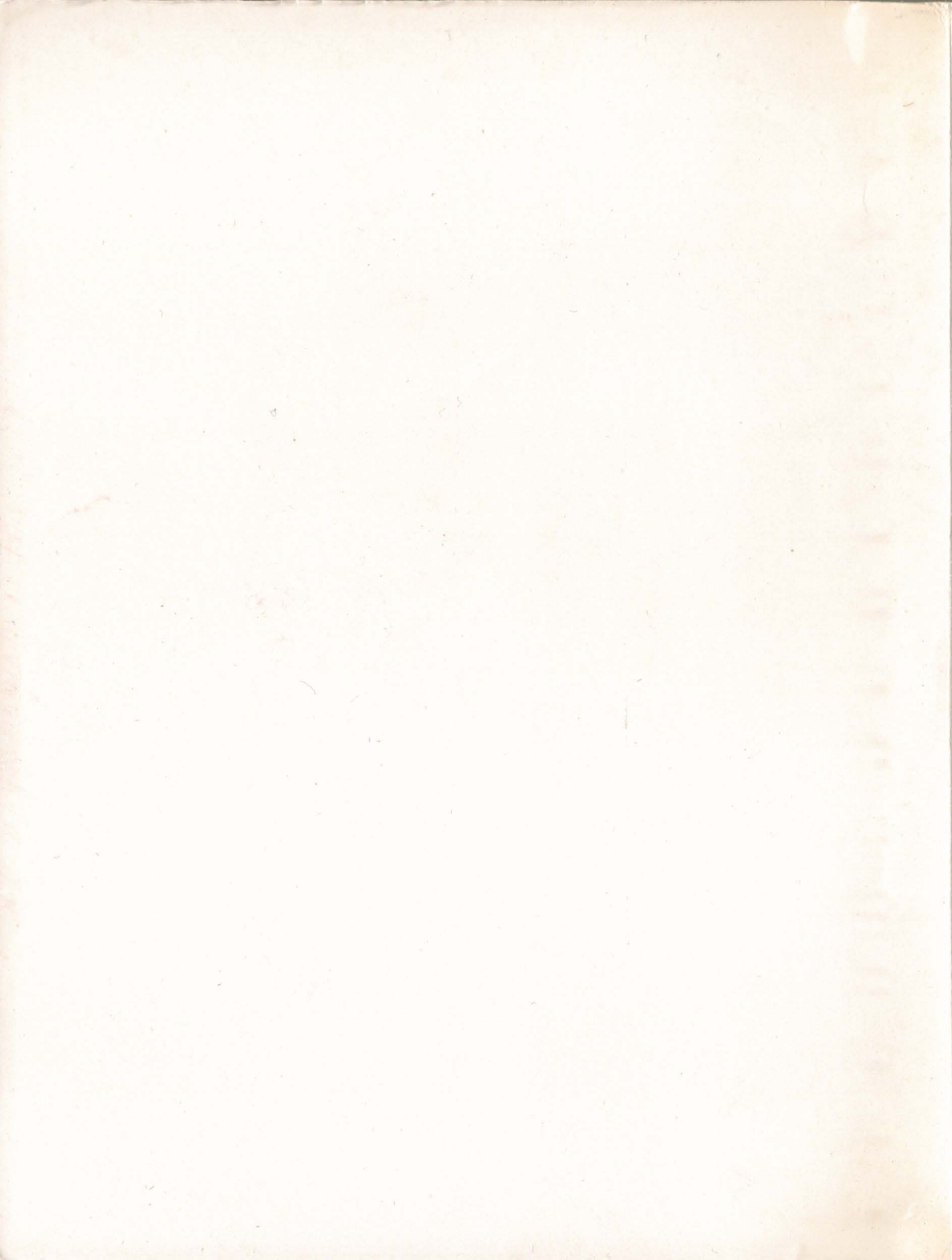


Prospero

C

Language

```
#include (stdio.h)
#define sievesize 8190
#define iterations true 1
#define false 0
char flags[sievesize+1];
main()
{
    int i,prime,k,count,iter;
    for (iter = 1; iter (<=sievesize); iter++)
        count = 0;
    for (i = 0; i (<=sievesize); i++)
        flags[i] = true;
    for (i = 0; i (<=sievesize); i++)
        if (flags[i])
            prime = 2 * i;
            while (k = i + prime)
                flags[k] = false;
    }
    count
```



Prospero

C

Language

September 1990

Prospero Software

LANGUAGES FOR MICROCOMPUTER PROFESSIONALS

COPYRIGHT

Copyright © 1988, 1990 Prospero Software. All rights reserved.

This document is copyright and may not be reproduced by any method, translated, transmitted, or stored in a retrieval system without prior written permission of Prospero Software.

Permission is granted to Prospero C licence holders to abstract and use any of the programming examples.

DISCLAIMER

While every effort is made to ensure accuracy, Prospero Software cannot be held responsible for errors or omissions, and reserve the right to revise this document without notice.

TRADEMARKS

Acknowledgement is made for references in this manual to Apple, Lisa and Macintosh, which are trademarks of Apple Computer Inc., to WordStar, which is a trademark of MicroPro International Corp., to Digital Research and GEM, which are trademarks of Digital Research Inc., to Atari and Atari ST, which are trademarks of Atari Corp., to Motorola and MC68000, which are trademarks of Motorola Inc., and to Unix, which is a trademark of AT&T Bell Laboratories.

Prospero C, Pro Fortran-77, Prospero Fortran, Pro Pascal and Prospero Pascal are trademarks of Prospero Software.

Prospero Software, Inc.
100 Commercial Street, Suite 306
Portland, Maine 04101
U.S.A

Prospero Software Ltd.
190 Castelnau
London SW13 9DH
England

C

C is a programming language originated by Dennis Ritchie, working at AT&T's Bell Laboratories in the early 1970's. It is a general purpose programming language which has achieved widespread popularity by combining a wealth of operators with a simplicity that allows extremely efficient programs to be coded. Many operating systems have been coded in C, most notably UNIX, and the language has been implemented on a very wide range of computer systems.

In 1982, an standardization committee was set up by the American National Standards Institute (ANSI) in order to produce a standard for the language. The latest draft of the Standard has not yet been officially approved, but is expected to become an official Standard in the next year or so, with only minor changes.

PROSPERO C

Prospero C is a complete implementation of the August 1987 draft of the proposed ANSI standard for C for use on the Atari ST range of computers. Prospero C includes:

- Workbench for easy programming
- Four window Programmer's editor
- C syntax checker
- C compiler
- Linker
- Librarian
- Cross reference generator
- Symbolic debugger
- Complete ANSI C library with many extensions
- Complete GEM AES and VDI bindings
- 1000 pages of documentation.

The Prospero C compiler is a true compiler, generating native machine code for efficient program execution.



FORMAT OF THIS MANUAL

Prospero C is accompanied by a user manual in four volumes. This is Volume 1, and is divided into several parts.

Part I contains the directions for installing and operating the software (Workbench, compiler, object programs, etc.), the options available, format of diagnostics, hints on program testing, and suchlike matters. There are also details of hardware requirements and installation and configuration procedures.

Part II covers various aspects of the Prospero C implementation, including details of the hardware and operating system interfaces.

Part III forms a detailed reference manual for writing programs in Prospero C, and describes all features of the language.

There are appendices giving the formal syntax, the compile-time and run-time error codes, the ASCII character set, observations on mixed-language programs, and definitions of terms, limits, and implementation specific behavior.

Volume 2 describes the Prospero C library functions.

Volumes 3 and 4 contain descriptions of the GEM bindings for making use of the GEM VDI and AES functions.

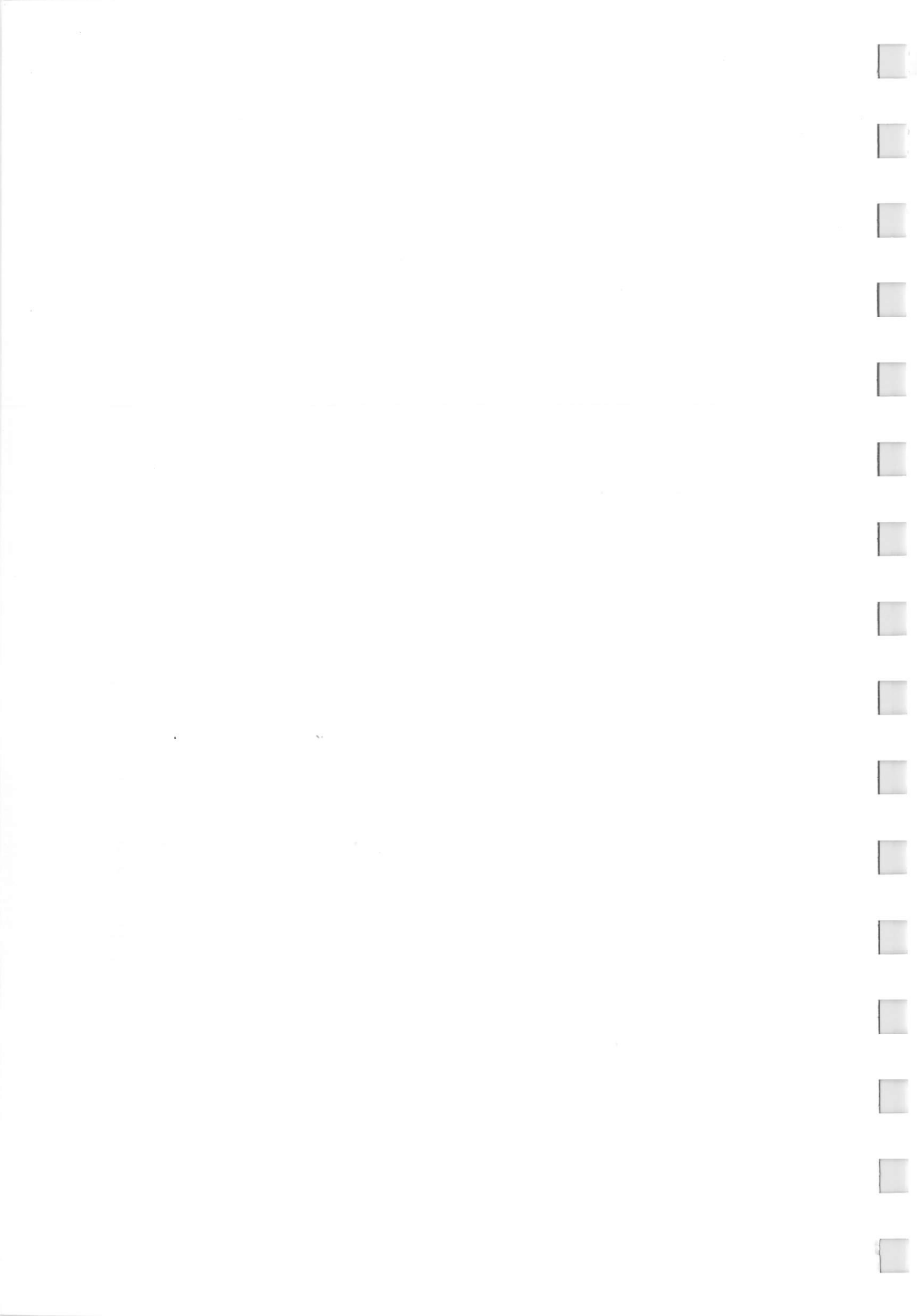
PART I – PROSPERO C OPERATION

1	Introduction	1
1.1	Prospero C	1
1.2	GEM	2
1.3	About this manual	4
1.4	Contents of the package	5
1.5	Installation details	6
1.5.1	Hardware requirements	6
1.5.2	Delivery	6
1.5.3	Installation on a hard disk	7
1.5.4	Installation on floppy disks	8
1.5.5	Installation on a RAM disk	9
1.5.6	Workbench configuration	10
2	Simple edit, compile and link	12
3	Operation of the Workbench	17
3.1	The Workbench menus	23
3.1.1	Desk menu	23
3.1.2	File menu	24
3.1.3	Block menu	27
3.1.4	Find menu	30
3.1.5	Compile menu	35
3.1.6	Link menu	37
3.1.7	Run menu	38
3.1.8	Options menu	40
3.2	Workbench key combinations	44
3.2.1	WordStar style control key combinations	44
3.2.2	Alt key combinations	46
3.2.3	Special key combinations	47
4	Operation of the compiler	48
4.1	Compile-time options	49
4.1.1	Compiler output to LOG file – option G	49
4.1.2	Source listing to PRN file – option L	50
4.1.3	Include source line information – option N	50
4.1.4	Check array indexes – option I	50
4.1.5	Check assignments against bounds – option A	51
4.1.6	Check pointers – option P	51
4.1.7	Accept strict ANSI C Standard only – option S	51
4.1.8	Char is unsigned – option U	52
4.1.9	Generate compact code – option C	52
4.1.10	Wait after errors – option W	52
4.1.11	Autosave after compilation – option V	52
4.2	Check syntax	53

4.3	Command line version	53
4.3.1	The one-line command	53
4.3.2	Conversational mode	53
4.3.3	Making use of variables	54
4.3.4	Command-line macros	54
4.4	Compiler messages	55
5	Operation of the linker	57
5.1	Simple use of the Linker	58
5.1.1	Link	58
5.1.2	With small libraries	58
5.1.3	With CGEM	58
5.1.4	With control file	59
5.1.5	Link other file	59
5.2	Linking using a control file	60
5.2.1	Stack allocation	61
5.2.2	Linker maps	61
5.2.3	Module Specifications	62
5.3	Command line version	63
5.3.1	The one-line command	63
5.3.2	Conversational mode	63
5.3.3	Indirect mode	64
5.4	Linker messages	64
5.4.1	Non-fatal errors	64
5.4.2	Fatal errors	65
6	Operation of object programs	67
6.1	Arguments to main	67
6.2	Pre-connected files	67
6.3	Run-time errors	67
6.4	Miscellaneous error messages	68
7	Operation of the librarian	70
7.1	Forms of invocation	70
7.1.1	The one-line command	70
7.1.2	Conversational mode	71
7.1.3	Indirect mode	72
7.2	Report options	72
7.2.1	Module listing (M)	72
7.2.2	Cross-reference listing (X)	73
7.2.3	Unsatisfied references listing (U)	73
7.2.4	Suppress '.' names (N)	73
7.2.5	Listings to disk (D)	73
7.3	Module selection	74
7.4	Librarian messages	75

Contents

7.4.1	Normal messages	75
7.4.2	Error messages	75
8	The symbolic debugger	77
8.1	General description	77
8.2	Sample session	79
8.3	General guidance on using Probe	80
8.3.1	GEMDOS aspects	80
8.3.2	The start-up file	80
8.3.3	Entry after runtime errors	81
8.3.4	Working with SID and other programs	81
8.4	Probe command parameters	82
8.4.1	Character set and source tokens	82
8.4.2	Identifiers	82
8.4.3	Number constants	82
8.4.4	String constants	83
8.4.5	Special symbols	83
8.4.6	Identifiers and qualifiers	84
8.4.7	Break and watch specifiers	85
8.5	Probe commands	86
8.5.1	assign	86
8.5.2	break	86
8.5.3	calls	88
8.5.4	display	88
8.5.5	echo	88
8.5.6	go	89
8.5.7	help	89
8.5.8	key	90
8.5.9	list	90
8.5.10	output	91
8.5.11	profile	91
8.5.12	quit	92
8.5.13	route	92
8.5.14	step	93
8.5.15	trace	93
8.5.16	view	94
8.5.17	watch	94
8.5.18	X (execute)	95
8.5.19	Z (hexadecimal display)	95
8.6	Command syntax summary	96
9	The cross-reference generator	98
9.1	Operation from the Workbench	98
9.2	Operation from the command-line	99



1 INTRODUCTION

1.1 Prospero C

Welcome to Prospero C. Thank you for purchasing this product. We hope you enjoy using it; please let us know if you encounter any problems using it or if you have any suggestions for improvements.

