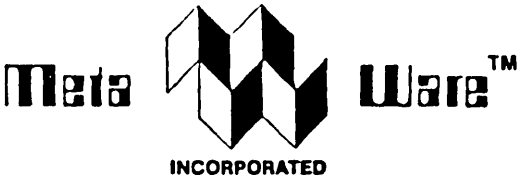


High C ™

Programmer's Guide

Version 1.2 for Concurrent DOS

by MetaWare™ Incorporated

High C ™

Programmer's Guide

Version 1.2 for Concurrent DOS 286

NOTICES

The software described in this guide is licensed, *not sold*. Use of the software constitutes agreement by the user with the terms and conditions of the End-User License Agreement packaged with the software. Read the Agreement carefully. Use in violation of the Agreement or without paying the license fee is unlawful.

Every effort has been made to make this guide as accurate as possible. However, MetaWare Incorporated shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this guide, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this guide and the software that it describes.

MetaWare Incorporated reserves the right to change the specifications and characteristics of the software described in this guide, from time to time, without notice to users. Users of this guide should read the file named "README" contained on the distribution media for current information as to changes in files and characteristics, and bugs discovered in the software. Like all computer software this program is susceptible to unknown and undiscovered bugs. These will be corrected as soon as reasonably possible but cannot be anticipated or eliminated entirely. Use of the software is subject to the warranty provisions contained in the License Agreement.

A. M. D. G.

Trademark Acknowledgments

The term(s)	is a trademark of
Digital Research	Digital Research Inc.
Concurrent, CP/M	Digital Research Inc.
High C, MetaWare	MetaWare Incorporated
MS-DOS	Microsoft Corporation (registered tm.)
Professional Pascal	MetaWare Incorporated
UNIX	AT&T Bell Laboratories

Contents: Programmer's Guide page(s)

for The MetaWare High C ™ Compiler ..... *total 226 pp.*

Contents: Programmer's Guide page(s)

Contents: Programmer's Guide page(s)

Contents: Programmer's Guide page(s)

Contents: Programmer's Guide page(s)

Feedback, Please

(Upon first reading.)

We would greatly appreciate your ideas regarding improvement of the language, its compiler, and its documentation. Please take time to mark up the manual on your *first* reading and make corresponding notes on this page (front and back) and on additional sheets as necessary. Then mail the results to:

MetaWare™ Incorporated
412 Liberty Street
Santa Cruz, CA 95060

MetaWare may use or distribute any information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use that information. If you wish a reply, please provide your name and address. **Thank you** in advance, The Authors.

Page Comment

Feedback, Please

Page Comment

Preface

This is a guide to the operation of the High C compiler as implemented for the Concurrent DOS 286 1.2 or later operating system — hereafter abbreviated to just "Concurrent" per Digital Research custom — running on the Intel 80286 microprocessor and using the Intel Object-Module Format (OMF). The compiler generates code for any of the Intel 8086/88/186/188/286 family of microprocessors.

Due to the time delay required for typesetting, this is a "beta" Programmer's Guide derived from the Programmer's Guide for High C running under MS-DOS on the 8086. As such, occurrences of MS-DOS-isms remain in sections where we have merely reproduced the typeset MS-DOS pages rather than dot-matrix-print the Concurrent equivalent. In these cases the Concurrent equivalent is almost identical to the MS-DOS version, except for minor name replacements.

Therefore, please bear in mind that:

- occurrences of "MS-DOS" should be replaced by "Concurrent";
- occurrences of ".EXE" should be replaced by ".286";
- occurrences of ".LIB" should be replaced by ".L86";
- there is no AUTOCFIG program for Concurrent that sets the number of tree pages for the compiler (see the Options section);
- the page numbers in the Table of Contents are not always exact, but are close;
- the index has not been updated;
- options **cram** and **tmpi3** are not applicable;
- documentation for use with embedded applications has been retained, although there are no tools we know of that take Concurrent's .286 load format and down-load it to an embedded system. If embedded applications are desirable we

recommend instead linking under MS-DOS. Tools are available for down-loading MS-DOS .EXE files.

MS-DOS compatibility. Programs can be compiled under Concurrent and the object modules transferred for linking under any PC/MS-DOS 2.0 or later operating system. Similarly, programs can be compiled under MS-DOS and the object modules transferred for linking under Concurrent. As long as no use is made of operating-system-specific facilities, programs link properly under either operating system.

This document does not treat linking under MS-DOS; see the High C Programmer's Guide for PC/MS-DOS for such information. Also note that the Concurrent-resident compiler generates 80286-specific code by default; such code will not run on MS-DOS systems not hosted on an 80286 — the 80286 code can be turned off by turning Off the toggle **286**.

1
Introduction

This is a guide to the operation of the High C compiler as implemented for PC/MS-DOS 2.0 or later operating systems running on the Intel 8086/88/186/188/286 family of microprocessors using the Intel Object-Module Format (OMF).

Unless otherwise stated, any reference to "8086" throughout this guide applies equally to all of those processors. Likewise, any reference to "8087" applies equally to the 80287 numeric co-processor.

Thus, the guide applies to the Victor 9000, IBM PC (XT, AT), HP 150, Wang Professional Computer, TI Professional Computer, Compaq, and others.

Embedded applications. Programs may be linked on an MS-DOS machine and down-loaded to an "embedded application", i.e. an 8086-family processor without MS-DOS. The code may even be placed in ROM.

The source code of the initialization module that establishes the High C environment is provided and may be tailored for specialized applications. To bring up High C's I/O facilities for the embedded application, a few well-defined modules must be rewritten. See Section *Embedded Applications.*

High C is designed to facilitate serious professional programming on the 8086 — with or without the 8087 or 80287 numeric co-processor. It is both true to ANSI standard C (still under development at this writing) as a subset and somewhat extended.

C is a mixed-level systems language designed by Dennis Ritchie at AT&T's Bell Laboratories. It grew in popularity because of its use in implementing UNIX, its elegant (and deceptive) simplicity, and its close-to-the-machine features. As its

popularity grew, many software developers have used it for "real-world" applications as well as for systems software.

Later implementations of C were extended to add enumeration types and a few other features. More recently many extensions have been proposed to make C a safer language while still being consistent with the philosophy of the original language. Today there is a core language being standardized by the American National Standards Institute (ANSI) and many C compilers with divergent extensions.

High C includes (what most likely will be) ANSI standard C and also provides extensions that were carefully designed to be consistent with the philosophy of C. Generally some of the best features of such other languages as (MetaWare's Professional) Pascal, Modula, and Ada, were borrowed as extensions. Incompatibilities were minimized by introducing a minimum of new key words and by not otherwise modifying the original syntax. Yet the extensions are such that they will be flagged by any standard-conforming compiler.

Portability. Standard C programs can be compiled with an ANSI option that turns off the extensions and reduces the compiler to the standard core. Or such programs can be gradually up-graded by not choosing the ANSI option and adding more and more extensions; a minimum of program modifications are needed to start with because the extensions generally do not conflict with the standard.

Furthermore, High C is hosted on, and cross-compiles consistently among, mainframes, (super-)minis, and micros, with function and quality paramount throughout the implementations. That is, it is without the typical limitations found on other implementations of C on microcomputers — and even mainframes — that defeat serious professional programming.

Safety, efficiency. While the close-to-the-machine features of C are available, High C supplies all of the new strong type-checking specified in ANSI C. In addition, the compiler provides many checking features that are usually available only in a separate "lint" program. Thus one gets both efficiency

and reliability. It is an excellent language for both applications and systems programming. We regard only Professional Pascal as being superior in this regard.

Other important features and extensions include:

- nested functions complete with up-level references as in Pascal;
- nested functions passable as parameters to other functions as in Pascal;
- support for the entire 8086 family: 8086, 8088, 80186, 80188, 80286, and the 8087 and 80287 co-processors;
- support for ROM-able code for "embedded applications";
- a full set of memory models: Small, Compact, Medium, Big, and Large;
- three Integer ranges, and three IEEE Real precisions;
- intrinsic functions such as `_abs`, `_min`, `_max`, `_fill_char`, etc., for efficiency;
- many compiler controls and options, including one for strict ANSI standard checking; and
- many optimizations, some of which are normally found only in mainframe compilers. These include:

  common subexpression elimination,
  retention and reuse of register contents,
  dead-code elimination,
  jump-instruction size minimization,
  constant folding,
  short-circuit evaluation of Boolean expressions,
  numerous strength reductions,
  fast procedure calls, and
  automatic allocation of variables to registers (except on the 8086, where it does more harm than good).

Thus, High C generates better code than most C compilers.

Guide. This guide contains all MS-DOS/8086-specific information necessary for using the compiler effectively. The major components of the product are listed below.

The reader who is new to the product should scan the Table of Contents to get an overview of the guide. Briefly, there is a section each on how to compile, link, and run. Then there are three sections on how to use the compiler controls: options, profiles, "ipaths", configurations, pragmas, and toggles.

Then there are sections on machine specifics, such as floating-point, memory models, storage mapping, machine architecture, and debugging. Then there are sections on external naming conventions and communicating among modules written in MetaWare's High C, plain C, MetaWare's Professional Pascal, and assembly language.

The final sections are on the large library of utilities that are provided, on embedded applications, listings, error messages, cross-references, and defaults and limits.

An extensive index is also provided for quick reference to all subsections that discuss or significantly relate to each topic.

Some sections of this guide are also used as sections of guides for High C and other MetaWare compilers on other machines and operating systems, most particularly in cross compilers to the 8086 family. Thus, in those sections there are occasionally references to UNIX and DEC's VMS, e.g.

This guide does *not* explain the High C language or its extensions. They are treated in the **MetaWare High C Language Reference Manual**. Nor does the guide or the manual attempt to teach C programming; consult the **Reference Manual** for references to several standard-C textbooks. The Library provided with High C is described in the **High C Library Reference Manual**. For further information on the 8086 consult:

Morse, S. P. **The 8086/8088 Primer**, Second Edition, Hayden Book Company, Inc., 1982.

Components. The High C software package has five parts:

1. the compiler,
2. the run-time libraries necessary for producing an executable program from the output of the compiler,
3. a set of source-code "header" files used to access various utilities, some of which are target-dependent,
4. a cross-reference mechanism that will work on several program "modules" at one time to produce a cross reference and optionally an annotated cross-reference listing of the program, and
5. a set of useful UNIX™-like utility programs that make MS-DOS more livable. Included among the utilities are programs to recursively list directories (`ls`) and to search for strings (`fgrep`). `fgrep` is an attractive alternative to a bulky printed paper cross-reference, since it is very fast. Another utility (`find`) is used by the INSTALL program to load the compiler. The utilities are described in the on-line documentation.

Requirements. The memory and disk requirements of High C are given in the next chapter.

Installation. An **Installation Guide** is on the distribution media — see the file named INSTALL.DOC — in addition to being in the typeset documentation. The compiler is essentially self-installing.

The distribution media also contain several benchmark and demonstration programs; see the **Installation Guide**. And the file named README contains last minute notes and release information.

Key Words and Phrases. Following most subsection headers, in brackets and small typeface, is a collection of [key words and phrases] related to that subsection. They give a quick idea of what the section is about.

2
Invoking the Compiler

2.1 The Compile Command

[.C; .OBJ]

The compiler is named HC.EXE, so it is invoked with the hc command. The syntax of the command is illustrated next:

```
hc path_name [options]
```

where "path_name" is the path name of the file containing the module to be compiled. The file name extension may be omitted, in which case the default extension ".C" is assumed.

The "options" activate various compiler options, which are summarized in the table below. A detailed description of each is presented in Section *Compiler Controls.*

Command Option	*Default*
-[no]ansi	-noansi
-[no]asm	-noasm
-[no]debug	-nodebug
-define Macros	
-ipath Path	
-lines_per_page nn	-lines_per_page 60
-list File	-list @
-mm Model	-mm Small
-[no]object [File]	-object source.OBJ
-off Toggles	
-on Toggles	
-[no]profile [File]	-profile HC.PRO
-tmpi1 File	
-tmpi2 File	
-tmptp File	
-tpages nn	
-[no]xref File	-noxref

If no errors are detected, the compiler produces a relocatable object file in the current working directory with the same name as the input file but with the extension ".OBJ".

The default extensions ".C" and ".OBJ" are configurable; see Subsection *Configuring the Compiler*, Section *Compiler Controls.*

Examples:

The following command compiles the program in file MY_DIR/SORT.C and generates an object file named SORT.OBJ in the current working directory:

```
hc my_dir/sort
```

To indicate that input should come from the standard input, use "@" in place of the input file name. Likewise, to direct any output to the standard output rather than a file, use "@" in place of the output file name. For example, to compile a (presumably short) program directly from the keyboard, type

```
hc @
```

followed by the program to be compiled. ^Z terminates keyboard input.

2.2 Search Paths for Input Files

[option ipath]

The compiler has the ability to search a list of directories for an input file, such as an Include file. The list of directories to be searched is specified by the various "ipath" environment symbol, -ipath option, and Ipath pragma. See Section *Compiler Controls* for details.

2.3 Disk Storage Requirements for Temporary Files

[MS-DOS: options tpages, tmptp] *(MS-DOS only.)*

The compiler requires storage for temporary files during compilation. When a program exceeds a certain size, depending upon the memory available and the setting of certain parameters (see -tpages in Section *Compiler Controls*), the compiler's internal representation of the program is placed in a file that is randomly accessed. Since floppy disk access is slow, a hard disk is highly recommended for the temporary file; see the -tmptp option in Section *Compiler Controls.*

2.4 Memory Requirements *(Concurrent only.)*

[Concurrent: options ansi, tpages]

Because of the large amount of function in the compiler, it is large — almost 1/2 megabyte of object code — and should be run from a hard disk.

The distribution comes with two versions of the compiler. The first version loads slowly due to Concurrent's slow loader, but uses minimum memory since it is overlayed; it occupies ???K of code. The second version loads very quickly but occupies 453K of code space. The compiler needs a minimum of 100K of data space to operate, even with tpages set to 40 (this is small). We recommend a tpages setting of at least 100 for reasonabe-size compilations; 200 or more is preferable for large compilations. Each tree page costs 768 bytes.

See Section *Compiler Controls* under the **-tpages** option for more information about space needs.

NOTE: At this writing, the overlayed version of the compiler is not available.

3

Linking a Compiled Program

3.1 Compilation Units or "Modules"

[object modules, load module]

A High C program consists of one or more compilation units or "modules" that are compiled separately to produce object modules. Exactly one of these modules is a "main (program)" module; the rest are "non-main" modules.

The main module is that module containing a definition of the function "main". This function is called after the High C run-time environment has been established.

To execute the program, one or more object modules must be linked with various functions from one of the High C Run-Time Libraries to form a load module.

3.2 Run-Time Libraries; Link Errors

[memory models: Small, Compact, Medium, Big, Large; 8087 co-processor or emulator; linkage errors: unresolved external; "SMALL?", "COMPACT?", "MEDIUM?", "BIG?", "LARGE?"; library names]

There are two Run-Time Libraries for each of the five 8086 "memory models" supported: one that contains an 8087 floating-point emulator and one that does not; see Sections *Floating-Point Support* and *Memory Models.* Thus there are ten libraries in all.

Each Run-Time Library contains functions that set up the environment, perform I/O, manage the heap, provide floating-point support, etc. See Section *Utility Packages.*

Library Names. The naming convention of the libraries is as follows. The first two characters are HC. The next is the first letter of the model name: Small, Compact, Medium, Big, or Large. Finally, if the library contains the 8087 emulator, its name ends with "E", but if the library is for use only with an 8087 co-processor then "C".

Memory Model	Library Name with 8087/80287	
	emulator	co-processor
Small	HCSE.LIB	HCSC.LIB
Compact	HCCE.LIB	HCCC.LIB
Medium	HCME.LIB	HCMC.LIB
Big	HCBE.LIB	HCBC.LIB
Large	HCLE.LIB	HCLC.LIB

Small is the default model so either HCSC(M) or HCSE(M) is used if no explicit memory model is specified.

Link errors. Object modules compiled for different memory models are incompatible so should not be linked together. To prevent such an error the compiler emits external symbol references that are resolved only if like object modules are linked. The names of the symbols are:

SMALL?, COMPACT?, MEDIUM?, BIG?, and LARGE?.

The main module defines the appropriate symbol and all non-main modules reference it.

Thus if the linker diagnoses LARGE? as being unresolved, for example, then one or more of the object modules being linked were compiled with the Large model, but the main program

3.3 Linking under Concurrent: LINK86 and LINK

[segment class restrictions]

VAX/VMS cross. Programs compiled on the VAX that are to be run under Concurrent may be linked under Concurrent using the standard linkers. This requires that the object modules be down-loaded from the VAX to the Concurrent machine prior to linking.

The Concurrent version of the run-time libraries must reside on the Concurrent system in order to link a program. These libraries may be down-loaded from the distribution supplied with the cross compiler. Or, if the Concurrent-resident compiler is available, its libraries may be used.

Concurrent-resident. Two linkage editors available under Concurrent are Digital Research's LINK86 and MetaWare's modification of that linker, LINK. For detailed information on LINK86 consult the **Programmer's Utilities Guide for the CP/M-86 Family of Operating Systems and Concurrent DOS-286** by Digital Research, Inc. The modifications that MetaWare has made to that linker are presented below. Here we present a brief summary of how to use the linkers in conjunction with MetaWare compilers.

Linking with LINK86

Create a load module and symbol table file from MetaWare object modules by invoking **link86** as follows:

link86 <Load_module>[li,ma]=<Object_list>,HC??.L86[se]

<Load_module> is the name to be assigned to the resultant load-module file. Its default extension is .CMD.

[li,ma] causes the inclusion of symbols from the library in the symbol table file (li), and the generation of a map (ma). The symbol table file has extension .SYM and the map file .MAP. The symbol table file can be used for symbolic debugging in conjunction with Digital Research's

SID-286 symbolic debugger. See the **Programmer's Utilities Guide for the CP/M-86 Family of Operating Systems and Concurrent DOS-286** for more information on SID-286.

<Object_list> is a list of object-module or library file names separated by plus signs or blanks. The file name extensions may be omitted, in which case .OBJ is assumed. For libraries, the extension .L86 must be supplied, and the [se] option given to cause a library search (as opposed to inclusion of the entire library in the link).

HC??.L86[se] indicates searching ([se]) of the appropriate High C Run-Time Library. The "]" can be omitted if it is the last character on the line.

Lengthy link specifications can be placed in a linker "input" file suffixed with .INP. Such a file can be specified only last on the command line.

Examples:

```
link86 SORT[ma]=SORT,HCSC.L86[se
```

links the object module SORT.OBJ to produce the load module SORT.CMD and the load map SORT.MAP.

```
link86 GAMMA=ALPHA,BETA,DELTA.L86[se],HCSC.L86[se
```

links the two files ALPHA.OBJ and BETA.OBJ with the user library DELTA.L86 to produce the load module GAMMA.CMD.

```
link86 GAMMA=INPUTS[in]
```

where file INPUTS.INP contains

```
ALPHA,BETA,DELTA.L86[se],HCSC.L86[se
```

does the same. You can also use

```
link86 INPUTS[in]
```

where file INPUTS.INP contains

```
GAMMA=ALPHA,BETA,DELTA.L86[se],HCSC.L86[se
```

Linking with LINK

MetaWare's modification of LINK86, LINK, operates the same way as LINK86 does, with the following additional features:

- LINK takes full path names as input.
- Unlike LINK86 "[se" is unnecessary after .L86 files for LINK, which assumes this by default. Specify "[nose" ("[nosearch") to include the entire libraray rather than to search it for needed objects only.
- The default output file extension is .286 instead of .CMD, reflecting standard usage under Concurrent.
- There is no limitation of 2048 characters on the length of a .INP file.
- .INP files can be nested.
- .INP files can appear anywhere a file can appear on the command line, not just at the end of the command line.
- More than one comma can separate file names. This permits the building of a convenient batch file that links up to nine object modules:

  link %1=%1, %2, %3, %4, %5, %6, %7, %8, %9, hcsc.186

  When only three parameters (e.g. f1, f2, and f3) are supplied to this batch file, the result is illegal to LINK86 but legal to LINK:

  link f1=f1, f2, f3, , , , , , hcsc.186

- A new option "pubcode" treats all private code segments as public.

Segment class restrictions

Although Intel OMF supports any segment class names, the Concurrent's linkers and Concurrent itself supports only four class names: CODE, DATA, HEAP, and STACK. Do not use the compiler options Dclass and Cclass to change those class names; the linker will produce an invalid executable without warning.

3.4 Linking for Embedded Applications

[MS-LINK]

Under Concurrent. Programs that are to run within embedded applications may be linked under Concurrent. However, we know of no tools that can load a Concurrent executable file in an embedded system. We recommend instead using Microsoft's MS-LINK linker; there are tools that support the loading of MS-DOS executables in an embedded system.

Prior to linking, system-dependent modules of the runtime libraries must be modified as required for the embedded application. See Section *Embedded Applications.*

3.5 Linking High C and Professional Pascal

High C and Professional Pascal modules can be linked together, and nearly the full resources of each language can be used in this mode. Generally linking consists in supplying *both* Pascal and C libraries to the linker, and sometimes in supplying certain additional ".OBJ" modules, as is described in the Notes below.

Two questions must be answered to determine the appropriate link command.

1. In which language is the main program written (where execution begins)?

2. Does any module written in the other language use the I/O system of that language? For example, do the Pascal routines call the intrinsics Writeln and Readln, or do the C functions call the library functions "printf" and "scanf"?

After these two questions are answered, the HC/PP Link Table below can be used to determine how to link the program. We assume that "main.obj" is the object module corresponding to the main program. The library suffix letters "xx" in the table stand for one of the appropriate memory models and co-processor/emulator options; "xx" must be the same for both the Pascal and C libraries. For example, the C

library "HCME.LIB" (Medium model, Emulator library) must be used with the Pascal library "PPME.LIB".

HC/PP Link Table.

- Main program language is C; no I/O done by Pascal:

```
link prog=main, <other objects>, hcxx, ppxx
```

- Main program language is C; I/O done by Pascal:

```
link prog=main, <other objects>, finit, system1, hcxx, ppxx
```

See Notes 3 and 4 below.

- Main program language is Pascal; no I/O done by C:

```
link prog=main, <other objects>, ppxx, hcxx
```

- Main program language is Pascal; I/O done by C:

```
link prog=main, <other objects>, cfinit, ppxx, hcxx
```

See Notes 3 and 5 below.

Notes:

1. If the main program is written in High C, the Professional Pascal command-line argument package in ARG.PF cannot be used. If the main program is written in Professional Pascal, the High C variables "argc" and "argv" are unavailable. In short, modules in only one language can access the arguments on the command line.

2. Pascal and C share the same global variable "errno" (or ErrNo, in Pascal documentation). In C, this variable is defined in the header file "stdio.h", and in Pascal, in the interface file STATUS.PF. Any modifications made to "errno" in modules in one language affect modules in the other language.

3. Normally in C "errno" is never cleared by library functions; it is instead the responsibility of the user to check and clear "errno". However, when Pascal I/O is used in combination with a C program, "errno" is cleared upon successful Pascal I/O completion.

4. FINIT.OBJ is provided for each memory model in the Professional Pascal distribution. Its inclusion overrides a "dummy" version in the HCXX libraries and permits the initialization of the Pascal I/O system.

5. CFINIT.OBJ is provided for each memory model in the High C distribution. Its inclusion overrides a "dummy" version in the PPXX libraries and permits the initialization of the Pascal I/O system.

6. When the Pascal Run-Time Library is linked before the C Library, any C functions registered with the C Library function "onexit" are *not* called at program termination. Furthermore, after any successful call to Concurrent, the variable "errno" is cleared. Such calls are made by the C I/O system; when dealing with High C only, "errno" is never cleared by the library.

3.6 Post-Mortem Call-Chain Dump

When the run-time system detects a fatal error and aborts, a dump of functions currently active may be obtained if the program is linked with the object module STKDMP.OBJ that is provided for each memory model. By default a dummy dump function is linked in that instead of producing a dump prints a message that the dump is unavailable. See Section *Debugging* for more information.

When linking in STKDMP.OBJ, the Run-Time Library PTOC.L86 must be specified; it is supplied for each memory model. Specify PTOC.L86 *before* all other libraries in the list.

3.7 Post-Mortem Heap Dump

When the run-time heap manager detects an error, a dump of the current contents of the heap may be obtained if the program is linked with the object module HEAP1.OBJ that is provided for each memory model. By default a dummy dump function is linked in that instead of producing a dump prints a

message that the dump is unavailable. See Section *Debugging* for more information.

When linking in HEAP1.OBJ, the Run-Time Library PTOC.L86 must be specified; it is supplied for each memory model. Specify PTOC.L86 *before* all other libraries in the list.

3.8 Case Sensitivity in Linking

C is a case-sensitive language. For example, the identifiers Paycheck and paycheck are regarded as distinct in a C program. In addition, since Concurrent linkers always respect case, one must observe case across compilations.

3.9 Minimizing Program Size

```
[toggle Optimize_for_space; dummy argument processor _mwset_up_args;
fixed-size heap C_HEAP.C; dummy file close C_CLOSE.OBJ; dummy
C_SCANF.OBJ, C_PRINTF.OBJ]
```

Many applications require minimal program size. There are several options provided the programmer for reducing the size of a program.

First, the toggle Optimize_for_space should be turned on in source modules for which code density is preferred over execution speed. Use "**pragma** On(Optimize_for_space);"; see Section *Toggles*.

Second, some facilities provided by default for use at run-time can be eliminated. This removes the program code that implements the facilities, thus saving space. In general the facility is removed by providing a dummy function that does nothing. The facilities are:

- **The argument processor.**

This code is used to compute the values for argc and argv to be passed to the main program. If "argc" and "argv" are *not* used, the code for processing them can be eliminated by supplying a dummy function named "_mwset_up_args". The dum-

my function gets called instead of the one that sets up "argc" and "argv", for a savings of around 300 bytes. Here is a sample declaration of such a dummy function:

```
_mwset_up_args() {return 0;}
    /* To suppress argument processing;   */
    /* the returned 0 gets put into argc. */
```

- **The heap manager.**

If little or no heap is used, the heap manager can be essentially eliminated: link in the object produced from compiling C_HEAP.C, supplied with the distribution. This simple heap manager provides a heap whose fixed size is specified in the source. See the source for more information.

- **The I/O file close functions.**

If the program does no I/O other than to the terminal, the functions that close any open disk files can be eliminated: link in C_CLOSE.OBJ. This object file is supplied for each memory model.

- **"scanf" for real numbers.**

If the program does not use "scanf" to scan literals of types **float**, **double**, and **long double**, the functions that convert their source representations to (internal) real numbers can be eliminated: link in C_SCANF.OBJ. This object file is supplied for each memory model.

- **"printf" for real numbers.**

If the program does not use "printf" to print values of types **float**, **double**, and **long double**, the functions that convert real numbers to their printed representations can be eliminated: link in C_PRINTF.OBJ. This object file is supplied for each memory model.

3.6 Post-Mortem Call-Chain Dump

When the run-time system detects a fatal error and aborts, a dump of functions currently active may be obtained if the program is linked with the object module STKDMP.OBJ that is provided for each memory model. By default a dummy dump function is linked in that instead of producing a dump prints a message that the dump is unavailable. See Section *Debugging* for more information.

When linking in STKDMP.OBJ, the Run-Time Library PTOC.LIB must be specified; it is supplied for each memory model. Specify PTOC.LIB *before* all other libraries in the list.

3.7 Post-Mortem Heap Dump

When the run-time heap manager detects an error, a dump of the current contents of the heap may be obtained if the program is linked with the object module HEAP1.OBJ that is provided for each memory model. By default a dummy dump function is linked in that instead of producing a dump prints a message that the dump is unavailable. See Section *Debugging* for more information.

When linking in HEAP1.OBJ, the Run-Time Library PTOC.LIB must be specified; it is supplied for each memory model. Specify PTOC.LIB *before* all other libraries in the list.

3.8 Case Sensitivity in Linking

C is a case-sensitive language. For example, the identifiers Paycheck and paycheck are regarded as distinct in a C program.

4
Running a Program

4.1 The Run Command under MS-DOS

[load module; I/O redirection]

After a composite program has been linked together it may be run. The load module is invoked simply by typing its name.

For example assume the file HELLO.EXE contains the load module corresponding to the following program.

```
main() {
   printf("Hello.\n");
   }
```

Then typing the command line:

```
hello
```

causes the line "Hello." to be displayed on the terminal and typing

```
hello > output.fil
```

redirects the output of the program to file OUTPUT.FIL.

4.2 Command-Line Parameters

[argc, argv; minimizing program size]

It is possible to send data to the program about to be run by supplying parameters on the command line that executes the program. Parameters are supplied positionally; what separates parameters on a command line is described later.

The parameters are passed to the main program as an array of strings. To access the parameters declare the main program as follows:

```
main(ArgC,ArgV)
   int ArgC; char *ArgV[];
   {
   ...
   }
```

ArgC is the number of arguments, numbered 0 through ArgC - 1. ArgV is an array of pointers to the arguments: ArgV[i] points to the i^{th} argument, $0 \leq i < \text{ArgC}$. Each argument is represented by a C NUL-terminated string. In addition, ArgV[ArgC] is defined and always points to an empty string.

The zeroth argument is the name of the running program.

The other arguments consist of the text appearing on the line used to invoke the program, where each argument is separated from another by a sequence of blanks or tabs. In addition, if an argument begins with the double-quote character ", then it should end with ", and " can be denoted within such an argument by using the two-character sequence \". The enclosing quotes are stripped off and each \" is compressed to just ".

For example, the following program prints out all its arguments, including the program name if made available:

```
main(ArgC,ArgV)
   int ArgC; char *ArgV[];
   {
   int j;
   for (j = 0; j < ArgC; j++) {
      printf("Argument #%d is %s\n",j,ArgV[j]);
   } }
```

When invoked via, for example,

```
myprog first second "third\" arg" last
```

the program prints

```
Argument #0 is C:/CC/MYPROG.EXE
Argument #1 is first
Argument #2 is second
Argument #3 is third" arg
Argument #4 is last
```

assuming the full path name of the program executable is `C:/CC/MYPROG.EXE`.

If `ArgC` and `ArgV` are *not* used, the code for processing them can be eliminated. See Subsection *Minimizing Program Size* of the prior section.

5
Compiler Controls

MetaWare compilers support many controls that direct compilation and cause various information to be produced by the compiler. There are two classes: "command-line options" (or "command qualifiers" on VAX/VMS) and "pragmas".