Prospero C is a professional quality ANSI-standard C compiler for the Atari ST.

It is designed for use by the most demanding professional programmers who want to exploit the power of this machine to the full, and by educators and students who prefer to learn about C on this attractive machine using a standards-conforming language.

Prospero has a reputation for producing high quality languages and their Pascal and Fortran compilers for the Atari ST, IBM PC and other machines are used by programmers in government service, universities, the professions and in science and technology laboratories around the world.

At the time of writing, the C standard has not been adopted by ANSI, so officially there is no such thing as ANSI-standard C. The Prospero C compiler meets the standard as defined in the latest draft of the proposed standard; when the standard is accepted we will reissue this product to take account of any developments. If you find any ways in which you believe our implementation of C does not conform to the standard, please write to us.

Prospero C has many aspects: it is an excellent C compiler, it is a programming environment, it is an effective and accurate calculating machine, it gives access to all the facilities of the Atari ST, both directly through operating system calls, and indirectly through the GEM graphics interface. In addition, it provides and documents the bindings which are necessary to use GEM facilities within your programs, if that is what you want to do. The programming environment provides an excellent example of the sort of GEM application you can produce, and demonstrates some of the features GEM offers. However, it is certain to be initially more difficult to write programs to run under GEM than using the 'vanilla' scrolling screen mode of interaction. GEM is introduced in the following section and discussed in greater detail in Volumes 3 and 4.

1.2 GEM

The Workbench provided with this product runs under GEM, and two of the four volumes of this manual are devoted to the use of GEM, so some explanation of GEM, its function and purpose follows. However, the use of GEM in your C programs is entirely optional, and certainly it is far simpler to write programs which do not use it. However, if you invest the time to learn how to use GEM in your programs, you can produce powerful and user-friendly software. As the GEM graphics calls are independent of the specific hardware, and GEM is implemented on a number of machines as well as the Atari, you can also write extremely portable software.

GEM is an applications programming environment provided as part of the ROM firmware on your Atari ST computer; it is a piece of software that sits between the applications program which you write, and the microcomputer's operating system. It has two simple objectives: firstly to make your programming job easier in the long term; secondly to make it easier for the user of your programs.

GEM is active in two important areas: graphics input and output, and the user interface. Both of these are a problem in traditional "A prompt" programming.

Graphics poses technical problems because the graphical input and output requirements of computers are not standard – the Atari has three different screen resolutions, all of which are driven slightly differently – and many different ways of telling a computer to draw a square.

GEM provides a hardware-independent interface between your programs and the computer; this means it has two layers – a standard layer which interprets whatever your program asks it to do, and a variable, device-specific, part with sections of program called drivers which are written specially for each type of screen, keyboard, printer, plotter etc. The program makes calls to the standard part, knowing that the output will be the same (as far as possible) regardless of what the hardware in use is, and therefore the program needs no knowledge of what hardware will be in use when it is run, and portable graphics programs can be written. Thus, for example, if your program uses a window with a close box, when you tell GEM to draw the window it will use the information from the screen driver in use to calculate what size the close box should be, and then issue a draw command for a square of appropriate size and fill color. Your program may have been written using one screen resolution; when it is run on another screen resolution – or on a different computer where GEM is available with a completely different screen – GEM will ensure that it works correctly.

The user interface problem is one of standardization – some programmers like to use menus which are selected by letters, others like function keys, some programmers use a particular key for one thing, and some use it for something completely different. Any new users of a program will expect to have finger trouble – they will use commands they learnt in another program and be surprised when they don't work. There have been some attempts at standardization – for example the control key commands used in WordStar are copied to a greater or lesser extent in other programs. The result is almost total confusion, and results in a big training problem for computer users.

GEM provides what is needed – a radical new approach. The first step involves separating the control functions from the keyboard so that everyone can forget whatever key sequences he or she knows and start again with a clear mind. The device used by GEM is the mouse – a small box moved around the user's desk which causes a pointer to move round on the screen. The mouse isn't the only pointing device that has been invented, but it has several advantages.

The conventional screen with lines of text has, added to it, various borders and menu areas which always work in a consistent way. For example any user knows that a box at the top left-hand corner of a window means "close". So any user looking at a GEM program he has never seen before will know that he can move the pointer so it points at a box at the top left of whatever he sees on the screen and click on the left hand button, and the whatever will go away.

These visual ideas originated with the Smalltalk language developed at Xerox PARC. These have become popular on the Apple Lisa and Macintosh, and have now been brought to your Atari in GEM. Without going into detail, GEM provides a large vocabulary of user interface devices; with them you can make your programs much easier to understand and use – avoiding the need for elaborate manuals and complex training programs.

1.3 About this manual

This manual is presented in four volumes and runs to some 1000 pages. About half of this is the description of the GEM bindings and how to use them, and this will perhaps indicate how much GEM provides. It is subdivided as follows:

Volume 1

Part I	How to use Prospero C
Part II	Prospero C Implementation details
Part III	Prospero C Language Definition
App.	Appendices

Volume 2

Library	Prospero C Library details
---------	----------------------------

Volume 3

VDI	GEM VDI Bindings
-----	------------------

Volume 4

AES	GEM AES Bindings
-----	------------------

Part I tells you how to install and run the programs in this package. Part II tells you about the way in which the C language is implemented, mentioning the standard features and describing Prospero extensions in more detail. Part III is a complete reference guide to Prospero C. This is not intended as a teach yourself guide to C; many excellent books have been written to fill this gap. The Appendix contains much useful information and includes a source language syntax definition and various error code listings. The second volume describes the C library functions which you can use from your programs. The third volume describes the VDI bindings, which concern GEM's graphics output primitives. The fourth volume describes the library of AES bindings, which concern the user of GEM's user interface facilities, such as windows and pull-down menus.

1.4 Contents of the package

The implementation follows the normal practice for compiled (as opposed to interpreted) C in that a source program is first created using the built-in editor, converted to machine code by the compiler, and then linked with a selection of support routines from the library using the link editor, to produce an executable version of the program. The executable version can then be directly invoked by the operating system (many times, if desired) without any further direct use of the C software. Thus four essential parts of the package are the editor, compiler, the link editor, and the library of support routines.

- The editor forms part of the 'Workbench' – a GEM application called C-BENCH.PRG, which is used to control the creation and modification of C source text files and the invocation of the compiler, linker and utility programs. The final executable program (or any other program) can also be invoked from the Workbench if desired, or it can be executed as a 'stand-alone' program completely independently.
- The compiler can be invoked from the Workbench when required. It consists of two main processing programs C1.OVL and C2.OVL. These may search for the file of error message texts C.ERR if the need arises. A stand-alone version C.TTP is also supplied.
- The link editor program can also be invoked from the Workbench, and is called PROLINK.OVL. A stand-alone version of the linker PROLINK.TTP is also supplied, which can be executed independently from the Workbench.
- Three libraries are provided. CLIB.BIN contains the standard C functions and Prospero C extensions and run time support. CLIBS.BIN is an alternative version of CLIB which can be used to produce smaller programs when no floating point support is required. One of the above is always required. CGEM.BIN contains the GEM AES and VDI bindings, and is required for any program making use of GEM.

In addition to the compiler, link editor and standard libraries, there are three other major items in the package. These are the symbolic debug program PROBE.PRG (with its file of help texts PROBE.HLP), the cross reference program CXREF.TTP, and the library manager PROLIB.PRG which will be important for users wishing to construct their own libraries. All these can be invoked from the Workbench program.

1.5 Installation details

1.5.1 Hardware requirements

The hardware required to run Prospero C is an Atari ST computer, a screen, a user memory area of at least 400K bytes, and at least 720K bytes of disk storage.

1.5.2 Delivery

The Prospero C software is delivered on disks containing the following files:

C-BENCH.PRG	Prospero GEM Workbench (control)
C-BENCH.RSC	Workbench resource file
C1.OVL	Compiler (Pass 1) overlay
C2.OVL	Compiler (Pass 2) overlay
C.ERR	Compile-time error messages
CFIRST.BIN	Standard library header
CLIB.BIN	Standard library
CFIRST0.BIN	Alternative (small) library header
CLIB0.BIN	Alternative (small) standard library
CGEM.BIN	GEM AES and VDI bindings library
LAST.BIN	Standard library footer
PROLINK.OVL	Linker overlay
PROLIB.PRG	Librarian program
PROBE.PRG	Symbolic debugger
PROBE.HLP	Symbolic debugger "Help" file
CXREF.TTP	Cross-reference program
C.TTP	Stand-alone version of compiler
PROLINK.TTP	Stand-alone version of linker
H*.H	Header files defining library functions and types – see volumes 2, 3, and 4

Also supplied are a few example source program (.C) files, and a program CCHECK.PRG for checking the integrity of the supplied files. This can be run (from the GEM Desktop or the Workbench) to verify that the programs supplied are not damaged. If there are any special comments relating to the software, for instance descriptions of extra files included on the disk, they are placed in a file called READ.ME. If this file is present, consult it before installing the software.

1.5.3 Installation on a hard disk

Installation on a hard disk does not present any special problem. The following assumes that the hard disk is C:.

The Prospero C overlays and utility programs (the files supplied with an extension .PRG, .TTP or .OVL) must all be contained in the same folder – the name of this folder is a matter of personal preference, but the total path name including the drive must be no more than 32 characters. Normally the folder will be a single level in from the root, and be called something like PROC.

Having created a new folder for these files, using the GEM Desktop **New Folder** command, the .PRG, .TTP and .OVL files from the issue disks should be copied into it; the files PROBE.HLP and C.ERR should also be copied.

The folder in which the Workbench looks for the libraries can be specified independently of the overlays, so a separate folder can be set up to contain them. It will usually be simpler to place them in the same folder as the overlays.

The folder containing the header files can also be specified separately, but it is normally best to keep them in a subdirectory called H within the folder containing the overlays.

Source files will normally be kept in a separate folder – or several folders according to their nature. The supplied source files can be placed in whichever folder is most convenient.

The Workbench and its resource file may be placed in the same folder as the overlays, or it can be placed in the root level of the disk – the latter allows the Workbench to be installed so that double clicking on any .C file will automatically open the Workbench and load the specified .C file into its memory ready to edit.

1.5.4 Installation on floppy disks

Floppy disk installation is more difficult, because the compiler files have to be spread over several disks. With single sided disks of 360K capacity, there is not room for the Workbench and all the overlay files on a single disk. However, the Workbench itself does not need to be available once the program has started up, so that the overlays can be placed on a separate disk.

The source files being compiled will normally be kept on a disk in drive B – this disk will remain in the drive for the duration of the Workbench session, unless a different set of source files are to be compiled. The disk in drive A contains the Workbench overlays, and may have to be changed occasionally if there is not room for all overlays on one disk.

There are several ways of dividing the files to minimize the amount of disk changing that is required – the best approach is largely a matter of personal preference. Three possible schemes are outlined below :-

Scheme 1. Separate compiling and linking disks

The files required for compilation (C1.OVL, C2.OVL, C.ERR, and optionally CXREF.TTP) are placed on one disk – the compiling disk. The files required when linking or running a program (PROLINK.OVL, CFIRST.BIN, CLIB.BIN, CGEM.BIN, LAST.BIN, and optionally PROBE.PRG, PROBE.HLP and PROLIB.PRG) are placed on a second disk – the linking disk. Source programs are kept on separate disks as required, though the header files may be kept on the compiling disk. The Workbench can be placed on the compiling disk if there is room, or on a separate disk which is removed once the program has started.

This scheme requires a disk change between compiling and linking a file, which is irritating for small files, where the compile-link-run cycle will be quite rapid. However it is good for developing large (or very large) programs, particularly those where a number of source segments are linked into a single executable file.

Scheme 2. Separate main and auxiliary disks

The files absolutely required for compilation and linking (C1.OVL, C2.OVL, C.ERR, PROLINK.OVL, CFIRST*.BIN, CGEM.BIN, CLIB*.BIN and LAST.BIN) are placed on a single disk – the main disk. A second, auxiliary, disk contains the files required less frequently – PROBE.PRG, PROBE.HLP, CXREF.TTP and PROLIB.PRG. Source programs are kept on separate disks as before. The Workbench is placed on a separate disk which is removed once the program has started – perhaps on the auxiliary disk.

If two disk drives are available, this scheme requires no disk changes for simple compile-link-run cycles, unless any of the auxiliary utility files are required. However for single drive systems, linking will require some disk swapping as the linker reads both the object file and the library files – scheme 3, where these are all on the same disk, is preferable under such circumstances.

Scheme 3. Libraries on source disk

The overlay files, which the Workbench expects to find in the same folder (C1.OVL, C2.OVL, C.ERR, PROLINK.OVL, CXREF.TTP, PROBE.PRG, PROBE.HLP and optionally PROLIB.PRG) are placed on one disk – the code disk. The libraries, which the Workbench can be instructed to search for in a separate folder, are placed on a second disk together with the source code being compiled. The Workbench can be placed with the overlays on the code disk, or can be placed on a separate disk which is removed once the program has started.

This scheme requires no disk changes at all on a twin drive system, and the minimum on a single drive system, but requires copies of any libraries and include files used to be placed on any disk containing source code to be compiled.

If double sided floppies of capacity 720K are available, all overlays, utilities and libraries and source can be accommodated on a single disk without difficulty.

1.5.5 Installation on a RAM disk

Installation of the Workbench overlays on a RAM disk, if one is available, gives a very significant improvement in the speed with which they can be loaded, and therefore on compilation and linking speeds. Source files can then be kept either on the RAM disk (and copied periodically to permanent floppies) or on floppies as required. On a 1040 ST there is sufficient

required overlays, with plenty of memory left for editing, compiling and running sizeable programs. On a 520 ST a RAM disk is less practical, though a small RAM disk can be used to hold the compiler workfiles to speed up compilation.

1.5.6 Workbench configuration

When the Workbench starts up, it searches for its resource file, and for a configuration file named C-BENCH.CFG. This file stores all the settings of the Workbench options, including the instructions about where to locate the compiler overlays and libraries. If the Workbench, all its overlays, and the source files to be edited are in the same directory, the Workbench will function without this configuration file – normally, however, a configuration file of this name should be created the first time the Workbench is used. This is done using commands from the **Options** menu as follows.

The first step is to specify the directory pathnames where the overlays, libraries and source files are to be found. Selecting **Set drive/path names** from the **Options** menu produces the following form.

Set drive/path names

Path for compiler overlays :- A:\PROC_____

Drive for workfiles :- B:

Path for user files :- B:_____

Path for include files :- B:_____

Path for Libraries :- A:\PROC_____

The diagram shows the suggested values that should be entered for a floppy disk installation following scheme 1 or 2 – note that the compiler overlays and libraries are stored in a folder rather than the root in the example above. For a hard disk system the entries might be as follows:

Compiler overlays	:- C:\PROC\
Workfiles	:- C:
User files	:- C:\SOURCES\
Include files	:- C:\SOURCES\H\
Libraries	:- C:\PROC\

Having selected an appropriate configuration, and clicked **OK** to put away the form, the configuration can be saved. However, it may be preferable to adjust a few of the other configurable options at this stage before saving them – for example the tab width, the compiler options and the function key settings. When the configuration is correct, selecting **Save configuration** from the **Options** menu presents the File Selector form to obtain the configuration file name. Clicking **OK** or pressing Return or Enter without altering the **Directory** or **Selection** fields will save the configuration to the file C-BENCH.CFG in the folder where the resource file was found, ready to be loaded automatically next time the Workbench is started. The Workbench is now ready for use.

2 SIMPLE EDIT, COMPILE AND LINK

To prove that the software is installed and functioning correctly double click on the icon marked C-BENCH.PRG. A blank screen appears with a white menu bar at the top. Move the mouse so the point is over the word **File**. This menu drops down:

Desk	File	Block	Find	Compile
	Edit .C file			ⓂE
	Edit other file			

	Save file			ⓂS
	Save as , , ,			
	Delete File			ⓂD
	Print			

	Write block to file			^KW
	Read block from file			^KR
	Print block			^KP

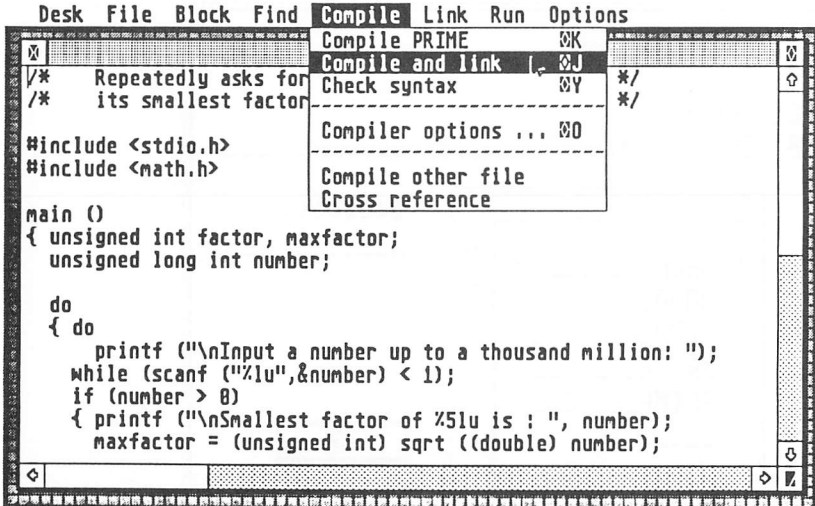
	Close			ⓂW
	Quit			ⓂQ

Move the mouse down so that the words **Edit .C file** are highlighted in reverse video. Click on the mouse button. The menu goes away and a form like this appears:

The screenshot shows a window titled "ITEM SELECTOR". At the top, it says "Directory:" followed by "B:\SOURCES*.C" and a dashed line. Below this is a list box with a title bar containing a close button (X) and the text "*.C". The list contains the following items: "BACKUP" (with a small icon to its left), "DOODLE.C", "GEMDEMO.C", "MANDEL.C", and "PRIME.C". There are also several dashed lines representing empty space in the list. To the right of the list box is a vertical scrollbar with an upward arrow at the top and a downward arrow at the bottom. To the right of the list box is a text field labeled "Selection:" with a dashed line below it. At the bottom right of the window are two buttons: "OK" and "Cancel".

This will show the program files that are in the diskette in drive B. The top line, marked **Directory:**, shows the current directory path that the computer is using, and the slider window below it the current list of files in the folder. You can select one of the files shown by double-clicking on it. Select PRIME.C. A window will then open containing the text from the file. It may then be edited. However since this is a demonstration file it can be compiled straight away without any edits being required.

To compile the source of this program from memory take the mouse pointer into the menu bar at the top of the screen over the word **Compile**. A menu will drop down:




Note that the first line says **Compile PRIME**. Move the mouse so that the pointer moves down the menu to the line **Compile and link**. When the pointer is on this line (and it is highlighted as above) click the left hand mouse button.

A new form entitled **Compiling B:\PRIME** will appear:

Compiling B:\PRIME

Last Error :- Line no :- 0

After a short pause while the compiler overlay is loaded, compilation will begin. The line number at the top right of the form will go up, in increments of twenty, while Pass 1 converts the source to an intermediate file on disk. Assuming that no errors are encountered (there should be none, unless the source of PRIME has been altered), Pass 2 will then load, and produce a relocatable file PRIME.BIN. The next stage is to “link edit” this file to produce an executable program. As the **Compile and Link** option was selected, the Workbench immediately invokes the Linker, and the following form appears:

Last Error :-	Linking B:\PRIME.PRG	Reading :- PRIME.BIN
		
<input type="button" value="Abort"/>		<input type="button" value="Continue"/>

The file PRIME does not make use of the additional library CGEM.BIN, so the object file name displayed after **Reading :-** in the top right should say FIRST.BIN, PRIME.BIN, CLIB.BIN then LAST.BIN, then the same again for the linker’s second pass. No errors should be reported unless PRIME has been modified. The result of linking is the file PRIME.PRG, in the same folder as the source PRIME.C was loaded from. If you select the **File** menu and click on **Edit other file** you will see three files whose name begin with PRIME: the source PRIME.C, the executable program PRIME.PRG, and also the relocatable version PRIME.BIN which is generated by the compiler and read by the linker. If the compiler options G, L or N were specified, other files might be generated – see section 4 for details.

To run the program PRIME.PRG select the **Run** menu :

Compile Link B:\PRIME.C a number, and , or Prime	<table border="1"> <tr> <td style="text-align: center;">Run Options</td> </tr> <tr> <td> Run PRIME <input type="checkbox"/> R Run other file ----- With command tail Run under GEM ----- Debug program <input type="checkbox"/> P </td> </tr> </table>	Run Options	Run PRIME <input type="checkbox"/> R Run other file ----- With command tail Run under GEM ----- Debug program <input type="checkbox"/> P
Run Options			
Run PRIME <input type="checkbox"/> R Run other file ----- With command tail Run under GEM ----- Debug program <input type="checkbox"/> P			

and click on **Run PRIME**. PRIME should be run without a command tail, not under GEM – the menu should therefore appear as above.

Program PRIME reads from standard input and writes to standard output, which are by default assigned to the screen. It repeatedly asks for a number and prints the smallest factor (or else 'Prime'). The screen is cleared and a first line of text appears:

Input an integer up to a thousand million :

Enter a suitable number:

999999989

The program continues:

Smallest factor of 999999989 is : 4327

Input an integer up to a thousand million : 999999937

Smallest factor of 999999937 is : Prime

and so on. (No computation takes longer than about a second.)

In addition to entering a zero, PRIME can be terminated by typing control-C. In this case, the Workbench will resume immediately, and redraw its window ready for further edits. If PRIME is terminated by entering zero, the Workbench will display the message

Press any key to continue

Pressing any key will return to the Workbench ready for further editing.

The extra facilities of the editor, compiler and linker are explained in the next three sections.

3 OPERATION OF THE WORKBENCH

*Note: In the text of the following descriptions words that appear on the screen during program operation are printed in **this font**. Following an industry convention, ^ is used to mean Ctrl or Control and ♦ is used to mean Alt or Alternate. Control and Alternate are second and third function shift keys: whereas the shift key is held down, as on a conventional typewriter, to turn a lower case 'a' into an upper case 'A', Control and Alternate are held down to turn 'l' into 'Repeat Find' or 'x' into 'Cut'.*

The Workbench is designed to help you display and edit program source text files, and to operate the compiler, linker and other utility programs provided as part of this package.

The Workbench can be operated in four ways:

- by selecting commands from the menus which drop down when the pointer is moved into the menu bar at the top of the screen. This is usually done using the mouse.
- by using the function keys. No preset values are attached to these keys but they may be set up to expand to any sequence of key strokes, including control codes.
- by using the keyboard. The editor accepts the commands marked on the keytops such as left arrow and Page Up, the commonly accepted WordStar control key combinations such as ^S, ^D, ^E, ^X, and a selection of Alt key combinations such as ♦X for cut, ♦C for copy and ♦V for paste. As a reminder these appear to the right of corresponding menu commands.
- by using the mouse to place the cursor in the text, and to select blocks of text. This is an alternative to using the cursor control keys (left arrow etc.)

The editor is designed to be as familiar as possible to people who have had programming experience using different editors on different machines. This is an impossible ideal, since different editors work in mutually incompatible ways. However, wherever possible we have provided several familiar ways to do the same thing.

The Workbench is a normal GEM application, and uses menus, window control points, dialog boxes and so on in the normal manner. Thus anyone who is familiar with the operation of other GEM applications should immediately find the control of the Workbench familiar. If you are not familiar with other GEM applications, the most important concepts common to all are described below.

The Workbench application is contained in the file C-BENCH.PRG – this is a program designed to use the WIMP interface provided by GEM AES (WIMP stands for 'Windows, Icons, Mouse and Pull-down menus'). A companion file, called C-BENCH.RSC, contains descriptions of the menu bar and various dialog boxes (described later) which the Workbench uses; this is called the resource file, and must always be present if the Workbench is to be successfully executed.

GEM applications are driven using the mouse, which can be moved around a flat surface causing a pointer or other cursor to move around the screen (this cursor is also frequently referred to as 'the mouse'). By moving the mouse cursor about the screen, and clicking the mouse button, you can select items from menus, select or move windows, answer questions, make choices and so on. Throughout this manual the term click will refer to the left hand mouse button being pressed then released – to 'click on' means to move the mouse pointer over an object and then click the button. The mouse can also be used to 'drag' – the mouse button is depressed and held down while the mouse cursor is moved to another area of the screen. 'Double clicking' – clicking twice in rapid succession – is also used by many applications, though not extensively by the Workbench.

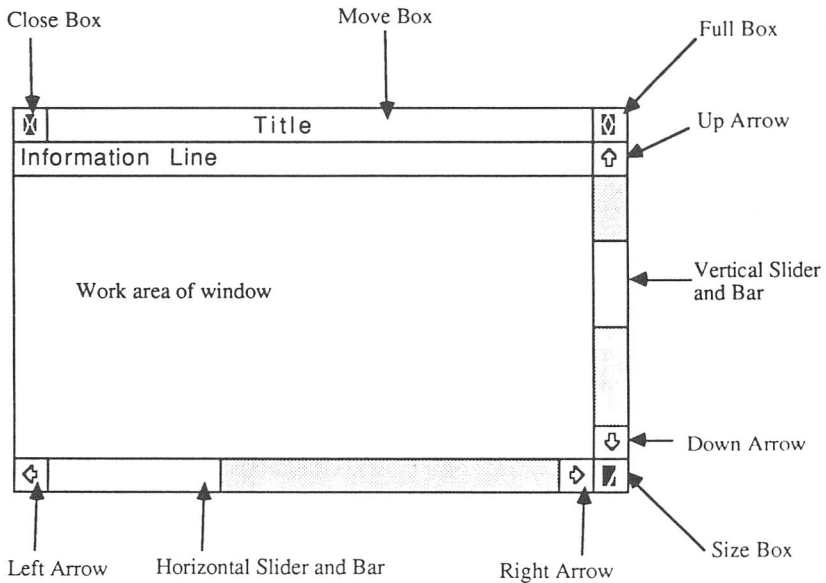
To run the Workbench you should double click on the program icon called C-BENCH.PRG on the GEM desktop. The program will then load, and a blank screen will appear with a white menu bar at the top with these words:

Desk File Block Find Compile Link Run Options

Each word in the menu bar corresponds to a group of items which can be selected, and is known as a menu title. If the mouse is over a title in the menu bar, a list of items pops down. An item from this list can be selected by moving the mouse pointer to the appropriate line, then clicking the button. Each available item on the menu (unavailable items are shown dimmed) will be highlighted as the mouse is passed over it, to indicate which item would be selected if the mouse was clicked at the time. To put the menu away without selecting an item, the mouse can be clicked while it is not over a highlighted item, or it can be moved over a different menu title in the menu bar to cause that title's menu to appear.

Note that no normal operations can proceed while a menu is on the screen, so that windows may not show what they should until the menu has been removed. This is a limitation of the GEM operating system.

The second important feature of almost all GEM applications is the use of windows. A window is a rectangular area of the screen which contains information of some sort from the application – in the Workbench up to four windows may be used, each containing the text of a different file. These windows can overlap just like sheets of paper, so that not all of a window may be visible at any given time. One window (the ‘active’ or ‘current’ window) will be on top, and will not be obscured by any other windows. To make a different window active, and bring it to the top, the mouse can be clicked on any visible part of the window.



Around the edge of the window are a number of areas which can be clicked on to alter the appearance of the window. These are shown in the diagram above. Clicking in the close box in the top left hand corner indicates that the window is to be removed from the screen. The full box in the top right hand corner will increase the window's size to occupy the entire screen (or shrink it to its former size if it is already full size). The position of the window can be moved by pressing the mouse button over the title bar and dragging it to a new position, while the size and shape can be altered by dragging the size box in the bottom right hand corner.