Pragmas are described in Section *Compiler Pragmas.* Some of them can revise what is specified in the command line.

The first two subsections here apply to both Professional Pascal and High C. The few places where the distinction of languages is relevant are clearly delineated.

5.1 Command-Line Options (Qualifiers)

```
[compiler-execution environment; toggle options, non-toggle options; ansi = standard;
asm = machine_code - assembly listing; cram - 8086 memory requirement reduction;
debug - symbol-line-type records; Emit_line_records, Emit_line_table; define -
#define macros; ipath - initial value; lines_per_page - set the number; list -
file-name; mm = memory_model; object - file-name; off, on - toggles; profile -
file-name; tmpil-2-3 - 8086 temporary intermediate file-name; tmptp - 8086 tree page
file-name; tpages - 8086 number of tree pages; xref = cross_reference - listing,
file-name]
```

Command-line "options" or "qualifiers" are given to the compiler in the execution environment. They apply for the entire compilation unless they are overridden (where allowed) by `pragma` options within the program being compiled. However, `mm` is unusual in that it supersedes pragma `Memory_model`.

The command-line options are all "directives". Two of them, `on` and `off`, can be used to set the initial value of any compiler "toggle". Toggles can also be turned On or Off in program source via pragmas. All toggles are documented in Section *Compiler Toggles.*

Some toggles that are typically used when running the compiler are List — produce a listing, Quiet — do not

announce each compiler phase as compilation progresses, Summarize — produce a statistics summary, and 186 (286) — generate code that uses the special instructions of the 80186 (80286) processor.

The other options specify such things as the " ANSI mode", getting a (pseudo-) assembler listing, and the names of listing, intermediate, paging, debug, and object files.

Options are given to the compiler on the command line by preceding each by a hyphen "-" and following it by parameters where applicable. Most option names may be truncated to just a few leading characters. In addition, many options may be negated by prefixing the name with "no", as in "noansi".

The table below gives the name of each option, its minimum truncation, whether it is negatable, a short summary of the option's effect, and the default value of the option. More detailed explanations follow the table.

Most of the options have a fixed default value, but for some this default value may be configurable with the config program; see Subsection 5.4. In the table, "*Default:* ddd" indicates that the default is fixed and is ddd. The phrase "*Default (configurable):* ddd" indicates that the default value is configurable, but as the compiler is distributed, the default is ddd. No default value is given when such would be irrelevant, e.g. for an option that has no effect unless it is present. As a final wrinkle the autocfig program may change the values of the options cram and tpages, when the compiler is installed, depending upon available memory.

In general the compiler interprets the character "@" as denoting the standard input when it appears where an input file name is expected, and standard output in an output position.

Examples: *(Replace* pp *with* hc *for High C.)*

```
pp input -obj obj.fil -tpages 150
concoct | pp @ -obj concoct.obj -asm > list.fil
```

The second example shows the compiler taking its input from the standard input ("@"), as produced by a hypothetical program concoct.

Maximum truncation	*Negatable?*	*Option name, parameters*	*Effect, Default*
an	Y	ansi	Accept only ANSI standard programs. *Default (configurable):* noansi.
as	Y	asm	Produce a (pseudo-) assembler listing to the listing file. *Default:* noasm.
cr	Y	cram	Try to conserve memory, which is limited. *Default (configurable):* nocram, but the configuration may be changed automatically by autocfig; see Subsection 5.4. (Not on VMS.)
debug	Y	debug	Produce Intel OMF debug records. *Default:* nodebug.
def	N	define Mdefs	#define the listed Mdefs — macro definitions — before processing the source file.
ip	N	ipath Path	Supply an initial value for the Ipath pragma.
lin	N	lines_per_page nn	Specify the number of lines per page to be nn. *Default:* lines_per_page 60.
lis	N	list File	Specify the listing file name to be File. *Default:* list @.
mm	N	mm Model	Specify the memory Model: Small, Compact, Medium, Big, or Large. *Default (configurable):* mm Small.

ob	Y	**object** File	Specify the object file name to be File. *Default (configurable):* **object** source.OBJ.
off	N	**off** Toggles	Turn the Toggles off; see Section *Compiler Toggles.*
on	N	**on** Toggles	Turn the Toggles on; see Section *Compiler Toggles.*
pr	Y	**profile** File	Specify the profile file name to be File. *Default (configurable) for PP:* **profile** pp.pro *(for HC:* hc.pro*).*
tmpil	N	**tmpil** File	Specify the instruction file name to be File. (Not on VMS.)
tmpi2	N	**tmpi2** File	Specify the intermediate-language file name to be File. (Not on VMS.)
tmpt	N	**tmptp** File	Specify the tree paging file name to be File. (Not on VMS.)
xr	Y	**xref** File	Produce a cross reference listing. Specify File as the cross-reference file name, suffixed by .XRF if no extension is given. *Default:* **noxref**.

ansi. Specifying **ansi** causes only ANSI standard programs to be accepted by the compiler. For this to work the compiler must have access to the files PPANSI.ST and PPANSI.PT (for High C: HCANSI.ST, HCANSI.PT, and HCANSIP.PT) in its search path; see Subsection 5.3. These files contain tables used for scanning and parsing programs that must obey ANSI restrictions; the files are included on the distribution media. (Also known as STANDARD on VMS.)

Notes: The Pascal standard is specified in ANSI document X3J11-85-102, August, 1985. The ANSI standard for C is currently under development; the **ansi** option reflects the draft standard as of the publication date of this guide. The main use of this option is to turn off MetaWare extensions.

asm. Specifying **asm** causes a (pseudo-) assembler listing of the generated code to be put in the listing file. The assembler listing is annotated with lines from the compiled source file — but not with lines from any other (included) files, for technical reasons. These lines appear as comments just preceding the corresponding assembler instructions. (VMS: called MACHINE_CODE.)

debug. Specifies that Intel OMF debug records are to be produced in the object file, i.e. Intel symbol, line, and type records. This information can be used by debuggers on Intel development systems, but Concurrent does not support a debugger that can take advantage of any of the records.

tmpi3. The compiler normally sends certain debug information to a default temporary sequentially-accessed file. This directive specifies an alternate file name. It can be used when there is not enough room on the current drive to hold the file. (Not on VMS.)

on Emit_line_records. This toggle is off by default. It is a part of the debug directive and as such is active when debug is. It specifies that only the line records are to be produced. Emit_line_records can be turned on independently of debug.

on Emit_line_table. This toggle is on by default. It causes line numbers to be emitted in the object file for use in call-chain stack dumping, such as during debugging and when a program aborts at run time. The line table takes up no more than one byte per source line; it does not affect the speed of the compiled code.

define. Supplies initial definitions of macros. One or more definition may be specified after define. Each definition takes the form

```
      name
or    "name expression"
```

(including the " quote marks).

These two forms "turn into" the lines

```
      #define name
and   #define name expression
```

respectively, and are processed by the compiler as its first input, even before reading the profile.

ipath. Sets the initial value of the Ipath pragma. *Note:* when the Ipath pragma is specified in the source program, the ipath option is overridden from that point on. See Subsection 5.3 for an explanation of "ipaths".

lines_per_page nn. Every nn lines on a listing, a page eject is issued. The default (60) is appropriate for most 6-lines-per-inch printers, which have a total of 66 lines per page. The

setting of lines_per_page is intended to allow some blank space at page boundaries. When using 8-lines-per-inch mode on some printers, typically there are 88 lines per page and so lines_per_page should be set to 80 or 82.

If lines_per_page is set to zero, the number of lines per page is assumed infinite, so the periodic page ejects are eliminated.

list. The compiler normally sends all output to the listing file which is by default the standard output. To re-direct the output to a file, use MS-DOS output redirection, e.g. "pp xyz > output.fil" (or hc). Alternatively the list directive tells the compiler to send the listing output to a specified file. If the file name specified after list has no extension, ".LST" is appended to it. Note that this option does *not* turn on the listing of the compiled source; use on List for that (List is also the name of a toggle; see Section *Compiler Toggles*).

mm. Specifies the memory model for which the compiler is to target its generated code. The five memory models supported are Small, Compact, Medium, Big, and Large. This option overrides any Memory_model pragma appearing in the program. See Section *Memory Models* for a description of the models. (Known as MEMORY_MODEL on VMS.)

object. The object file name defaults to the name of the input file with ".OBJ" as the extension, and with any directory-path prefix removed. The compiler's default object file name may be changed with the object option. If the name after object has no extension, ".OBJ" is appended. noobject specifies that no object code is to be emitted; this is a good way to obtain syntax checking or a cross-reference without the expense of code generation.

The default extension ".OBJ" may be changed by configuring the compiler; see Subsection 5.4.

on, off. Turns On or Off various compiler toggles. All toggles are documented in Section *Compiler Toggles*. Follow-

ing **on** or **off** are one or more toggle names, as in "**on** List Emit_line_records".

profile. Specifies a profile. A profile is a file that is read by the compiler prior to reading the source file. The default profile is named "PP.PRO" ("HC.PRO"). **noprofile** specifies that no profile is to be read. See Subsection 5.2 for details.

tmpi1, tmpi2. The instruction file and intermediate-language file are sequentially-accessed temporary files used by the code generator phases. If the current drive does not have enough room for them, the **tmpi1** and **tmpi2** options may be used to re-specify their locations. Although each of these files can be on a floppy drive since each is sequential, they are both accessed at the same time so they should be on separate floppies if not on a hard disk. (Not on VMS: irrelevant; nor is tmpi3.)

tmpi3. See the paragraph after the **debug** paragraph above.

tmptp. The drive and file name of the tree paging file can be specified by using the **tmptp** parameter. The paging file is randomly accessed so it should be on a hard disk — that is, unless there is sufficient memory for a **tpages** setting large enough that no tree paging is necessary, in which case the file will not be used at all. (Not on VMS: irrelevant.)

tpages. The compiler constructs a tree representation of the source program during compilation. Only portions of the tree are needed in memory at any given time; unneeded tree portions are paged to disk. Disk traffic can be reduced and compilation speed increased by increasing the number of page buffers in memory. (Not on VMS: irrelevant.)

That increase happens automatically up to the maximum specified with the **tpages** option. The default is set by **autocfig** based on available memory at installation time; see Subsection 5.4. Generally the maximum should be set as high as possible short of mortgaging all remaining memory to the tree pager and thus leaving no memory for other compiler data structures.

If compilation aborts due to lack of memory, the tpages parameter should be reduced. However if the abort occurs during compiler initialization, the new value must be below 250 to have any effect. This "magic number" is due to an optimization in the implementation of the tree pager.

In particular, page buffers allocated when the compiler is started up are more efficiently accessed than all buffers are later. The reduction in efficiency occurs when the first additional buffer is allocated or when the first buffer is paged to the tree page file, whichever is first. The optimization is implemented for at most 250 buffers due to hardware limitations.

Thus it is advantageous at start-up time to allocate as many page buffers as possible up to 250 so that compilation can proceed for as long as possible before a new page must be allocated or a page must go to disk. Thus the compiler initially allocates either 250 or nn, whichever is smaller, where nn is the tpages parameter.

In summary the tpages parameter nn not only specifies the maximum number of page buffers that will ever be allocated but, if it is less than 250, it also specifies the initial number of buffers to be allocated; otherwise the initial number is 250.

Each page buffer holds 64 tree nodes of 12 bytes each for a total of 768 bytes per buffer. Thus 100 buffers (the default) requires 76,800 bytes of memory and the maximum initial number of 250 requires 192,000 bytes.

The paging file uses a single 1024-byte disk block to store each 768-byte page buffer (I/O is most efficient on 512-byte boundaries). When the program being compiled requires more tree pages than reside in memory, the paging file is used to hold some of the pages. The compiler aborts if for any reason there is insufficient space available for that file.

xref. See Section *Cross References* for details.
(Known as CROSS_REFERENCE on VMS.)

5.2 Profiles

[compiler-execution environment; source file prefix; HC.PRO, PP.PRO, .PRO]

A *profile* is a file that is treated as a prefix to the source file. Its purpose is to provide a way to customize the compiler to particular needs.

The default profile name is "PP.PRO" ("HC.PRO") but an alternate name may be specified with the `profile` option; see Subsection 5.1. `noprofile` means not to search for a profile.

One can use a profile to initialize toggles as desired, specify the desired memory model, `#define` some macros, and set any other pragma option. Then these new "defaults" can be applied to several or all source modules in a global fashion. In other words, the profile can be used to effectively alter compiler defaults.

The profile can also be used to "predefine" new identifiers and to change the meaning of existing predefined names.

Example. The profiles illustrated below would cause:
(1) the apparent default to be that one *does* get a listing,
(2) the memory model to be `Big` rather than `Small`, and
(3-HC) some "standard" definitions to be provided —
(3-PP) the type Integer to behave exactly like `LongInt`:

High C:

```
pragma  On(List);
pragma  Memory_model(Big);
#define UNIX
typedef
   enum{False,True} Boolean;
```

Professional Pascal:

```
pragma On(List);
pragma Memory_model(Big);
package; -- Unnamed so that
   -- no open is necessary.
   type  Integer = LongInt;
   const MaxInt  = MaxLong;
   const MaxInt  = MaxLong;
   end;
```

Implementation. The compiler reads the profile immediately prior to reading the source file. It opens the profile in the same manner that it opens an include or other input file, searching "ipaths" if necessary; see Subsection 5.3. If the profile is found, the programmer is notified; if the profile is not

found, the compiler immediately proceeds to read the source file without notice — even if `profile` was specified on the command line.

The latter is because `profile` only changes the name to look for, but does not "demand" that the file be found.

5.3 "Ipaths": Input File Search Facility

[define compiler-execution logical name; #include and pragmas Ipath, Include; AUTOEXEC.BAT = LOGIN.COM]

Input files. When the compiler must access an input file, there are three strategies for searching for the file.

(1) This strategy is used when the input file was specified via any of the three forms

```
#include "F"
pragma  R_Include("F");
pragma RC_Include("F");
```

Strategy 1. The compiler first attempts to open file F relative to the directory containing the file that contains the include. If the file open fails, Strategy 3 is then used.

(2) This strategy is employed when the input file was specified via a `#include` <F> directive, a "< >-include":

Strategy 2. The compiler iterates through a list of string prefixes specified by an < >-include "ipath" (see below for how to set it). It appends F to each prefix in turn, until the result is the path name of a file that can be opened successfully. If no open succeeds, Strategy 3 is used.

(3) This strategy is employed when (1) or (2) fails, and when an open request is not due to either form of `#include` directive or either of the relative-include pragmas, such as when the compiler attempts to open the primary source file, the profile, or a file specified by pragma Include or `C_include`:

Strategy 3. The compiler attempts to open the file in the current working directory. If the open fails, the compiler then proceeds to iterate through a list of string prefixes. It

appends the specified file name to each prefix in turn, until the result is the path name of a file that can be opened successfully. These prefixes are taken first from any specified by the programmer via pragma Ipath, and then from a Concurrent "logical name" normally called IPATH, a value for which is established by the Concurrent **define** command; see below.

Ipath specification. There are three distinct "ipaths" the compiler uses when opening an input file. Each uses the same syntax — a list of strings separated by ";" as illustrated below.

(1) The < >-include "ipath" can be specified only by configuring the compiler, and should be done when the compiler is installed and the decision is made where to put the standard ".h" header files provided in the distribution. The < >-include "ipath" typically references just the directory containing those files, but can be made to reference any sequence of directories. See Subsection 5.4 on configuring the compiler.

(2) Pragma Ipath can be set anywhere and changed any number of times. Typically, though, it is set in a profile.

(3) An appropriate place to set the Concurrent environment IPATH is in the Concurrent AUTOEXEC.BAT file. One of the string prefixes might be the directory path of the "standard include library", where the standard ".h" header files provided in the distribution are put.

Examples. In the profile or a source file

```
pragma Ipath("A:/INC/;C:/INC/;C:/USR/INC/");
```

or equivalently the Concurrent command

```
define IPATH=A:/INC/;C:/INC/;C:/USR/INC/
```

sets up three input-file prefixes. (The absence of blanks *is* significant.) If the compiler should subsequently encounter the statement:

```
#include "QSORT.CF"
```

it would try to open "QSORT.CF" in the directory that contains the file containing the **#include**. Were that to fail, the compiler would then try to open "QSORT.CF" in the current working directory (the directory in which the compiler was invoked). Were that to fail, it would try to open "A:/INC/QSORT.CF", then "C:/INC/QSORT.CF", and finally "C:/USR/INC/QSORT.CF", until it has a successful open. Were all attempts to fail, the compiler would abort with a "File not found." message.

(We have assumed used-defined header files have the extension ".CF" rather than ".H" so as to eliminate any possible conflict with any ANSI-standard header file names.)

Note that the name of the Concurrent logical name IPATH can be changed by configuring the compiler; see the next subsection. For example, the name could be changed to INCLUDE (or perhaps CIPATH so as not to conflict with the IPATH name of another MetaWare compiler, if it is in use, too). Then, the Ipath specification would be set by:

```
define INCLUDE=A:/INC/;C:/INC/;C:/USR/INC/
```

5.4 Configuring the Compiler

[configuring file extensions, global aliasing convention; configuring IPATH names: MS-DOS = DCL, <>-include, pragma Ipath; configuring options ansi, cram on 8086, ipath, mm = memory_model, tpages on 8086; configuring Check_stack, 186, 286; configuring Emit_line_table, Literals_in_code; config, autocfig]

MetaWare compilers are provided with a program named CONFIG that permits the programmer to change certain compiler defaults. That is possible because one can modify the compiler executable file without damaging its contents. Some of the defaults are neither options nor toggles and can therefore be changed only by configuring the compiler.

The configurable defaults are:

- The source program file extension when none is supplied. ".C" is the default for the distributed compiler, but users may change it to ".SEA", ".SEE", or ".HC", e.g.
- The extension for the object file output of the compiler. ".OBJ" is the default for the distributed compiler.
- The global aliasing convention; see Section *Externals.*
- The name of the Concurrent "Ipath" logical name. It is normally "IPATH" but can be changed to something else, e.g. "INCLUDE", to agree with conventions used by other compilers.
- The <>-include "ipath" value; see the prior subsection.

Options:

- ansi: Enforce ANSI standards?
- cram: Assume limited memory?
- ipath: The initial value of pragma Ipath.
- mm: The compiler memory model.
- tpages: The number of tree pages.

Toggles — see Section Compiler Toggles:

- Check_stack: Emit stack checking code?
- 186: Generate code for the Intel 80186?
- 286: Generate code for the Intel 80286?
- Emit_line_table: Emit line numbers in the object file for use by the post-mortem StackDump facility?
- Literals_in_code: Place literals in code?

The CONFIG program is self-documenting and users may run it (in the same directory as the compiler) for a complete explanation of how to configure the compiler:

```
config
```

Another supplied program AUTOCFIG sets the tpages option based on available memory and may set the cram option. This program is executed automatically during installation and it should be re-executed when memory resources are changed:

```
autocfig
```

This command must be given when in the directory containing the compiler.

6
Compiler Pragmas

High C compilers provide a myriad of "pragmas" (the term comes from Ada) that direct compiler operations. Furthermore, a "profile" file can be supplied to override the defaults for those pragmas and thereby customize the compiler to a local environment; see Section *Compiler Controls.*

Pragmas control the inclusion and listing of source text, the production of object code files, the generation of optional additional program and debugging information, and so on.

6.1 Syntax of Pragmas

Compiler pragmas take one of the following general forms:

```
pragma <Pragma_name>(<Pragma_parameters>);
```

or

```
pragma <Pragma_name>;
```

where <Pragma_parameters> is a list of constant expressions separated by commas. The number and nature of the expressions are dependent upon the particular <Pragma_name>. A pragma can appear anywhere a statement or declaration can appear; see the High C Language Reference Manual for a specification of the precise placement of pragmas.

6.2 Compiler Pragma Summaries

```
[On, Off, Pop; Alias, Calling_convention; Cgroup, Code, Data,
Dgroup; Global_aliasing_convcention, Literals, Static_segment;
Memory_model; Include, C_Include, R_Include, RC_Include, Ipath]
```

The following pragmas, listed alphabetically within logical groups, are available:

Pragma	*Its Purpose*
--->	*Toggles* — see Section *Compiler Toggles:*
On, Off, Pop	Turn On or Off, or reinstate a prior status of, various compiler switches or "toggles".
--->	*Externals* — see Section *Externals:*
Alias	Specify the external object module name for an internal identifier.
Calling_convention	Control the way linkage to subroutines is done. Allows calling other language such as Pascal, PL/M, or FORTRAN from C.
Cgroup	Specify the code group name for small-code memory models.
Code	Specify the segment into which generated code is to be placed. Useful for overlaying code.
Data	Specify the use of named common for data storage allocation. Can be used for overlaying data but its primary purpose is to provide another way of sharing data between compilation units.
Dgroup	Specify the data group name for small- and medium-data memory models.

`Global_aliasing_convention`
Specify automatic construction of external object module names for internal identifiers.

`Literals` Specify the name of the segment to contain literals.

`Static_segment` Specify the segment into which defined variables go. Useful for overlaying data.

`--->` *Models* — see Section *Memory Models:*

`Memory_model` Specify the 8086 memory model.

`--->` *Source File Inclusions* — see Subsection 6.3:

`Include` Include the source of another file in the compilation unit.

`C_include` Conditionally include the source of another file in the compilation unit.

`R_include` Include the source of another file in the compilation unit treating the path name as Relative to the directory containing the file containing the include.

`RC_include` Conditionally include the source of another file in the compilation unit treating the path name as Relative to the directory containing the file containing the include.

`Ipath` Specify a search path for the include pragmas.

6.3 Include Pragmas: Inclusion of Source Files

[pragmas Include, C_Include, R_Include, RC_Include, and Ipath; conditional source file inclusion; directory search for input files; include file search path]

The Include pragma is used to include source from other files while the compilation unit is being compiled. The pragma operates slightly differently from the standard C **#include** directive. There are four forms of the include pragma:

```
pragma    Include(<File_name>);
pragma  C_include(<File_name>);
pragma  R_Include(<File_name>);
pragma RC_include(<File_name>);
```

where "<File_name>" is a string *constant* denoting the name of a file. *Examples:*

```
pragma    Include("a_lot");
pragma R_Include("dclns");
pragma C_include("math.h");
```

The Include pragma directs the compiler to unconditionally include a file. The C_include pragma causes the file to be included only if it has not been included before — "conditionally included". The R_Include pragma has exactly the same effect as the standard C **#include** directive. The RC_include does likewise except that the file is conditionally included.

The difference between the R ("Relative") include pragmas, R_Include and RC_Include, and the pragmas Include and C_include is that for the latter two, the directory containing the file containing the pragmas is *not* searched for the included file, whereas in for relative includes, it is. See Section *Compiler Controls* for more information about the compiler searches for an include file.

An Include *pragma may not be followed by anything else on the line containing the pragma. After the* Include*-d file is processed, processing resumes on the line after the one containing the* Include *pragma. In effect the rest of the line is a comment!*

Ipath. The compiler is able to search up to three different lists of directories for an include or input file. These lists are specified by (a) pragma Ipath; (b) the "ipath" environment symbol; and (c) the configurable < >-include "ipath". The syntax of (a) is illustrated by:

On MS-DOS, or UNIX or Xenix if ":" replaces ";":

```
pragma Ipath("/usr/jones/;/usr/jones/inc/");
```

On VAX/VMS:

```
pragma Ipath("[DIR1]|DISK1:[DIR1]|DISK1:[DIR2]");
```

See Section *Compiler Controls* for more information about how all the different Ipaths are specified and in what order the compiler searches them.

Identity of file names. For the `C_include` and `RC_include` pragmas, file names, including path, are considered the same on some systems only if they are textually identical. Thus, the following two pragmas may cause two includes to occur:

On MS-DOS or UNIX or Xenix:

```
pragma C_include(       "string.h");
pragma C_include("../inc/string.h");
```

On VAX/VMS:

```
pragma C_include(      "STRINGS.H");
pragma C_include("HCLIB:STRINGS.H");
```

even though both includes may really refer to the same file.

This can happen if one of the Ipath directory lists contains "../inc/" on MS-DOS or "HCLIB" on VMS. The problem is that host operating systems do not always provide a method for determining whether two file names describe the same file; e.g. neither MS-DOS nor VMS has the ability, but UNIX does.

Also for the purposes of textual comparison, file name casing is significant only on operating systems that support such. On UNIX casing is significant; on MS-DOS and VMS it is not.

Methodology. The primary utility of conditional includes lies in supporting modularity. Assume file "trees.cf" is merely a collection of declarations defining the interface to a trees module. Suppose further that trees.cf makes reference to a typedef Symbol in another module defined in "symbols.cf". If a standard "**#include** "symbols.cf"" were placed within trees.cf, a duplicate declaration of Symbol would occur in any compilation unit that **#include**-d both trees.cf and symbols.cf. If, instead, a conditional include were used in both trees.cf and in any compilation unit including symbols.cf, at most one copy of symbols.cf would be included.

(We have assumed used-defined header files have the extension ".CF" rather than ".H" so as to eliminate any possible conflict with any ANSI-standard header file names.)

With conditional includes, each interface file F can conditionally include all other interface files F′ that are necessary for the definition of the resources in F. Therefore any user of F can simply include F and will automatically get other resources that are needed, without duplication.

7
Compiler Toggles

7.1 Toggle Pragmas

[On, Off, Pop, compiler switches or toggles]

One of the purposes of pragmas is to turn On and Off various compiler switches or "toggles". In such cases, the pragma syntax is simply

pragma <Pragma_name> (<Pragma_parameter>)

The <Pragma_name> is either On, Off, or Pop, and the single <Pragma_parameter> is the name of the toggle to be affected. All compiler toggles are described in Sections 7.2 and 7.3.

"On" turns the toggle on; "Off" turns it off; and "Pop" reinstates it to a prior value. Toggles operate in a stack-like fashion, where each On or Off is a "push" of on or off, and a Pop "pops" the stack. The stack for each toggle is at least 16 elements deep, but no diagnostic is given if the stack overflows or underflows. *Examples:*

```
pragma On (List);   /* Turns on the source listing. */
pragma Off(Check_stack); /* Turns off the run-time */
pragma Off(List);          /* stack overflow checks. */
pragma On (List);   /* Turns on the source listing. */
pragma Pop(List);   /* Back to off for the listing. */
pragma Pop(List);   /* Back to on  for the listing. */
```

Recall that toggles can also be initialized on the command line, with on and off. See Section *Compiler Controls*.

Below we present the default values, names, and meanings of the compiler toggles. The presentation is divided into two categories: system-independent and system-dependent toggles. In each category the toggles are alphabetized.

The default value of some toggles can be changed by configuring the compiler; see Subsection *Configuring the Compiler* of Section *Compiler Controls.* Configurable toggles are indicated by "(configurable)" after the specification of the default value; the stated default is that provided in the compiler as configured for distribution.

7.2 System-Independent Toggles

Asm -- Default: Off

[assembly listing]

When On, causes a (pseudo-) assembly listing to be generated, annotated with source code as assembly comments. To get the listing for just the body of a given non-level-one routine, be careful to place the "`pragma On(Asm);`" just after the body's { and place the corresponding "`pragma Pop(Asm);`" just before the corresponding }.

Callee_pops_when_possible -- Default: Off

To allow varying numbers of parameters to a function F, the standard C calling convention has F's callers pop F's arguments from the stack after F returns. That is less efficient than having F pop the parameters, since the code for the pop is replicated at each call rather than occurring exactly once at the epilogue of F.

When this toggle is On, the more efficient scheme is used when possible. In particular, each non-exported function that is never passed as a parameter pops its parameters, and its callers do not.

This is a safe optimization only if the number and sizes of parameters passed in each call to such a function matches the number and sizes of parameters in the declaration of F. Otherwise, unpredictable and sometimes disastrous results may occur. By contrast, if the toggle is Off, incorrect parameter passing will generally only cause F to access garbage in

its parameters, but the stack will be restored properly after the call.

This toggle is not applicable on target machines with automatic parameter popping such as the VAX, nor machines without an explicit stack, such as the IBM 370, but applies to such machines as the Intel 8086, MC68000, and NS32000.

Check_stack -- Default: Off (configurable)

[stack overflow]

When On, causes code to be emitted to check for stack overflow. Note: this toggle does not apply to (cross) compilers targetting to machines with automatic stack expansion, such as the VAX, MC68010, and NS32000.

Int_function_warnings -- Default: On

When Off, suppresses warning messages normally generated when (a) a function returning `int` has no "`return (expression);`" statement within it; and (b) a function returning `int` contains a "`return;`" within it.

This is to remove frequent warnings for old C source that did not use the reserved word `void` to indicate a function returning no result, since such functions return `int`, by default.

List -- Default: Off

[compiler or source listing]

When On, causes the compiler to produce a listing. It is typically given when starting the compilation but may appear in the source file to turn the listing On or Off around a particular section of source.

Make_externs_global -- Default: Off

When On, any local declaration of an object with storage class `extern` is made global if there does not already exist a global declaration of the object. Early C compilers sloppily promoted an `extern` declaration within a function to the global scope. This toggle supports programs depending upon that "feature".

Optimize_for_space -- Default: Off

[code optimization]

When On, causes the generation of more space-efficient but potentially less time-efficient code. An example of this is using a multiply rather than a sequence of adds and shifts to compute array subscripts: multiply is much more expensive than shifts on an 8086/88, for example.

Parm_warnings -- Default: On

In C it is permissible to pass arbitrary arguments to a non-prototype (old-style) function F, without any type checking to ensure that the passed arguments match in type with the declared formal parameters of F. High C compilers produce a warning whenever just such an inconsistency is detected. Frequently this inconsistency is a source of disastrous or difficult-to-find bugs:

```
double square(x) double x; {return x*x;}
...
printf("%lf\n",square(3));
```

The call to square passes the integer 3, not the double 3.0, and the compiler complains. The C language definition *prohibits* the compiler from casting 3 to a `double` before passing it.

To eliminate the compiler warnings, turn Off the toggle `Parm_warnings`. We recommend, however, that the program text be repaired to eliminate the offending function calls rather than eliminating the potentially useful feedback from the compiler.

Pointers_compatible -- Default: Off

[pointer compatibility]

When On, allows pointers of any type to be compatible with each other. Although this is violation of the ANSI standard and High C specifications, many old C programs improperly assign pointers of different types to each other. This toggle allows such programs to be compiled without modification.

Pointers_compatible_with_ints -- Default: Off

When On, allows pointers of any type to be compatible with `int`s. Although this is violation of the ANSI standard and High C specifications, many old C programs improperly assign pointers and `int`s back and forth. This toggle allows such programs to be compiled without modification.