The remaining control areas affect the window's contents rather than its screen position. The sliders indicate what portion of the total information is currently displayed – by dragging them to a new position a different part of the information can be viewed. To move in smaller steps (a page at a time), clicking in the area of the slider bar above or below (or for the horizontal slider, to the right or left of) the slider causes the window contents to be scrolled in the appropriate direction. To scroll in still smaller steps, the appropriate arrow can be clicked.

A third important concept in GEM applications is the dialog box – these are GEM's way of prompting a user for information. A simple dialog box, typically requiring a yes/no response, is called an alert, while a more complicated one requiring various pieces of information is often referred to as a form, as it behaves rather like a paper form to be filled in. Most GEM applications will have a number of different forms which can appear when information is required, but all will be driven in the same way. One particular form which is used extensively by most applications (including the Workbench) is the File Selector form, and will serve as an example of the features of other forms.

When an application wants to obtain the name of a file from the user, either to read from or write to, it uses the File Selector form. This is drawn in the middle of the screen, obscuring what was there before, and looks approximately as shown in the diagram – the precise appearance varies between versions of GEM.

In the bottom right hand corner are two boxes containing the words **OK** and **Cancel** respectively. These are known as buttons, and are 'pushed' by clicking on them with the mouse. The **OK** button is drawn with a thick black border – this indicates that it is the default button, and that Return or Enter can be pressed as an alternative to clicking on the button. The **Cancel** button has a slightly thickened border, indicating that it is an exit button – in other words, clicking on it indicates that you have finished with the form. Buttons with a thin border (there are none on this form) indicate an option to be selected – when selected, the button is displayed highlighted. Sometimes selecting one button will cause others to become unselected – these are known as radio buttons because they behave like waveband buttons on traditional radios.

ITEM SELECTOR

Directory:
B:\SOURCES*.C

Selection:
MANDEL...C

*.C

- BACKUP...C
- DOODLE...C
- GEMDEMO...C
- MANDEL...C**
- PRIME...C
-
-
-
-

OK

Cancel

The File Selector form contains two 'editable text fields' where text can be entered by the user. Only one of these contains a cursor (a vertical bar) – if characters are typed they will appear at this point. Unused character positions are marked with underscores. The cursor can be moved to the left or right using the cursor keys, and backspace and delete can be used to edit the text. The escape key clears the current text. To move the cursor to a different text field, the up or down cursor keys or tab or backtab as appropriate can be used. Alternatively, an editable text field can be made current by clicking on it with the mouse.

The two fields in the File Selector box are labeled **Selection:** and **Directory:** – these indicate the filename and path specifier of the selected file respectively. The directory specifier also indicates a wildcard specifier such as '*.C', indicating that only files with the given extension are to be displayed in the list of available files. To select a file and pathname, you could type in the required name and path in these two text fields. However, the File Selector form is slightly different from other forms in that it allows these fields to be filled in by clicking in the directory listing box in the lower left hand corner. Any files in the directory indicated by the **Directory** field which match the given wildcard specifier are listed in this box, and clicking on a filename will fill in the **Selection** field automatically. If the directory contains any sub-directories, these will be listed too, with a mark beside them to distinguish them from the filenames, and clicking on one of these will update the **Directory** field so that the contents of the sub-directory is

displayed. To close a directory and return to the parent, the close box in the top left hand corner of the directory listing should be clicked. In GEM version 2.0, closing the root directory gives a list of the available drives to be selected. In earlier versions, the root cannot be closed, and the only way to change to a different drive is by editing the contents of the **Directory** editable text field. Note that any alterations to this field made by typing do not take effect until you click in the the directory listing box – in the title bar is best.

To select an existing file, you should obtain the listing of the appropriate directory, scroll it using the arrows or slider bar until the required file is visible, then click on the filename. To select a new filename, the appropriate directory should be selected as before, then the filename typed into the **Selection** field. The list of files can still be of use, to make sure that the filename you are about to select is not already taken.

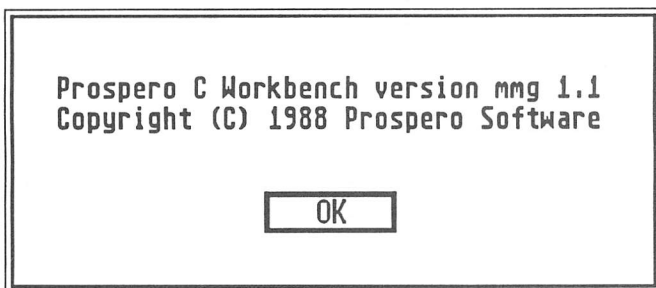
Once the **Directory** and **Selection** fields are correct, the **OK** button is clicked to confirm the choice. Return or enter may be typed as an alternative. Clicking **Cancel** indicates that you do not want to proceed with the operation, and it will be aborted. As a short cut, double-clicking on a filename in the directory listing has the same effect as clicking on it then clicking **OK**.

3.1 The Workbench menus

Throughout operation of the Workbench a menu bar is displayed across the top of the screen, from which menu items can be selected in the usual GEM manner as described above. The items are divided into 8 groups according to their function, described in the following sections.

3.1.1 Desk menu

The **Desk** menu provides access to GEM Desk Accessories such as calculator and clock programs which the user may wish to install. The first line of the **Desk** menu, **About C-BENCH ...**, provides the Prospero C copyright notice:



Note that under GEM 2.0 this menu title is moved to the right hand side of the menu bar, and its text changed to reflect the name of the current application. Thus it will be called **C-BENCH**, but will be the same in all other respects.

3.1.2 File menu

File	Block	Find	Compile
Edit .C file			⌘E
Edit other file			

Save file			⌘S
Save as , , ,			
Delete File			⌘D
Print			

Write block to file			⌘RW
Read block from file			⌘RR
Print block			⌘RP

Close			⌘W
Quit			⌘Q

This menu contains commands for loading and saving files to be edited, and associated functions. It also contains the Quit command for ending a Workbench session.

3.1.2.1 Edit .C file

Edit .C file is used to open a window in which to edit a program source file (with a .C extension). Clicking on this option produces a GEM File Selector form as described above. Having selected the required file name and directory, click **OK** to confirm the choice. If the specified file already exists, the source will be loaded from disk into a window to be examined or edited as required. If the file does not exist, an alert asks you to confirm that you wanted to create a new file – if you click **OK** or type Return or Enter, an empty window is opened into which the new text can be typed.

Clicking on **Cancel** at any stage aborts the entire operation. Note that ⌘E can be used as an alternative to selecting this item.

3.1.2.2 Edit other file

Edit other file works in the same way as **Edit .C file**, except that no default extension of .C is assumed, and none is applied to file names with no extension. It is intended for use in editing ASCII files which are not C program files, or for editing header (.H) files.

3.1.2.3 Save file

Save file records the text in the current top window on disk. If the words **Save file** are dimmed (as in the diagram above) the file on disk is the same as the file in memory and there is no need to save, or there are no windows open to save. Note that ♦S can be used as an alternative to selecting this item.

3.1.2.4 Save as ...

Save as... works in the same way as **Save file**, except that it presents the File Selector form (see above) to allow the name of the file to be specified. If the name chosen is that of a file which already exists, an alert box is presented for confirmation that the current file on disk is to be replaced. Clicking on **Cancel** at any stage aborts the operation. This is dimmed and unavailable (as above) if no windows are open.

3.1.2.5 Delete file

Clicking on **Delete file** presents the File Selector form (see above) to allow the name of a file on the disk to be selected for deletion. An alert box is presented to allow you to confirm that deletion is required. Clicking on **Cancel** at any stage aborts the operation. Note that ♦D can be used as an alternative to selecting this item.

3.1.2.6 Print

Print is used to print out the current top window to the printer. If only part of it is wanted, then **Print block** (3.1.2.9) may be used.

3.1.2.7 Write block to file

Write block is used to place a selected block of text from the top window onto the disk in a file whose name is chosen using the File Selector. If the specified file exists, an alert is presented to confirm that it is to be overwritten. Clicking on **Cancel** at any stage aborts the operation.

Write block is only available when text is selected (see section 3.1.3 for selection methods) and is otherwise dimmed, as above. The WordStar command ^KW can be used as an alternative to selecting this item.

3.1.2.8 Read block from file

Read block is used to read a block of text from a file on disk and insert it into the current top window. The text file is chosen using the File Selector form, and the whole of the text from the file is inserted into the top window at the current cursor position. Clicking on **Cancel** in the File Selector aborts the operation. This item is dimmed and unavailable (as above) if no windows are open. The WordStar command ^KR can be used as an alternative to selecting this item.

3.1.2.9 Print block

Print block is used to print a selected block of text from the top window. This item is only available when text is selected (see section 3.1.3 for selection methods) and is otherwise dimmed, as above. The WordStar command ^KP can be used as an alternative to selecting this item.

3.1.2.10 Close

Close is used to close the current top window. If the text in it has not been saved, an alert box first offers the chance to save it. Click **Yes** or type Return or Enter to save the text before closing the window. Clicking **Cancel** will abort the close operation, while clicking **No** will close the window and discard any changes made to the text since it was last written to disk. This item is dimmed and unavailable (as above) if no windows are open.

The same effect can be achieved by clicking on the close box in the top left hand corner of the window, or by typing ♦W.

3.1.2.11 Quit

Quit stops execution of the Workbench program and returns you to the GEM Desktop. An alert box offers the chance to save any unsaved files before closure. Note that ♦Q can be used as an alternative to selecting this item.

3.1.3 Block menu

Block	Find	Compile	Link
Mark start of block			^KB
Mark end of block			^KK

Copy block		ⓄC	^KC
Cut block		ⓄX	^KX
Paste block		ⓄV	^KV
Delete block			^KY
Unmark block		ⓄH	^KH

The **Block** menu is concerned with operations affecting blocks of text. A block of text may be any amount of continuous text which is highlighted on the screen by being shown in reverse video and which is to be the subject of a further operation – so a block of text can be marked for cutting, copying, deleting, or writing to disk. There are four ways to mark text :

- by using the **Mark start** and **Mark end** commands in this menu.
- by dragging the mouse across the required text, with the mouse button depressed. Double click may also be used to select a single word. Note that that the block and the cursor are separate distinct things. When dragging, the text cursor is left at the last end of the drag, either start or end depending on the direction of drag, and so is difficult to see. When double clicking to select a word, the text cursor is left where it would be with a single click, just under the arrow.
- by using the WordStar block commands ^KB (Mark start) and ^KK (Mark end). These are provided for people who use them naturally; the other WordStar block commands ^KV (Paste), ^KX (Cut), ^KC (Copy), ^KY (Delete), ^KH (Hide), ^KR (Read Block) and ^KW (Write Block) also work. However for people who have not previously used WordStar they are not the easiest commands to learn.
- the subject of a Find operation (see section 3.1.4) will be selected when found.

None of the commands in this menu are available unless a window is open.

3.1.3.1 Mark start of block

Mark start makes the current text cursor position the start of the block. The block start and block end are maintained independently, and can be moved independently – the block is defined only if both start and end are defined and the block start precedes the block end. Moving the block start may therefore cause the block to become defined or undefined according to the block end position. If a block is defined, it is displayed in reverse video. This item is dimmed and unavailable if no windows are open.

The WordStar command ^KB can be used as an alternative to selecting this item.

3.1.3.2 Mark end of block

Mark end makes the current text cursor position the end of the block. If this is after the current block start, a block will be defined. If it is before the current block start, or the block start has not been defined, the block will be undefined, and the block operations will not be available. This item is dimmed and unavailable if no windows are open.

The WordStar command ^KK can be used as an alternative to selecting this item.

3.1.3.3 Copy block

Copy block places a copy of the marked text in an internal buffer. If this buffer contains text, the **Paste block** item in this menu is undimmed, and can be used to insert the buffer contents into the text at the cursor position. **Delete block** (3.1.3.6) and **Write block to file** (3.1.2.6) do not affect the contents of this internal buffer. This item is dimmed and unavailable unless a valid block is marked in the top window.

The WordStar command ^KC or the Alt key command ♦C can be used as alternatives to selecting this item.

3.1.3.4 Cut block

Cut block works in the same way as **Copy block** except that it deletes the marked text after copying it to the internal buffer. This item is dimmed and unavailable unless a valid block is marked in the top window.

The WordStar command ^KX or the Alt key command ♦X can be used as alternatives to selecting this item.

3.1.3.5 Paste block

Paste block inserts the contents of the internal buffer created by **Copy block** or **Cut block** at the current text cursor position. It is not affected by the current marked block, and is only available if text has previously been placed in the internal buffer using **Copy block** or **Cut block**. The buffer is not emptied, so that a single cut followed by several pastes can be used to make multiple copies of a section of text.

The WordStar command ^KV or the Alt key command ♦V can be used as alternatives to selecting this item.

3.1.3.6 Delete block

Delete block removes the current marked block of text. There is no way to retrieve this text, so it may be better practise to use **Cut block** to delete text as it can then be retrieved if necessary. The text is then unmarked. This item is dimmed and unavailable unless a valid block is marked in the top window.

The WordStar command ^KY can be used as an alternative to selecting this item.

3.1.3.7 Unmark block

Unmark block removes the highlight from the current marked block, and clears the start and end block markers.

The WordStar command ^KH or the Alt key command ♦H can be used as alternatives to selecting this item.

3.1.4 Find menu

Find	Compile	Link	Run
Find		⌘F	^QF
Find and replace		⌘A	^QA
Repeat find			^L

Start of text			^QR
End of text			^QC
Start of block			^QB
End of block			^QK

Goto line number			⌘G

The **Find** menu is concerned with locating text, block markers and line numbers within the text in the current top window. The text cursor is positioned before whatever is found. The following WordStar and Alt key commands also perform find functions :

WordStar	Alt key	Command
^QF	◆F	Find
^QA	◆A	Find and replace
^L		Repeat find
^QR		Find start of text
^QC		Find end of text
^QB		Find start of block
^QK		Find end of block
	◆G	Goto line number

None of the commands in this menu are available unless a window is open.

3.1.4.1 Find

Find is a sophisticated search function. It will search forwards or backwards from the current cursor position, or forwards from the start of a text file, for a string of characters up to 32 characters long, optionally ignoring their case. The search process is rapid, due in part to the fact that searches are not made across line barriers, and leading spaces at the start of each line are ignored.