ANSI and High C disallow this dangerous practice because pointers are not necessarily the same size as `int`s. The programmer should ensure that intermixed pointer and `int` types have the same size; otherwise a pointer stored in an `int` may not be retrieved as expected later.

Public_var_warnings -- Default: On

When Off, suppresses the warning messages "Variable used before set.", "Variable set before used.", and "Variable not used." for all variables exported, i.e. those non-automatic variables not declared `static` or `extern`.

Such warning messages only occur for such variables that are not within an `#include`-d. If one adheres to the discipline that all imported variables are defined in included files, the message will not occur.

Quiet -- Default: Off

[compilation phase announcements]

When On, causes each compilation phase to be announced in turn as the compilation progresses.

Summarize -- Default: Off

[compilation statistics and summary]

When On, causes the production of summaries of compilation activities. The summaries are produced at various stages of compilation.

Warn -- Default: On

When On, causes warning messages to be suppressed.

7.3 System-Dependent Toggles

186 -- Default: Off (configurable)

[Intel 80186/80286 processors]

When On, causes the generation of 80186 instructions. The 8086/88 code generator is capable of generating code for any of the 8086/88/186/188/80286 family. The 80186 has a few more instructions than the 8086/88, which can increase the efficiency of programs; the 80286 has those and even more instructions, but the latter are for operating system applications.

The 80186 instructions include: Push immediate, multiply immediate, shift left/right immediate, and the procedure "leave" instruction. The procedure "enter" instruction is not used unless it will save space and toggle `Optimize_for_space` is On because the code the compiler normally generates to enter a procedure is faster than the "enter" instruction.

286 -- Default: On (configurable)

[Intel 80186/80286 processors]

When On, causes the generation of 80186 and 80286 instructions. The only difference between this and the 186 toggle is that the FWAIT instruction that synchronizes the 8086/88 and the 8087 is omitted, since the 80286 and 80287 automatically synchronize.

Emit_line_records -- Default: Off

[emitting debugging information]

When On, causes the Intel OMF line-number records that associate line numbers with addresses within code segments to be emitted. No debuggers under Concurrent use line records.

Emit_line_table -- Default: Off (configurable)

[emitting debugging information; call-chain stack dump]

When On, causes information to be emitted that allows the determination of the line number of a given code address A, given the entry point of the routine containing A. This is used by the run-time call-chain stack dump mechanism to produce line numbers for the dump rather than code addresses.

Emit_line_table causes consumption of code space in the code segment containing the routines for which the line number table is emitted. Usually the overhead is small: a little less than one byte per source line. No speed loss occurs.

Emit_names -- Default: Off

[call-chain stack dump; run-time error]

When On, causes the prologue of each routine R to be preceded with R's name so the StackDump routine can find the name should a run-time error occur while R is active. The name used is the "internal" name, the one relevant to source-code programming, rather than any "external" name specified via pragma Alias for the sake of a linker. Also, each code segment begins with its name. Thus, in large-code models, displaying the current CS at offset zero yields the name of the current code segment.

Floating_point -- Default: On or Off per the host

[native floating-point instructions; Intel 8087 support]

When On, causes the compiler to emit "native" 8087 floating-point code rather than library calls. It is set On if the machine on which the compiler is running has an 8087, and Off otherwise. This is dynamically determined by the compiler at start-up.

Care must be taken with the exercise of this option. If code is being prepared that will be run on many machines, some of which have no numeric processor chip, emitting native code can be harmful even if users are warned that "not all software features will work" with the omission of the chip. For example, on the 8086/88, most floating-point instructions are preceded by a WAIT opcode, which may hang the CPU if no 8087 is present. Thus software containing native 8087 code may hang on machines without an 8087.

The MetaWare co-processor Run-Time Libraries include a call interface to all 8087 functions. Code compiled with Floating_point Off goes through the Library to access the 8087. The co-processor Libraries check for the presence of the 8087 before any floating-point operation is attempted and aborts if the 8087 is not present.

The MetaWare Run-Time Libraries with the floating-point emulators does not need the 8087 chip, although it will make use of one if present. Code linked with this library will run on any machine. The emulator Libraries closely simulates the 8087. See Section *Floating-Point Support* for more information, and for a way to "turn off" the 8087 so that the library does not use it.

The 8086-resident compiler itself is linked with the emulator library; therefore, it does not need an 8087 to compile floating-point expressions.

Literals_in_code -- Default: Off (configurable)

[ROM-able code, literals in data vs. code space, pragma Literals; 8086 extended memory; post-mortem call trace call-stack dump]

Lengthy literals in a program are normally be placed in the program's data space. Such literals include string and floating-point constants. With the Literals_in_code toggle On, they are placed instead in the code space (but see the first and second notes below). This can be beneficial where dynamic code loading is performed by the operating system. In such circumstances code is most often read-only so literals can be swapped out of memory without the need to write them to the paging medium used by the operating system.

Literals_in_code should be turned On when "ROM-able" code is desired and the memory model is not small-data. ROM-able code is destined to be burned into a ROM. In such applications typically the RAM contents are undefined when the program begins execution. Therefore all the literals must exist in the code space and are therefore also burned into ROM. See

Subsection *Storage Classes* in Section *Storage Mapping* for a different method for small-data.

NOTE: This toggle is ignored if the program is using a small-data model. By definition all data and literals must be addressable from the DS segment register in a small-data model. See Section *Storage Mapping.*

NOTE: C string literals cannot normally be placed in code, since they are writeable data items. For example, the 80286 processor forbids writing to code when running in protected mode. Hence, `Literals_in_code` normally has no effect for string literals. However, if the toggle `Read_only_strings` is On, C string literals are assumed *not* writeable and turning On `Literals_in_code` *will* cause them to be placed in code. See the description for toggle `Read_only_strings` for more.

We recommend the combination of `Read_only_strings` and `Literals_in_code` for Large memory model modules that have no local static variables declared. Under these circumstances it is not necessary for the compiler to dedicate the DS register to point to the module's static segment, since it has none. Therefore DS can "float", and much better pointer-dereference code can be generated when both the DS and ES registers are available.

See also pragma `Literals` for a way to specify the data segment into which all literals not placed in code are put.

NOTE: Alternatively, on some machines it is necessary to place the literals in the data space. For example, some 8086 systems make use of a signal from the chip that indicates whether the memory reference is from the CS register versus other segment registers. System designers can take advantage of this and cause the CS memory references to go to a separate one-megabyte memory bank, thus effectively obtaining a two-megabyte memory for the 8086. However, this requires that code running on such a system *not* try to access data in code memory with the DS or ES register, or data in data memory with the CS register. If literals are placed in code memory, the compiler will sometimes reference them with the

7x
Compiler Toggles

7.3 System-Dependent Toggles: Addenda

... (After toggle Literals_in_code:)

NEC -- Default: Off

When On, the compiler generates instructions recognized by the NEC 8086-compatible V20 and V30 processors — these include the TEST1, CLR1, and SET1 instructions. Furthermore, instructions recognized by the Intel 80186 processor are generated, since the NEC processors recognize them also.

For example, for the Pascal program fragment

```
var I:  Integer;
Primes: 0..8190;
begin
if I in Primes then ...
```

the code

```
mov   si,@sieve+24
mov   cx,si
shr   si,3
and   cl,7
mov   al,1
shl   al,cl
test  @sieve[si],al
jz    ...
```

is generated with the NEC toggle Off and 186 On, and

```
mov   ix,@sieve+24
mov   cw,ix
shr   ix,3
test1 byte ptr @sieve[ix],cl
bz    ...
```

with NEC and NECasm (see next) On — 6 bytes shorter, for a 27% savings in space.

NECasm -- Default: Off

When On, the compiler generates NEC instruction mnemonics and register names rather than the standard Intel mnemonics and names. For example, the Intel instruction "`adc ax,-10[bp+si]`" is known in NEC conventions as "`addc aw,-10[bp+ix]`". The setting of this toggle in no way affects the object module generated by the compiler. Turning it On makes sense only if an assembly listing is requested.

DS or ES register, so the Literals_in_code pragma should be Off. Note also that on such a two-bank memory system, the post-mortem call-chain stack dump facility will not be able to print the names of the modules and routines in the dynamic call chain at the time of any error, because the textual names for those routines are in the code space and there is no way to reference them.

Read_only_strings -- Default: Off

[ROM-able code, literals in data vs. code space]

C string literals are not true literals, since they are writeable data items. This means that they cannot be placed in code space, since many processors — the 80286 is an example — can protect against writing to code. Furthermore, two identical string literals must be duplicated in a program's object code, since one might be modified and the other not.

With the toggle Read_only_strings On, string literals are considered true literals. Identical string literals are written to object code only once, and the Literals_in_code toggle (see above) takes effect for string literals, causing them to be placed in code. With data space at a premium on an 8086, this is a useful way to shift potentially large amounts of program from data to code. Furthermore, with string literals in code, an overlaying linker will automatically overlay string literals with the code of an overlayed module.

Segmented_pointer_operations -- Default: On

On the 8086/88, when comparing two pointers with the relational operators >, <, >=, and <=, the comparison generally does not make sense if the segment portions of the compared pointers differ. This is because MetaWare's 8086 compilers restrict data objects to the architecture-imposed limitation of 64KB. Therefore, by default, only the 16-bit offset portion of such 32-bit pointers is compared for >, <, >=, and

<= (for equality and inequality, a full 32-bit comparison is used).

For programmers that are managing pointers to individual objects that span more than 64KB, this toggle can be turned Off to force 32-bit comparisons.

In addition to comparisons, differences of pointers are done with respect to offset only if the toggle is On. For example,

```
diff = p1 - p2
```

does a 16-bit subtraction of the offset portions of "p1" and "p2", dividing the result by the pointed-to size to determine "diff". The subtraction becomes 32-bit if the toggle is turned Off.

The addition (subtraction) of an integer value to (from) a pointer is *always* done in 16-bit arithmetic: MetaWare's 8086 compilers do not support 32-bit operations here. For example,

```
q = p+i   /* p's offset, plus i, into q's offset. */
q = p-i   /* p's offset, less i, into q's offset. */
```

The Segmented_pointer_operations toggle has no effect here. If you want 32-bit pointer arithmetic you must provide functions to increment and decrement pointers.

8

Floating-Point Support

8.1 The 8087 or 80287 Co-Processor

The 8087 is a fast floating-point processor that runs as a "slave" to the 8086 CPU. The 8087 has its own set of instructions that it "intercepts" from the host CPU.

On 80286 processors, the equivalent of the 8087 is the 80287 chip. Any reference to the 8087 in this section applies equally to the 80287 unless an exception is made.

Since not all 8086 machines come with an 8087 co-processor, the compiler provides two methods for evaluating floating-point expressions.

8.2 Floating-Point Evaluation and Run-Time Libraries

[toggles Floating_point, 286; 8087 emulation libraries]

The compiler generates code to evaluate floating-point expressions according to the setting of a toggle named `Floating_point`.

If `Floating_point` is Off, calls are emitted to library functions that closely emulate the 8087 instruction set.

If `Floating_point` is On, native 8087 instructions are generated. Additionally, if the 286 toggle is On, 80287 instructions are emitted. (The 286 toggle is never automatically activated by the compiler, e.g. by detecting that it is running on a 286, but must be configured On or set by the user via pragma On in the program or profile.) The only difference between 8087 and 80287 instructions is that the 80287 instructions need not be preceded by the 8086 `FWAIT` instruction that is used by the 8086 and 8087 to synchronize; the 80287 synchronizes with the 80286 automatically.

The setting of Floating_point during compilation determines which Run-Time Libraries can be linked with the program, as described next.

For each "memory model" two sets of functions are provided for evaluating floating-point expressions: the 8087-natives and the emulators. The 8087-native functions simply invoke the appropriate 8087 instructions and abort with a diagnostic message if no 8087 is present. The emulation functions can run on a machine with or without an 8087. If the 8087 is present, it is used; otherwise, it is emulated. The emulators are, of course, bigger than the 8087-natives. But as long as memory can be afforded, the emulation functions are an excellent choice in obtaining speed when the 8087 is present and emulation when it is not.

Code compiled with Floating_point On can *only* be linked with 8087-native functions. This achieves the fastest possible floating-point arithmetic, and also the smallest program size. Code compiled with Floating_point Off can be linked with either Library. The reason that such code can be linked with the native functions is that the natives support all the entry points to the emulator, but merely invoke the 8087 rather than emulating. However, even when linked with the 8087-natives, such code runs slower than that compiled with Floating_point On, due to the overhead of the calls.

In summary, for maximum speed and minimum space, compile with Floating_point On and link with the native functions. For maximum flexibility, compile with Floating_point Off and link with the emulators.

A word of caution: if code is being developed that is to run on a multitude of 8086 machines, Floating_point should be explicitly turned Off in every module that contains real-type expressions — or the toggle can be explicitly turned Off in the profile; see Section *Compiler Controls.* If code containing 8087 instructions is executed on a machine without an 8087, the machine may hang. This is only a hazard with user func-

tions: the Run-Time Libraries ensure that the 8087 is present before using it, i.e. they were compiled with the toggle Off.

See Section *Linking a Compiled Program,* Subsection *Run-Time Libraries; Link Errors,* for the library names and other details. See Section *Memory Models* for descriptions of the models.

8.3 Detecting the Presence of an 8087

[toggle Floating_point; NO87 environment variable; MS-DOS set-command]

At start-up time for the 8086-host compiler, a run-time intialization routine performs a test to determine if the host machine contains an 8087 co-processor. If the 8087 is present, the `Floating_point` toggle is initialized to On, otherwise Off, so code is generated accordingly unless specifically overridden in the profile, e.g.

Cross compilers from all other machines have the toggle Off by default. That default may be configurable: use the `config` command to find out for the machine of interest; see Section *Compiler Controls,* on configuring the compiler.

Compiled programs use the same run-time intialization routine at start-up time. If the 8087 is *not* present, the 8087-native library functions abort with a diagnostic message the first time a floating-point library function is called. The emulation functions use the 8087 if it is present and emulate it otherwise.

NO87. Under Concurrent, users can disable the use of the 8087 by using the logical name "NO87". When NO87 is defined (as anything), the run-time support acts as though there is no 8087: the 8087-native library functions abort upon attempted use, and the emulation functions emulate only. On programs linked with the emulators, this feature can be used to determine how much slower the program runs without using the 8087 (if the computer has one). On programs linked with the 8087-natives, the feature can be used to determine whether library functions requiring the 8087 are ever called.

At run-time initialization the string value that NO87 is defined as is printed on the standard output, unless the value is all blanks. For example, the Concurrent command

```
define NO87=Use of co-processor disabled.
```

causes the message "Use of co-processor disabled." to appear at the beginning of execution of any program, including the compiler. On the other hand,

```
define NO87=        (One or more blanks.)
```

causes no message to be displayed, but the 8087 to be disabled. Finally,

```
define NO87= (Immediate ENTER; no blanks)
```

"undefines" the NO87 variable so that the 8087 is made availaable for use. One can determine which logical names are defined simply by typing "`define`".

9
Memory Models

9.1 The 8086 Memory Architecture

[code, data, run-time stack and extra segments; CS, DS, SS, ES; dynamic versus static registers; data areas in one segment; memory models; Small, Compact, Medium, Big, Large]

The 8086 family of processors reference memory through a set of four "segment registers": a code segment (CS), a data segment (DS), a stack segment (SS), and an extra segment (ES). Up to 64K bytes can be addressed with a single segment register.

The compiler supports five so-called "memory models" on the 8086 family that differ from each other in whether these registers stay fixed or vary during the execution of the program, and if fixed, whether the data areas reside in a single 64K-byte segment. If a given register stays fixed, the related addressing capability is of course limited to 64K bytes.

We describe the five memory models below first according to how they treat code and then how they treat data. In the next two subsections we define five adjectives to capture the essence: small-code versus large-code, then small-, medium-, and large-data. Then we detail the five models: Small, Compact, Medium, Big, and Large, each of which is described by a code-data pair of these adjectives. (The sixth pair seems not to result in a sufficiently useful model to bother implementing. Perhaps the present reader will provide the impetus for us to support it by finding an important application that just fits this sixth niche.)

9.2 Small-Code versus Large-Code Models

[static versus dynamic CS; short versus long calls]

The code of a program can be modeled in two ways. A *small-code* model requires all code to reside in a single segment referenced by the CS register. The content of the CS register does not change during the program's execution. Thus the code may not exceed 64K bytes in length.

In a *large-code* model program code may be scattered across several code segments. However the code of any given routine (function) must reside in a single code segment. With this scheme the CS register is dynamically modified to reflect the currently active code segment.

The chief advantage of a large-code model is that it permits programs to execute that have more than 64K bytes of code. On the other hand a small-code model is more efficient in code space and execution time than a corresponding large-code model.

Small-code models employ a short `call` instruction to perform routine (function) calls. The instruction is three bytes long and it uses a self-relative displacement. In contrast large-code models use the long `call` instruction which is five bytes long and is not self-relative. Self-relative displacements require no relocation fix-ups so the overhead of loading a small-code program is substantially smaller than that of a large-code program.

9.3 Small- versus Medium- versus Large- Data Models

[pointer and address sizes; memory reference costs]

The compiler also supports three schemes for referencing data.

A *small-data* model incorporates a single data segment (≤64K) in which the stack, heap, and static storage are all located. The segment registers DS, SS, and ES are all permanently set to reference that one data segment. With this scheme, pointers and addresses are only two bytes long, consisting of a 16-bit displacement within the data segment, but they cannot refer to code or indeed any memory outside the data segment.

A *large-data* model employs multiple data segments: one for the stack, one or more for the static data, and one or more for the heap. With this scheme, pointers and addresses are four bytes long, consisting of a 16-bit displacement followed by a 16-bit "paragraph address" of a segment (a paragraph is 16 bytes). Thus all of memory can be referenced.

Memory references in a large-data model are significantly more expensive than in a small-data model. To "dereference" a pointer in a large-data model, the DS or ES register must be loaded with the paragraph address of the segment being referenced. To access static data, e.g. variables in a Common block, the DS or ES register must also be loaded with the paragraph address; but since the address is a constant, two instructions are required to load the segment register because immediate values can not be directly loaded into a segment register.

A *medium-data* model is like a large-data model in that pointers are four bytes long. It differs, however, in that all static data reside in a single data segment permanently referenced by the DS register. This scheme permits static data to be referenced as cheaply as with the small-data scheme. Thus, only explicit pointer dereferences in C, and references through

pointer and address variables and parameters passed by reference in Pascal, require the expense of 32-bit dereferences.

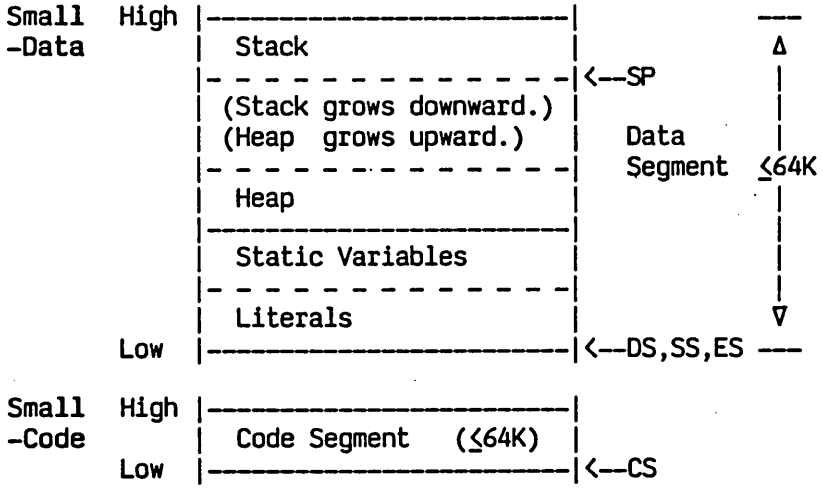
9.4 Small Model: Small-Code, Small-Data

The Small model can be described as small-code, small-data. All logical code segments are grouped into a single, physically contiguous code segment by the linker. Likewise all logical data segments are grouped into a single data segment. All segment registers stay fixed throughout program execution. The CS references the code segment and the DS, SS, and ES, all reference the data segment, i.e. they each have the same content; the stack pointer (SP) dynamically references the top of the run-time stack at the opposite end of the data segment:

By default the linker specifies that the data segment should be as large as possible. This can be changed; see the linker documentation on the command file option parameters ABSOLUTE, ADDITIONAL, and MAXIMUM. Segment size specifications as set by linker defaults or by these option parameters are written to a simple header at the beginning of the load module and can be "poked" after the fact if desired, although there is no standard utility to modify the header.

The stack is carved out of the top of the data segment. Reducing the data segment size therefore also reduces the possible maximum size of the stack.

Diagram of the Small Memory Model



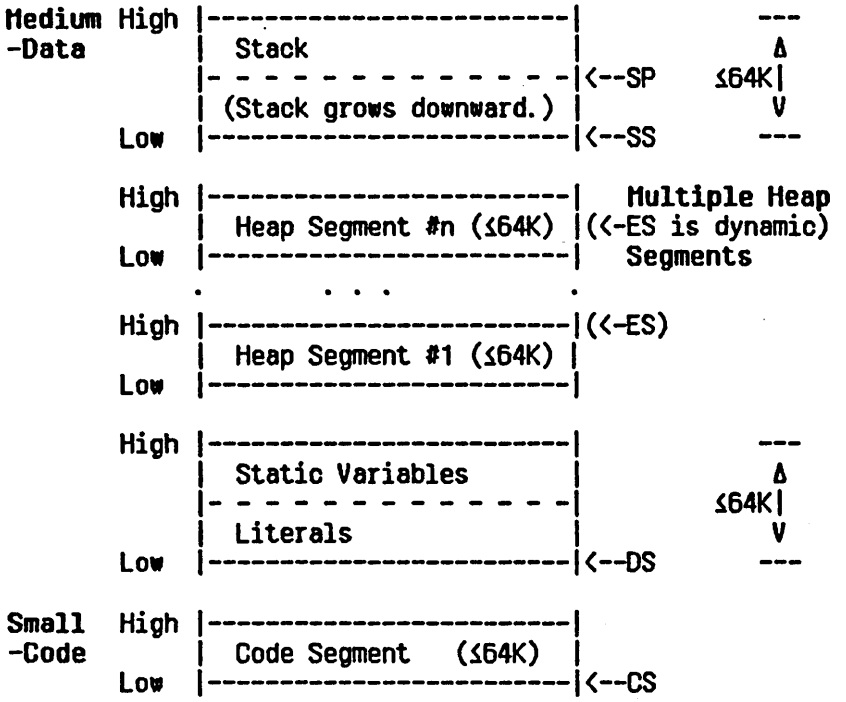
9.5 Compact Model: Small-Code, Medium-Data

Compact can be described as small-code, medium-data. Unlike the previous model (and the Medium model, Subsection 9.6), this model has four-byte pointers and addresses.

The static variables and literals together form a single data segment. The stack segment's address and size is determined by the loader. The heap is dynamically allocated from Concurrent and may consume all of memory (up to 16MB). It does not necessarily occupy contiguous memory as implied by the picture below.

The stack size is fixed at link time, and is by default 2,000 bytes, as defined in the run-time initializer (the source of which is supplied). This can be changed; see the linker documentation on the command file option parameters ABSOLUTE, ADDITIONAL, and MAXIMUM. Segment size specifications as set by linker defaults or by these option parameters are written to a simple header at the beginning of the load module and can be "poked" after the fact if desired, although there is no standard utility to modify the header.

Diagram of the Compact Memory Model

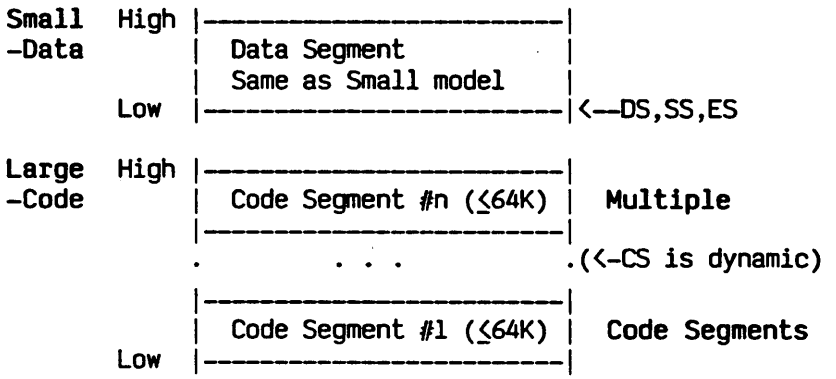


9.6 Medium Model: Large-Code, Small-Data

The Medium model can be described as large-code, small-data. The code resides in multiple code segments. The CS register changes dynamically to reference the active code segment.

Perhaps a good way to remember the name Medium is that the average of Large(-code) and Small(-data) is Medium. (There seems to be little demand for the sixth combination of small-code, large-data.)

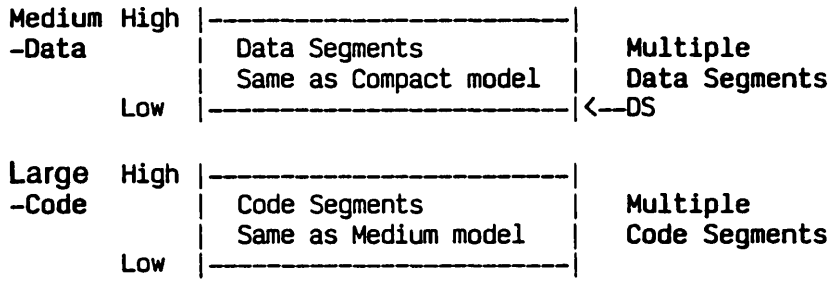
Diagram of the Medium Memory Model



9.7 Big Model: Large-Code, Medium-Data

The Big model can be described as large-code, medium-data. This scheme is like the Compact model except that the code occupies multiple segments.

Diagram of the Big Memory Model



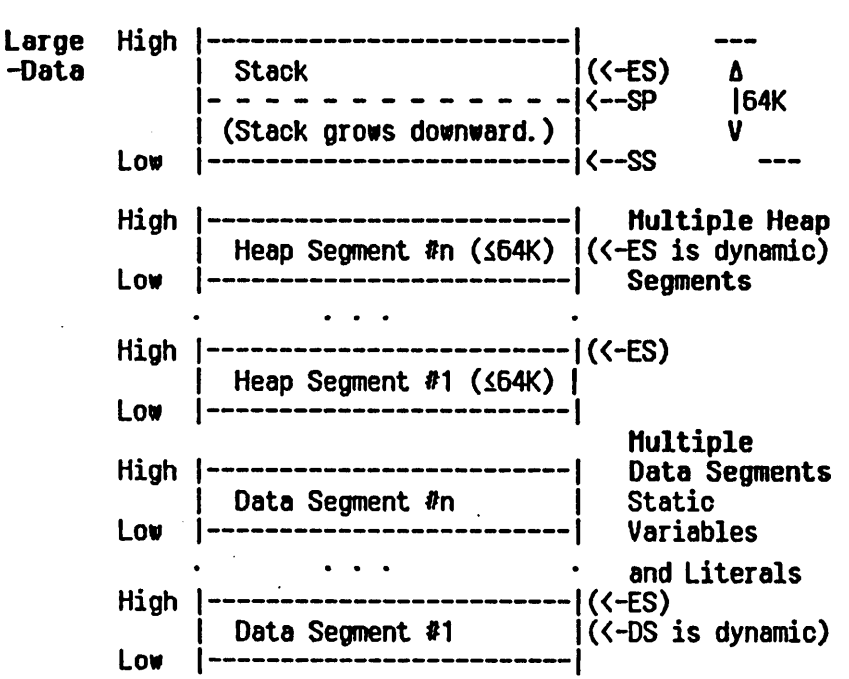
9.8 Large Model: Large-Code, Large-Data

The Large model can be described as large-code, large-data. It is the most general of all the models in that it permits multiple data segments for mapping static variables and multiple code segments.

The SS register is the only segment register that stays fixed in the Large model. The DS and ES change dynamically to reference various data segments, but the DS is fixed per module, referencing a default data segment containing static variables for the module (compilation unit).

Section *Run-Time Organization* describes how the DS register is maintained for large-data.

Diagram of the Large Memory Model



9.9 Pragma Memory_model; Default Model: Small

[Small, Compact, Medium, Big, Large]

Pragma Memory_model is used to specify a model by its name:

pragma Memory_model(<Model_name>)

where <Model_name> is one of the following names: Small, Compact, Medium, Big, or Large. (Casing is not significant.) The pragma may appear anywhere syntactically allowed in the source file. If more than one is specified, only the last has effect; if the mm option (/MEMORY_MODEL qualifier on VMS) is used on the command line that invokes the compiler, it overrides any Memory_model pragmas in the program or profile.

The default model is Small. If some other model is to be used, the desired model must be specified by a pragma for every compilation unit or "module". For example, the pragma could be placed in the profile; see Section *Compiler Controls.*

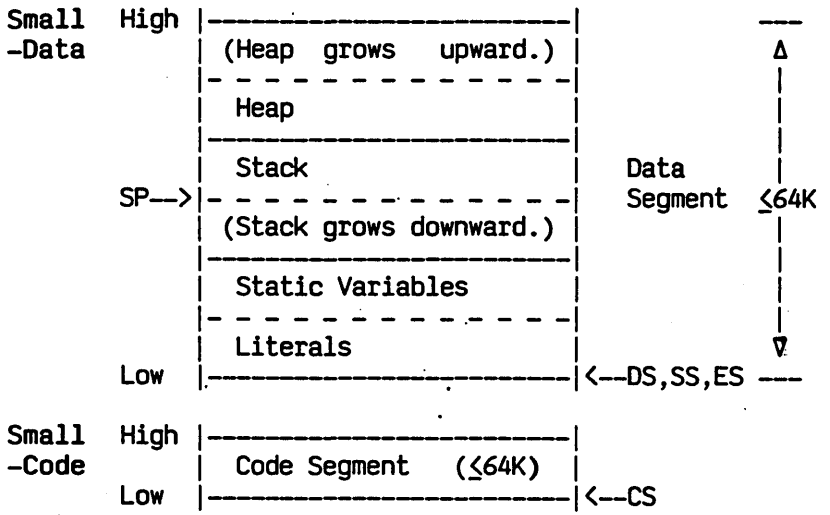
Linking. There are two run-time libraries for each memory model: one that contains an 8087 floating-point emulator and one that does not; see Section *Floating-Point Support.* Thus there are ten libraries in all. Their names and how to link to the appropriate library are discussed in Section *Linking a Compiled Program,* Subsection *Run-Time Libraries; Link Errors.*

9.10 Using a Fixed-Size Stack

[Small, Compact, Medium, Big, Large; toggle Check_stack]

For Concurrent, the stack size is fixed for all but small-data memory models. MetaWare compilers provide the option of arranging memory in those models so that the stack is of a fixed size, determined at the time the program is linked. The stack resides immediately above the program's static data, and the heap resides above the stack. This different arrangement is illustrated below for the Small memory model.

Diagram of the Small Memory Model with Fixed-Size Stack



This alternative memory arrangement is available for small-data memory models (Small and Medium), but only by changing the run-time initializer INIT.ASM provided in the compiler distribution. A variable Stack_size is provided in INIT.ASM that, if defined, causes the alternative memory arrangement to be used. For example, one might write "Stack_size = 2000" to obtain a 2000-byte stack. Re-assembling the initializer requires an assembler compatible with Concurrent's RASM.

10
Storage Mapping

10.1 Data Types in Storage

[data type alignments and sizes, struct padding, bit fields]

The table below summarizes the sizes and alignments of various C data types. "Alignment" means that when an object of the type is declared, its storage address modulo its alignment is zero. *This does not apply to declarations within structures:* fields are never aligned so that structures can be tightly packed.

The `char` and `int` types have the same size regardless of whether they are signed; therefore the table does not mention the sign.

Data Type	Size (bytes)	Alignment
`char`	1	1 (bytes)
`short int`	2	2
`int`	2	2
`long int`	4	2
`float`	4	2
`double`	8	2
`long double`	10	2
pointer to T where	2 (Small-data)	2
T is not a	4 (Medium-data)	2
function type	4 (Large-data)	2
pointer to T where	2 (Small-code)	2
T is a	4 (Large-code)	2
function type		
Extended-function	4 (Small-code)	2
type	6 (Large-code)	2
`struct`	Sum of field sizes	§
`union`	Biggest field	§
T[E]	sizeof(T)*E	Same as T

where § means 1 if the size is 1, otherwise 2.

Bit-fields. Only bit fields of type `unsigned int` and `unsigned long int` are supported. Any other type draws a diagnostic and is changed to either `unsigned int` or `unsigned long int`, according to whether the field size is 16 bits or less or not.

A bit field may not exceed 32 bits and is packed in each consecutive byte from right to left. A bit field of size 16 bits or less may not cross more than one byte boundary, i.e. it must be contained within two consecutive bytes. A bit field of size 17 bits or more may not cross more than three byte boundaries, i.e. it must be contained within four consecutive bytes.

A bit field of length zero causes alignment to occur at the next byte boundary.

A bit field that is byte-aligned and one or two bytes long is treated as if it were type `unsigned char` or `unsigned int`, respectively. These are more efficiently accessed than the corresponding bit fields.

Structure alignment and padding. No padding is supplied within structures, with two exceptions: (1) when a bit field does not fit starting at the current bit position (because it would cross more than one byte boundary, for fields of size 16 bits or less, or more than three byte boundaries, for fields of 17 bits or more), the bit field is aligned to the next byte; and (2) when a non-bit-field member follows a bit-field member, the former starts on the next byte boundary.

The choice of no padding was made so that structures could be packed tightly. The programmer who wants greater efficiency from, say `int`s, should align them on an even-byte boundary when running programs on the 8086 or 80286. Such alignment makes no difference on the 8088, which has an eight-bit bus.

For example, the structure definition

```
struct {unsigned x:3,y:9,z:13,w:1; char c; int i;}
```

is mapped to memory as follows:

```
7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
<---Byte 0 --->  <---Byte 1 --->  <---Byte 2 --->  <---Byte 3 --->
y y y y y x x x        y y y y    z z z z z z z z      w z z z z z

7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
<---Byte 4 --->  <---Byte 5 --->  <---Byte 6 --->
c c c c c c c c  i i i i i i i i  i i i i i i i i
```

Note that i is not aligned on an even byte boundary, and that four bits in the second byte are wasted due to the size of z. The structure size is 7 bytes, and an n-element array of such structures occupies exactly 7*n bytes.

Enumeration types. A type specifier of the form "enum {...}" denotes signed char, signed short int, signed int, or signed long int, depending upon the size of the explicit or implicit signed constants within the specifier. Thus the storage size for an enum object is that of the denoted type. For example:

```
enum {Red, Green, Blue} x;  /* type: signed char     */
enum {Red, Green, Blue = -130} x; /* signed int       */
enum {Red, Green=65_000, Blue} x; /* signed long int */
```

10.2 Storage Classes

[_MWLITERALS, toggle Literals_in_code, ROMable code]

Associated with each module (compilation unit) is a private data segment where local static variables are mapped. The name of this data segment is ? followed by the file-name "stem", i.e. without any ".c". For example, for file "z.c" it would be "?z". In a large-data model literals are also mapped there (unless the Literals_in_code toggle is turned On; see Section *Compiler Toggles*); in a small-data model literals are placed in a public segment called _MWLITERALS so they can be manipulated by linker commands.

Note: in small- and medium-data models such data segments are "grouped" into a single physical data segment by the linker and referenced by the DS segment register by MS-DOS. Thus in such a case the sum total of all data segments may not exceed 64K bytes.

10.3 The Stack Frame

[BP, addressing locals and parameters]

By default, variables declared within a function are mapped at consecutive negative offsets within the function's "stack frame". A *stack frame* is an area of storage allocated on the stack when a function is invoked. The BP register references the stack frame of the function currently executing.

A local variable is addressed by some negative displacement off of the BP register. Parameters are addressed with a positive displacement off of the BP.

See Section *Run-Time Organization* for a description of the stack frame for various code models.

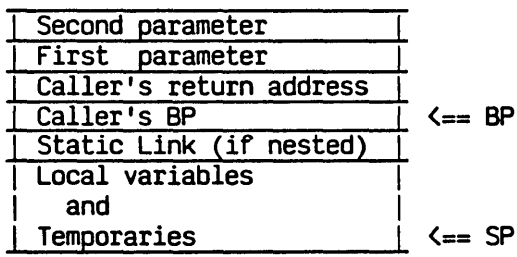
11
Run-Time Organization

11.1 Stack Frame Layout

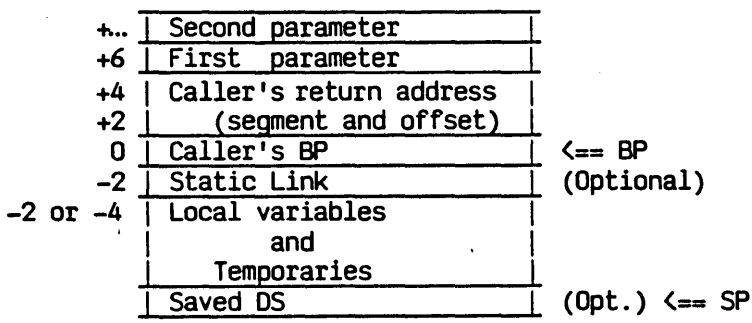
[local variable addressing, parameter alignment; small-code, large-code; static link, up-level addressing; stack growth; saving registers; DS altered by some functions in large-data models]

The stack frame of a C function has one of the following formats under the assumption of reverse parameter layout (the default).

Stack Frame in a Small-Code Model



Stack Frame in a Large-Code Model



The BP register points to the base of the local stack frame and is used to address the function's parameters and its local variables. The value in the word pointed to by BP is the value of the caller's BP and is used for restoring BP upon exit. Just above this (in increasing memory address) is the return address where execution resumes after the function is finished. Above the return address are the parameters to the function.

Parameters. Each parameter to a function takes an even number of bytes on the stack. For example a character value that is normally stored as a single byte is pushed as two bytes on the stack.

By default, parameters are pushed in reverse order so the first parameter in the formal parameter list has the least displacement off of BP. However see Subsection 11.3 below.

Saving registers. A function that is nested within another function is passed a "static link" in the BX register. The *static link* is the displacement, relative to the stack segment, of the stack frame of the immediately enclosing function. The link is used to reference "up-level" variables, i.e. locals of containing procedures. It is saved at -2 off the BP. Note that nested functions are an extension over standard C, so that the static link will never be encountered in standard-C programming.

If no static link must be saved, local variables and temporaries begin at offset -2 in the stack frame. Otherwise they begin at offset -4.

DS altered by some functions in large-data models. In a large-data model, any function that is exported or passed as a parameter and that is contained within a module having a non-empty local data segment saves the DS register on the stack after the local variables and temporaries have been allocated. In addition, such a function then sets up the DS register to reference the local data segment for the module. When the function returns, the previous content of the DS register is restored. (Recall that in small- and medium-data models the DS register is fixed throughout program execution.)

Likewise, any function R containing a label L that is the target of a `goto` in a contained function C similarly saves and restores the DS register, *whether or not* it is contained within a module that has a non-empty local data segment. But if it is, the DS register is likewise set up after being saved. The reason for saving and restoring DS even if there is no local data segment is that C could be passed as a parameter to an external function R′ that sets up DS for its own purpose. If, when R′ calls C, C jumps to L, control has been returned to R, but with DS now changed. R must therefore restore DS upon return.

Currently, High C does not allow a `goto` from a contained function to its parent; this can be done only by using the library function "`setjmp`". However, this extension is a likely one, since it is an important facility, and "`setjmp`" is a very low-level way of achieving the same thing.

11.2 Prologues and Epilogues

[Small, Compact, Medium, Big, Large]

The instructions used to set up a function's stack frame differ depending on the memory model being used and whether the function is nested, exported, or passed as a parameter.

Prologue/Epilogue for Small, Compact Memory Models

```
PUSH   BP              ;Save caller's BP at offset 0.
MOV    BP,SP           ;Set up new stack frame address.
                       ;First parameter is at 4[BP].
PUSH   BX              ;Store static link at offset -2.
                       ;  (Omitted if level 1).
SUB    SP,Frame_size   ;Allocate (link and) locals.
                       ;Locals start at -4[BP],
                       ;  or if no link at -2[BP].
...                    ;Body of the subroutine.
MOV    SP,BP           ;Deallocate link and locals.
                       ;  (Omitted if Frame_size = 0.)
POP    BP              ;Restore caller's BP.
RETS                   ;Short return to caller.
                       ;Caller must pop parameters.
```

n is the number of bytes occupied by the parameters. By default the caller must pop the parameters from the stack. However with the use of the Calling_convention pragma the user may specify that the callee is to pop the parameters; this convention is used in Professional Pascal, for example, because it generates smaller code. Thus with the proper use of the Calling_convention pragma, High C modules can communicate with Professional Pascal routines.

Prologue/Epilogue for Medium and Big Memory Models

```
PUSH   BP              ;Save caller's BP at offset 0.
MOV    BP,SP           ;Set up new stack frame address.
                       ;First parameter is at 6[BP].
PUSH   BX              ;Store static link at offset -2.
                       ;  (Omitted if level 1).
SUB    SP,Frame_size   ;Allocate (link and) locals.
                       ;Locals start at -4[BP],
                       ;  or if no link at -2[BP].
...                    ;Body of the function.
MOV    SP,BP           ;Deallocate link and locals.
                       ;  (Omitted if Frame_size = 0.)
POP    BP              ;Restore caller's BP.
RETF                   ;Far return to caller.
```

The code for Medium and Big models is the same as that for the Small and Compact models except that a far return is executed instead of a near return and the formal parameters begin at 6[BP] instead of 4[BP].

Prologue/Epilogue for Large-Model Exported Function

```
PUSH   BP              ;Save caller's BP at offset 0.
MOV    BP,SP           ;Set up new stack frame address.
                       ;First parameter is at 6[BP].
PUSH   BX              ;Save caller's static link.
                       ;  (Omitted if level 1).
SUB    SP,Frame_size   ;Allocate locals.
                       ;Locals start at -4[BP],
                       ;  or if no link -2[BP].
; See paragraphs above on Large-data model and DS
; for when the next four instructions are applicable.
PUSH   DS              ;Save caller's DS register.
MOV    AX,Static_seg   ;Load address of static segment.
MOV    DS,AX           ;Store into DS register.
...                    ;Body of the function.
POP    DS              ;Restore caller's DS.
MOV    SP,BP           ;Deallocate locals.
POP    BP              ;Restore caller's BP.
RETF                   ;Far return to caller.
```

Pro-/Epilogue for Large-Model Passed, Nested Function

```
(Same as the latter, except for one place.  )
      (Add just before the SUB SP,Framesize )

PUSH   BX              ;Save caller's static link.

      (Now the locals start at -4[BP]:      )
      (  the static link is saved.          )
```

11.3 Parameter Passing

[parameters passed by value in reverse order]

Actual parameters are pushed on the stack in reverse order by default (but see the Calling_convention pragma in Section *Externals*). Since the stack grows backwards, the parameters appear in ascending order on the callee side. As stated before, all parameters occupy an even number of bytes.

All parameters are passed by value. That means that the called function may modify its parameters without affecting the

values on the calling side. When a structure is passed by value the entire structure is pushed.

11.4 Function Results

[_RETURN_POINTERS_IN_ES_BX]

Function results are returned to the caller in a variety of ways depending on the function's return type. In what follows, "scalar" denotes `int`, `char`, `long int`, `short int`, and their `signed`/`unsigned` variants. "Record" denotes `struct` or `union` types.

1) Scalar or record type occupying a byte: AL register. (AH is not set.)
2) Scalar or record type occupying a word: AX register.
3) Scalar or record type occupying two words: DX:AX register pair with the most significant two bytes in DX and the least significant two bytes in AX.
4) Pointer type — in a small-data model: AX; in a large-data model: paragraph address in DX; displacement in AX. However, if the calling convention attribute _RETURN_POINTERS_IN_ES_BX is used, the function returns a small-data pointer in BX and a large-data pointer with paragraph address in ES and displacement in BX.
5) Extended lambda type "()!" — if it occupies two words, then as in 3) above; otherwise (three words) as in 7) below.
6) `float`: DX:AX register pair with the most significant two bytes in DX and the least significant two bytes in AX.
7) The following are returned in a temporary whose address is implicitly passed by the caller as the first parameter:

   record types occupying three bytes or longer than four bytes;

   `double` and `long double`.

12
Debugging

12.1 Post-Mortem Call-Chain Dump

[run-time error; producing a call-chain stack dump; FStackDump library function; line number debugging; toggles Emit_line_table, Emit_line_records; STKDMP.OBJ, DEBUGAIDS.CF]

When the run-time system detects an error, the typical action is to abort the program. It is possible to arrange that the abort is accompanied by a dump of active routines (functions) at the time of the error.

The first line of the dump names the most recently called routine. Each subsequent line names the caller of the routine named on the preceding line. The dump has this format:

ROUTINE	AT	IN MODULE	WAS CALLED NEAR	WITH ACTUAL PARAMETERS
Rtne_name	Line#lll	Mod_name	Line#lll	xxxx |xxxx |xxxx |xxxx |xxxx
or:	seg:off		seg:off	

Rtne_name is the name of the called routine; Mod_name is the module in which it resides. Beneath "AT" is the line number of the routine within Mod_name, or entry-point address of the routine. For small-code memory models, Mod_name is always the name of the first module in memory, so it is not very useful.

"WAS CALLED NEAR" gives the line number of the call in the module of the caller (mentioned on the *next* line) or the address of the call. The line numbers are produced only for those modules that were compiled with the toggle Emit_line_table On; see Section *Compiler Toggles.*

Actual parameters are divided into 16-bit quantities and are listed with the first parameter first. Each "xxxx" is hexadecimal for a 16-bit quantity. Because the 8086 reverses the byte ordering on integers the bytes in "xxxx" must be reversed before the value can be read. For

example, "050D" is the 8086 internal representation of hexadecimal "0D05", or 3333.

This post-mortem dump can actually be invoked at any time during the execution of a program. See the description of the interface file DEBUGAIDS.CF in Section *Utility Packages* or read the on-line version of the file.

The dump facility is not available unless the object file STKDMP.OBJ, provided in the distribution for each memory model, is linked with the program. By default a dummy dump routine is linked in that instead of producing a dump prints a message that the dump is unavailable.

12.2 Post-Mortem Heap Dump

[heap corruption, HEAP1.OBJ]

If the run-time heap manager detects an error, the typical action is to abort the program. It is possible to arrange that the abort is accompanied by a dump of the contents of the heap at the time of the error. The dump identifies each heap (there may be several), prints a portion of the contents of each item allocated in the heap, and prints the heap's free chain.

Heap corruption can occur when a pointer is mistakenly used after it is freed. The heap manager places free-chain information at the location of the pointer when the storage is freed. At the next call to "`malloc`" or "`free`" the free chain may be found to be corrupt, and the heap dump invoked. Corruption also occurs when a program stores into memory past an allocated area, damaging links in the list of allocated areas.

To make the heap dump of value in finding out why the program is corrupting the heap, change all calls to "`malloc`" in the program to call a function that calls "`malloc`" and prints out the address returned: use the "`%p`" format directive with "`printf`" to print out a pointer. Similarly, change all calls to "`free`" in the program to first print the address freed before calling "`free`". This produces a record of the pointers (de-) allocated.

The heap dump pinpoints the heap area whose links are corrupt. Determine if any pointers were overrun or used after they were freed, thus clobbering the links.

This determination is difficult since writing arbitrary information on top of the links can thoroughly confuse the heap manager and sometimes produce confusing dumps. This dump should be used only as a last resort; instead, take care in using pointers and avoid the problem altogether.

The heap dump facility is not available unless the object file HEAP1.OBJ, provided in the distribution for each memory model, is linked in with your program. By default a dummy dump routine is linked in which, instead of producing a dump, prints a message that the dump is unavailable.

12.3 Concurrent Assembly Language Debugging

[toggles Emit_names, Emit_line_records]

Concurrent provides the symbolic debugger SID. Here we give some additional information about the run-time environment to get the programmer started debugging. We do not explain how to use SID; for detailed information on SID consult the **Programmer's Utilities Guide for the CP/M-86 Family of Operating Systems and Concurrent DOS-286** by Digital Research, Inc. SID is invoked by typing:

```
sid prog prog $t <parameters to prog>
```

where PROG.286 is the file resulting from linking a program, and PROG.SYM is the symbols file. $t signifies that the following <parameters to prog> are to be passed to prog as if it were executed with the command "prog <parameters to prog>".

If there is no .SYM file, start SID with

```
sid prog $t <parameters to prog>
```

Single-step the first instruction with the "T" command, and then display the first few instructions with the "L" command. The result is something like this:

```
jmp   …
call  <user_main> -- Start tracing here.
call  …
ret
```

The **jmp** instruction transfers control to run-time initialization, including initializing the user stack, heap, and the 80-87/80287 chip. The initialization returns to the call instruction following the **jmp**. Therefore one should go to this **call** instruction with the "G" command and trace thereafter.

Each function is preceded by its name if toggle Emit_names is turned On (by default it is Off). Thus, to find out which function is being called just display memory preceding the call address. Doing this for the target of the second **call** above produces "main", the name of the user-written C function that the initializer calls. By using this display one can keep track of the execution flow path without much difficulty. This is unnecessary, of course, if a . SYM file is used.

If the program is being run with a large-code model, the name of each code segment is within the first 16 bytes of the segment.

For more information on how the architecture is used see Section *Run-Time Organization*.

13
Externals

13.1 Interfacing to Other Languages

[interfacing to Pascal, FORTRAN, PL/M; pragma Calling_convention, pass-by-reference parameters; parameter passing]

To facilitate using software written in languages other than High C, High C compilers provide means of specifying a variety of calling conventions for functions. This allows the user to link High C code with programs written in various other languages, such as FORTRAN, Pascal, and PL/M.

The responsibility for getting data communicated properly is left to the programmer, since different languages and compilers map data in different ways, but allowing compatible calling conventions solves the most difficult problem. See Sections *Inter-Language Communication* and *Storage Mapping.*

The two forms of this pragma are as follows:

```
pragma Calling_convention(Expression);  /* and */
pragma Calling_convention(Expression,_DEFAULT);
```

The Expression must be of a scalar type and is interpreted as the calling convention. The second form not only sets the current calling convention but also sets the default calling convention. *Note:* _DEFAULT must be spelled exactly that way, including UPPER case.

The calling convention is a bit pattern where each bit specifies a particular calling convention attribute. In writing the Expression certain pre-defined constants may be used to obtain the bits to be combined. Here are the constant names and their meanings:

Name	Semantics
_BY_REF	Parameters are passed by reference, as in FORTRAN, so they must be declared as pointer types.
_CALLEE_POPS_STACK	The called function pops its parameters off the stack upon return. Otherwise the caller must pop the stack. (Not applicable for High C on machines with automatic parameter popping, such as the VAX, or that do not support an explicit stack, such as the IBM 370 series.)
_INTERRUPT	Specifies that the function is an interrupt handler. On some machines this is required because a hardware interrupt may use a different calling sequence than for a normal call; for example, the machine status is typically saved and restored across interrupts. Interrupt (handling) functions are supported by the Interrupts package documented in Section *Utility Packages*. (On machines/operating systems in which interrupts are not supported, this package does not exist.)
_REVERSE_PARMS	Push the parameters in reverse order on the stack. (The first parameter is pushed last.)
_RETURN_POINTERS_IN_ES_BX	Applicable to the 8086 architecture only. Specifies an efficient return convention for functions returning type "pointer-to-...": the result is returned in the ES:BX pair if it is a 32-bit pointer, and BX if it is 16-bits. See Section *Run-Time Organization* for information about function return conventions.

In addition, two other pre-defined names indicate the current convention being used and the default calling convention (to which the current calling convention is initially set):

```
_CALLING_CONVENTION      /* Current convention. */
_DEFAULT_CALLING_CONVENTION          /* Default. */
```

The calling convention has an effect on all function declarations, whether the declaration is a definition or not. The default convention for C is _REVERSE_PARMS.

By contrast, for Professional Pascal it is (_REVERSE_PARMS | _CALLEE_POPS_STACK). The C convention is less efficient than the Pascal convention, but the former permits passing varying numbers of arguments to a function, since in the former the caller pops the arguments from the stack. However, if varying numbers of arguments is not needed, the more efficient convention may be chosen. See also toggle `Callee_pops_when_possible` for a way to direct the compiler to choose the Pascal convention when possible.

For example, the following may be used to communicate with Pascal:

```
#define Pascal (_REVERSE_PARMS | _CALLEE_POPS_STACK)
#define C _REVERSE_PARMS

pragma  Calling_convention(Pascal);
extern  void In_Pascal(int i, int j);
pragma  Calling_convention(C); /* Return to C convention. */

main () {
   In_Pascal(2,3);
   }
```

A suggested way of using this facility is to use macros as above to define the calling convention for each language of interest, and thereafter use only the macros.

Undeclared functions. C permits calling undeclared functions. High C compilers supply the declaration of such a function at the global level, and its calling convention is *always* the default calling convention.

Such undeclared functions are generally used for two purposes: to call an externally defined function such as "`printf`" without bothering to define it, and to call a function

declared later in the current module. If the default convention is unsatisfactory for such functions, the programmer must explicitly declare them; the explicit declarations can be encompassed by `Calling_convention` pragmas.

If there is any question as to the effect of a calling convention pragma in terms of the code sequence generated for a call, the answer may be found by compiling any High C program with the calling convention in question set as desired and looking at the compiler's generated code listing. See the command-line option `-asm` (command qualifier `/MACHINE_CODE` on VMS) in Section *Compiler Controls*.

13.2 Aliasing Pragmas

```
[external name clashes: linker limitations; aliasing variable,
function names; pragmas Alias, Global_aliasing_convention; case
shifting in aliasing conventions]
```

The names of variables and functions that are communicated across module boundaries are normally made public in the resultant object module. In large programs there may be hundreds or even thousands of such names, so name conflicts are likely to occur.

Unfortunately neither C nor most linkers provide for a structured name space — for named packages of resources, for example. Worse yet linkers often put low limits on the number of characters per name. Thus the well-chosen "internal" names in a program may not also be useable as "external" names (those known to the linker) as they should be. Thus some, preferably automatic, method of aliasing internal names to externals is needed, and High C provides it.

It is important to be able to alias such names to avoid conflicts in the linker's external symbol dictionary, rather than being forced to pervert the internal names themselves. It is the internal names that are most important to be well-chosen "containers of meaning", for program maintainability.

(The external names are also important in that respect, but we feel that the proper solution there is a "module interconnection language" and associated linker with a structured dictionary to match the overall structure of the program.)

`Global_aliasing_convention`. This convention has effect everywhere unless specifically overridden. It specifies the automatic aliasing of names. Its syntax is

```
pragma Global_aliasing_convention(<Form>);
```

where <Form> is a *constant* string expression with value F that specifies the way each resource's external name X is to be derived from its internal name R. Specifically, X is the actual text of F except for the following substitutions for substrings within F: "`%r`" (resource) denotes R; "`%%`" denotes just "`%`".

In addition "`%r`" can be followed by substring designators of the following form: "`:Start:Len`" denotes the substring starting at character `Start` and going for `Len` characters. The character positions are numbered starting at one. If `Start` is negative, it denotes the starting position from the right end of the string where the last character is numbered -1. If "`:Len`" is omitted, it means "to the end of the string".

For example, if the `Global_aliasing_convention` were set to "`_%r`", a function named "`printf`" would be known as "`_printf`" in the object module.

The `Global_aliasing_convention` can be turned off by specifying no parameter:

```
pragma Global_aliasing_convention();
```

When the above pragma is given, the compiler does not supply any aliasing; i.e. each resource is known externally by its internal name.

By default, the `Global_aliasing_convention` is turned off. It is also configurable. See Section *Compiler Controls,* Subsection *Configuring the Compiler.*

The Global_aliasing_convention can be overridden by a programmer-specified Alias pragma for a given name — i.e. the Alias pragma takes precedence. See the discussion below.

Case shifting in aliasing conventions. In addition to "%r": "%C", "%c", and "%a"are allowed in aliasing-convention specifications. "%C" means to convert subsequent letters in the external name to upper Case. "%c" means to convert them to lower case. "%a" means not to convert subsequent letters at all, but to use them as is, each with its given case; this mode holds at the beginning of the specification.

Alias pragma. This pragma specifies for a specific internal name another name for external or public purposes. It is the alternate name that appears in the object module.

The form of the Alias pragma is as follows:

```
pragma Alias(<Internal_name>,<External_name>);
```

where <Internal_name> is the function or variable identifier being aliased and <External_name> is a *constant* string expression whose value denotes the alternate or external name.

The Alias pragma must appear *in the scope of* the declaration of the internal name. *Example:*

```
void Initialize();

   pragma Alias(Initialize,"x_initialize");
   /* The function Initialize is referenced in the  */
   /* object-module symbol table as "x_initialize". */

int A;
   pragma Alias(A,"_A");
   /* "A" is referenced in the O-M ST as "_A". */
```

13.3 Code Segmentation: the Code Pragma - (Not on UNIX.)

[naming code segments, code overlays, pragma Code]

"Code segmentation" means grouping code into named units that are manipulated by a linker. The concept of segmenting generated code is useful whether or not a program is modularized and separately compiled. However, it is irrelevant on some systems, such as UNIX.

The concept applies only to top-level or "level-one" functions: level-two and greater functions must be in the same code segment as their containing level-one function; code for one function cannot be split across segments. Note that level-two and greater functions are a High C extension; standard C permits only level-one functions.

By default, compiler-generated code is placed in a segment whose name is the name of the file being compiled. The default code segment name can be overridden by using the Code pragma.

The Code pragma explicitly specifies the name of the code segment in which the code of the subsequent functions is to reside. The pragma has two forms that must be used in pairs to bracket the relevant source code:

```
pragma Code(<Segment_name>);
...     --Affected source code goes here.
pragma Code();
```

where <Segment_name> is a *constant* string expression that denotes the name of the code segment. The second form is used to terminate the effect of the previously specified Code pragma so that subsequent code will be placed in the default segment. The pragma applies only at the outer level of the program so the pairs are not nestable. *Examples:*

```
/* Assume default segment "MAIN". */

   void A() {...}
        /* A's code is placed in segment "MAIN". */

      pragma Code("SEG1");
   void B() {...}
        /* B's code is placed in segment "SEG1". */
      pragma Code; /* Ends pragma Code("SEG1"). */

   void C() {...}
        /* C's code is placed in segment "MAIN". */
   ...
```

Note: The Code pragma that is active over the body of a function applies to the function's code. The Code pragma active at any prior declaration for the function is irrelevant.

Rationale

An overlaying loader may be employed to reduce the memory requirements of a program. Then only the parts of the program that are actually necessary need be in memory at any given time. Virtual memory operating systems, such as Berkeley UNIX, provide this facility for all programs, but many primitive systems, such as MS-DOS and CP/M, do not.

Overlaying loaders typically allow one to specify the object files to be loaded in any given overlay. Good overlaying loaders automate the loading of overlays so that the source code in no way contains, or reflects the existence of, any calls to an overlay manager.

One way to achieve the desired effect is to break up a program into separate source modules, thus producing separate object modules, and instruct an overlaying loader to place certain object modules in certain overlays. Similarly, a large module might be further sub-divided into more modules.

Breaking up a module is not always appropriate, however. As an example, a string-table module may contain "initialization" and "working" code. The initialization code should be loaded in a first overlay, and the working code in another over-

lay that replaces the first. If the module is small enough, it may be a nuisance to break it up and manage more source and object files.

Therefore High C provides fine-grained control over the placement of code. The Code pragma directs generated code into distinct, specified code segments, even though all that code goes into a single object file. Overlaying loaders can then be instructed to include code segments in various overlays.

13.4 Data Segmentation: the Data Pragma

[global and automatic data; data communication in separately compiled modules; segment names; Common segments]

Audience. Read this section only if interested in either (a) communicating with programs written in Professional Pascal or (b) using a data communication convention different from that of standard C.

Communication between separately-compiled modules is achieved by using the `extern` storage class in C. On non-UNIX systems, we follow the ANSI standard requirement for declarations that there be exactly one defining declaration of each variable. On UNIX systems, multiple defining declarations of a variable `x` are allowed, as long as at most one of them initializes `x` (thus the `extern` storage class is not required).

(In the UNIX scheme, omission of the keywod `static` in a declaration may cause an accidental "merging" of one variable with another. For example, if two distinct modules contain "`int x;`", and the intent in each was to have a local, private `x`, the keyword `static` is necessary. If accidentally omitted, the two `x`'s denote the same object.)