The dialog box shown below is produced, so that the relevant search string and options can be entered. The options are selected using two sets of radio buttons. When the options have been set, clicking **OK** will start the search. If the specified string is found, it will be selected as the current block, and the cursor will be positioned at the start of it. **Repeat find** can then be used to find the next occurrence, searching forwards or backwards as appropriate. If the text is not found, an alert is displayed and the cursor remains where it was.

FIND

Find string :- form_alert_____

Direction Forwards Backwards **Global**

Match Exact **Caseless**

The WordStar command \wedge QF or the Alt key command \blacklozenge F can be used as alternatives to selecting this item.

3.1.4.2 Find and replace

Find and replace is similar to the **Find** string function, but allows the text if found to be replaced selectively with another text. The function can replace a single occurrence then stop; it can go through each occurrence, prompting the user whether it should be replaced; or it can replace all occurrences without any further intervention. This last feature should be used with caution, as there is no way to undo all these changes (except the hard way – one by one).

A dialog box like the one below is used to contain the two strings of text and the various options :

FIND & REPLACE			
Find string :-	check_name _____		
Replace with :-	check_file_name _____		
Direction	<input checked="" type="checkbox"/> Forwards	<input type="checkbox"/> Backwards	<input type="checkbox"/> Global
Match	<input checked="" type="checkbox"/> Exact	<input type="checkbox"/> Caseless	
Replace	<input type="checkbox"/> One	<input type="checkbox"/> Some	<input checked="" type="checkbox"/> All
	<input type="button" value="Cancel"/>		<input type="button" value="OK"/>

Once the required text strings and options have been entered, clicking **OK** starts the replace function, while clicking **Cancel** will abort the process. If the **Some** option was selected, each occurrence in turn is highlighted, and a dialog appears to confirm the replacement as follows :

Replace ?	<input type="button" value="Yes"/>	<input type="button" value="No"/>	<input type="button" value="Stop"/>
-----------	------------------------------------	-----------------------------------	-------------------------------------

Clicking Yes or typing Return or Enter causes the highlighted text to be replaced, and then the next occurrence found; clicking No skips to the next occurrence without replacing; while clicking Stop aborts the replace operation at that point.

The WordStar command \wedge QA or the Alt key command \blacklozenge A can be used as an alternative way to select this item.

3.1.4.3 Repeat find

Repeat find repeats the previous **Find** or **Find and replace** command, without prompting for new search strings or options. If you wish to quickly skip through all the occurrences of a certain letter or word, it is much easier to use the equivalent WordStar command ^L than to repeatedly select the menu item. Each time you press ^L or select **Repeat find** will find or replace the next occurrence. This command is only available if a find or replace command has been performed.

3.1.4.4 Start of text

Start of text moves the text cursor to the start of the text in the current top window.

The WordStar command ^QR can be used as an alternative to selecting this item, as can the key combination control-home if a key marked 'home' is available.

3.1.4.5 End of text

End of text moves the text cursor to the end of the text in the current top window.

The WordStar command ^QC can be used as an alternative to selecting this item, as can the key combination control-end if a key marked 'end' is available.

3.1.4.6 Start of block

Start of block move the cursor the the start of the current marked block in the top window. If the block start is not defined this command is dimmed and unavailable.

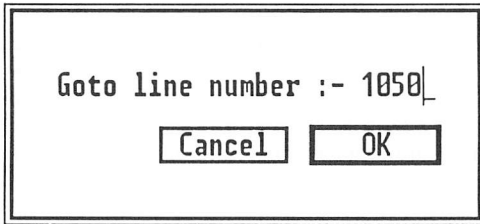
The WordStar command ^QB can be used as an alternative to selecting this item.

3.1.4.7 End of block

End of block move the cursor the the end of the current marked block in the top window. If the block end is not defined this command is dimmed and unavailable.

The WordStar command ^QK can be used as an alternative to selecting this item.

3.1.4.8 Goto line number



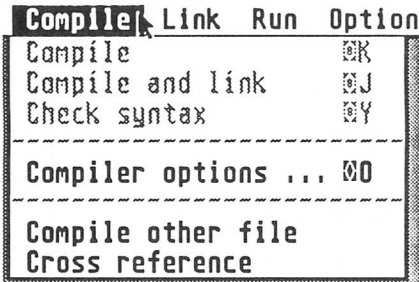
Goto line number :- 1050|

Cancel OK

Goto line number is useful when using error reports that refer to the line number in the source file, or simply to find out how many lines there are in a file. A form is produced as above, asking for the required line number. If you select **OK**, the cursor will be placed at the start of the appropriate line (provided that there are sufficient lines in the text). The default value is the current line number, so that this command can be used to discover the current line number without altering it.

The Alt key command **◆G** can be used as an alternative to selecting this item.

3.1.5 Compile menu



The **Compile** menu is used to control the compilation of C source programs. The menu is divided into three sections.

3.1.5.1 Compile from memory

The first section of the menu contains commands which operate on the current top window, and are only available when a window is open :

Compile runs the Prospero C compiler. If text is available to compile, the menu changes to give the name of the source which would be compiled if this option was selected (the diagram above shows the menu when no text is available). The source is compiled directly from memory, to produce an object file with a .BIN extension on disk. See section 4 for more details on compiler operation. Note that ♦K can be used as an alternative to selecting this item.

Compile and link is equivalent to **Compile**, except that if the compilation is successfully completed it will then go on to link. See section 3.1.6 for more details on linking. Note that ♦J can be used as an alternative to selecting this item.

Check syntax uses the Prospero C compiler to check the source in the current top window, without producing an object file. This considerably speeds up the process, and is useful for finding errors when it is likely that some exist. Note that ♦Y (or ♦Z on German keyboards) can be used as an alternative to selecting this item.

3.1.5.2 Compiler options

The second section of the menu is used to set the compiler options. This is done using the following form :

Compiler Options

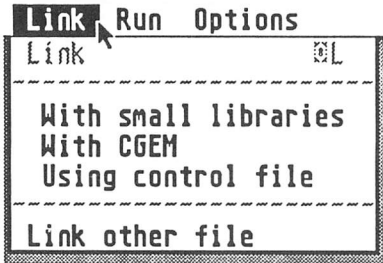
G	Compiler output to LOG file
L	Source listing to PRN file
N	Include source line information
I	Check array indexes
A	Check assignments against bounds
P	Check pointers
S	Accept strict ANSI Standard C only
U	Char is unsigned
C	Generate compact code
W	Wait after errors
V	Autosave after compilation

Options which are selected are shown with a highlighted letter; they remain selected until next altered, and if the **save configuration** facility in the **Options** menu is used (see 3.1.8.1), they will still be selected next time work is resumed. The significance of each option is described in section 4.

3.1.5.3 Compile from disk

The final section of the compile menu is for operations on disk files: **Compile other file** is used to compile a source file from the disk. In this case the File Selector form is presented and the user is invited to select any .C file for compilation. In other ways the operation is the same as for the **Compile** option. **Cross reference** is used to generate a cross-reference listing of any source file on the disk, selected using the File Selector form. The operation of the cross-reference generator is described further in section 9.

3.1.6 Link menu



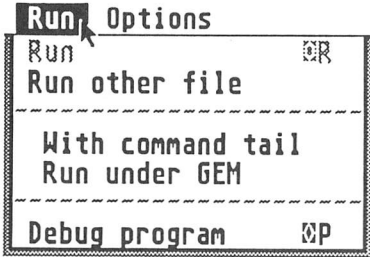
The **Link** menu is used to control the linking of the object programs produced by the compiler.

Link runs PROLINK, the Prospero linker. If text is available to compile, the menu changes to give the name of the source file (the diagram above shows the menu when no text is available). If this source has been successfully compiled since its last edit, the corresponding .BIN file on disk will be linked using the options specified using the rest of this menu, otherwise the command will be dimmed and unavailable. See section 5 for more details on linker operation. Note that **♦L** can be used to select this item.

The **With small libraries**, **With CGEM** and **Using control file** options are used to set the linker options. Options which are selected are shown with a check mark; they remain in force until next altered, and can be saved using the **Save configuration** facility in the **Options** menu (see 3.1.8.1). The significance of each option is described in detail in section 5.

The **Link other file** command is used to link any other object file on disk. The file selector form is presented and the user is invited to select any .BIN file from the disk for linking. In other ways the operation is the same as for **Link**.

3.1.7 Run menu



The **Run** menu provides the means to execute other programs or GEM applications without quitting from the Workbench. Typically this will be the program just compiled and linked, but any other .PRG, .TOS or .TTP file on the disk can be executed. The Workbench program remains in memory and will continue normally after termination of the specified program.

3.1.7.1 Run

If text is available for compilation in the top window, this menu item changes to give the name of the source file. Provided that the text has been successfully compiled and linked since its last edit, selecting this item provides a quick way to run the resulting program, otherwise it will be dimmed and unavailable.

The Alt key command $\diamond R$ can be used instead of selecting this item.

3.1.7.2 Run other file

Run other file presents the file selector form from which any executable file can be chosen. The default extension is .PRG – however it is possible to edit the file specification in the path field, or simply type in a filename with a .TTP or .TOS extension, in order to execute files of other type. If the selected file has a .PRG extension, it will be run using the options specified below. .TTP and .TOS programs are run as if **Run under GEM** was not selected, respectively with and without a command tail, as from the GEM Desktop.

3.1.7.3 With command tail

With command tail allows the user to indicate whether a command tail is to be passed to programs executed using the **Run** or **Run other file** commands. If selected, a check mark is displayed in the menu, and a dialog is presented to allow the command tail to be entered whenever a program is to be run – the diagram overleaf shows the command tail 'myfile.dat myfile.out' being entered ready to be passed to a program :-

Enter command tail :- myfile.dat myfile.out _____

Cancel

OK

3.1.7.4 Run under GEM

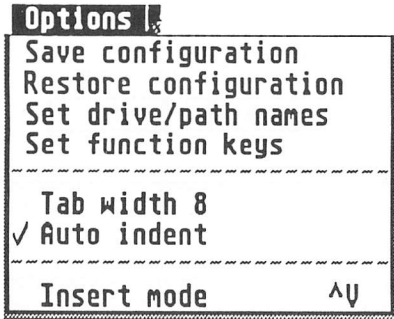
Run under GEM allows the user to indicate whether programs executed using the **Run** or **Run other file** commands are to run as GEM applications or not. If selected, a check mark is displayed in the menu, and the screen will be left in graphics mode when the program is executed. If not selected, the screen is cleared before running the program, and the Workbench will wait for a keystroke after termination of the program before redrawing its windows, menu bar etc.

3.1.7.5 Debug program

Debug program is used to run PROBE, the symbolic debugger. In order to be debugged, a program must have had the N option (Include source line information) and preferably the L option (Source listing to PRN file) set when it was compiled. See section 8 for more details on how to operate PROBE.

The Alt key combination ♦P can be used as an alternative to selecting this item.

3.1.8 Options menu



The commands in the **Options** menu are concerned with various aspects of the Workbench configuration, so that it can be tailored more precisely to the individual user's requirements.

3.1.8.1 Save configuration

The **Save configuration** facility records the current settings of the various Workbench options in a disk file, so that they can be reloaded when the Workbench is next used. The file selector form is used to select the directory and name of the file in which the configuration is to be saved – if **OK** is clicked without altering the **Directory** or **Selection** fields, the file C-BENCH.CFG will be used in the same folder as the resource file was found in. The Workbench searches for a file of this name when it starts up, and automatically loads its options from it if found. It is also possible to specify a different filename, so that several sets of options can be saved ready for use in different circumstances – for example different function keys might be defined when editing different programs, or when different people are using the editor.

3.1.8.2 Restore configuration

The **Restore configuration** facility sets the various Workbench options from those stored in a disk file. The file selector form is used to allow you to select the directory and name of the .CFG file from which the configuration is to be restored. The options in C-BENCH.CFG will be loaded when the Workbench starts, as described above, but this command may be useful if other configurations have been saved for different circumstances.

3.1.8.3 Set drive/path names

Set drive/path names	
Path for compiler overlays :-	A:\PROC_____
Drive for workfiles :-	B:
Path for user files :-	B:_____
Path for include files :-	B:_____
Path for Libraries :-	A:\PROC_____
<input type="button" value="Cancel"/> <input type="button" value="OK"/>	

The **Set drive/path names** command is used to specify the drives and directories in which the editor looks for the compiler and linker overlays and libraries, where the compiler places its workfiles and looks for include files whose path is not fully specified, and the default path where the Workbench looks for source files when using the file selector. The path used for user files is automatically altered whenever it is changed in a file selector form. These pathnames are saved in the configuration file when **Save configuration** is used (see 3.1.8.1).

3.1.8.4 Set function keys

Set Function key sequences

```

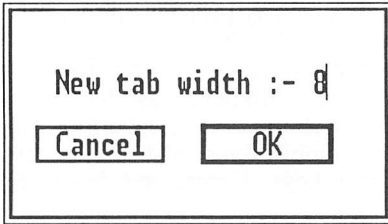
F1 string  :- #include <stdio.h>_____
F2 string  :- #define _____
F3 string  :- printf ("_____
F4 string  :- for (;;)_____
F5 string  :- if (altered) do_altered_____
F6 string  :- {3 8_____
F7 string  :- {3008_____
F8 string  :- Prospero Software_____
F9 string  :- Prospero Software_____
F10 string :- Prospero Software_____

```

This command can be used to define the key sequences produced when one of the function keys is pressed. Each function key has a corresponding string of up to 32 characters, defined using this command, so that pressing the function key has the same effect as that sequence of characters. There are several ways in which the function keys can be usefully defined – for example, if typing in a C program, it might be useful to define F1 as producing the keys `int`, F2 producing `char *` and so on. As another example, if you are used to an editor where pressing F3 meant ‘find’, then F3 could be defined to produce the keys `^QF` – the WordStar find command. Placing control keys in the function key sequences is particularly useful – for example the sequence `^QS^G^G^X` will move the cursor to the start of the line, delete the first two characters, then move down to the next line. If a function key is defined to produce that sequence, pressing it repeatedly will cause a group of lines to be moved left two characters.

When entering control characters into editable text fields, the characters displayed will be strange symbols, as in the diagram above, but the effect when the function key is pressed will be that of the corresponding control code. It is not possible to enter some keys into function key sequences – for example the cursor keys, `tab`, `Return`, `Enter` and `Alt` key combinations. The WordStar equivalents are particularly useful here – note also that `^M` and `^I` can be entered into function key strings to mean `Return` and `tab` respectively.

3.1.8.5 Tab width



The **Tab width** command is used to specify the width in characters of the tab stops. When the tab key is pressed, it inserts spaces at the current cursor position until the cursor column is a multiple of this tab width. The tab width selected must be between 1 and 9, and the text of this menu item alters to indicate the selection. Alternatively, a tab width of zero can be requested – this selects '**Special tabs**', intended where information needs to be entered in columnar fashion. Pressing the tab key will repeatedly insert spaces at the current cursor position until the cursor lies below the start of the next word (i.e. a non-space character immediately preceded by a space) on the line above.

3.1.8.6 Auto indent

When **Auto indent** mode is selected, indicated by a check mark placed next to this menu item, pressing return will cause the same number of spaces to be entered at the start of the new line as there are at the start of the line above. This is frequently useful when entering C source, as the required indentation is usually the same as or close to that of the line above. If **Auto indent** mode is already in use, selecting this item will turn it off.

3.1.8.7 Insert mode/Overwrite mode

If this menu item reads '**Insert mode**', characters typed will be inserted into the text without replacing those already there. If it reads '**Overwrite mode**', characters typed will replace those already in the text. Selecting this item toggles the state between **Insert mode** and **Overwrite mode**, and alters the menu text accordingly. The WordStar command ^V and the key marked 'Ins' or 'Insert' have the same effect as selecting this menu item.

3.2 Workbench key combinations

The Workbench is designed to be familiar to as many people as possible, so that, in most cases, pressing a particular key will do what they would expect it to. To this end, the majority of WordStar control key combinations are supported, and have the same effect in the Workbench as they do in WordStar. The Alt key combinations used as a short cut to select menu items are based on the Macintosh command key combinations. Also, any special keys on the keyboard, such as the cursor keys, Home, and Insert are supported.

3.2.1 WordStar style control key combinations

Three types of WordStar type key sequences are supported – the simple commands, which are obtained by holding down the control key while typing a letter, the control-Q commands, which are obtained by typing control-Q followed by a second key to indicate which command is required, and the control-K commands, which are obtained similarly to the control-Q commands, except that they are prefaced by control-K.

The simple commands are organized around the diamond formed by the keys E, S, D, and X – these are the basic cursor movement keys in WordStar, conveniently close to the control key. Slightly further out from the centre of the diamond are keys which move the cursor further – thus A and F move a word at a time, R and C a page at a time. Various other keys further to the right of the keyboard perform sundry other commands. The complete list is as follows :

^E	Cursor up
^S	Cursor left
^D	Cursor right
^X	Cursor down
^A	Word left
^F	Word right
^R	Page up
^C	Page down
^W	Scroll window up (without moving cursor)
^Z	Scroll window down
^V	Enter or leave insert mode
^G	Delete character to right of cursor
^H	Delete character to left of cursor (same as backspace key)
^I	Tab
^L	Repeat find
^M	New line (same as return key)
^Y	Delete line
^N	Insert line
^T	Delete word

The control-Q combinations are concerned with find operations, or moving the cursor to specific points. Typing ^Q before one of the simple cursor commands makes the cursor go further. Thus ^S moves the cursor left one space, ^QS moves the cursor left to the start of the line. There is a similar logic behind most of the other commands in this group (except ^QY, which is a bit of an odd man out, and ^QB and ^QK, which mirror the ^KB and ^KK commands described later). The complete list is as follows :

^QE	Start of page
^QS	Start of line
^QD	End of line
^QX	End of page
^QR	Start of document
^QC	End of document
^QY	Delete to end of line
^QA	Replace
^QF	Find
^QB	Start of block
^QK	End of block

The control-K combinations are concerned with block operations. The WordStar block mechanism is slightly different to the Workbench one, but the following commands have an analogous effect in the Workbench to their original WordStar function :

^KB	Mark block start
^KK	Mark block end
^KX	Cut block (to clipboard)
^KC	Copy block (to clipboard)
^KV	Paste block (from clipboard)
^KY	Delete block
^KH	Hide block
^KW	Write block to disk
^KR	Read block from disk
^KP	Print block

3.2.2 Alt key combinations

Many of the more common menu selections can, as a short cut, be made by holding down the Alt key and typing a letter – frequently the first letter of the menu command. These key combinations are based on the standard (and semi-standard) Macintosh command key combinations. The following Alt key combinations are supported :

- ◆E **Edit .C file**
- ◆S **Save file**
- ◆D **Delete file**
- ◆W **Close**
- ◆Q **Quit**

- ◆C **Copy block**
- ◆X **Cut block**
- ◆V **Paste block**
- ◆H **Unmark block**

- ◆F **Find**
- ◆A **Find and replace**
- ◆G **Goto line number**

- ◆K **Compile**
- ◆J **Compile and link**
- ◆Y **Check syntax (◆Z on German keyboards)**

- ◆O **Compiler options ...**

- ◆L **Link**

- ◆R **Run**
- ◆P **Debug program**

3.2.3 Special key combinations

The standard ST keyboard has a number of special keys, marked with various editing functions such as home, insert, delete and so on, as well as the cursor keys. These keys should all work as expected. If control is held down while pressing one of these keys, the effect is modified, in much the same way as the control-Q combination modifies the WordStar cursor control codes. The following special keys are supported :

Left arrow	Cursor left
Right arrow	Cursor right
Up arrow	Cursor up
Down arrow	Cursor down
Home	Start of window
Insert	Enter or leave insert mode
Delete	Delete character to right of cursor

When typed with the control key held down, the meanings are modified as follows :

^Left arrow	Start of line
^Right arrow	End of line
^Home	Start of text

In addition, the function keys are supported, each expanding to a sequence of characters which can be specified using the **Set function keys** menu command – see section 3.1.8.4.

OPERATION OF THE COMPILER

Desk	File	Block	Find	Compile	Link	Run	Options
<input checked="" type="checkbox"/>				Compile PRIME			<input checked="" type="checkbox"/> K
<input checked="" type="checkbox"/>				Compile and link			<input checked="" type="checkbox"/> J
<input checked="" type="checkbox"/>				Check syntax			<input checked="" type="checkbox"/> Y
-----				Compiler options ... <input checked="" type="checkbox"/> O			
-----				Compile other file			
-----				Cross reference			

```

/* Repeatedly asks for
/* its smallest factor

#include <stdio.h>
#include <math.h>

main ()
{ unsigned int factor, maxfactor;
  unsigned long int number;

  do
  { do
    printf ("\nInput a number up to a thousand million: ");
    while (scanf ("%lu",&number) < 1);
    if (number > 0)
    { printf ("\nSmallest factor of %5lu is : ", number);
      maxfactor = (unsigned int) sqrt ((double) number);

```

An invocation of the Prospero C compiler processes one source file, containing a C translation unit, and converts this into a binary output file consisting of a single module in relocatable format. The source file can be any ASCII text file stored on disk, or can be contained in memory in the Workbench.

Compilation is a 2-pass process under the overall control of the Workbench. Pass 1 (the program C1.OVL) reads the source file and generates a temporary work file, with the name PTEM\$IL2.\$\$\$, which contains a semi-compiled "intermediate-language" representation of the source program. When processing by Pass 1 is complete, the Workbench gives control to Pass 2 of the compiler, the program C2.OVL. This program reads the work file, and generates the relocatable .BIN file. When compilation is complete, Pass 2 deletes the work file and gives control back to the Workbench.

It should be noted that the current default folder, in which the .BIN file (and possibly the .LOG and .PRN files) will be produced, will be the folder from which the source was loaded. If a path for include files has been specified (see 3.1.8.3), this will be added to the start of any include file name which does not begin with a drive specifier or backslash before attempting to open the file.

4.1 Compile-time options

The compile-time options are selected using the following form, which is produced when the **Compiler options ...** command is selected :

Compiler Options

G	Compiler output to LOG file
L	Source listing to PRN file
N	Include source line information
I	Check array indexes
A	Check assignments against bounds
P	Check pointers
S	Accept strict ANSI Standard C only
U	Char is unsigned
C	Generate compact code
W	Wait after errors
V	Autosave after compilation

Clicking on the letters in the left hand column selects or deselects the corresponding options. The letters are those used in other Prospero compilers to select options, and are written to the .LOG file and the .PRN file to indicate which options were set.

The meaning of each option is described in the following sub-sections. The default setting for each option is “off” when the software is shipped, but this can be altered using **Save configuration**, as described in section 3.1.8.

4.1.1 Compiler output to LOG file – option G

When this option is specified, a record of the compilation, with any errors or compiler messages generated, is written to a file, whose name is the same as that of the source, with “.LOG” added. This file can be useful either for inspection of compile-time errors and/or for recording the compilation status of each source file (code size, etc.).

4.1.2 Source listing to PRN file – option L

A listing of the pre-processed source (with all macros expanded) can be generated as a by-product of compilation. Each line is preceded by its line number within the file. The listing is output to a file with the name of the source but ending in “.PRN”, in the same folder as the source. After the compilation it may be printed or displayed as desired. The debugger PROBE will use the .PRN file (if available) to list the source lines as they are executed.

4.1.3 Include source line information – option N

This option instructs the compiler to insert extra code into the object program to maintain during execution a record of the source file name and line number corresponding to the code currently being obeyed. This information will be displayed in the event of any run-time error, and the calling stack which is printed out (see section 6.3) will contain these file names and line numbers (for all calls which occurred in source files compiled with this option).

This option also has the effect of causing stack overflow checking to be performed on procedure entry.

This option is needed if the PROBE symbolic debugger is to be used – see section 8.

When this option is in force, the macro `_NINFO` is predefined (with an empty body).

4.1.4 Check array indexes – option I

This option causes code to be compiled to determine whether or not array index expressions are within the correct limits. The checks are carried out just before an index value is to be used, and can have the effect of generating more code.

Range checks can be valuable in the early stages of program testing. If code size or speed is at a premium, they may be switched off once the program has been tested.

When this option is in force, the macro `_ICHECK` is predefined (with an empty body).

4.1.5 Check assignments against bounds – option A

This option causes checks to be introduced during execution when assignments of values are made and when values are passed as actual parameters (and may have the effect of increasing the size of the generated code). In the case of ordinal-type values, the check is on the range allowed for such quantities.

When this option is in force, the macro `_ACHECK` is predefined (with an empty body).

4.1.6 Check pointers – option P

This option causes checks to be inserted at each “pointer dereference” (*p) in the program. The check will detect any attempted use of a pointer which has been set to NULL, and therefore has a good chance of picking up cases where no value has been assigned.

When this option is in force, the macro `_PCHECK` is predefined (with an empty body).

4.1.7 Accept strict ANSI C Standard only – option S

When this option is invoked, the compiler disables use of the non-standard features of Prospero C, namely:

\$ permitted in identifiers,
pascal and fortran keywords

It also has the effect of checking the code more carefully for dubious or non-portable constructs, such as calling a function without a prototype, or failing to return a value from a function. Several less serious errors which produce warnings when the S option is not invoked become errors when it is. If a program is to be transferred to a different C implementation, this check helps to pick out any points which may call for attention.

When this option is in force, the standard predefined macro `__STDC__` is defined, expanding to the constant 1. If it is not in force, the macro expands to the constant 0.

4.1.8 Char is unsigned – option U

By default, objects declared as plain char are sign extended when used in an expression, i.e. plain char is equivalent to signed char. If the U option is used, plain char objects do not sign extend in expressions, and plain char is equivalent to unsigned char.

When this option is in force, the macro `_UCHAR` is predefined (with an empty body).

4.1.9 Generate compact code – option C

If the compact code option is invoked, the compiler substitutes shorter (but somewhat slower) alternatives for certain object code sequences. The amount of difference this will make depends on the nature of the program. Use of the option would only be recommended for particularly large programs.

When this option is in force, the macro `_COMPACT` is predefined (with an empty body).

4.1.10 Wait after errors – option W

This option is somewhat different from the others, as it concerns the way the compiler (and linker) behave when an error or warning is encountered. If the option is on, each error reported will cause the compiler to wait until either **Abort** or **Continue** is selected. If it is off, the compiler will pause after each error report to allow the user to abort, but if **Abort** is not selected within a short time compilation will continue, so that a report of all errors can be generated in the LOG file without having to click **Continue** after each one.

4.1.11 Autosave after compilation – option V

This option is not strictly a compiler option, but rather it instructs the Workbench to automatically save a file after successful compilation. This option is recommended because if the program runs and crashes, the sourcefile may be lost from memory. If compilation is unsuccessful, no action is taken. The file is saved with a date stamp corresponding to the start of the compilation (and is thus earlier than the .BIN file produced) to assist in the use of MAKE like utilities.

4.2 Check syntax

If this command is used, the compiler Pass 1 is invoked as normal on the source in the current top window, but will not produce any intermediate code. Selecting this option speeds up compilation considerably, and would be recommended if a source is known to contain errors (perhaps because it is being imported from another system), or if the only purpose of the compilation is to generate a source listing (.PRN) file, for example. The compiler still uses the current options as described above, although not all are meaningful when no code is produced. If a .LOG file or .PRN file is produced, the list of options used will include option Y, to indicate that it was produced by a syntax check rather than a full compilation. When using the command line version of the compiler, option Y can be specified to perform a syntax check only.

4.3 Command line version

The C compiler can also be driven independently of the Workbench, using the supplied program program C.TTP. This might be useful when used in conjunction with a command-line processor's batch-file or Make facilities, or when compiling very large programs where the compiler runs out of memory when executed from the Workbench. The command line version of the compiler can be driven in two ways.

4.3.1 The one-line command

This is when a command tail containing the source filename is specified. If no extension is given, the default .C is supplied automatically. The filename may optionally be followed on the command line by the character "/" together with one or more letters, as in:

```
B:PRIME/LM
```

Each letter stands for the corresponding compile-time option (see section 4.1). The letters may be run together, as in this example, or may be separated by spaces, commas, or further / characters. It makes no difference whether they are in upper or lower case.

4.3.2 Conversational mode

If no command tail is specified, a conversational mode of operation is entered. The first request is for the name of the source file, the response being e.g. B:PRIME, terminated by Return. If no filename extension is given, the default .C is supplied automatically.

There is then a series of prompts enabling some or all of the options to be altered for this compilation. There are three possible responses to each question in the list:

- Y or y to select the option
- N or n to reject the option and go to the next
- . to terminate the prompting and use the defaults from then on.

(Note that any characters other than these are ignored, and that it is not necessary to press Return for the reply to be accepted.) When the list is terminated, or the end is reached by Y and N responses, the compilation process begins.

4.3.3 Making use of variables

Three environment variables are inspected by the C compiler, for determining the location of include files, temporary files and the compiler overlays. A further environment variable (DEFINE) may be used to pre-define macros, and is described in section 4.3.4 below. These environment variables are normally set using the SET command in a command shell. Several paths may be specified, separated by semi-colons.