The High C `Data` pragma provides an alternative method of sharing data, using named common segments. Its general usage is illustrated by:

```
pragma  Data( ... specify "Common" here ... );
int     X,Y,Z: ...;
... /*Other normal C declarations may appear here. */
pragma  Data; /*"Turns off" the prior Data pragma.*/
```

Scope. Each `Data` pragma must be terminated or "turned off" as illustrated above in the same scope that it is turned on. The storage class specification applies only to variable declarations between the specification and its termination *but not to any inside contained (lower-level) functions.* That is, variables declared at lower levels — local to surrounded (nested) function declarations — are not affected: at a function declaration, any active `Data` pragma temporarily becomes inactive and the default applies through the end of the function.

A compile-time warning is issued if a `Data` pragma is specified when a prior `Data` pragma is still active (in which case the subsequent pragma applies), or if a `Data` pragma is active at the end of a function declaration or at the end of a compilation unit. Thus `Data` pragmas cannot be nested within a single function, though they can be nested if they apply to the local variables of distinct functions.

Parameters. The specifying `Data` pragma takes as parameters the storage class designator Common and an optional segment name. The segment name, if specified, must be a constant expression whose value is a string: the desired segment name. The ending `Data` pragma has no parameters.

The two forms of the specifying `Data` pragma, with and without the optional segment name, differ essentially in the number of names made known to the linker. Without it, the names of all the affected variables are made known individually. With it, only the given segment name is made known: each variable is addressed at a fixed offset within the segment.

For example, in the following program fragment:

```
pragma Data(Common);              --Multi-segment form.
int    Tables_are_loaded: Boolean;
struct {...} Tables;
pragma Data;
```

both names, Tables and Tables_are_loaded, are made known to the linker and each variable goes into a distinct segment in the linked program: the first is named Tables and the second Tables_are_loaded. But in the next fragment:

```
pragma Data(Common,"Tabs");     --Single-segment form.
int    Tables_are_loaded: Boolean;
struct {...} Tables;
pragma Data;
```

neither name is made known to the linker. Rather both go into a segment named Tabs in the linked program.

On UNIX systems, "`pragma Data(Common);`" is of little use, since by default each unitialized, non-`extern` variable (e.g. "`int x;`") is treated as if that pragma were in effect. Unnamed common is the standard practice on UNIX.

Multi-segment form. A desirable aspect of the first illustrated form is that the order of variable declarations does not matter. One can swap the two declarations above and not need to re-compile all the source referencing them. Such named references make the declarations position-independent.

An undesirable aspect of this form is the number of names known to the linker. A program with a large number of external names can have linker name clashes, especially if the linker imposes an unreasonably low limit on the lengths of names.

Single-segment form. The alternative form, with the optional segment name N, causes each affected variable to be addressed by a unique offset within the named segment. Thus the variable names do not appear in the external symbol dictionary. So in the second program fragment above Tables_

are_loaded is at offset zero in segment Tabs, and Tables is at offset two. The generated code contains these offsets. The linker knows only of the name Tabs and its length.

Use of this single-segment form of the pragma also means that, if two declarations were swapped, all source files referring to those declarations must be re-compiled. And if the data types used in the declarations are changed in such a way that the sizes of the variables are changed then re-compilation is necessary.

In summary, the Data pragma offers extra control and a trade-off. Without the optional segment name, affected variable declarations may be re-ordered or changed with regard to sizes, or new declarations may be added or old ones deleted, without necessarily recompiling all source files referring to the declarations.

Of course, if a size is changed, all the users of that particular variable must be re-compiled. The price to be paid is the potentially large number of external names, which in turn may cost in terms of linker name clashes.

On the other hand, with the optional segment name, the order and sizes of declarations are significant: changes require all source files referring to the declarations to be re-compiled. But one is protected from name clashes on linkers with limited name lengths.

Storage Allocation. Storage for Common variables is allocated in a static region of memory by the linkage editor. Common areas with the same name occupy the same address. If pragma Data(Common, "N") is used, all the pragma-bracketed variables go into a single segment named N. If pragma Data (Common) is used, each variable goes into a separate segment named by the variable name.

Common variables can be used in the following simple manner. A single copy of the declarations of Common variables may be placed in a single include file. Each module that references

these variables simply includes this file. This scheme reliably prevents inconsistent declarations of the same Common variable. *Example:*

File main.c:
```
#include "d.cf"
main () {
   v = 1;
   s = "%d";
   sub();
   }
```

File sub.c:
```
#include "d.cf"
sub () {
   printf(s,v);
   }
```

File d.cf:
```
pragma Data(Common);
int v;
char *s;
pragma Data;
```

Name clashes can be a minor problem with Common. Distinct Common declarations can be mapped to the same storage location by a linker that recognizes only limited name lengths. This may happen even if the two Common areas have different sizes, although some linkers will complain. Thus, the two declarations

```
pragma Data(Common);
int Much_big;
pragma Data;
```

```
pragma Data(Common);
int Much_bigger[500];
pragma Data;
```

may resolve to the same memory address under an eight-character limitation, even though the two Much_big's are entirely different. Thus, care must be taken in using the Common storage class.

Rationale

The notion of Common permits a single declaration of a shared data resource, increasing program reliability. It also facilitates porting UNIX programs to non-UNIX environments, since standard UNIX C compilers permit duplicate defining declarations of a resource — implementing such via Common.

To remain compatible with standard C and avoid the labor of having to modify each and every declaration when its Common is used, the bracketing Data pragma was chosen. Thus normal declarations need not be modified to assign the variables to a

different storage class. This allows for greater program portability and ease of modification.

The two Data-pragma forms — with and without the optional segment name — were provided to give better control over linker name space. With the segment name, linker name space is reduced considerably at the cost of more re-compilation when changing the contents of the segment, because the compiler's references into the segment are by position, not by name. Without the segment name, compilation is reduced but linker name space is increased.

13.5 Data Segmentation: the Static_segment Pragma

[overlaying data, pragma Static_segment]

In all cases but the Large memory model, all variables defined in a compilation unit are placed in a public data segment whose name S is constructed from the name of the file being compiled by prefixing it with "@". In all but small-data models, literals also go into S.

S can be changed with the Static_segment pragma:

```
pragma Static_segment(<Seg_name>);
```

where <Seg_name> is a *constant* string expression with value S that is the name of the segment into which defined variables will be placed.

In addition, if literals were going into the same segment as variables, they will subsequently go into S. The Literals pragma in the next subsection can change where literals are placed.

By assigning a different segment to variables, they can be easily overlayed. In addition, a single compilation unit can have some variables that are overlayed and some that are not by simply changing the static segment and instructing the overlaying linker to overlay the desired segments. *Example:*

```
pragma Static_segment("overlay1");
int x,y,z;
static a,b,c;
pragma Static_segment("no_overlay");
static char q,r,s;
```

Here an overlaying linker such as PLINK86 can be instructed to overlay the segment "overlay1" but *not* overlay "no_overlay".

Note: In the Large memory model each exported variable is assigned its own segment. Thus in the example above, only the variables of static storage class (a,b,c,q,r,s) are affected by the Static_segment pragma.

13.6 Specifying a Literals Segment

[overlaying literals, pragma Literals, toggle Literals_in_code]

Normally, the compiler places all literals, consisting of floating-point constants and quoted strings, in a data segment of the compiler's invention. In small-data models the segment is _mwLITERALS. In medium- and large-data models the literals join all variables defined in the compilation unit in a segment whose name is constructed from the name of the source file by preceding it with an "@". In both cases the segment is public.

The toggle Literals_in_code directs the compiler to place all non-string literals in code for non-small-data models; it has no effect for small-data models. While this toggle helps by allowing literals to be overlayed along with code in large programs, in and of itself it still does not allow string literals to be overlayed, since string literals are *not* placed in code —

they are by default writeable in C and hence must be data. But see Section *Compiler Toggles* for the `Read_only_strings` toggle, which causes string literals to be treated the same as floating-point literals, and thus potentially placed in code.

Pragma `Literals` specifically directs the compiler where to place literals:

```
pragma Literals(<Seg_name>);
```

where `<Seg_name>` is a *constant* string expression with value S, causes the placement of any subsequently encountered literals in a public data segment known as S.

After separating out literals from other variables in a compilation unit, the literals can be overlayed while the variables can remain non-overlayed. This is important if the variables are modified between calls to the overlay. But if the values of the variables do not need to be retained between calls to the overlay, the `Literals` pragma need not be used: the single data segment into which all literals and variables go can then itself be overlayed.

13.7 Group Names: Pragmas `Cgroup` and `Dgroup`

To enforce the 64KB code and data restrictions on small-code and small- and medium-data memory models, Intel OMF uses the notion of "group". The code group is where all code goes in small-code models; the data group is where all static data goes in small- and medium-data models. For more information on groups, consult Intel's OMF manual **8086 Relocatable Object Module Formats**, part number 121748-001.

The compiler uses code group name `CGROUP` and data group name `DGROUP`. This is reflected both in the code generated for user programs and in the object modules in the Run-Time libraries. Other compilers may use different conventions. To change the group names to conform to external conventions, the pragmas `Cgroup` and `Dgroup` are provided:

```
pragma Cgroup("Group_name");

pragma Dgroup("Group_name");
```

Do not use these pragmas unless you are experienced with OMF and its design. For example, no code with groups named other than CGROUP and DGROUP will work correctly with the Run-Time Libraries (except for memory model Large, which has no groups). Use the pragmas for stand-alone applications or for compiling code that will be linked with programs and libraries supplied by others.

14

Inter-Language Communication

14.1 Communication between HC, PP, and Asm

This section shows how to share data and code between MetaWare's Professional Pascal (PP), MetaWare's High C (HC), "plain" C, and assembly language (Asm) modules. By "plain C" we mean the language implemented by other C compilers — typically just Kernighan and Ritchie C or a subset of it. The term "C" alone will include both plain C and High C, since High C is a superset of plain C.

In all of the languages mentioned, modules can be separately prepared and later linked to form a composite program. Separate modules share data and code by means of using `extern` in C data declarations, or the `Data` pragmas of High C, or the packages and `Data` pragmas of Professional Pascal.

This section is organized as follows. First, a complete example of communication for each of three pairs of languages is presented: Pascal and C, C and Assembler, and Pascal and Assembler. In each example both code and data originating in one language are used in the other. Brief notes on critical details in the communication are given after each example. Second, a general description is presented in several additional sections.

The Pascal and C languages make slightly different uses of the machine architecture. Therefore in their case we present two examples: one with C as the main program and the other with Pascal as the main program. In the first we force the Pascal program to adhere to plain C conventions; thus this example is suitable for both plain C and High C. In the second we force the C program to adhere to Pascal conventions; thus this example is suitable for High C only, since the conventions

of plain C are unchangeable. This illustrates how both High C and Professional Pascal are flexible in using the conventions of other languages.

The reader should first read the example for the particular language pair of interest. If that is not enough to answer relevant questions, continue on the rest of the chapter where we discuss in detail: (1) how the data types of the three languages correspond, (2) the naming conventions of each language, (3) how to communicate data between the languages, (4) the parameter type correspondences in function/routine declarations, and (5) how to call one language from the other.

14.2 Example: PP and C with C Main Program

High C or plain C:

```
int cvar = 5;      /* Defined in C,          value 5. */
extern int pvar;   /* Defined in Pascal,     value 10.*/
extern int pfcn();/* No argument checking here.     */

int cfcn (Multiplier,Adder) int Multiplier,Adder; {
   return (cvar+pvar)*Multiplier + Adder;
   }
void main () {
   int sum_times_4_plus_6;
   sum_times_4_plus_6 = pfcn(2,3);
   /* Prints 66. */
   printf("%d\n",sum_times_4_plus_6);
   }
```

Professional Pascal:

```
Export (Pascal_package);

pragma   C_include('Language.pf');

package     Pascal_package;

            -- Override default convention:
   pragma   Routine_aliasing_convention('%r');
            -- Adhere to C calling conventions:
   pragma   Calling_convention(Language.C);
```

```
    function  Pfcn(Mult,Add:Integer):Integer;External;
    pragma    Calling_convention();

    pragma    Data(Export);   -- Export to C.
    var Pvar: Integer;
    pragma    Data;

    end; --  Pascal_package;

program  Pascal_subroutine;

pragma    Data(Import);       -- Import from C.
var Cvar: Integer;
pragma    Data;

pragma    Calling_convention(Language.C);
function cfcn(Mult,Add:Integer):Integer;     External;
pragma    Calling_convention();

value Pvar := 10;

function Pfcn(Mult,Add:Integer):Integer;
   begin
   Pfcn := (Cvar+Pvar)*Mult + Add + cfcn(Mult,Add);
   end;
```

Notes:

1. Pascal is case-insensitive: the compiler maps all identifiers to lower-case. Thus the C program was written with identifiers in lower-case, in case case-sensitive linking is used.

2. The Pascal and C calling conventions are different: In C the caller pops the parameters off the stack. The Pascal program was made to conform to the C calling conventions.

3. The Pascal and C conventions for allocating data are different. Pascal was made to conform to C's conventions.

4. There may be only one main program: either Pascal or C. As the C program is the main program, one should link with the C Run-Time Library specified ahead of the Pascal Run-Time Library. In this particular example there is no need at all

for the Pascal Library. See Section *Linking a Compiled Program* for more information on linking Pascal with C.

5. Plain C does not allow the specification of the parameters of external functions. See the next example where we assume High C and obtain the protection of such parameter specification.

14.3 Example: PP and HC with PP Main Program

High C:

```
/* Adhere to the Pascal data-definition convention: */
pragma Data(Common,"?from_c");
int cvar = 5;        /* Defined in C,       value 5. */
pragma Data();

pragma Data(Common,"?pascal_package");
extern int pvar;     /* Defined in Pascal,  value 10.*/
pragma Data();

/* Adhere to the Pascal calling convention: */
pragma Calling_convention
  (_DEFAULT_CALLING_CONVENTION | _CALLEE_POPS_STACK);

extern int pfcn(int Multiplier, int Adder);
int cfcn (int Multiplier, int Adder) {
   return (cvar+pvar)*Multiplier + Adder +
                  pfcn(Multiplier, Adder);
   }
pragma Calling_convention
  (_DEFAULT_CALLING_CONVENTION);
```

Professional Pascal:

```
Export (Pascal_package);

package Pascal_package;
              -- Override default convention:
   pragma    Routine_aliasing convention('%r');
   function  Pfcn(Mult,Add:Integer):Integer;External;
   var Pvar: Integer;
   end;
```

```
package From_C;
                -- Override default convention:
   pragma    Routine_aliasing_convention('%r');
   function  Cfcn(Mult,Add:Integer):Integer;External;
   var Cvar: Integer;
   end;

program Pascal_main_program;

value Pvar := 10;
function Pfcn(Mult, Add:Integer): Integer;
   begin
   Pfcn := (From_C.Cvar+Pvar)*Mult + Add;
   end;

var Sum_times_4_plus_6: Integer;
begin
Sum_times_4_plus_6 := From_C.Cfcn(2,3);
Writeln(Sum_times_4_plus_6);
end;
```

Notes:

1. Pascal is case-insensitive: the compiler maps all identifiers to lower-case. Thus the C program was written with identifiers in lower-case, in case case-sensitive linking is used.

2. The Pascal and C calling conventions are different: In C the caller pops the parameters off the stack. The C program was made to conform to the Pascal calling conventions.

3. The Pascal and C conventions for allocating data are different. C was made to conform to Pascal's conventions.

4. There may be only one main program: either Pascal or C. As the Pascal program is the main program, one should link with the Pascal Run-Time Library specified ahead of the C Run-Time Library. In this particular example there is no need at all for the C Library. See Section *Linking a Compiled Program* for more information on linking Pascal with C.

5. When the Pascal Run-Time Library is linked ahead of the C Library, any C functions registered with the C Library function onexit are *not* called at program termination. Also,

library, I/O calls may clear the C Library variable "errno", whereas with C alone "errno" is always "sticky": the user must explicitly clear it to zero after an error has occurred. See Section *Linking a Compiled Program* for more information on linking Pascal with C.

14.4 Example: HC and Asm with HC Main Program

High C:

```
int cvar = 5;      /* Defined in C,         value 5. */
extern int avar;   /* Defined in Assembly,  value 10.*/

extern int afcn(int Multiplier, int Adder);

int cfcn (int Multiplier, int Adder) {
   return (cvar+avar)*Multiplier + Adder;
   }
void main () {
   int sum_times_4_plus_6;
   sum_times_4_plus_6 = afcn(2,3);
   /* Prints 66. */
   printf("%d\n",sum_times_4_plus_6);
   }
```

Assembly:

```
        extrn   SMALL?:word

        extrn   cfcn:near

        public  avar
ADATA   cseg    word
        extrn   cvar:word
avar    dw      10

DGROUP  group   ADATA

ASM     cseg    word
CGROUP  group   ASM
```

```
        public  afcn
        db      'afcn',4    ; For StackDump.
afcn:   push    bp
        mov     bp,sp
        push    6[bp]       ; Parameter Add.
        push    4[bp]       ; Parameter Mult.
        call    cfcn
        add     sp,4        ; Pop the parameters.
        mov     cx,ax       ; Save the fcn result.
        mov     ax,cvar
        add     ax,avar
        imul    word ptr 4[bp]
        add     ax,cx       ; Add cfcn(Mult,Add).
        add     ax,6[bp]
        mov     sp,bp
        pop     bp
        ret                 ; Pop no parameters.
        end
```

Notes:

1. In C the caller pops the parameters off the stack. Thus the assembly language program pops the parameters after calling "cfcn", and does not pop its own parameters at return. However, in this case the pop after the call to "cfcn" could be eliminated since SP is restored at the return of the assembly subroutine.

2. Arguments are pushed on the stack in reverse order of appearance in an argument list, since the stack grows down in memory. Therefore the arguments appear in ascending order on the callee side.

3. The insertion of the name of the assembly language subroutine in ASCII text just preceding the subroutine body is for the use of the post-mortem call-chain dump facility. It can be omitted, but then the subroutine will not be identified in the

trace. For the call-chain dump to work, the BP register must be pushed in the location indicated.

4. This example is for the Small memory model, hence the use of near in and "extrn cfcn:near". Parameters start at 4[BP], and the assembly language program must group code and data into groups CGROUP and DGROUP, respectively. The C external "cfcn" must be declared within a code segment that is CGROUP-ed.

5. The Pascal and C compilers assume the direction bit is clear at all times. Therefore the assembly subroutine should not leave it set. See the CLD and STD instructions in any 8086 instruction set description.

6. The assembly language code and data segments are word-aligned following the practice of the Professional Pascal and High C compilers. Although this is not strictly necessary in Intel OMF, some cross-linkers complain if shared segments do not agree in their alignment. Word alignment permits tight allocation of segments in a memory image. Byte alignment, although affording tighter allocation, frustrates the compiler's attempts at aligning variables occupying more than a byte on a word boundary for greater efficiency with the 8086 16-bit memory bus.

14.5 Example: PP and Asm with PP Main Program

Professional Pascal:

```
Export (Pascal_package);

package Pascal_package;

                -- Override default convention:
   pragma       Routine_aliasing_convention('%r');
   function     Pfcn(Mult,Add:Integer):Integer;External;
   var Pvar: Integer;
   end;
```

```
package From_assembly;
                -- Override default convention:
   pragma    Routine_aliasing_convention('%r');
   function  Afcn(Mult,Add:Integer):Integer;External;
   var Avar: Integer;
   end;

program Pascal_main_program;

value Pvar := 5;
function Pfcn(Mult, Add:Integer): Integer;
   begin
   Pfcn := (From_assembly.Avar+Pvar)*Mult + Add;
   end;

var Sum_times_4_plus_6: Integer;
begin
Sum_times_4_plus_6 := From_assembly.Afcn(2,3);
-- Should print 66:
Writeln(Sum_times_4_plus_6);
end;
```

Assembly:

```
          extrn    BIG?:word

          extrn    pfcn:far

?FROM_ASSEMBLY    dseg common word
avar      dw       10

?PASCAL_PACKAGE   dseg common word
pvar      rw       1          ; Reserve 1 word.

DGROUP    group    ?FROM_ASSEMBLY, ?PASCAL_PACKAGE
          db       3,'asm',3  ; For StackDump.
                              ; 3 = Length('asm').

ASM       oseg     word
```

```
        public  afcn
        db      'afcn',4    ; For StackDump.
afcn    proc    far
        push    bp
        mov     bp,sp
        push    8[bp]       ; Parameter Add.
        push    6[bp]       ; Parameter Mult.
        callf   pfcn
        mov     cx,ax       ; Save the fcn result.
        mov     ax,cvar
        add     ax,avar
        imul    word ptr 6[bp]
        add     ax,cx       ; Add pfcn(Mult,Add).
        add     ax,8[bp]    ; Add parameter Add.
        mov     sp,bp
        pop     bp
        ret     4           ; Pop 4 bytes of parms.
        end
```

Notes:

1. In Pascal the callee pops the parameters off the stack. Thus the assembly program does not pop the parameters after calling "pfcn", and pops its own parameters upon return.

2. Arguments are pushed on the stack in reverse order of appearance in an argument list, since the stack grows down in memory. Therefore the arguments appear in ascending order on the callee side.

3. The insertion of the name of the assembly language subroutine in ASCII text just preceding the subroutine body is for the use of the post-mortem call-chain dump facility. It can be omitted, but then the subroutine will not be identified in the trace. Similarly, the name of the assembly language code segment is given at the beginning of the segment, along with the length of the name both preceding and following it. For the call-chain dump to work, the BP register must be pushed in the location indicated.

4. This example is for the Big memory model. Hence the use of `far` in and "`extrn pfcn:near`". Parameters start at 6[BP], and the assembly language program must group data into DGROUP. The Pascal external "`pfcn`" must *not* be declared within any code segment.

5. The Pascal and C compilers assume the direction bit is clear at all times. Therefore the assembly subroutine should not leave it set. See the CLD and STD instructions in any 8086 instruction set description.

6. The assembly language code and data segments are word-aligned following the practice of the Professional Pascal and High C compilers. Although this is not strictly necessary in Intel OMF, some cross-linkers complain if shared segments do not agree in their alignment. Word alignment permits tight allocation of segments in a memory image. Byte alignment, although affording tighter allocation, frustrates the compilers' attempts at aligning variables occupying more than a byte on a word boundary for greater efficiency with the 8086 16-bit memory bus.

If it were left to us, however, "dw ?" and "dw 1 dup(?)" would mean exactly the same thing — no initial value supplied.

14.6 Data Type Correspondences

[storage mapping; padding; memory models; size of pointers]

In sharing data between languages it is necessary to know how the languages represent data types. The table below gives a brief sketch of corresponding data types. Read the two columns that correspond to the two languages of interest. The last few rows of the table inform of High C data types not present in plain C.

C Data Type	Pascal Data Type	Assembly Type
char	Char, or 0..255	**rb** 1
signed char	-128..127	**rb** 1
short	Integer	**rw** 1
int	Integer	**rw** 1
long int	LongInt	**rd** 1
unsigned short int	Cardinal	**rw** 1
unsigned int	Cardinal	**rw** 1
unsigned long int	See Note 1 below.	**rd** 1
float	Real	**rd** 1
double	LongReal	**rd** 2
type *	^*type* or Address(*type*)	**rw/rd** 1 Note 2
type [n]	**packed array**[0..n-1] **of** *type*	Note 3
Note 5	**array**[0..n-1]**of** *type*	Note 3
type (*) ()	Note 4(a)	**rd** 1
struct {*type* a, b;}	**packed record** a, b: *type* **end**	Note 3
Note 5	**record** a, b: *type* **end**	Note 3
Note 4(b)	**set of** *type*	Note 3
High C only	**Pascal Data Type**	**Assembly Type**
type ()!	**function**(): *type*	**rw/rd** 1;**rw** 1 Note6
void ()!	**procedure**()	
long double	ExtReal	**rw** 5

(*type* In the left column means a C type specifier; *type* in the middle column means the corresponding Pascal type denoter.)

Notes:

1. There is no exact equivalent to C's `unsigned long` in Professional Pascal. In C, `unsigned long` holds values from `0..MaxLong*2-1`, where `MaxLong` is from Pascal. The Pascal type `0..MaxLong` approximates the C type: unsigned operations are performed on variables of this type, but the value of such a variable cannot exceed `MaxLong` unless range checking is Off, in which case it may reach `MaxLong*2-1`.

2. The size of pointers in C and Pascal depend upon the 8086 memory model. Small-data models use 16-bit pointers (`rw 1`); medium- and large-data models use 32-bit pointers (`rd 1`), with the 16-bit offset preceding the 16-bit segment.

3. An array, record, or set in assembly language consists of a series of declarations that constitute the contents of the array, record, or set. However, alignment of records and array elements must be observed and sufficient padding introduced when necessary. See also Note 5. Bit mapping within Pascal sets is described in Section *Storage Mapping* of the **Professional Pascal Programmer's Guide.**

4. There is no correspondent. (a) C pointer-to-function types are Pascal routine-types but without the environment pointer (static link). (b) Pascal set types have no correspondent in C.

5. In Pascal unpacked records and unpacked arrays may contain padding; packed records and arrays do not. Therefore, there is no array type correspondent in C for a Pascal array containing padding. For a Pascal `record` containing padding, the corresponding C `struct` must contain extra declarations to provide the padding. Pascal padding is defined relative to the alignment of Pascal types; see Section *Storage Mapping* of the **Professional Pascal Programmer's Guide.**

6. The Pascal routine type and the High C "extended-lambda" type `"()!"` are comprised of a function address and

an environment pointer (static link). The address is 16 or 32 bits long depending upon the memory model: small- versus large-code. The environment pointer is always 16 bits.

14.7 Parameter Correspondence

[function prototypes; padding]

HC-Asm or PP-Asm communication. If communication between HC and Asm or PP and Asm is desired, consult Section *Run-Time Organization* of the **Programmer's Guide** for the appropriate high-level language. Described there are the parameter passing conventions for the high-level language that effect proper communication of parameters and function results to and from assembly language subroutines.

Some of these details are given in the examples at the beginning of this section and in the general assembly language templates specified in the previous Subsection.

C-PP communication. In the rest of this section we cite the differences between the C and Pascal parameter-passing conventions.

Passing parameters between Pascal routines and C functions requires an understanding of how the data types of the two languages correspond to one another, as specified in Subsection 14.6. In this subsection "corresponding (data) types" means as defined in the table in Subsection 14.6.

In C all parameters are passed by value. Arrays cannot be passed at all; instead, the address of the first element of the array is passed. Pascal allows not only passing arrays as parameters but also both by-reference and by-value parameter passing techniques. The remainder of this discussion treats Pascal by-reference and by-value excluding arrays, and finally arrays as a special case.

Pascal by-reference. A Pascal `var` or `const` parameter of type T corresponds to a C parameter of type `*T'` (pointer to T'), where T and T' are corresponding types.

Pascal by-value. Passing parameters by value is done differently depending on whether one uses the plain C function declaration that does not specify the parameter types of `external` functions or the new function-prototype declaration that does.

Plain C. Consider a plain-C-style function header. Arguments of C types `char` and `short` are widened to `int`, and those of type `float` are widened to `double`. C arguments of all other non-array types are passed as such. Therefore, Pascal function headers must be written with the data type corresponding to `int` when the parameter to be passed is `char` or `short`, with the data type corresponding to `double` when the parameter to be passed is `float`, and otherwise with the corresponding data type in all other non-array cases.

High C prototypes. Assume that the `external` C function is specified via a function-prototype declaration. Here C parameter passing semantics agrees exactly with Pascal in that parameters are passed as if by assignment to a variable of the type of the formal parameter. Thus, for scalar types and structure types — pointers, integers, characters, reals, and `struct`/`union`s — the type declared in the C prototype and in the corresponding Pascal routine header must correspond.

Pascal arrays. Because C cannot pass an array by value, a Pascal value parameter of an array type is never appropriate in communicating with C. There are three progressively lower-level approaches for by-reference arrays, each applying only to certain situations.

High-level approach. High-level Pascal arrays can be used by declaring the array on the Pascal side as passed by reference (`var` or `const`), but *only* when there is no padding in the Pascal array. In this case, the address of the first element is passed, as in C. *Example:*

C:	Pascal:
	`type T = array[1..10] of Integer;`
`int Pfcn(),a[10];`	`function Pfcn(var A:T)...;`
`Pfcn(a);`	`... A[I] := 3;`

C arrays on the 8086 are never padded. If type T above were "`var A:array[1..10] of record I:Integer; B:Boolean; end;`", Pascal would use four bytes per array element to align each Integer on an even boundary. But in 8086 C, "`struct {int I; char B;} A[10];`" uses three bytes per element, so an extra byte of padding in the C `struct` type is necessary if the C array is to be shared with Pascal.

Note that structure padding for C is machine-dependent.

Mid-level approach. Rather than use the Pascal `var` parameter, one can explicitly declare the parameter as a pointer-to or address-of the array. Use pointer-to if the array is on the heap, and address-of otherwise. ***Example:***

C:	Pascal:
	`type T = array[1..10] of Integer;`
	`type PT = ^T;` *—Array in heap.*
or:	`type PT = Address(T);` *—Not in heap.*
`int Pfcn(),a[10];`	`function Pfcn(A:PT)...;`
`Pfcn(a);`	`... A^[I] := 3;`

Professional Pascal compilers can sometimes generate better code when told, via ^ rather than Address, that an object is in the heap instead of anywhere in memory.

Low-level approach. Alternatively, since Professional Pascal supports C pointer arithmetic, one can model communicated arrays in the low-level way C models arrays: as pointers to the elements. Thus, we effectively abandon the idea of declaring the communicated parameter as an array in Pascal. Once again, use pointer-to if the array is on the heap, and address-of otherwise. ***Example:***

C:	Pascal:
	`type PT = ^Integer;` *—In heap.*
or:	`type PT = Address(Integer);` *—Not.*
`int Pfcn(),a[10];`	`function Pfcn(A:PT))...;`
`Pfcn(a);`	`... (A+I)^ := 3; -- A[I] := 3;`

14.8 Calling Routines in Other Languages

[pragma Calling_convention; direction bit]

To call a routine in a different language, both parameter conventions and routine calling conventions must be observed. Parameter conventions were treated in the previous section; here we treat calling conventions.

Pascal-C communication. The only difference between the C and Pascal calling conventions is that the caller of a C function F is responsible for popping the arguments passed to F, whereas in Pascal the callee pops the arguments. While the C convention permits the flexibility of passing a varying number of arguments to functions, it is more expensive in code space and time.