Thus

```
SET INCLUDE=C:\PROC\H\C:\MYPROG\INC\
```

directs the compiler to search the current directory, and then the paths C:\PROC\H and C:\MYPROG\INC for include and header files.

Similarly, the compiler will place temporary files in the first path specified by the environment variable TMP, or use the current directory if no path is specified.

As with other programs, the environment variable PATH will be used to search for the compiler overlays.

4.3.4 Command-line macros

When operating the compiler from the command-line using C.TTP in the one-line mode, it is possible to pre-define macros using the /D option. The letter /D in the options part of the command tail is followed by the name of a macro to be defined, followed optionally by an '=' or space and then the body to which it should expand. The body is terminated by either the end of the command-line or by a '\ ' character. Multiple macros can be defined by separating them with semicolons, or by specifying the /D option several

times. The macros are defined exactly as if, before the start of the source, there appears the statement

```
#define <macro> <body>
```

This may provoke error reports from line 0 if, for example, the macro name is badly formed, or a macro is specified more than once.

The environment variable DEFINE may also be used to specify macros to be predefined, using the same format.

4.4 Compiler messages

When the compilation process begins, messages are output to the form titled **Compiling..** (or to standard output, for the command-line version) and optionally to the log file, to report progress and any irregularities. The source file name, current line number and last error are shown, as in the example below :-

Compiling B:\SOURCES\MANDEL

Last Error :- Line no :- 35

```

iterations++;
  ^
Error : Identifier not declared - assumed 'extern int'
iterations

```

Errors in the source program may be detected during either of the passes, though the majority generally appear in Pass 1, which can also generate warnings. Each error or warning is reported in the progress form, and compilation pauses, for a short time if option W (Wait after errors) was not selected, or until the user clicks on **Abort** or **Continue** if it was. In either mode, the user may click on **Abort** at any time to return to editing – if an error has been reported, and the line in question is in the top window (rather than in an included file, or being compiled from disk), then aborting will cause the text cursor to be positioned at the line where the compiler detected the error.

Each error is reported in the following format: Either **Error :** or **Warning :**, followed by an explanatory message if the file C.ERR is found, or an error number if it is not, preceded by the text of the line in error (Pass 1 errors only). In Appendix B of this manual is a list of the error codes, with

somewhat fuller descriptions where appropriate. If option G is in force, the error message will also be written to the .LOG file

A single error, as the programmer sees it, may sometimes give rise to a number of reports. An obvious instance is a missing declaration, which will be signalled at each reference. It is also possible for one error to have a “cascading” effect. Large error counts should, therefore, not be taken at face value.

If Pass 1 is error free, Pass 2 will be executed to translate the intermediate work file into a .BIN file ready to be linked. The line number is not updated during Pass 2, but the user can still abort if desired.

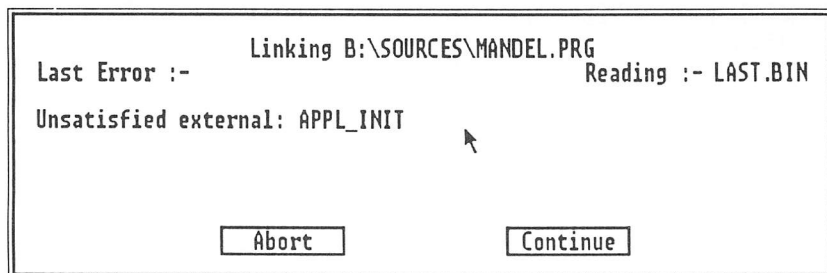
When operating the compiler from the command-line, the following status codes are returned, indicating the success or otherwise of compilation:

Status	Meaning
0	Clean compilation without errors or warnings
1	Compilation completed, but there were warnings
2	Compilation failed due to errors

5 OPERATION OF THE LINKER

The linker processes a sequence of one or more files in relocatable object format and combines them into an executable program file.

During linking, messages are output to the form titled **Linking...** to report progress and any errors etc., and allow the user to interrupt the process if required. The executable file name, current object file and last error are shown, as in the example below :-



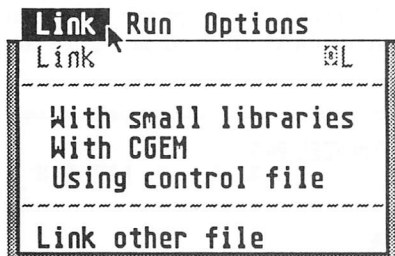
The linking process can be interrupted at any stage by clicking on **Abort**. If errors are reported, the linker will use the Wait after errors option (see section 4.1.10) to determine whether to wait for a button to be selected before continuing, or to simply pause for a while before continuing.

The linker builds a .PRG file in the normal GEMDOS format: a 28-byte header followed by the Text Segment image.

All addresses and values output, or requested, by the linker are in units of bytes and in hexadecimal notation.

5.1 Simple use of the Linker

For most programming requirements the linking process is very simple. The Link menu presents two ways of selecting the file to be linked, and a selection of optional libraries:



5.1.1 Link

The **Link** option links the current top window (which will be shown in the menu), and is only available if the file has been successfully compiled since it was last edited. The other options on the menu determine which libraries are included.

5.1.2 With small libraries

The small libraries (CFIRSTO.BIN and CLIBO.BIN) can be used by a program which uses no floating point operations of any kind, and can significantly reduce the size of the program generated. Note however that using the small libraries when the floating point routines are in fact used may not give rise to any errors at link time, but cause unpredictable behavior when the program is executed. If in doubt, it is better to use the standard libraries, by not selecting this option.

5.1.3 With CGEM

Selecting this item causes the CGEM.BIN library to be included in the link. The CGEM library is required for linking programs which make calls to the GEM AES or VDI bindings described in the AES and VDI manuals – if this library is required, but not included in the link, the Linker will report undefined external symbols, as in the example above, where CGEM has inadvertently been missed out. The .PRG file produced is unlikely to be usable in this case, so it is usually best to click **Abort**, then restart the link with the correct libraries.

5.1.4 With control file

This option specifies that a control file is to be used to specify the files to be linked – this might be used when several separately compiled modules are to be linked together, or if non-default linker options are to be used. The name of the control file to be used in subsequent links is specified using the file selector form. Link control files are described in detail in section 5.2.

5.1.5 Link other file

If **With control file** is selected, the files to be linked are defined by the contents of the control file, and this option is identical to **Link**. Otherwise, a filename is specified in the usual way using the file selector form, and this file is linked using the libraries specified by the options described above.

The remainder of this chapter concerns advanced use of the linker and may be skipped by most programmers.

5.2 Linking using a control file

The linker can be used in a more powerful way by driving it with a control file. This is a text file created by the editor with an extension of `.LNK`, which is read by the linker to specify what files are to be linked, and various other options. To link using a control file, the **Using control file** item from the **Link** menu should be selected. This presents the file selector form so that the link control file can be nominated, and disables the **With CGEM** and **With small libraries** items. The menu is altered to give the name of the link control file to be used. Selecting this item again reverts to the simple mode of linking described above, and enables the **With CGEM** and **With small libraries** items.

The control file consists of 4 sections, each of which should start on a fresh line, with no blank lines between or within sections. Note that upper and lower case are not distinguished in the control file – all the examples below are given in upper case, but would work equally well in either.

Section 1 The name and optional drive and path of the executable program file to be produced. If no extension is given, `.PRG` is supplied by the linker.

E.g. MYDEMO.TTP

Section 2 This section can be used to alter the stack allocated to the program (see section 5.2.1). If the section consists of the single line 'N', the default stack size (4 K) will be allocated. If the first line is 'Y', the next line should contain the required stack size, in hexadecimal.

E.g. Y
4000

would allocate a 16 K stack.

Section 3 This section is used to specify whether the linker is to produce any link map reports (see section 5.2.2). The first three lines should contain either 'Y' or 'N', indicating whether the linker should produce a module map, symbol map and section map respectively. If the answer to any of the above is 'Y', there should be a further line in this section, again containing 'Y' or 'N', specifying whether the linker is to include names beginning with '.' in the maps. These names are reserved for use by the Prospero C system, and are unlikely to be of great interest most of the time. The maps are written to a file whose name is the same as the executable file being produced, with a `.MAP` extension. This can be examined using the Workbench or printed out as required.

E.g. Y
N
Y
N

would produce a module map and segment map, with '.' names not included.

Section 4 This section specifies what object modules are to be linked to produce the executable file. This consists of a list of object module specifications, separated by spaces, commas, or newline characters, and terminated with a full stop. Usually this will be the names of all the object files and libraries to be linked; the module CFIRST0 (or its alternative version FIRST0) must appear at the start of the list, and the module LAST must be at the end. Note that the default folder will be that containing the control file, so that the filenames of the CFIRST and LAST object files and the standard libraries will usually require a path specifier. Any libraries being linked must have a /S qualifier to indicate that they are to be scanned selectively. Further details of object module specifiers are given below in section 5.2.3.

E.g. \PROC\CFIRST
PRIME
\PROC\CLIB/S
\PROC\LAST

5.2.1 Stack allocation

The linker will allocate a stack of 4K, unless a link control file has been used which specifies the stack size to be used – see above.

5.2.2 Linker maps

The linker can optionally produce maps of the executable file organized by module, by Public/External names and by sections – see above for details of the link control file.

The module map consists of address and other information for the various sections which contribute to each module's storage requirements.

The symbol map consists of the addresses, relative to the start of the Text Segment of all Public or External symbols resolved by the linker, printed both in alphabetical and in numerical ordering.

The section map indicates the address and size of each section and common block produced. For sections, the address is relative to the start of the Text Segment (i.e. the code image); for common blocks, the address is relative to the start of the block of memory acquired at program start-up for all the common blocks together.

5.2.3 Module Specifications

The final section of the control file consists of line(s) containing the names of all object files to be linked, separated by commas or spaces, and terminated with a full stop. Each file name may be followed by a “module selector” to specify that only some of the modules are selected. (The default is to select all modules from the file.) For this purpose, two kinds of “selector” are provided.

The first kind is the “selective scan” of an input file, and is specified by following the filename with the two characters /S. Only those modules that have been referenced by previously selected modules will be incorporated into the executable file (and so into any reports). This should be used for all libraries being linked – in particular all control files will normally contain the specifier CLIB/S or CLIBS/S, immediately before the LAST object file.

When reading an input file in the “selective scan” (/S) mode, the linker and the librarian (see section 7) adopt identical selection criteria.

The second kind of selection is by “module enumeration”, and is specified by following the filename with the character “[”, then a collection of module names, and finally the character “]”. This “collection” of module names is to be written as a list of names, separated by commas; optionally, in place of a module name, the list can contain, at any point, two names separated by “-” (i.e. name1-name2), signifying “all modules from name1 to name2 inclusive”. Note that the case of the letters in the module name is not significant.

Example: FNAME1 [MOD1, MOD4-MOD8, MOD16]

A particular filename can be followed by at most one of these two kinds of selector.

An example of an input line containing all the above features is:

```
FNAME1, FNAME2 [M6], FNAME3 [MOD3-MOD9], LIBNAME/S
```

5.3 Command line version

The linker can also be driven independently of the Workbench, using the supplied program PROLINK.TTP. This might be useful when used in conjunction with a command-line processor batch processing or Make facilities. The command line version of the linker can be driven in three ways.

5.3.1 The one-line command

This is when a command tail containing all the required information is specified. The command tail consists of a specification of the executable filename, and then one or more input object filenames. The executable filename must be followed by the '=' character; if no extension is given, the extension .PRG is supplied by the linker.

As an example:

```
A:PRIME=CFIRST, B:PRIME, CLIB/S, LAST
```

The input filenames are separated from one another by commas and/or spaces; if no extension is given, .BIN is supplied by the linker. Any of the input filenames may be followed by a "module selector" (see 5.2.3).

The most usual such "selector" consists of the two characters /S, to indicate that a "selective" scan of that file is to be made, i.e. that only those modules are to be incorporated that have been referenced by previously-encountered modules. (In this context, a "module" is the output from one C compilation, or the output from one execution of an Assembler.)

In the case of the C run-time library, a selective scan must always be specified, i.e. CLIB/S. The file CFIRST (or CFIRST0) must be the first file input, and LAST must be the last file input, immediately preceded by the run-time library CLIB (or CLIB0). CFIRST and LAST should not be selectively scanned.

5.3.2 Conversational mode

If no command line is specified, the conversational mode of operation is entered. The name of the executable file is first requested, then there is a series of questions relating to link-time options, followed by an invitation to input one or more lines containing filename(s). The link time options are described in detail in section 5.2.

5.3.3 Indirect mode

To operate the linker using the indirect mode, a command tail of the form

```
^filename
```

should be supplied, where “filename” is the name of a linker command file, as described in section 5.2. Note that if “filename” does not have an extension, none is supplied by the linker. The “^” character may be optionally followed by spaces.

5.4 Linker messages

5.4.1 Non-fatal errors

The most common situation leading to an error message is when all the input files have been processed and yet there are still External references outstanding for which no corresponding C function or external object (or Assembler Public name) has been encountered. This may be because a .BIN filename has been inadvertently omitted from the list in a control file, or the program has used GEM calls, but **With CGEM** was not selected. The message

```
Unsatisfied external:
```

is printed on the form, followed by the unmatched names. If the Wait after errors option is in force (see section 4.1.10), the linker will wait until either **Abort** or **Continue** is clicked – otherwise there will be a short pause before continuing.

If the user does not abort, the linking process will continue to its conclusion. In particular, an executable file will be produced which is normal except that any location containing a reference to a missing routine will not contain a sensible value. Caution should therefore be exercised if execution of the program is attempted: a run-time error Y can occur.

Other non-fatal errors that may be reported are as follows :

If a character other than ‘s’ is supplied after ‘/’, the linker reports this, and ignores the spurious character.

5.4.2 Fatal errors

If any other error situation occurs, continuation is not possible. A message is output to the form describing the problem, and the linker will wait until either Abort or Continue is clicked (if Wait after errors is in force – see section 4.1.10) or there will be a short pause (if it is not) before aborting. An alert indicates that the link was not successful, and no usable .PRG file will have been produced.

The first group of such messages are caused by driving the linker incorrectly. There are 5 of these:

Command file not found

When **Using control file** is in force, the specified control file name is illegal or the file does not exist.

No executable filename supplied
Object file not found
Illegal Stack size

These are most likely to be caused by errors in the linker control file, or by attempting to link (perhaps using **Link other file**) a non-existent object file.

Illegal module-selection syntax

The rules given in 5.2.3 have been broken. In particular, “-” must have a module name on either side of it, and “[” must have a matching “]” on the same line.