The PP and HC compilers provide control of calling conventions through the `Calling_convention` pragma. Each language can adhere to the convention of the other. For a discussion of how to specify calling conventions see Section *Externals*. Also see Subsections 14.2 and 14.3 above where one example shows a Pascal routine using the C calling convention, and vice-versa.

PP-Asm and C-Asm communication. All issues involved here are treated by examples in Subsections 14.4 and 14.5. Furthermore, since the communication intimately involves naming conventions, Subsection 14.9.4 below presents templates for preparing assembly language subroutines that communicate code with Pascal and C. Therefore we present only a summary of the issues here.

- In Pascal, the callee pops parameters from the stack, and in C the caller does.
- The parameters must be pushed on the stack in reverse order of appearance in an argument list. The parameters therefore appear in ascending sequence on the stack on the callee side.

- Parameters start at 4[BP] in small-code and 6[BP] in large-code memory models.
- Any segment register modified in an assembly subroutine must be restored before returning to the caller, except that in a large- or medium-data model the ES register need not be restored.
- PP and HC code is compiled assuming that the direction bit is clear at all times, so that if the direction bit is set in an assembly language subroutine called from PP or HC, it *must* be cleared before returning to the Pascal or HC environment; see the CLD and STD instructions in any 8086 instruction set description.
- Data must be grouped in a group called DGROUP in small- and medium-data memory models.
- Code must be grouped in a group called CGROUP in small-code memory models.
- Never place in CGROUP any `extrn` declarations of code entry points.

14.9 External Name Communication

```
[Routine_aliasing_convention; Data_aliasing_convention; pragma
Alias; Named Common, Common, Export, Import, module interface;
underscore prefix]
```

A resource R — data object or routine (C function or Pascal function, procedure, or iterator) — shared across modules must have information provided to the linker that associates a name with the address of R. This is done by causing the placement of a corresponding name in the name table of the object module with an attribute that informs the linker that the name may be shared across modules.

This name need not be the same as the name of R in the source program text containing R's declaration; it need only

be derivable from it in a unique way. We refer to such a name as the *external* name.

For modules in different languages to communicate, they must agree on how they specify the external name of a shared resource. Therefore we now consider how external names are constructed from names within modules.

Sample declarations illustrating general forms. There are four general forms for declaring external data and two for declaring external routines in each language. Below we illustrate the four data forms for each language, called: (E) Export, (I) Import, (C) Common, and (N) Named common.

Plain C uses E and I only. High C and Professional Pascal can use all four, but High C defaults to E and I. Each language supports both importing (FI) and exporting (FE) functions, and both cases are illustrated for each language.

Suppose we wish to export a variable from one module so it can be imported into another module, whatever the language on either end. There are three choices: E, C, and N. If we choose form N for exportation, we must use N also to import the variable into the other module; similarly, form C requires form C. However, form E requires I, independent of the language used in any case.

The E/I combination carries with it protection against multiple exports of any given variable. Such protection is provided by the linker. No such protection is available for the C and N forms.

Warning: You may not initialize an exported variable declared using cases E, C, and N more than once. The results depend upon the linker and are unpredictable. Initializing an imported variable — case I — is illegal and draws a diagnostic from the compiler.

Below are the forms available in each language, organized by language. Please ignore the section on any language that is not of interest.

14.9.1 Plain C Naming Conventions

In plain C, the external name associated with R is lexically exactly the same as that appearing in the declaration of the function, with a leading underscore added or not depending upon which plain C compiler is used. The table below gives the results for the Lattice 2.14 and Microsoft 3.00 C compilers.

Declaration	External names, using: Lattice	Microsoft
Case E: Exporting data:		
`int i,j;`	`i,j`	`_i,_j`
Case I: Importing data:		
`extern int i,j;`	`i,j`	`_i,_j`
Case FE: Exporting a function:		
`int fcn(a,b) int a,b;{ return a+b; }`	`fcn`	`_fcn`
Case FI: Importing a function:		
`extern int fcn();`	`fcn`	`_fcn`

14.9.2 High C Naming Conventions

In High C the external name associated with R is determined by one of two aliasing mechanisms, the `Alias` pragma and the `Global_aliasing_convention`; see Section *Externals* of the **High C Programmer's Guide.**

The `Alias` pragma specifies a literal replacement for the external name. If no `Alias` pragma is used for a particular name, the `Global_aliasing_convention` applies. The default value of this convention is that the external name is lexically exactly the same as that appearing in the declaration of R.

In addition the `Data` pragma can be used to group declarations in a segment such that only the segment's name is

known externally. Then there are no external names for the variables and the aliasing mechanisms discussed above do not apply.

The table below runs the gamut of High C declarations and illustrates the corresponding applicable aliasing mechanism and external Names. The Mechanism column indicates the aliasing mechanism that applies: Global or Alias for `Global_aliasing_convention` or `Alias` pragma; and None for cases where the resources themselves have no external names.

Declaration ---------- Mechanism: Names

Case E: Exporting data:

```
int    i,j; ------------------ Global: i,j
```

or, equivalently,

```
int    x,y; ------------------- Alias: i,j
pragma Alias(x,"i");
pragma Alias(y,"j");
/* (No further examples of Alias are given.) */
```

Case I: Importing data:

```
extern int i,j; -------------- Global: i,j
```

Case C: Unnamed common data:

```
pragma Data(Common);
int    i,j; ------------------ Global: i,j
pragma Data;
```

Case N: Named common data:

```
pragma Data(Common,"?block");
int    i,j; ------------------- None
pragma Data;
```

Note: The High C `Data` pragma totally overrides specified storage classes, i.e. storage classes are irrelevant.

Case FE: Exporting a function:

```
int    fcn(int a,int b) { ----- Global: fcn
   return a+b;
   }
```

Case FI: Importing a function:

```
extern int fcn(int a,int b); -- Global: fcn
```

ANSI-standard declarations. In implementing High C declarations we chose to follow the ANSI standard requirement for declarations that there be exactly one defining declaration of each variable, rather than support the UNIX practice of allowing multiple defining declarations. This ANSI requirement is followed by other popular C compilers for the 8086.

However, since forms N and C do not require a single defining declaration, they can be used to support UNIX practice. The advantage of forms N and C is that they require only a single textual declaration of a declared variable, which can be placed in a file and included by each user of the variable:

File main.c:
```
#include "d.h"
main () {
  v = 1;
  s = "%d";
  sub();
  }
```

File sub.c:
```
#include "d.h"
sub () {
  printf(s,v);
  }
```

File d.h:
```
pragma Data(Common);
int v;
char *s;
pragma Data;
```

The disadvantage of this scheme arises only when a shared variable is to be initialized. If "`int v`" were replaced with "`int v = 1`", for example, the compilation of either main.c or sub.c produces an object module that specifies the initialization of `v` to `1`. Now if later the initial value of `v` should instead be `2`, both "`sub.c`" and "`main.c`" need recompilation. If both are not recompiled, one would specify initialization to `1` and the other to `2`. The linker may arbitrarily choose which initialization to obey, with no warning message. By contrast, were the strict ANSI C conventions obeyed, only the exporter of `v` could initialize it, so that only a single module need be recompiled should the initial value change.

In short, the trade-off is between convenience and reliability. As usual, we chose reliability through discipline over convenience that can lead to bugs that are difficult to find later.

14.9.3 Professional Pascal Naming Conventions

In Professional Pascal every resource has an applicable aliasing mechanism that specifies the computation of its external name, if it has one. These mechanisms are the `Alias` pragma, the `Routine_aliasing_convention`, the `Data_aliasing_convention`, and the `Global_aliasing_convention`. All but the last are explained in the Professional Pascal Language Extensions manual, Section *Externals,* Subsection *Aliasing Pragmas for Interface Packages.*

The `Global_aliasing_convention` applies everywhere the other mechanisms do not. Under MS-DOS this convention specifies no transformation of the name, but that default can be configured into the compiler should the user want to change it. See Section *Compiler Controls,* Subsection *Configuring the Compiler.*

All identifiers in Pascal are converted to lower case, despite the use of any aliasing mechanism: Pascal is not a case-sensitive language. However when the `Alias` pragma is used to specify a string literal L as the external name, the internal lower-casing is irrelevant since exactly L is used as the external name of the resource. However the aliasing conventions derive the external from the internal, lower-cased name.

In addition the `Data` pragma can be used to group declarations in a segment such that only the segment's name is external. Then there are no external names for the variables and the aliasing mechanisms discussed above do not apply.

The table below runs the gamut of Professional Pascal declarations and gives the corresponding applicable aliasing mechanism and external Names. The Mechanism column indicates the aliasing mechanism that applies: Routine, Data, Global, or Alias for `Routine_aliasing_convention`, `Data_aliasing_convention`, `Global_aliasing_convention`; `Alias` pragma; and None for cases where the resources themselves have no external names. The listed external names are based on

the default values for the Routine_ and Data_aliasing_conventions of '%r@%P' and '%r@%P' respectively, and the default for Global_ aliasing_convention of '%r'.

Declaration ---------- Mechanism: Names

Case E: Exporting data:

```
pragma Data(Export);
var    I,J: Integer; ---------- Global: i,j
pragma Data;
```

or, equivalently,

```
pragma Data(Export);
var    X,Y: Integer; ---------- Alias: i,j
pragma Alias(X,'i');
pragma Alias(Y,'j');
-- (No further examples of Alias are given.)
pragma Data;
```

Case I: Importing data:

```
pragma Data(Import);
var    I,J: Integer; ---------- Global: i,j
pragma Data;
```

Case C: Unnamed common data:

```
pragma Data(Common);
var    I,J: Integer; ---------- Global: i,j
pragma Data;
```

Case N: Named common data:

```
pragma Data(Common,'?block');
var    I,J: Integer; ---------- None
pragma Data;
```

(*Note:* "?block" Names the segment into which i and j are placed. Any user-defined string is acceptable there: "?block" was chosen to match what Professional Pascal form as the external name of the package below:)

or, equivalently, in an interface package,

```
package  Block;
   var   I,J: Integer; --------- None
   -- But note use of Data_aliasing_convention:
   pragma Data(Common);
   var   X,Y: Integer; Data: x@block, y@block
   pragma Data;
   end;-- Block;
```

(*Note:* Enclosing declarations in a Professional Pascal interface package declaration, as above, is equivalent to using form N where the common segment name is the character "?" followed by the package name.)

Case FE/FI: Exporting/Importing a routine, before the `program` *header, using an interface package:*

```
package  Block;
   function Fcn(A,B:Integer):Integer;
            External; -------- Routine:  fcn@block
   end;--Block;
```

or, using a specific Routine_aliasing_convention:

```
package  Block;
   pragma   Routine_aliasing_convention('%r');
   function Fcn(A,B:Integer):Integer;
            External; -------- Routine:  fcn
   end;--Block;
```

Case FE: Exporting a routine, after the `program` *header:*

```
function Fcn(A,B:Integer):Integer; External;
function Fcn(A,B:Integer);Integer;
         ------------------- Global:  fcn
   begin
   Return(A+B);
   end;
```

Case FI: Importing a function, after the `program` *header:*

```
function Fcn(A,B:Integer):Integer;
         External; ---------- Global:  fcn
```

14.9.4 Assembly Language Naming Conventions

Here we assume usage of the Digital Research RASM-86 assembler. For detailed information on RASM-86 consult the **Programmer's Utilities Guide for the CP/M-86 Family of Operating Systems and Concurrent DOS-286** by Digital Research, Inc.

There are two classes of external names: (a) segment, group, and segment-class names and (b) names referenced in assembly **public** and **extrn** statements, such as the names of variables and subroutines.

Due to limitations of Microsoft software, and for staying compatible with MS-DOS, the names in class (a) must be presented in *upper case* in assembly programs when communicating with High C or Professional Pascal programs. The names in class (b) must agree exactly in case with any other program sharing the name, if any. In C, names are typically in lower case and so class (b) names must be all lower case in assembly programs. In Professional Pascal, all names are lower-cased by the compiler so they must appear in lower case in assembly programs.

RASM-86 normally upper-cases all names. To avoid this, run RASM-86 with the "$nc" option. This preserves the case of the names as presented in the program:

```
rasm sample.a86 $nc
```

Not only does the table below give the proper declarations for communication of resources between assembly language and Pascal and C, but sample assembly language usage of the communicated resources are provided.

The illustrations assume the use of the High C compiler or Professional Pascal compiler and do not necessarily reflect conventions used by other Pascal or C compilers for MS-DOS.

Case E: Exporting data:

```
        public  i,j
DUMMY   dseg
I       rw      1
J       rw      1

; If not large-data memory model:
DGROUP  group   DUMMY

; If large-data memory model, nothing to add.
```

Case I: Importing data:

```
; If not large-data memory model:
DUMMY   dseg
        extrn   i:word,j:word

DGROUP  group   DUMMY
        mov     ax,i
        add     ax,j

; If large-data memory model:
        extrn   i:word,j:word
; Do not enclose the above line in a data segment.
        mov     ax,seg i
        mov     es,ax
        mov     ax,word ptr es:0
        mov     bx,seg j
        mov     es,bx
        add     ax,word ptr es:0
```

Case C: Unnamed common data:

```
I       dseg    common
ivar    rw      1

J       dseg    common
jvar    rw      1

; If not large-data memory model:
DGROUP  group   i,j
        mov     ax,ivar     ; References i.
        add     ax,jvar     ; References j.
```

```
; If large-data memory model:
        mov     ax,I
        mov     es,ax
        mov     ax,es:ivar ; References I.
        mov     bx,J
        mov     es,bx
        add     ax,es:jvar ; References J.
```

Case N: Named common data:

```
?BLOCK  dseg    common
i       rw      1
j       rw      1

; If not large-data memory model:
DGROUP  group   ?BLOCK
        mov     ax,i
        add     ax,j

; If large-data memory model:
        mov     ax,?BLOCK
        mov     es,ax
        mov     ax,es:i
        add     ax,es:j
```

Case FE: Exporting a subroutine:

Small-code case: (Small or Compact memory model.)

```
        extrn   SMALL?:word   ; Or COMPACT?.
CODESEG cseg    word
; 'CODESEG' is a name chosen freely.
CGROUP  group   CODESEG

        public  fcn
        db      'fcn',3 ; Used by StackDump.
fcn:    push    bp
        mov     bp,sp
        sub     sp,stack_frame_length: If necessary.
        ⟨Body of fcn goes here⟩
        ; Formal parameters start at [bp+4].
        mov     sp,bp
        pop     bp

        . . .
```

```
        ; Choose one of the next two lines depending upon
        ; whether called from Pascal or C.
Pascal: ret     n       ;Number of bytes in arguments.
C:      ret             ;Callee does not pop arguments.
        end
```

Large-code case: (Medium, Big, or Large memory model.)

```
        extrn   MEDIUM?:word    ; Or BIG?, or LARGE?.
CODESEG cseg    word
; 'CODESEG' is a name chosen freely.
; Next line, if present, is used by StackDump.
        db      7,'codeseg',7
                                ; 'codeseg' is 7 letters.
        public  fcn
        db      'fcn',3;Used by StackDump if present.
fcn:    push    bp
        mov     bp,sp
        sub     sp, stack_frame_length ; If necessary.
        <Body of fcn goes here>
        ; Formal parameters start at [bp+6].
        mov     sp,bp
        pop     bp
; Choose one of the next two lines depending
; upon whether called from Pascal or C.
Pascal: retf    n       ;Number of bytes in arguments.
C:      retf            ;Callee does not pop arguments.
        end
```

Case FI: Importing a routine:

Small-code case: (Small or Compact memory model.)

```
        extrn   SMALL?:word     ; Or COMPACT?.
; Do not enclose the next line within a code segment.
        extrn   fcn:near

CODESEG cseg    word
; 'CODESEG' is a name chosen freely.
CGROUP  group   CODESEG

;     function  fcn(I, J: Integer):Integer; External;
;or  extern int fcn(int I, int J);
        ...
```

```
        ;       fcn(5,10)  ; Called:
        mov     ax,10      ; or "push 10" on a 80286.
        push    ax
        mov     ax,5       ; or "push 5"  on a 80286.
        push    ax
        call    fcn
; Add the next line only if calling a C function:
        add     sp,4       ; Pop parameters.
        ...
```

Large-code case: (Medium, Big, or Large memory model.)

```
        extrn   MEDIUM?:word  ; Or BIG?, or LARGE?.
; Do not enclose the next line within a code segment.
        extrn   fcn:far

;    function   fcn(I,J: Integer):Integer; External;
;or  extern int fcn(int I, int J);
        ...
        ; fcn(5,10) called:
        mov     ax,10      ; or "push 10" on an 80286.
        push    ax
        mov     ax,5       ; or "push 5"  on an 80286.
        push    ax
        callf   fcn
; Add the next line only if calling a C function:
        add     sp,4       ; Pop parameters.
        ...
```

15
Utility Packages

Supplied with each High C compiler are the standard C library ".h" header files that specify the interfaces to collections of functions, macros, constants, and variables, along with their corresponding object modules. For a specification of the contents of those header files see the **MetaWare High C Library Reference Manual**. In this section we present additional library facilities that are non-standard in that they are unrelated to the ANSI-standard specification of the C run-time library.

15.1 Utility Packages: ".CF" Interface Files

Supplied with each High C compiler is a collection of "interface" text files, called "packages", that may be included in a compilation to get access to certain functions supplied in the Run-Time Library. The files have the extension .CF for C interFace. The names of the files and a brief description of their contents are given below in alphabetical order.

The interface files may change, and new ones may be added, as updates and new releases are installed. Thus, although the information presented here may be considered reliable, it is necessarily general, so the programmer should consult the interface file actually distributed for possible new information or modifications and for further detailed documentation.

Embedded applications. The utility packages run under all MS-DOS systems (Release 2.0 or greater) without modification. Most of the packages may also be ported to a non-MS-DOS system, i.e. an "embedded application", straightforwardly. A note concerning the adaptation of each relevant package to an embedded application is provided following the package's

description below. Also consult the Section *Embedded Applications.*

15.2 DEBUGAIDS -- Run-Time Debugging Aids

```
void stackdump(int F);
```

"stackdump(F)" produces a call-chain stack dump of the currently active functions on file F: 1 indicates standard output and 2, standard error. Other run-time debugging functions may be added in the future.

To obtain the stack dump facility, file STKDMP.OBJ must be specifically linked in; see Section *Linking a Compiled Program.*

Embedded application note. This package relies on the STDIO.H package to perform all I/O operations. STDIO.H, in turn, relies on the SYSTEM package to perform low-level I/O. Thus, if the SYSTEM package is implemented appropriately, this package should work without modification.

Not relevant to Concurrent

15.4 LANGUAGE – Calling Conventions for C, Pascal, PL/M

This package provides the typical calling conventions for various languages such as C, Pascal, and PL/M. See Section *Externals,* for a discussion of the `Calling_convention` pragma.

15.5 LINETERM -- Line Terminator Convention

```
struct {short Length; char S[2];}
     LTconv_in,LTconv_out;
```

This package provides access to the line terminator definitions used by the High C Run-Time Library. The line terminators for both input and output are specified, and may be dynamically altered by the programmer. This allows conversion from one line terminator convention to another by setting the desired line terminator for input and output and simply using High C I/O or calling the STDIO.H functions to read and write files.

For example the UNIX line terminator convention is a single character: LF. Under MS-DOS CR,LF is required. UNIX files could be converted to MS-DOS files by setting the input line terminator to LineFeed (LF) and the output line terminator to CR,LF and executing roughly the following code:

```
LineTerm.LTConv_in.Length = 1;
LineTerm.LTConv_in.C[0] = 10; /* Input ends with LF.*/
while (gets(S) != 0) puts(S);
```

15.6 SORTS -- Sorting Algorithms

```
/* Quicksort items L..H given a < relation and  */
/*                             a swap procedure: */
void  qsort(
      unsigned L, unsigned H,
      int  Less_than(unsigned,unsigned)!,
      void Swap     (unsigned,unsigned)!
      );
```

This package is to contain sorting algorithms. For now it contains just one: a generic quick sort algorithm that sorts anything.

15.7 STATUS -- Values for "errno"

This package provides a list of all the possible values for the standard library variable "errno".

15.8 SYSTEM -- Operating System Services

```
-- The interface package is over 100 lines long;
-- see the distributed file: SYSTEM.CF.
```

This package provides access to low-level I/O functions and other system-dependent services.

Embedded application note. This package is referenced from various sources throughout the Run-Time Library. It must be implemented if any I/O function or heap-management function is referenced directly or indirectly.

15.9 CCDOSIF -- Access to Concurrent Functions

This package provides a method of invoking all of the over 50 Concurrent services documented in Digital Research's **Concurrent DOS Programmer's Guide**. The compiler distribution additionally contains the source to the implementation of this package — one .C file per function.

Embedded application note. This package is used in the implementation of the System package.

16
Embedded Applications

This section explains how to modify the source code of various MS-DOS-dependent modules of the High C 8086 Run-Time Libraries for use within non-MS-DOS environments.

Unless otherwise indicated below, the object version of each function is aliased to the source name preceded by "_mw" to avoid conflict with standard C library and likely user-coined names. For example, INIT is known externally, i.e. to the linker, as _mwINIT.

The reader should also consult Section *Utility Packages* for additional embedded application considerations for each of the packages.

16.1 MS-DOS-Dependent Modules

[INIT, TERM, EXIT, SYSTEM, INTRUP, ALLOC, CONSOLE]

Few modules in the Run-Time Libraries are directly dependent upon MS-DOS. Some dependent modules are esoteric to MS-DOS, e.g. ENV, and are not referenced elsewhere in the Run-Time Libraries; such modules are not dealt with here.

The initialization module (INIT) must be implemented on any non-MS-DOS system to establish the run-time environment. Five other modules must be ported if any significant portion of any Run-Time Library is used by the application. These five modules are:

TERM	terminates the environment by closing files, etc.
EXIT	provides the C functions exit, _exit, and abort.
SYSTEM	provides low-level system services.
INTRUP	provides interrupt handling.
ALLOC	allocates and frees memory in 1K increments.
CONSOLE	provides input/output from a console.

The source of the Concurrent implementations of these modules is provided with the distribution. The reader should study these files and use them as a guide. Non-assembly source is written in either MetaWare's Professional Pascal ™ or High C language; therefore a user who wishes to modify the source may need both compilers.

16.2 INIT -- Environment Initialization

[argc, argv, and I/O initialization; main program termination]

INIT is an assembly language module that contains the function INIT which is the initial entry point of each linked program. INIT establishes the environment in which High C programs are to run. More specifically, it:

- sets up the stack,
- initializes public variables as required,
- initializes the 8087 or 80287 floating-point processor,
- calls a function to set up interrupt vectors,
- calls a function to initialize I/O for High C,
- calls the main program,
- calls a function to close any open files, and
- terminates the program.

Minimum environment. The MS-DOS version of this module is much more involved than is typically required for an embedded application. The stack is placed at the high end of memory and is allowed to grow downward until it collides with the heap. For an embedded application, a fixed-size stack may be adequate. The module provides an option to use a fixed-size stack by setting the variable Stack_size.

If the application is completely independent of Library support, INIT need only set up a stack, invoke the main program, and then terminate.

I/O initialization. If the High C STDIO module is used, the function "cfinit" must be invoked as part of the initialization process to set up various control blocks and buffers. The

function "cfterm" closes all open files and frees up buffers; this function is called from the TERM module.

Main program. The name of the main program of any linked program is main.

argc, argv support. If "argc" and "argv" are to be used, the public variables "arglen" and "argp" must be initialized and the function "set_up_args" within INIT must be called prior to calling "main". "arglen" is a word that is set to the length of the parameter string. "argp" is set to the address of the first byte of the parameter string.

Program termination. The MS-DOS version of INIT calls a function named "endd" (alias "_mwend") to terminate a program. "endd" is in the TERM module; see the next subsection.

This function closes open files, calls a function to restore interrupt vectors, and then invokes the "dos_exit" function to terminate the process. endd is also called by the exit library function. "dos_exit", which is called by "endd", is in the SYSTEM module.

Line terminator convention. The INIT module initializes the common block that establishes the default line-terminating convention for the host system. See Subsection *LINETERM* in Section *Utility Packages.*

Under MS-DOS, lines are terminated with the two characters CR (13) and LF (10).

16.3 TERM – Environment Termination

The TERM module implements the Pascal "halt" library routine, called by the C "exit" function, and the program termination function "endd" (alias "_mwend") which is invoked from INIT, at least in the MS-DOS version.

"halt" simply calls "endd" but contains code to prevent infinite recursion if a severe error should occur while "endd" is active.

"endd" calls "cfterm" to close all open files, restores interrupt vectors to their prior condition, and then calls "dos_exit" to terminate. "dos_exit" is in the SYSTEM module.

16.4 EXIT – Functions "exit", "_exit", and "abort"

The EXIT module implements the C functions "exit", "_exit", and "abort". "exit" is implemented by calling the Professional Pascal ™ routine "halt". "_exit" calls a routine to restore interrupts and calls "dos_exit" (in SYSTEM), avoiding a call to "cfterm" to terminate the file system. "abort" is a synonym for "_exit(-1)", which produces a return code of 255 under MS-DOS since return codes are limited to 0..255.

In simpler environments, the TERM module can be eliminated by implementing "exit", "_exit", and "abort" in the INIT module and back-substituting the call to "endd".

16.5 SYSTEM – System Services

[source files, I/O model, file-system-less; close, create, c_create, c_create_text; dos_exit; fileclass; lseek, lseek_; open, c_open, c_open_text; read, write, write_; c_unlink]

The SYSTEM module typically takes the most work in porting to another system. Of course only those functions in SYSTEM that are explicitly referenced elsewhere in the Run-Time Libraries need be ported. Details of only those functions are described here.

Source files. The implementation of the SYSTEM module under MS-DOS occupies seven source files: SYSTEM1.P through SYSTEM7.P.

I/O model. The SYSTEM module models I/O in the same way MS-DOS and UNIX do: files are treated as unformatted streams of bytes. Open files are referenced through so-called file handles. A *file handle* is a small integer value within the range zero to MaxFiles, where MaxFiles+1 is the maximum number of files that can be opened at one time. An arbitrary number of

bytes may be read from or written to a file with the read and write functions.

I/O is device independent. Devices are accessed in exactly the same way as disk files.

No file system. Many embedded applications do not support a file system. In such cases, I/O functions within the module can be made to access ports. The "file name" can be used to designate a port address when an open or create operation is performed. Alternatively, file handles can be permanently assigned to specific ports.

Standard input/output. File handles 0, 1, and 2 are pre-initialized to *standard input*, *standard output*, and *standard error output*, respectively. The STDIO module relies on this convention in supplying the FILE* variables `stdin`, `stdout`, and `stderr`.

Function descriptions. Described below is each function of the SYSTEM module that is referenced elsewhere by other modules in the Run-Time Libraries. No other function in the SYSTEM module need be implemented unless the user's program has an explicit reference to it.

`c_close(F)`

Closes the file associated with file handle F. The value of F may then be subsequently reassigned in a call to "c_open" or "c_create".

`c_create (Name,Mode)`
`c_create_text(Name,Mode)`

Creates the file named Name if it does not exist, otherwise the file is truncated to zero bytes. Mode is the set of attributes to be assigned to the file; the interpretation of Mode is system dependent. If Mode is the empty set, a default mode is used that is appropriate for the host system. A file handle is returned that may be used to reference the file.

"c_create_text" is called to create an file ASCII-formatted file. On systems that make no distinction between ASCII

and non-ASCII files these two functions can be considered synonymous.

`dos_exit(RC)`

Terminates the program with a return code RC. This function is referenced only from the TERM module and need not be implemented if the application has an alternate way of terminating a program.

`fileclass(F)`

Returns the type of file associated with file handle F: Console_input, Console_output, or Disk_file.

This function is called by the STDIO module to determine whether a file should be buffered. I/O to the console is unbuffered so that prompting is possible.

```
get_date(Day,Month,Year)
get_time(Hrs,Mins,Secs)
clock()
```

These are used by the functions implementing the standard C library functions in header file "time.h". "get_date" returns 1-31 in Day, 1-12 in Month, and 1980-2099 in Year. "get_time" returns elapsed wall-clock time: 0-23 in Hrs and 0-59 in Mins and Secs. "clock" returns elapsed wall-clock time in hundredths of a second.

```
lseek (F,Loc,Method)
lseek_(F,Loc,Method)
```

"lseek" repositions the *file pointer* associated with file handle F so that the next read or write operates on the n-th byte of the file (origined at zero) where n is computed as follows:

```
n = Loc                        if Method = From_beginning
n = Loc+"current position"     if Method = From_current
n = Loc+"file size"            if Method = From_end
```

"lseek_" performs the same function as "lseek" but "lseek_" also returns the new file pointer position.

"lseek/lseek_" need not be implemented unless random file access is performed via the STDIO function "fseek".

```
c_open      (Name,Method)
c_open_text(Name,Method)
```

Opens the file named Name for input, output, or updating. If the file is not found, "errno" is set to Error_file_not_found; otherwise a file handle is returned that may be used to reference the open file. The "c_open_text" function is used to open a file that is to be interpreted as an ASCII file with embedded end-of-line characters. On systems that make no distinction between ASCII and non-ASCII files these two functions can be considered synonymous.

Text files need not be supported for updating.

```
read(F,BufP,Cnt)
```

Reads Cnt bytes from the file or device associated with file handle F into the buffer with address BufP. Returns the number of bytes actually read, which may be less than Cnt if fewer than Cnt bytes remain in the file or if reading from a line-at-a-time device, e.g. a keyboard.

When end-of-file is encountered, "read" returns zero.

```
write (F,BufP,Cnt)
write_(F,BufP,Cnt)
```

Writes Cnt bytes to the file or device associated with file handle F from the buffer with address BufP. The procedure "write" sets "errno" to "error_write_failed" if fewer than Cnt bytes were written (often caused by a disk overflow). The function "write_" returns the number of bytes actually written and does not indicate an error condition if fewer than Cnt bytes are written.

```
c_unlink(Name)
```

Deletes the file named Name. This function is called when temporary High C files are closed.

16.6 INTRUP – Interrupt Handling

Not relevant to Concurrent

16.7 INTERRUPTS – Generalized Interrupt Handling

Not relevant to Concurrent

16.8 ALLOC – Memory Allocator

```
[sysalloc, sysfree, allocated, least_free_memory]
```

The ALLOC module serves as the foundation of the heap manager (HEAP module). It organizes available memory into an array of 1024-byte pages. There follows a description of each function of the module:

sysalloc(Len)

Allocates as many consecutive free pages as necessary to accommodate Len bytes. The address of the allocated memory is returned. (The heap manager always requests an integral number of pages).

sysfree(A, Len)

Frees the allocated page(s) addressed by A. Len rounded up to a multiple of 1024 determines how many pages are to be freed. Memory not allocated by sysalloc may not be freed.

allocated(A, Len)

Is periodically called by the heap manager when heap-integrity checking is On. It returns 1 (TRUE) if the pages indicated by address A and byte-length Len are allocated; 0 (FALSE) otherwise.

If heap-integrity checking is not important, this function can be implemented to unconditionally return 1 (TRUE).

`least_free_memory()`

Returns the the least number of bytes available in memory thus far. This function need not be implemented because it is not used by the Run-Time Libraries.

16.9 CONSOLE -- Console Input/Output

[console gets, puts, newline]

The CONSOLE module reads strings from the keyboard and writes strings to the screen.

Source files. The CONSOLE module under MS-DOS comprises two source files: KB1.P and KB2.P. For historical reasons the names are not CONSOLE1.P and CONSOLE2.P.

By `string S` below we mean a Professional Pascal string: two bytes of length followed by "length" characters. Such a string is passed by address to "gets" and "puts".

`gets(string S)` reads a line from the keyboard and stores it in string S. If the line is too long to fit in S, the result is unpredictable. This is in accordance with the standard C gets function which is inherently dangerous in this respect.

`puts(string S)` writes the string S to the screen, without terminating the line.

`newline()` writes out the necessary characters to the screen to terminate the current line.

Under MS-DOS, file handle two is always directed to the screen for output. Thus "puts" and "newline" simply write to file handle two via the SYSTEM function "write".

Since standard input is not necessarily from the keyboard, the first call to "gets" opens the keyboard for input and invokes "read" from the SYSTEM module. The terminating CR,LF is then stripped from the string read in.

17
Listings

17.1 Pragmas Page(n), Skip(n), and Title(T)

To cause n page ejects at some point in the listing, insert:

```
pragma Page(n);  -- where n is the number of ejects.
```

To cause n lines to be blank at some point in the listing, insert:

```
pragma Skip(n);  -- where n is the number of blanks.
```

To cause a title T to appear at the top of each successive page, place the following pragma in the source:

```
pragma Title(T);  -- where T is a string constant.
```

Each successive Title pragma changes the title for the next pages. The only way to title the first page is to place the Title pragma in the profile, and not use the -on List command-line option (/list command qualifier on VMS) but to use "pragma On(List);" at the end of the profile or wherever the listing should start in the source file; the command line form causes the listing to start too early. See Section *Compiler Controls.*

17.2 Format of Listings

```
[listing ruler, line-numbers, scope-level, nesting-level; include file]
```

Ruler. The first line after any header and title lines on each page is a "ruler" that defines three fields for each line. The fields are for: (1) three level numbers, (2) the line number, and (3) the line contents. The ruler is as follows:

```
Levels LINE #  |----+----1----+----2----+----3----+----4----+----5----+----6
```

Level numbers can be used to find a missing } or comment terminator when a message such as "Unexpected end-

of-file." is produced by the compiler. All three level numbers are initially zero, but they are printed as blank rather than "0".

The first level number indicates the scope nesting level for declarations. It is incremented at entry to each function, `struct`, or `union` declaration. It is decremented at the corresponding end-of-construct.

The second level number indicates the statement nesting level. It is incremented at each { and decremented at the corresponding }.

The third level number indicates the structure initialization nesting level. It is incremented at each { and decremented at the corresponding }.

Include files. A first-level include file named `File_name` is indicated as starting after a line containing "`+(File_name`" in the line number field, and ending just before a matching "`+)File_name`" line. The included lines have "+" in the leftmost column of the line-number field, and they are numbered independently of the main source file.

An included file inside an include file has an extra "+" on each of its lines for each level of inclusion, except that the line numbers take precedence over "+"s in the line-number field if and when the "+"s would otherwise intrude into the line number field.

The profile, if any, is listed as an include file when a listing has been requested.

Example. Since in this area a picture is worth a thousand words, we present a sample program listing on the next two pages, enhanced with boldface reserved words and followed by the optional (pseudo-)assembly listing requested by `-asm` (on VMS: `/machine_code`) on the following compile command line:

```
hc queens.c -on List -asm                    (OR:)
hc queens.c /list /machine_code              (VMS form.)
```

```
MetaWare High C Compiler, V. 1.2   27-Jun-85 08:51:20   queens.c    Page 1

Copyright (C) 1983-85 MetaWare Incorporated.  Serial 0-000000 INTERNAL ONLY
Target processor: i8086/88/186/286       (Code generator 2.5)
Scanner C_lexicon(15-May-85 07:29:16)
Parser  C_phrase_structure(22-Jun-85 14:22:54)
=====>  Profile hc.pro included in this compilation.

Levels  LINE #  |----+----1----+----2----+----3----+----4----+----5----+----
        +(hc.pro
        +     1 |pragma Off(Emit_names);
        +     2 |pragma Off(Check_stack);
        +     3 |pragma Off(Emit_line_table);
        +)hc.pro
              1 |/* From Wirth's Algorithms+Data Structures = Programs.      */
              2 |/* This program is suitable for a code-generation benchmark,  */
              3 |/* especially given common sub-expressions in array indexing.  */
              4 |/* See the Programmer's Guide for how to get a machine code    */
              5 |/* interlisting.                                              */
              6 |
              7 |pragma  Title("Eight Queens problem.");
              8 |
              9 |typedef enum{False,True} Boolean;
             10 |typedef int Integer;
             11 |
             12 |#define Asub(I)  A[(I)-1] /* C's restriction that array  *
             13 |#define Bsub(I)  B[(I)-2] /* indices start at zero       *
             14 |#define Csub(I)  C[(I)+7] /* prompts definition of       *
             15 |#define Xsub(I)  X[(I)-1] /* macros to do subscripting.  *
             16 |                           /* Pascal equivalents:        *
             17 |Boolean A[ 8 /* 1.. 8 */]; /* A:array[ 1.. 8] of Boolean *
             18 |Boolean B[15 /* 2..16 */]; /* B:array[ 2..16] of Boolean *
             19 |Boolean C[15 /*-7.. 7 */]; /* C:array[-7.. 7] of Boolean *
             20 |Integer X[ 8 /* 1.. 8 */]; /* X:array[ 1.. 8] of Integer *
             21 |
             22 |void Try(Integer I, Boolean *Q) {
1 1          23 |   Integer J = 0;
1 1          24 |   do {
2 2          25 |      J++; *Q = False;
2 2          26 |      if (Asub(J) && Bsub(I+J) && Csub(I-J)) {
3 3          27 |         Xsub(I) = J;
3 3          28 |         Asub(J) = False; Bsub(I+J) = False; Csub(I-J) = False;
3 3          29 |         if (I < 8) {
4 4          30 |            Try(I+1,Q);
4 4          31 |            if (!*Q) {
5 5          32 |               Asub(J) = True; Bsub(I+J) = True; Csub(I-J) = True;
5 5          33 |               }
4 4          34 |            }
3 3          35 |         else *Q = True;
3 3          36 |         }
2 2          37 |      }
1 1          38 |   while (!(*Q || J==8));
1 1          39 |   }
             40 |pragma Page(1); /* Page eject requested. */
```

```
MetaWare High C Compiler, V. 1.2   27-Jun-85 08:51:20   queens.c    Page 2

                Eight Queens problem.
Levels  LINE #  |----+----1----+----2----+----3----+----4----+----5----+----6
              41 |void main () {
1 1           42 |   Integer I; Boolean Q;
1 1           43 |   printf("%s\n","go");
1 1           44 |   for (I =  1; I <=  8; Asub(I++) = True);
1 1           45 |   for (I =  2; I <= 16; Bsub(I++) = True);
1 1           46 |   for (I = -7; I <=  7; Csub(I++) = True);
1 1           47 |   Try(1,&Q);
1 1           48 |pragma Skip(3); /* Skip three lines. */



1 1           49 |   if (Q)
1 1           50 |      for (I = 1; I <= 8;) {
2 2           51 |         printf("%4d",Xsub(I++));
2 2           52 |         }
1 1           53 |   printf("\n");
1 1           54 |   }

w L43/C4:        printf: Routine called but not defined.
```

If the -asm option (/machine_code qualifier on VMS) is specified, the source-annotated assembly listing on the next few pages is produced (the page boundaries have been adjusted to fit the present page sizes).

```
MetaWare High C Compiler, V. 1.2   27-Jun-85 08:51:20   queens.c    Page 3
                Eight Queens problem.
Addr  Object          Source Program and Assembly Listing
                              extrn   mw_INIT,printf
                      MODEL   segment 'DATA'
0000  53                     db      83
          ;/* From Wirth's Algorithms+Data Structures = Programs.   */
          ;/* This program is suitable for a code-generation benchmark, */
          ;/* especially given common sub-expressions in array indexing.*/
          ;/* See the Programmer's Guide for how to get a machine code  */
          ;/* interlisting.                                             */
          ;pragma Title("Eight Queens problem.");
          ;typedef enum{False,True} Boolean;
          ;typedef int Integer;
          ;#define Asub(I)  A[(I)-1]     /* C's restriction that array  */
          ;#define Bsub(I)  B[(I)-2]     /* indices start at zero       */
          ;#define Csub(I)  C[(I)+7]     /* prompts definition of       */
          ;#define Xsub(I)  X[(I)-1]     /* macros to do subscripting.  */
          ;                              /* Pascal equivalents:         */
          ;Boolean A[ 8 /* 1.. 8 */];    /* A:array[ 1.. 8] of Boolean */
          ;Boolean B[15 /* 2..16 */];    /* B:array[ 2..16] of Boolean */
          ;Boolean C[15 /*-7.. 7 */];    /* C:array[-7.. 7] of Boolean */
          ;Integer X[ 8 /* 1.. 8 */];    /* X:array[ 1.. 8] of Integer */
          ;void Try(Integer I, Boolean *Q) {
                      MODEL   ends
                      QUEENS  segment 'CODE'
0000                  .L0000:
                              public  Try
                      Try     proc near
0000  55                      push    bp
0001  8b ec                   mov     bp,sp
0003  83 ec 06                sub     sp,6
                      ;   Integer J = 0;
0006  c7 46 fe 0000           mov     word ptr -2[bp],0
                      ;   do {
                      ;      J++; *Q = False;
000b                  .L000b:
000b  ff 46 fe                inc     word ptr -2[bp]
000e  8b 76 06                mov     si,6[bp]
0011  c6 04 00                mov     byte ptr [si],0
                      ;      if (Asub(J) && Bsub(I+J) && Csub(I-J)) {
0014  8b 5e fe                mov     bx,-2[bp]
0017  82 bf ffffr 00          cmp     byte ptr @QUEENS-1[bx],0
001c  75 03                   jne     0021
001e  e9 0077                 jmp     0098
0021  8b 7e 04                mov     di,4[bp]
0024  03 fb                   add     di,bx
0026  82 bd 0006r 00          cmp     byte ptr @QUEENS+6[di],0
002b  74 6b                   je      0098
002d  8b 46 04                mov     ax,4[bp]
0030  2b c3                   sub     ax,bx
0032  96                      xchg    ax,si
0033  82 bc 001fr 00          cmp     byte ptr @QUEENS+31[si],0
0038  74 5e                   je      0098
                      ;       Xsub(I) = J;
```

```
MetaWare High C Compiler, V. 1.2   27-Jun-85 08:51:20   queens.c    Page 4
                Eight Queens problem.
Addr  Object          Source Program and Assembly Listing
003a  8b 46 04                mov     ax,4[bp]
003d  d1 e0                   shl     ax,1
003f  93                      xchg    ax,bx
0040  8b 46 fe                mov     ax,-2[bp]
0043  89 87 0026r             mov     @QUEENS+38[bx],ax
                      ;        Asub(J)= False; Bsub(I+J)= False; Csub(I-J)= False;
0047  93                      xchg    ax,bx
0048  c6 87 ffffr 00          mov     byte ptr @QUEENS-1[bx],0
004d  c6 85 0006r 00          mov     byte ptr @QUEENS+6[di],0
0052  c6 84 001fr 00          mov     byte ptr @QUEENS+31[si],0
                      ;        if (I < 8) {
0057  8b 46 04                mov     ax,4[bp]
005a  3d 0008                 cmp     ax,8
005d  7d 33                   jge     0092
                      ;            Try(I+1,Q);
005f  ff 76 06                push    word ptr 6[bp]
0062  40                      inc     ax
0063  50                      push    ax
0064  89 76 fc                mov     -4[bp],si
0067  89 7e fa                mov     -6[bp],di
006a  e8 ff93                 call    0000
                      ;            if (!*Q) {
006d  83 c4 04                add     sp,4
0070  8b 76 06                mov     si,6[bp]
0073  82 3c 00                cmp     byte ptr [si],0
0076  75 20                   jne     0098
                      ;                Asub(J)=True;Bsub(I+J)=True;Csub(I-J)=True;
0078  8b 76 fe                mov     si,-2[bp]
007b  c6 84 ffffr 01          mov     byte ptr @QUEENS-1[si],1
0080  8b 76 fa                mov     si,-6[bp]
0083  c6 84 0006r 01          mov     byte ptr @QUEENS+6[si],1
0088  8b 76 fc                mov     si,-4[bp]
008b  c6 84 001fr 01          mov     byte ptr @QUEENS+31[si],1
0090  eb 06                   jmp     0098
                      ;                }
                      ;            }
                      ;         else *Q = True;
0092                  .L0092:
0092  8b 76 06                mov     si,6[bp]
0095  c6 04 01                mov     byte ptr [si],1
                      ;        }
                      ;      }
                      ;   while (!(*Q || J==8));
0098                  .L0098:
0098  8b 76 06                mov     si,6[bp]
009b  82 3c 00                cmp     byte ptr [si],0
009e  75 09                   jne     00a9
00a0  83 7e fe 08             cmp     word ptr -2[bp],8
00a4  74 03                   je      00a9
00a6  e9 ff62                 jmp     000b
00a9                  .L00a9:
00a9  8b e5                   mov     sp,bp
```

```
MetaWare High C Compiler, V. 1.2   27-Jun-85 08:51:20   queens.c    Page 5
                Eight Queens problem.
Addr  Object          Source Program and Assembly Listing
00ab  5d                      pop     bp
00ac  c3                      ret
                      Try     endp
                      ;   }
                      ;pragma Page(1); /* Page eject requested. */
                      ;void main () {
00ad                  .L00ad:
                              public  main
                      main    proc near
00ad  55                      push    bp
00ae  8b ec                   mov     bp,sp
00b0  83 ec 04                sub     sp,4
                      ;   Integer I; Boolean Q;
                      ;   printf("%s\n","go");
00b3  b8 0000r                mov     ax,offset mw_LITERALS
00b6  50                      push    ax
00b7  b8 0004r                mov     ax,offset mw_LITERALS+4
00ba  50                      push    ax
00bb  e8 ----e                call    printf
                      ;   for (I =  1; I <=  8; Asub(I++) = True);
00be  c7 46 fe 0001           mov     word ptr -2[bp],1
00c3  83 c4 04                add     sp,4
00c6                  .L00c6:
00c6  8b 46 fe                mov     ax,-2[bp]
00c9  3d 0008                 cmp     ax,8
00cc  7f 0b                   jg      00d9
00ce  ff 46 fe                inc     word ptr -2[bp]
00d1  96                      xchg    ax,si
00d2  c6 84 ffffr 01          mov     byte ptr @QUEENS-1[si],1
00d7  eb ed                   jmp     00c6
                      ;   for (I =  2; I <= 16; Bsub(I++) = True);
00d9                  .L00d9:
00d9  c7 46 fe 0002           mov     word ptr -2[bp],2
00de                  .L00de:
00de  8b 46 fe                mov     ax,-2[bp]
00e1  3d 0010                 cmp     ax,16
00e4  7f 0b                   jg      00f1
00e6  ff 46 fe                inc     word ptr -2[bp]
00e9  96                      xchg    ax,si
00ea  c6 84 0006r 01          mov     byte ptr @QUEENS+6[si],1
00ef  eb ed                   jmp     00de
                      ;   for (I = -7; I <=  7; Csub(I++) = True);
00f1                  .L00f1:
00f1  c7 46 fe fff9           mov     word ptr -2[bp],-7
00f6                  .L00f6:
00f6  8b 46 fe                mov     ax,-2[bp]
00f9  3d 0007                 cmp     ax,7
00fc  7f 0b                   jg      0109
00fe  ff 46 fe                inc     word ptr -2[bp]
0101  96                      xchg    ax,si
0102  c6 84 001fr 01          mov     byte ptr @QUEENS+31[si],1
0107  eb ed                   jmp     00f6
```

```
MetaWare High C Compiler, V. 1.2   27-Jun-85 08:51:20   queens.c    Page 6
                Eight Queens problem.
Addr  Object          Source Program and Assembly Listing
                      ;   Try(1,&Q);
0109                  .L0109:
0109  8d 46 fd                 lea     ax,-3[bp]
010c  50                        push    ax
010d  ba 0001                   mov     dx,1
0110  52                        push    dx
0111  e8 feec                  call    0000
                      ;pragma Skip(3); /* Skip three lines. */
                      ;   if (Q)
0114  83 c4 04                  add     sp,4
0117  82 7e fd 00               cmp     byte ptr -3[bp],0
011b  74 24                     je      0141
                      ;       for (I = 1; I <= 8;) {
011d  c7 46 fe 0001             mov     word ptr -2[bp],1
0122                  .L0122:
0122  8b 46 fe                 mov     ax,-2[bp]
0125  3d 0008                   cmp     ax,8
0128  7f 17                     jg      0141
                      ;         printf("%4d",Xsub(I++));
012a  ff 46 fe                  inc     word ptr -2[bp]
012d  8b f0                     mov     si,ax
012f  d1 e6                     shl     si,1
0131  ff b4 0026r               push    word ptr @QUEENS+38[si]
0135  b8 0008r                  mov     ax,offset mw_LITERALS+8
0138  50                        push    ax
0139  e8 ----e                 call    printf
013c  83 c4 04                  add     sp,4
013f  eb e1                     jmp     0122
                      ;       }
                      ;   printf("\n");
0141                  .L0141:
0141  b8 000cr                 mov     ax,offset mw_LITERALS+12
0144  50                        push    ax
0145  e8 ----e                 call    printf
0148  8b e5                     mov     sp,bp
014a  5d                        pop     bp
014b  c3                        ret
                      main      endp
                      QUEENS    ends
                      mw_LITERALS       segment public 'DATA'
0000  676f                      db      'go'
0002  00                        db      0
0004                            org     4
0004  25730a                    db      '%s',0aH'
0007  00                        db      0
0008  253464                  db      '%4d'
000b  00                        db      0
000c  0a                        db      0aH'
000d  00                        db      0
000e                  mw_LITERALS       ends

No user errors    1 warning    107K of memory unused.
End of processing, 27-Jun-85 08:51:41                queens.c
```

18
Diagnostic Messages

Messages from the High C compiler report

(a) file I/O errors,
(b) system errors, and
(c) user errors and warnings.

18.1 File I/O Errors

File I/O errors are fatal. They can occur in attempting to open a non-existent file or in writing a compiler output file when not enough disk is available. The errors that are likely to be seen are:

Unable to open file fff: file not found.

This message is produced when any input source file, such as that specified on the compiler invocation line or in an Include pragma, cannot be found. In addition, when running the compiler in ANSI mode, the three files HCANSI.ST, HCANSI.PT, and HCANSIP.PT must be available. The compiler searches various "ipaths" for any input files, so the HCANSI files may be anywhere in the search paths. See Section *Compiler Controls*.

This message is produced twice: it is written once to standard output and once to standard error. If standard output is not redirected, the message appears on the screen twice.

*****Error occurred on writing instruction file: ...**
*****Error occurred on writing object file: ...**

8086 resident compiler. Same as next message below.

Cross compiler to 8086. These messages are followed by a system message explaining the cause of the problem.

***Write error occurred during tree paging.

8086 resident only. Usually caused by too little space on the disk. Remove unnecessary disk files and try again.

NOTE: Fatal errors may result in compiler temporary files being left on the disk. You should remove them. They are files with a ".TMP" suffix.

18.2 System Errors

System errors are fatal and should rarely occur. Their diagnostic messages take the following form:

>>>>> SYSTEM ERROR n <<<<<, in *Module:Function*
Error message text.

where n numbers the occurrences of system errors, *Module* is the module name, and *Function* is the function name. The only system error messages that the user should be concerned with are:

Dynamic array allocation/reallocation failed.

Out of memory.

8086 resident compiler. In both of these cases, there was too little memory to compile the program. Adding more memory to the computer can solve this problem. Alternatively the -tpages and the -cram options can be used to direct the compiler to use less memory; see Section *Compiler Controls.*

Cross compiler to the 8086. In both of these cases, the system failed to supply sufficient virtual memory to compile the program.

Exceeded Card_char_limit.

The input line was too long. Line length is normally limited to 2,000 characters (and to 256 with the -cram option specified: 8086 resident only). Shorten the line.

Recover: Exceeded the following limit: *Limit.*
In repairing a syntax error, a table overflowed. The table limit is fixed, so no increase in memory can improve the situation. Repair the syntax error.

There are many other system error messages that the compiler could produce, but they are associated with internal compiler errors or inconsistencies that should not occur.

Stack dump. Compiler system errors are always accompanied by a call-stack dump. The dump can usually be ignored, but when reporting a problem to the support staff, the history of called functions can be helpful; include a listing of the dump in any written correspondence. The following is a sample dump for the compiler residing on the Intel 8086 processor.

```
Call stack dump:

ROUTINE          AT     IN MODULE WAS CALLED NEAR WITH ACTUAL PARAMETERS
                 51C:071B              1E9C:0292   1C07|9C1E|6407|9C1E|0000
read_parse_tab 1E9C:0259 ptread        1E9C:0669   0100|EA02|80FF|4700|EB1D
                 51C:08E3              1DEB:0047   F6C5|E302|0100|E8FF|2E03
init_analyser  1DEB:001A analdrvr      1D24:032E   0100|A4C5|E302|F6C5|E302
                 51C:04CF              4D8:02CF    F6FF|1103|DB04|0000|0000
doit             4D8:02C2 skel         4D8:0311    0000|0000|2500|C022|5800
main             4D8:0306 skel         22C0:0025   5800|2B25|0000|0000|0000

Error was severe. Program terminated.
```

The "ROUTINE" and "IN MODULE" columns are the same for all host architectures, and in general are all that is relevant when reporting a problem to support personnel. Consult Section *Debugging* for how to interpret such dumps.

System errors due to a bug in the compiler's code generator are accompanied by a line "Code was being generated for program text near Ln/Cm." following the call-stack dump. This helps isolate the program text causing the problem and may facilitate reducing the problem program to a few lines, which then can be easily sent to compiler support personnel.

NOTE: When code generator errors occur, they can frequently be "cured" by inserting a label before the line causing the problem. Even if this cures the problem, please still report the problem to support personnel!

NOTE: Fatal errors may result in compiler temporary files being left on the disk. They should be removed. They are files with a ".TMP" suffix.

18.3 User Errors and Warnings

User error messages are grouped in the three categories: (1) lexical, (2) syntactic, and (3) constraint. Warnings do not terminate compilation; errors always do. Also, some diagnostic messages that are warnings become errors when running the compiler in ANSI mode.

All user diagnostics are accompanied by a line number n and column number m in the form Ln/Cm, and optionally the file containing the text where the error was detected. In addition, lexical and syntactic errors are generally accompanied by the erroneous line with a carat ^ beneath it at the point of error detection. Errors begin with "E" and warnings with "w" and usually occupy a single line.

Lexical error messages are produced when an improperly formed High C word is detected, such as a string with a missing closing quote. *Example:*

```
Levels  LINE #  |----+----1----+----2----+----3----+----4----+----5----+---
              1 |void main() {
1 1           2 |   char *S;
1 1           3 |   S = "Hello;
          C15 ----------------^
E L3/C15:  (lexical)  Unexpected end-of-line encountered.
1 1           4 |   }
```

Syntactic error messages are produced for High C programs that are ill-formed on the phrase level, such as a missing ";" or inserted spurious symbol. The message is accompanied by a statement of the REPAIR that the compiler effected so that it could keep processing input. *Example:*

```
Levels  LINE #  |----+----1----+----2----+----3----+----4----+----5----+---
              1 |void main() {
1 1           2 |   printf "Hello");
          C11 -------------^
1 1           3 |   }

E L2/C11:  (syntactic)  unexpected symbol:'<STRING>':"Hello"
REPAIR:   '(' was inserted before '<STRING>':"Hello"@L2/C11
```

Constraint error and warning messages diagnose more subtle problems, such as an undeclared identifier or type mismatch. There are over 160 such diagnostic messages, each of which is generally meant to be self-explanatory. Most of them prevent the generation of object code, but some are merely warnings and are intended to assist the programmer. Some warnings become errors when compiling in ANSI mode.

Messages that report errors terminate compilation after the phase issuing the diagnostic message, so errors that would otherwise have been detected by later phases are not reported until the earlier error is repaired and the compiler re-invoked.

As examples of these diagnostics:

```
Levels  LINE #  |----+----1----+----2----+----3----+----4----+----5----+---
                1 |void main() {
1 1             2 |   int i;
1 1             3 |   i = Undeclared_identifier;
1 1             4 |   }

E L3/C8:          Undeclared_identifier: This is undeclared.
1 user error      No warnings    453K of memory unused.

Levels  LINE #  |----+----1----+----2----+----3----+----4----+----5----+---
                1 |void main() {
1 1             2 |   int i, Unused;
1 1             3 |   i /= 0;
1 1             4 |   }

w L2/C8:          i: Variable is set but is never referenced.
E L2/C11:         Unused: Variable is never used.
E L3/C6:          Division by zero.
2 user errors     1 warning      457K of memory unused.
```

18.4 Error and Warning Messages, Explanations

The remainder of this section is a collection of all compiler diagnostic messages, presented in alphabetical order. Where appropriate an explanation is given. Often an explanation uses an example of the general case, for simplicity, rather than attempting to explain in detail the general case.

"=" used where "==" may have been intended.
"=" was detected as an operator in a Boolean expression, such as "`if(x = y) ...`". Often this is a mistake, as "`if(x == y)...`" was intended.

"`auto`" must appear within a function.
Storage class `auto` cannot be given for declarations that do not appear within a function.

"`break`" must appear within `while`, `do`, `for`, or `switch`.
"`case`" must appear within a "`switch`".
"`continue`" must appear within `while`, `do`, or `for`.
"`default`" must appear within a "`switch`".

"`pragma Data`" active at end of module.
"`pragma Data`" active at end of function.
A "`pragma Data(...);`" was given in a module or function, with no terminating "`pragma Data;`". This is permitted but the programmer may have forgotten to supply the terminating pragma, thus perhaps including more data declarations in a data segment than intended.

"`register`" is the only allowable storage class for a parameter.
Ignored.
In a function definition or declaration, a storage class other than `register` was given, such as in
`int f(i) static i; {...}.`

"`register`" must appear within a function.
Storage class `register` cannot be given for declarations that do not appear within a function definition.

"`void`" is illegal here.

A bit field is not valid as an argument to &.
One cannot take the address of a bit field, since such a field is not necessarily on a byte boundary.

A bit field is not valid as an argument to `sizeof`.
Since bit fields need not occupy an integral number of bytes, taking their `sizeof` is prohibited.

A function may not return a function (but may return a pointer thereto).

A function may not return an array (but may return a pointer thereto).

A function may not return an incomplete type.

A function cannot return a `struct` or `union` type whose fields have not yet been specified. For example, "`struct s; struct s *f() {...}`" is legal since f returns a *pointer* to an incomplete `struct` type, but "`struct s; struct s g() {...}`" is illegal.

A functionality `typedef` cannot be used in a function definition.

"`typedef int f(); f g {return 3;}`" is illegal: the type definition for f cannot be used to specify that g is a function.

A parameter may not be a function (but may be a pointer thereto).

A parameter name must be given here.

For function definitions, parameter names must be supplied. Thus, for example, "`void f(int, float g) {...}`" is illegal because the first parameter lacks a name.

A register-class function makes no sense.

For example, "`register f() {...}`" is illegal.

An array may not contain functions (but may contain pointers thereto).

An array must have a positive number of elements.

An array of objects of an incomplete type is illegal.

An array cannot contain a `struct` or `union` type whose fields have not yet been specified. For example, "`struct s; struct s *a[10];`" is legal since "a" contains *pointers* to an incomplete `struct` type, but "`struct s; struct s b[10];`" is illegal.

An interrupt function may not be called.

A function with the `_INTERRUPT` calling convention attribute cannot be called directly.

An object of type ttt cannot be initialized.

Argument to "include" must be a string.

Argument type ttt is not compatible with formal parameter type ttt'.

An attempt was made to pass an argument of a wrong type to a function, such as passing a `float` for a parameter that is a `struct`. When using standard C function definitions, this is a warning only, since C permits such mismatches; but when using prototype syntax, it is an error. This warning provides the security of Pascal function call semantics.

Array size exceeds addressability limits.

Bit field is byte-aligned and of the same size as ttt and so is being converted to that type for efficiency.

It so happens that a bit field is aligned on a byte boundary and is the same size as an integral type ttt, so it may as well be declared as such for efficiency.

Bit fields must fit in a register or register pair.

Cannot dereference a pointer to `void`.

Type `*void` was introduced as a means of defining a "generic pointer" compatible with other pointers. But there is no such thing as an object of type `void`. Therefore, dereferencing a pointer to `void` is illegal.

Cannot initialize a `typedef`.

Something like "`typedef int T = 1;`" was attempted.

Cannot initialize an imported variable.

Something like "`extern int T = 1;`" was attempted. A variable may be initialized only by its defining module.

Cannot pass this by address in a small-data model.

Cannot take `sizeof` a function type.

Cannot take `sizeof` an incomplete type.

The `sizeof` a `struct` or `union` type whose fields have not yet been specified is not known. For example, what

follows is illegal since the size of the structure is unknown: "`struct s; ... sizeof(struct s)...`".

Cannot take `sizeof` type `void`.
: There are no objects of type `void`, therefore taking `sizeof void` makes no sense.

Cannot take the address of a `register` variable.

Current calling convention requires pointer parameters only.
: The current calling convention contains "`_BY_REF`", requiring that all parameters be of pointer types.

Declared type is never referenced.

Divide by zero.
: This was detected in a constant expression at compile time.

Enclosing function's return type is "`void`"; therefore nothing may be returned.
: "`return E;`" for some expression E was found in a function whose return type is `void`.

End of file encountered within `#if` construct.

End of file encountered within arguments to a macro. Probably a missing right parenthesis.

End of file encountered within macro definition.

End of file encountered within macro formal parameter list.

Expression has no side effect and has been deleted.
: An expression used in a statement context has no side effect; therefore the expression is useless. For example, "`2+3;`".

External function is never referenced.

Fewer arguments given than function has parameters.

Field offset exceeds addressability limit.
: The size of `structs` can be limited by the target architecture, e.g. 64KB on an 8086.

`for` loop will never execute.

Function called but not defined.

Any function that was called but not defined is noted as a warning. Although such practice is permissible in C, especially useful when calling library functions, a common error is to misspell a function name. The error goes undetected until link-time without this warning. Furthermore, errors in parameter linkage can occur when a call is made to an undefined function. We recommend that the library ".h" header files always be included to get parameter checking, and that function prototypes be used for external function declarations, rather than making use of the "feature" of C for calling undefined functions.

Function expected.

The expression preceding the arguments (...) in a function call must denote a function.

Function parameter names are allowed only on function definitions, not declarations.

"`int f(a,b,c);`" is a function declaration that names the parameters (a,b,c). This is illegal unless function prototype syntax is used for the definition, as in "`int f(int a, int b, int c);`".

Function return value never specified within function.

A function with a non-`void` return type contains no `return` statement. This typically happens with "old" C programs that did not use `void` to indicate that a function returns nothing.

Functions may not be nested.

In ANSI-standard C, functions cannot be declared within functions. In High C they can. This message is produced when the compiler is doing ANSI checking.

Identifier required after `#ifdef` or `#ifndef`.

Identifier required. Pragma ignored.

Incompatible tag reference: The ttt tag class does not match the tag class ttt' defined at Ln/Cm.

Something like "`struct s; union s {int x;};`" was encountered. The tag s cannot simultaneously be the tag for a `struct`, `union`, and/or `enum`.

Incomplete type: the `struct`/`union` type at Ln/Cm must be completed before it can be used here.

A reference has been detected to a field of a `struct` or `union` type whose fields have not yet been specified.

Incorrect number of parameters to macro. Macro invocation ignored.

The number of arguments to a macro must agree exactly with the number of parameters in its `#define`.

Integer constant exceeds largest `signed` number.

Integer constant exceeds largest `unsigned` number.

Invalid calling-convention identifier.

An argument to the Calling_convention pragma must be one of the predefined calling-convention identifiers, such as _CALLEE_POPS_STACK.

Invalid digit in non-decimal number: X.

Local function is never referenced; no code will be generated for it.

A function of storage class `static` is not called anywhere in the compilation unit. Since it is not exported, there can be no reference to the function and it is essentially deleted.

Local stack frame exceeds addressability limit.

The stack frame size for a function exceeds the capabilities of the target architecture, e.g. 64KB for the 8086.

Lower bound of range is greater than upper bound.

This can only happen in High C case statements where range expressions are allowed as labels (an extension).

Macro name must be an identifier.

Macro parameter must be an identifier.

Members cannot be of an incomplete type.

A `struct` or `union` cannot contain a `struct` or `union` type whose fields have not yet been specified. For example, "`struct s; struct t {struct s *p;}`" is legal since p is a *pointer* to an incomplete `struct` type, but "`struct s; struct t {struct s p;}`" is illegal.

Memory model must be Small, Compact, Medium, Big, or Large.

Mismatched `#if-#elif-#else-#endif`.

More arguments given than function has parameters.

Must be a compile- or load-time computable expression.

The initializers for a static variable must be determinable when a program is loaded.

Must be a compile-time computable constant.
Must be a pointer.
Must be a scalar (`int`, `char`, floating, or pointer) type.
Must be a static variable reference.
Must be a string.
Must be a `struct` or `union`.
Must be a type.
Must be an identifier.
Must be an integral (`int` or `char`) type.
Must be of a pointer type.
Must be of an extended-function type.

Named parameter association is prohibited for this function since its declaration near Ln/Cm does not name all parameters.

An attempt was made to call a function F using named parameter association, but F's declaration did not name all of its parameters. For example,

```
void F(int a,float);  ..F(a=>37, 3.3);/*Illegal.*/
void F(int a,float b);..F(a=>37,b=>3.3); /*Fine.*/
```

No "pragma Data" is active.
"pragma Data;" was encountered without a preceding, and matching, "pragma Data(...);".

No member is declared here.
A declaration with no declared object was found within a struct or union. For example,

```
struct s {int; float; struct t {int y};}
```

contains three declarations, none of which declare an object. However, this construct is not entirely vacuous because the declaration of struct t is visible outside of struct s and therefore can be used to declare objects of type struct t.

No object may be of type void.

No parameter declarations may be given here.
In defining a function using prototype syntax, where the parameter types were specified in the parameter list, an attempt was made to re-declare the parameters following the parameter list. For example, "int x,y;" is illegal in "void f(int x, int y) int x,y; { ... }".

Non-decimal constant exceeds largest unsigned number.

Only a parameter may be declared here.
Preceding a function definition's {, only the function's parameters may be declared.

Only fields of type "unsigned int" or "unsigned long int" are supported.
Bit fields may be only of these two types. Any bit field of another type is coerced to one of them, depending upon the size of the bit field.

Only one "default" is permitted in a "switch".

Operand type inappropriate for operator.
An inappropriate operand was detected for a built-in operator such as &, |, ~, etc. For example, "float f1,f2; ... f1 = f1 & f2;" is illegal: & requires integral operands.

Parameter not found or specified more than once.
In a function call using named parameter association, a parameter was named twice, or a non-existent parameter was referenced.

Parameter ppp not supplied.
In a function call using named parameter association, parameter ppp was not given an argument value.

Parameter separator must be a comma.
In a `#define` of a macro with parameters, parameter names must be separated by a comma. For example, "`#define M(a b) c`" is illegal; "`a,b`" is required.

Pointer dereferencing disallowed in static context.

"`pragma Code`" may not occur within a function.
The `Code` pragma must appear only at the outermost declaration level — outside of all functions.

Pragma has too few parameters.

Pragma has too many parameters.

Previous "`pragma Data`" is still active.
"`pragma Data(...);`" was given in the context of an already active "`pragma Data(...)`". Insert "`pragma Data();`" preceding the offending pragma to "turn off" the active pragma.

Real constant has too many digits.

Result of comparison never varies.
An expression was found whose operands, while they are not all constants, are such that the value of the expression is always the same. For example, an expression of type `unsigned int` is always less than zero.

Right operand of shift operator is negative.

Since the first parameter was specified by the type "`void`", there may be no other parameters.
The special syntax exemplified by "`int f(void);`" denotes a function f taking no parameters. Because of

this, no parameter can be specified after "`void`": "`int f(void, float, int);`" is illegal.

Size change in cast involving pointer type: casted-to type ttt is not the same size as casted-from type ttt'.

Size of data segment exceeds addressability limit.

Size of local static storage exceeds addressability limit.
The size of a data segment exceeds an architecture-imposed limitation, e.g. 64K on an 8086. This can arise when there is too much global data declared in a compilation unit. Break up the unit into several, or use pragma `Data` to give different names to distinct sections of data. But note that on the 8086, only the Large memory model supports more than 64KB of static data.

Size of stack frame exceeds half of the addressability limit.
This is a warning that a function's local variable storage is close to an architecture-imposed limitation.

Size of stack frame exceeds quarter of the addressability limit.
This is a warning that a function's local variable storage is close to an architecture-imposed limitation.

Specified storage class for this declaration is unnecessary and was ignored.
In a declaration such as "`static struct s{int x;};`", the storage class "`static`" is useless since no object was declared.

Static initialization of bit fields is not supported.

Storage-class nonsensical for function definition.

String too long for initialized array.

Structure has no contents (is of size zero).

Subscripted expression must be an array or pointer.

The 2nd and 3rd operands of a conditional expression must be both arithmetic, or of the same type, or one a pointer and the other zero.

The declarator must be a function. This declaration has been discarded.

A declaration such as "`int f {...};`" was encountered, where a function body {...} was given for a non-function.

The rest of this line is extraneous.

The sign (**signed/unsigned**) has been specified more than once.

The storage-class (**auto**, **extern**, etc.) has been specified more than once.

The width (**long/short**) has been specified more than once.

This "**return**" should return a value of type ttt since the enclosing function returns this type.

This can be of an incomplete type only if it is "**extern**" or has an initializer supplying its size.

This code will never be executed.

This construct would have been deleted as an optimization had it contained no labels.

A construct such as "**while** (0) {...}" was detected but cannot be deleted due to the presence of one or more labels within {...}. This is questionable programming practice at best.

This function declaration is inconsistent with the "**int**"-returning function declaration imputed at Ln/Cm.

A function called before it is declared is assumed to be a function returning **int**, and any subsequent declaration of the function must declare it to be so. For example, "`main () { ... f(3);... } void f() {...}`" is illegal since f was called before being defined and therefore assumed to return **int**.

This function declaration is inconsistent with the declaration at Ln/Cm.

This is already defined as a macro. Redefinition ignored.

A redefinition of a macro is permitted only if the redefinition agrees exactly with the previous definition. To

otherwise redefine a macro, use `#undef` to explicitly undefine the macro before re-defining it.

This is multiply declared.
This is permissible only in conjunction with "`int`" or "`char`".
This is permissible only in conjunction with "`int`" or "`double`".
This is permissible only in conjunction with "`int`".
This is undeclared.
This may not be a pointer to a function (but may be a pointer to an object).
This tag name is more than 80 characters long.

This type lacks a tag and hence cannot be used.
A declaration such as "`struct {int x;};`" was encountered. Without a tag the `struct` cannot be referenced and hence is useless.

Toggle name required. Pragma ignored.

Too many initializers here.

Type ttt is not assignment compatible with type ttt'.
(a) In an assignment expression, the right operand, of type ttt, may not be assigned to the left operand, of type ttt'.
(b) In a function call, an argument, of the type ttt, may not be passed to a function that expects a parameter of type ttt'.

Type ttt is not compatible with type ttt'.
In a comparison, the left operand, of type ttt, may not be compared with the right operand, of type ttt'.

Unexpected symbol in expression. Line ignored.
Unknown preprocessing directive.
Unrecognizable Data class. Static assumed.
Unrecognizable Data class. Static assumed.
Unrecognizable field name.
Unrecognizable pragma name. Pragma ignored.
Unrecognizable toggle name. Pragma ignored.
Up-level addressing to a `register`-class variable is not allowed.
Variable is never used.

Variable is referenced but is never set.
Variable is set but is never referenced.
Variable referenced before set.
Variable required.

In this context a so-called "lvalue" is required but was not found. An *lvalue* is something whose address can be taken, and is required on the left side of an assignment expression and as an operand to &, ++, and —. The rules of C require the automatic conversion of some objects into non-lvalues. For example, an lvalue of type array-of-T is always converted to a (non-l)value of type pointer-to-T, so it is *never* allowable to take the address of an array. So, "`int a[10]; ... f(&a);`" produces the "Variable required." diagnostic due to the application of & to "a", which has been converted to the address of the first element. Remove the &.

Warnings have been completely disabled.

Define a "clean compile" as one containing no diagnostics from the compiler. A clean compile is a laudable goal for every compilation unit. To try to make the compile clean by turning Off warnings is "cheating", so the compiler produces this warning when the toggle Warn has been turned Off. This means that if the output of the compiler contains no diagnostics, the compile was truly clean.

Zero-length bit fields may not be named.

A declaration such as "`struct {int i:0,j:2};`" was encountered. "i" must be omitted. As is, it is possible to refer to the field. Such a reference would be illegal.

{...} inappropriate here for initializing a scalar.

19
Making Cross References

PP and HC. This section explains cross references and how to get them. The presentation is for both Professional Pascal and High C. Compile commands for the former are used in illustrations, so each occurrence of pp must be replaced by hc for High C programming. There are only ten such occurrences, all at the beginning of the second subsection.

On VAX/VMS. Command syntax is presented here for both VMS and non-VMS systems. Where VMS commands differ from non-VMS systems, a specific notation is made.

Cross compilers. To make this section work for a cross compiler simply replace pp with the appropriate command name. For examples, to make a cross reference using the High C VAX-to-8086 compiler use hc86 in place of pp below, or for Professional Pascal from VAX to an MC68000 use pp68.

19.1 Features of the Cross Reference

[annotated multi-modular, inter-modular, inter-lingual cross reference]

Cross-references have the following features:

- *References to source files.* All cross reference information refers to line numbers within files compiled, as opposed to line numbers within a listing. Therefore no listing is necesary to use the cross reference.
- *Include files.* Included source files are handled properly. That is, they do not interfere with the process, and their names are included correctly in the results.
- *Assignments versus uses.* References that assign values into variables are distinguished from references that use values of variables.

- *Annotated listing.* It is possible to generate an annotated source listing of one or more program files. The listing contains cross-reference information to the right of the source text listed.
- *Multi-module cross references.* A cross reference can span multiple compilation units by cross referencing many modules at once and showing references from one module into the other. Thus, a single cross reference can be produced for a program that is broken up into separately compiled modules.
- *Inter-module usage summaries.* A list of the names that one module uses that are located in other files can be produced, organized by file. This helps one understand the module interconnectivity of a large program.

19.2 How to Make a Cross Reference

Cross references are produced by a two-stage operation: First, the compiler produces cross-reference information in an intermediate file. Second, a separate cross-reference processor reads the intermediate file and produces the actual cross-reference listing.

One module. To compile a single source module named "M.P" ("M.C" and hc below for High C) and produce its cross reference in file "XLISTM.XRL" ("XRL" = "XRef Listing"), use the two commands:

```
pp m -xref                  VMS: pp m /cross
xref m > xlistm             VMS: xref m /output=xlistm
```

This produces object code as well as a cross reference listing. To avoid the former and therefore speed up the process, add `-noobj` (`/noobj` on VMS) to the command line.

To use the cross-referencer options described below for a single module, one should use the multi-module, "command file" approach described next.

Several modules. To compile and produce the cross reference for more than one source file, say modules M1.P, M2.P, and M3.P (".C" and hc below for High C), first compile each module with cross referencing on:

```
pp m1 -xref          VMS: pp m1 /cross
pp m1 -xref          VMS: pp m2 /cross
pp m1 -xref          VMS: pp m3 /cross
```

Again, add -noobj (/noobj) to each line to avoid code generation. Then enter the following pragmas in a file called, say, MS.CMD:

```
pragma Include('M1.XRF');
pragma Include('M2.XRF');
pragma Include('M3.XRF');
```

This gets the effect of concatenating the three ".XRF" files. Finally, to get the cross reference in file MS.XRL, type

```
pp m1 -xref          VMS: pp m1 /cross   (or hc)
xref ms > ms         VMS: xref ms /output=ms
```

The General Case. The -xref (/cross) of the compile command tells the compiler to produce a ".XRF" cross-reference information file. That file, or the concatenation of several such files, is then processed by the cross referencer *per se.*

The xref command has the format:

```
xref info [-on Toggle...] [-off Toggle...] [> list]
```

or on VMS:

```
xref info [/on=(Toggle,...)] [/off=(Toggle,...)]
                                        [/output=list]
```

where "info" is the name of an ".XRF" intermediate cross-reference information file or ".CMD" "command file" as illustrated above. Any ons and offs turn On and Off various Toggles described below. The cross reference is produced on the standard output unless redirected (>), in which case "list.XRL" is the file to which the listing is to be written (on VMS standard output is SYS$OUTPUT and redirection is specified via /output).

19.3 Cross-Referencer Pragmas

[cross referencer pragmas On, Off, Pop, Columns, Include]

The cross referencer is capable of producing other forms of output besides the standard alphabetically-sorted list of names. Communication to the cross referencer is done via pragmas placed in the ".CMD" file that have the same form as Professional Pascal pragmas, which is also the same as High C pragmas except that strings are specified with single-quotes (') rather than double-quotes (").

```
pragma  <Pragma_name> (<Pragma_parameters>) ;
```

The <Pragma_name>s supported are:

- *On*, *Off*, and *Pop*. These act identically to those of the source language; see Section *Compiler Pragmas*. In this case there may only be one <Pragma_parameter>, which may be any one of the following names:

```
Annotate_includes
Annotated_listing
List_module_usage
List_unused_includes
Statistics
```

  See Subsection 19.6 for a description of what these toggles do. They can also be entered on the `xref` command line using the `on` and `off` syntax. ***Example:***

```
xref whatever -on Annotated_listing Statistics
```

VMS:
```
xref whatever /on=(Annotated_listing,Statistics)
```

- *Columns (Declared_name, Info, Refs, Right_margin).* In this case the four parameters are integers. The values define the columns in which certain information is placed in the cross reference. The first specifies the column in which the declared name is placed; the second the column for related information; the third the column for the references to the *Declared_name*, and the last the right margin of the page. The default is *Columns(12,60,90,132).*

- *Include('File_name').* Here the parameter is a string denoting a file to be included. This works exactly like the Include pragma of Professional Pascal (and High C, except that the file name is enclosed in single, not double, quotes).

19.4 Cross-Referencer "Command Files"

[multi-module cross reference]

These pragmas can be edited right into the ".XRF" files produced by the compiler. But it is usually more convenient to make a small ".CMD" file containing the desired pragmas, together with an Include pragma specifying the name of each ".XRF" file of interest. *Example:*

```
pragma On(Annotated_listing);
pragma Columns(15,60,80,256);
pragma Include('PROG.XRF');
-- One Include for each .XRF file, if several modules.
-- Hyphenated comments are allowed as input to xref,
--          as illustrated by these latter three lines.
```

If file PROG.CMD contains the above lines, typing

```
xref PROG.CMD
```

produces the cross reference of PROG.XRF with an annotated listing and the columns as specified, all on the standard output.

Using such a "command file" is the preferred mode of operation when obtaining a multi-module cross reference. All of the .XRF files can be cross-referenced at once by constructing a command file that Include-s all the .XRF files, and xref-ing the command file.

Recall the several-module example above with modules M1.C, M2.C, and M3.C. Compile each one of them to obtain files M1.XRF, M2.XRF, and M3.XRF. Obtain a cross-reference for all three simultaneously by xref-ing a file containing

```
pragma Columns(15,60,80,256); -- For example.
pragma Include('M1.XRF'); -- See "Distinction of
pragma Include('M2.XRF'); -- File Names":
pragma Include('M3.XRF'); -- Subsection 19.7.
```

Caveat. Do not confuse the `Include` pragma for the cross-referencer with that for the source language. They have the same effect — that of including source text — but the former is directed to the cross-referencer and the latter to the compiler. Below we refer to files included in a compilation as "compiled include files" to avoid any confusion with files included in the input to the cross-referencer.

19.5 Cross-Reference Format

[components of cross-reference listings; Annotated_listing, List_module_usage]

Components. Each cross reference is self-documenting and consists of the following four items:

(1) An alphabetized list of all names declared in the program together with an ordered list of all the references to each name.

(2) An alphabetized table of all files used in the program and a file reference number for each.

(3) A list arranged by file of all the names declared in other files that each module uses — if requested.

(4) An annotated cross reference for each module — if requested.

When the components are produced:

Item (1) is always produced.

Item (2) is produced if the cross reference involves more than one file; this happens if either more than one module is cross referenced, or any compiled include files were involved in the modules being cross referenced.

Item (3) is produced if the `List_module_usage` toggle is On.

Item (4) is produced if the `Annotated_listing` toggle is On.

What each component consists of:

Item (1) presents the following information for each distinct name in the program:

- The line and column number of the declaration of the name. If the name occurs in a compiled include file, or if several modules are being cross-referenced, the file number is also given.
- The declared name N, and its *owner:* the name of the High C function, or the name of the Professional Pascal package, routine, or program, that contains N's declaration.
- Information about the named object, such as,

  in Professional Pascal, its "mode" (`type`, `const`, `var`, `procedure`, `function`, etc.), and in some cases the object's type or its value, such as a `const`'s value.

  in High C, its storage class (`static`, `extern`, `typedef`, `register`, etc.), and in some cases the object's type.
- The numbers of any lines containing references to the name. If the references are not in the module being cross-referenced, such as in an include file, or if several modules are being cross referenced, the line numbers are presented in the format "fn<...>" where n is the number of the file containing the references and "..." are the line numbers. Occasionally the entry in this field is of the form "resolved at *ref*" where *ref* is a line number or fn<...> reference as just described; this means that the name was introduced by a forward or external declaration whose actual definition was given at *ref*.
- References that assign or may assign a value to a variable are marked with the character "*".

Item (2) presents the correspondence between file numbers and file names. References in items (1) and (3) use only the file number rather than file names to keep the listing brief. Use item (2) to determine the corresponding file name.

Item (3) is optional. It is requested by turning on the toggle List_module_usage. The output produced is a listing for each module M of the names used by M that are declared in other files. The list is organized by file. This is useful for determining the interconnectivity between modules. For example, if module M1 refers to no function names within module M2, it may be possible to overlay the code of M1 and M2.

In Items (1) and (3) a reference to a name N declared at reference point P is changed to a reference to a point P′, if the definition at P′ resolves the declaration at P. Typically this will happen when N is declared in an interface file F, is used from a module M, and is defined at P′ in a module M′. The module usage in Item (3) will show that M refers to P′ in module M′, *not* P in interface file F. That is, one gets references to the implementations rather than the interfaces through which they were supplied.

Item (4) is optional and is requested by turning on the toggle Annotated_listing. The result is a line-numbered listing of the source of the program compiled, with each line annotated on the right with the line numbers where the names used on the line were defined.

If n names were used on the line, n line numbers appear to the right of the line, corresponding positionally. A line number alone is a reference into the file being listed. If the letter "i" appears instead, the name referenced is an intrinsic, such as Integer or Writeln in Pascal or "_find_char" or "_abs" in High C. Finally, a line number followed by "f" and another number means that the name was declared in a file other than the one being listed; the file number can be used to discover that file's name in Item(2). "Line#fFile#" was used instead of "File#<Line#>" as in Item (1) for brevity.

19.6 Cross-Referencer Toggles

[Annotate_includes, Annotated_listing, List_module_usage, List_unused_includes, Statistics]

`Annotate_includes`. Turn On this toggle to get an annotated listing of all files involved, not just files that are modules.

`Annotated_listing`. Turn On this toggle to get an annotated listing of all modules, but not "interface" or "header" files. See the description of Item (4) in the previous section.

`List_module_usage`. Turn On this toggle to get a listing for each module `M` of the names used by `M` that are declared in other files. See the description of Item (3) in the prior section.

`List_unused_includes`. Normally the cross referencer does *not* list any names found in compiled include files that were never referenced. This default can be overridden by turning On toggle `List_unused_includes`. The cross referencer also never lists unused intrinsic names. That cannot be overridden.

`Statistics`. Turning On this toggle causes a summary of cross-referencer statistics to be produced at the end.

19.7 Distinction of File Names

[sameness of include files for cross references]

In a multi-module cross reference, a particular interface file may have been included by several modules because each of the modules being cross referenced needs the resources in that file. The cross-referencer assumes that a repeated declaration of a name in a compiled include file is the same declaration if it appears at the same line and column number of the same include file.

For purposes of determining "sameness of include files" the cross referencer uses the text of the file name including the path, with casing ignored if the operating system ignores case in file names. Therefore to cross reference several modules successfully, avoid different names for one include file.

For example, if module M1 includes "../Utils/Trees.pf" and M2 includes "/Prog/Utils/Trees.pf" (".cf" or ".h" in High C), and if these two references denote the same file, the cross referencer will *not* recognize them as the same. (*On VMS:* "[-.UTIL]Trees.pf" and "[PROG.UTIL]Trees.pf".)

20
System Specifics

In this section are given some of the specific aspects of the 8086 implementation of High C.

20.1 Arithmetic

`int` arithmetic is much more efficient than `long int` arithmetic. `int` arithmetic is directly supported by the hardware, generally requiring one instruction per operation. `long int` addition and subtraction require several machine instructions per operation; `long int` multiplication requires a subroutine that uses three `int` multiplies; `long int` division requires an expensive subroutine that emulates division bit-by-bit.

Floating point is supported in the three formats of the 8087 numeric data co-processor.

`float`s are 32-bit values with an 8-bit exponent and a 23-bit mantissa. The absolute values of the representable numbers lie in the range $8.43 \times 10^{-37} .. 3.37 \times 10^{+38}$.

`double`s are 64-bit values with an 11-bit exponent and a 52-bit mantissa. The absolute values of the representable numbers lie in the range $4.19 \times 10^{-307} .. 1.67 \times 10^{+308}$.

`long double`s are 80-bit values with a 15-bit exponent and a 64-bit mantissa whose first bit is always 1. The absolute values of the representable numbers lie in the range $3.4 \times 10^{-4932} .. 1.2 \times 10^{+4932}$.

20.2 MS-DOS I/O

Not relevant to Concurrent

20.3 Addressing Limitations

Due to the 8086 hardware design, no single data structure may exceed 64K bytes in size. Therefore no aggregate variable, such as an array or structure, may exceed 64K bytes. Languages that support larger arrays on the 8086 "simulate" them with arrays of arrays (one can program the same in C).

Similarly, the run-time stack may not exceed 64K bytes, and no function may exceed 64K bytes of code. There are additional restrictions imposed by some memory models; see Section *Memory Models.*

20.4 Input Line Length

The default input line length for the compiler is 2,000 characters. When the `-cram` option is specified, it is reduced to 256 (not on the VMS cross compiler: irrelevant).

20.5 Heap-Item Size

Each item allocated on the heap incurs an overhead of ten bytes in small-data memory models and eight bytes in large-data models. Yes, those numbers are correct: the heap is protected, not a "cheap heap" without protection — but see `C_HEAP.C` in Subsection *Minimizing Program Size* of Section *Linking a Compiled Program.*

20.6 Default Segment Names: Pragma Code

The default code segment for each module is named by the file-name "stem", i.e. without any ".c". For example, if the module in file "x/y/z.c" ("[x.y]z.c" on VMS) is compiled, the code segment is named "z". Pragma Code can be used to specify another name. See Section *Externals.*

The default data segment for each module is named by the file-name "stem" preceded by "@". This data segment is public so that it can be moved into an overlay via an overlaying linker.

20.7 Overlays under PLINK86: Pragma Code

Not relevant to Concurrent

20.8 Some ANSI-Required Specifics

Here are some additional system specifics that the ANSI document (numbered X3J11/84-161 at this writing) requests that each C implementation provide.

char. The type specifier **char**, when not accompanied by an adjective, denotes type **unsigned char**.

Case of identifiers in generated object modules. Identifiers are emitted in object modules with exactly the same spelling as in the program text, including case. See Section *Linking a Program* about linking with respect to case.

Shift by a negative number. Shifting by a negative number produces unpredictable results and is flagged as an error by the compiler if detected at compile-time.

Sign of division remainder. The sign of the remainder from division is always the same as the sign of the dividend.

Signed right shift. A right shift of a signed integral type is an arithmetic shift, i.e. the sign bit is propagated from the most significant bit.

Truncation of a negative floating-point number. When a negative floating-point number is truncated, by virtue of converting it to an integral type, the truncation is toward zero. Thus -2.7 is truncated to -2 and -1.2 to -1.

Precision of floating-point arithmetic. Floating-point arithmetic is done in **float** precision unless one of the operands is **double** or **long double**, in which case the precision is **double** or **long double**, respectively. (Some C implementations do all floating-point arithmetic in **double** precision.)

Type of sizeof. The type of **sizeof(E)** is **unsigned int**, since no data structure can be longer than 64K bytes.

Pointer-integer casts. Pointer-to-integer and integer-to-pointer conversions act just like integer-to-integer conversions. That is, the integrity of the data involved is preserved only if the sizes of the involved pointer and integer types are the same.

For the sizes of data types, see Section *Storage Mapping*. If a pointer or integer value V is stored into a variable of a larger size, V may be retrieved intact from that variable.

Bit fields. A bit field may be only of type `unsigned int` or `unsigned long int`. The values `0..2**n-1` may be stored in a bit field of width `n`. Bit fields may straddle a single byte boundary and are allocated from right to left. For more information, see Section *Storage Mapping*.

Maximum number of cases. There may be at most about 5300 cases in a `switch` statement.

More Feedback, Please

(After some use.)

We would greatly appreciate your ideas regarding improvement of the language, its compiler, and its documentation. Please take time to jot down your ideas on this page (front and back) and on additional sheets as necessary *as you use the software*. Then, after you have some significant experience with the software, please mail the results to:

MetaWare™ Incorporated
412 Liberty Street
Santa Cruz, CA 95060

MetaWare may use or distribute any information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use that information. If you wish a reply, please provide your name and address. Thank you in advance, The Authors.

Page Comment

More Feedback, Please

Page Comment

I

Index

Starting on the next page is a "permuted key word in context" index for this document. In the center column is the particular key word W being indexed, in the context of a phrase or sentence containing W. The phrase appears to the left and right of W.

Occasionally the text of the phrase preceding W does not fit in the space to the left of W. In that case the index entry looks like

```
is text that was too long to precede the   WORD being indexed.  This ..... 7.4
```

where the first word "This" of the sentence did not fit on the left. Similarly the text to the right of W can be crowded:

```
                                the right. This  WORD is followed by too much text on ..... 7.4
```

where "the right" did not fit on the right.

If the texts both to the left and right do not fit, or the left (right) text cannot be completely wrapped around to the right (left), the entry is continued on another line. For example:

```
   but not too much text on the left. This  WORD is followed by far too much text on...
...the right                                ..................................... 7.4
```

After locating an entry, proceed directly to the referenced section(s). If a reference is to Section X.Y, look on page X-Y first and you will usually be within a page of the desired referent.

Acknowledgments

The authors of these manuals and designers of the High C language would like to thank the C standards committee, whose drafts of the C standard helped illuminate many dark areas of the language and assisted greatly in "chunking" the language concepts.

Paul Redmond's feedback was invaluable as he put dBase III through High C for Ashton-Tate. In the process he helped us polish the compiler in many ways.

David Shields' efforts in working with us were also very beneficial. He put tens of thousands of lines of C source code through High C, transliterated from the SETL version of the Ada-Ed compiler at New York University.

Professor William McKeeman and his research group at the Wang Institute of Graduate Studies supplied us with a collection of "gray expressions" that helped us verify the compiler.

The support of others who must needs remain nameless at this time is also appreciated.

Most of all we acknowledge that we are not self-made, but God-made. And we thank God for building into us the talents that made it possible for us to create High C. Praise God, from whom all blessings flow.

Ad majorem Dei gloriam (A.M.D.G.).

This ends the

MetaWare™ High C ™

Programmer's Guide