The second group of errors are when the linker is unable to continue execution. There are 3 of these:

Not enough memory

The linker has run out of work space (for its symbol tables etc.).

Executable file too big

Exceeds 16 Mbytes.

Disk/DOS error

Disk full, for example.

The third group of errors are most likely due to presenting the linker with a mutually inconsistent set of .BIN files:

Public name defined more than once

For example, two functions with the same name.

SECTION/Common inconsistency

For example an Assembler SECTION directive using the same name as a Pascal COMMON variable name, in a mixed language program.

The fourth group of errors are because a linker restriction has been violated. The messages are:

Absolute ORG not supported

Attempt to preset COMMON

The fifth group of errors should never occur. The most probable explanation is that an input file is not in the appropriate relocatable object format at all. The error messages are:

Cannot find section

Cannot find subsection

Cannot find symbol

End of input file encountered

Illegal directive encountered

Illegal id encountered

Illegal XREF relocation encountered

Input file relocatable format incorrect

Name in input file exceeds 32 characters

XREF expression out of range

The last group of errors should again never occur, and could indicate a linker malfunction:

Linker internal error

Paging error on .PRG file

6 OPERATION OF OBJECT PROGRAMS

The operation of an object program under GEMDOS is determined very much by the program itself. Programs can be run using the Workbench **Run** menu (see section 3.1.7), or from the GEM Desktop or other command shell.

6.1 Arguments to main

The object program starts up in the function called `main`. This may be declared without parameters, or with two parameters, usually known as `argc` and `argv` (you can call them what you like), whose contents are defined from the parameters passed to the object program in its command tail. The first parameter, `argc`, has type `int`, and describes the number of command strings passed. The second parameter `argv` is an array of pointer to `char`, containing pointers to null terminated strings containing the parameters themselves. `argv[0]` points to a null string – this would contain the name of the program if it were made available by the operating system. `argv[argc]` contains a null pointer. The semantics determining the parsing of the command tail into strings is described in Appendix H.

If the `main` function returns a value, this value will be passed to the parent process as the return code. If it returns without a value, the return code will be undefined.

6.2 Pre-connected files

Default assignments of files are as described in Part II. In particular, the standard C streams `stdin` and `stdout` are directed to standard input and output, i.e. to the keyboard and screen, respectively.

If a program uses the `spawn` facility to run a child program, its own standard streams `stdin` and `stdout` are automatically made available to the child program (through any number of parent-child levels).

6.3 Run-time errors

The only aspect of program operation not determined from the program itself arises if an error is detected by the run-time software.

Once program execution proper has commenced, errors may be detected in a number of situations: division by zero, floating point overflow and so on. In some cases they may be found by the checking code incorporated by one of the compile-time options (see section 4.1). In all cases a report is made on the console, giving error type – identified by a letter – and the hexadecimal machine address relative to the start of the code:

Error x at address aaaaaa

A list of the run-time error codes is given in Appendix C. The address aaaaaa is directly comparable with the addresses provided in the .MAP file generated by the Linker. For some errors, additional information appears with the standard message.

The standard error message is followed by trace information showing how the point in error was reached. This takes the form of a list of addresses at which function calls occurred. All addresses are relative to the start of the code. The first address given corresponds to the point where the main function called the next lower level of function, and so on up to the actual function in error.

For each source file which was compiled with the N (“track source line numbers”) option, the addresses in the error report are made more intelligible (without recourse to the linker’s .MAP file) by the addition of the source file name, the function name and the line number.

Finally, many classes of error allow continuation. In these cases, the message

Continue? (Y/N)

appears on the console. The program can be continued by pressing the key Y (or y), or aborted by pressing N (or n). (All other keys are ignored.)

6.4 Miscellaneous error messages

If a program was executed from the GEM Desktop, these messages will appear on the screen. If a program was initiated using `spawn`, the error is passed back as a return code to the initiating program. If such a return code is detected when running a program from the Workbench, the error is reported in an alert box.

Execution error: <error text>

where <error text> is one of the following:

wrong version (spawn... return code -6)

There is an inconsistency between the version of the library or the library header linked and the generated code. Most likely to be caused by upgrading to a new version of the compiler but using the old versions of the library, or vice versa.

out of memory (spawn... return code -7)

Insufficient memory is available for loading and/or running the user program.

`init. failure` (spawn... return code -8)

The program has successfully been loaded, but then an error occurred in one of the pre-execution steps:

- (a) processing the relocatable items in section `.ATAB`,
- (b) processing the data-initialization items in section `.INIT`

If the program consists purely of C code, this error implies a problem with the C software, and it should be reported. If user-provided assembler language routines were included in the link of the user program, they should be checked to ensure that:

- (a) They do not use section `.INIT`.
- (b) They only use section `.ATAB`, if at all, as described in Part II section 5, namely for achieving the relocation of external object addresses and of `JMP` instructions having 4-byte absolute operands. In particular, this error can be caused quite easily by not preceding the assembler instructions by a suitable `SECTION` directive, so that they become part of `.ATAB` instead.

It can quickly be verified whether or not assembler routines are the cause of this error, by linking the user program without them, then loading it. The program will then fail during execution, rather than during initialization.

`no parent pgm` (spawn... return code -9)

A program which is designed to be used only as an overlay, and share some of its parent's library code, will give this error if executed directly from the Desktop.

`stdio failure` (spawn... return code -10)

Unsuccessful attempt to read or write a standard file.

`shrink memory failure` (spawn... return code -11)

When a program starts execution, the run-time library initialization attempts to release the memory not required by the program. If this operation fails, this error results. (May be due to corruption of memory, or an invalid program file header.)

`linking order` (spawn... return code -12)

Unlikely to occur, but indicates an erroneous attempt to link with `.BIN` files generated by other compilers or assemblers.

7 OPERATION OF THE LIBRARIAN

The librarian is a standard GEMDOS executable program, which does not make use of GEM. It can be operated from the Workbench, but is not fully integrated (due to memory and disk space limitations). It should be run using **Run other file** to call the program PROLIB.PRG (with **Run under GEM** not selected), or from the GEM Desktop or other command shell. The **With command tail** option can be used to enter the command tail, or PROLIB can be run interactively if no command tail is specified.

The purpose of the PROLIB librarian utility program is to administer files which are in relocatable object format – such as those produced by the Prospero C, Prospero Pascal or Prospero Fortran compilers, or by various assemblers. Individual modules may be extracted, and/or files may be merged together into libraries. A number of report options are also available.

A file created by PROLIB will be in the same relocatable format, and so suitable for processing by PROLINK or other linkers capable of handling this format (such as GST's LINK).

7.1 Forms of invocation

There are three ways of operating the librarian: the “one-line”, the “conversational” and the “indirect” mode. All the options are available in each mode.

7.1.1 The one-line command

The command tail must be constructed as follows : First must come the name of the “library” file. This may optionally be followed by the character “/” together with one or more letters, as in:

```
B:PRIME/MX
```

Each letter stands for a particular option regulating the report(s) that are produced by the librarian (see 7.2). The letters may be run together, as in this example, or may be separated by spaces or further “/” characters; they may be in upper or lower case.

A one-line command of the above form indicates a “read-only” operation on the library file: the file must already exist, and the purpose of the PROLIB execution is solely to list certain information about this relocatable file.

Alternatively, the library filename (and any option letters) may be followed by an “=” sign and one or more input filenames, separated by commas, as in:

```
NEWLIB/M = MOD1, MOD2
```

A one-line command of this form indicates a “create” mode of operation: if the library file already exists it will be overwritten, and the purpose of the PROLIB execution is to combine the input filenames into a new library with the given name. (The librarian actually creates the file, in the first place, with an extension of .\$\$\$, and only renames this to the required library filename on successful completion of processing.) Any of the input filenames may be immediately followed by a “module selector” (see 7.3).

A variant of this “create” mode is to omit the library filename but still specify report(s). The latter will be produced as usual but no actual library file will be written. For example:

```
/MXD = MOD1, MOD2
```

If no filename extension is given (whether for the library or the component input file names), the extension .BIN is supplied automatically by the librarian.

7.1.2 Conversational mode

If no command tail is specified, the conversational mode of operation is entered.

The first request is for the library filename. A filename can be supplied, or simply c/r (Return) on its own to indicate that report(s) but no output library file are required. There is then a series of questions relating to the report options (cf. 7.2). Reply Y (or y) to select the option, otherwise N (or n).

The final question (which is only asked if a library filename has been supplied) is whether or not to create a new library with the given filename. If the answer is affirmative, or if no library filename was supplied, the librarian repeatedly issues an invitation to input a line containing filename(s). The filenames are entered just as for the one-line mode of operation, that is, they must be separated by commas and each may be followed by a “module selector”. To terminate this process, respond to the prompt

```
Input filename(s) -
```

with a full-stop (.) character, or with just c/r (Return) on its own.

If no filename extension is given (whether for the library or the input filenames), the extension .BIN is supplied automatically.

7.1.3 Indirect mode

The indirect mode of operating the librarian combines the features of the first two modes: PROLIB is executed with a command tail containing the name of a “command file” (preceded by the character ^ and optional spaces), this command file containing the answers to the questions which would be asked in the “conversational” mode.

For example, the command tail

```
^ MLIB
```

where the text file MLIB contains the lines

```
MLIB
N
N
N
Y
M1LIB, M2LIB, M3LIB
.
```

causes PROLIB to combine the modules from the 3 files M1LIB.BIN, M2LIB.BIN and M3LIB.BIN into the composite library file MLIB.BIN. Note that if the command file name has no extension, none is supplied by PROLIB.

7.2 Report options

The various report options are described in the following sub-sections. Each sub-heading contains (in brackets) the associated letter which must be written after the library filename in the one-line form of execution in order to invoke the option.

7.2.1 Module listing (M)

A report is produced which gives, for each module in the library file (in order of occurrence within the file), the name of the module, the Sections it contains, and all Public symbols defined and External symbols referenced within it. The “sections” are pieces of the code or data which go to make up an executable program; their sizes (in decimal) are printed.

7.2.2 Cross-reference listing (X)

The report consists of two parts. The first part gives, for each Public/External name in the library file (in alphabetical order), the name of the module in which it is defined (i.e. is a Public name) plus the names of all modules in which it is referenced (i.e. is an External name).

The second part is a listing of all Sections (in alphabetical order) together with the names of the modules which reference them.

7.2.3 Unsatisfied references listing (U)

This report is concerned with the requirement imposed by PROLINK (along with many other linkers) that, for a library which is to be “selectively” searched (cf. the /S option described in section 5.2.3), the component modules must be ordered in such a way that, if module A contains an external reference to an entry point in module B, then module B must follow module A in the library file. The report lists all External names (in alphabetical order) which do not obey this rule, either because they are defined in an earlier module or because they are not defined at all.

7.2.4 Suppress ‘.’ names (N)

(This option is only meaningful if at least one of M, U or X has been selected.)

In order to avoid conflict with user-defined names, most Public and Section names in the C library begin with ‘.’. Since they are rather numerous, it can on occasion be desirable to suppress these. By specifying this option, no name beginning with ‘.’ will appear in the report(s). The default is that all names, including those beginning with ‘.’, are listed.

7.2.5 Listings to disk (D)

(This option is only meaningful if at least one of M, U or X has been selected.)

The default destination for reports is the console. If this option is chosen, the reports are written instead to a disk file. The file is given the same name as the library file, but with the extension .PRN. If no library filename was given, the reports are written to \$\$\$\$\$\$.PRN.

7.3 Module selection

In the “create” mode of operation (only), the user may specify that only some of the modules in an input file are selected. (The default is to select all modules from each file.) For this purpose, two kinds of “selector” are provided.

The first kind is the “selective scan” of an input file, and is specified by following the filename with the two characters `/S`. Only those modules that have been referenced by previously selected modules will be incorporated into the output library file (and so into any reports).

Example: `FNAME/S`

The second kind is by “module enumeration”, and is specified by following the filename with the character “[”, then a collection of module names, and finally the character “]”. This “collection” of module names is to be written as a list of names, separated by commas; optionally, in place of a module name, the list can contain, at any point, two names separated by “-” (i.e. `name1-name2`), signifying “all modules from name1 to name2 inclusive”.

Example: `FNAME1 [MOD1, MOD4-MOD8, MOD16]`

Note that the case of the letters in the module-name is not significant.

A particular filename can be followed by at most one of these two kinds of selector.

An example of an input line containing all the above features is:

```
FNAME1, FNAME2 [M6], FNAME3 [MOD3-MOD9], LIBNAME/S
```

When reading an input file in the “selective scan” (`/S`) mode, the librarian and the linker adopt identical selection criteria. Use may be made of this to obtain an analysis of the composition of a fully-linked `.PRG` file. Suppose, for example, one wishes to know which modules from `CLIB` are needed by the `PRIME` program referred to in section 2 above. Execution of `PROLIB` with the command tail

```
TEMP/M=FIRST, B:PRIME, CLIB/S, LAST
```

will produce a report giving details of `PRIME.BIN` itself and of `FIRST`, `LAST`, and all the contributory modules from `CLIB`. At the same time, the relocatable file `TEMP.BIN` is produced. This file can subsequently be made the object of other reports, for example by executing `PROLIB` with the command tail:

```
TEMP/X
```

7.4 Librarian messages

7.4.1 Normal messages

If in the “create” mode, when it starts to process each input file the librarian writes the full filename to the console.

7.4.2 Error messages

7.4.2.1 Non-fatal errors

If an input file cannot be found (perhaps because its name has been misspelled), the librarian reports this and invites more filename(s).

If a character other than ‘s’ is supplied after ‘/’ following an input filename (i.e. where a “selective scan” directive is anticipated), the librarian reports this error and ignores the incorrect character.

7.4.2.2 Fatal errors

If any other error situation occurs, execution is aborted immediately, after outputting a message to the console.

The first group of such messages are caused by driving the librarian incorrectly. There are 5 of these.

Command line improperly terminated

In the one-line command mode, the library filename and switches have been read, followed by a character other than “=”.

Command file not found

In the indirect mode, the filename after the ^ character is illegal or the file does not exist.

No library filename supplied

In the indirect mode, the first line in the command file should contain a valid GEMDOS filename.

Library file not found

A report has been requested for a nonexistent library file.

Illegal module-selection syntax

The rules given in 7.3 have been broken. In particular, “-” must have a module name on either side of it, and “[” must have a matching “]” on the same line.

The second group of errors are when the librarian is unable to continue execution. There are 2 of these:

Not enough memory

The librarian has run out of work space (for its symbol tables, etc.).

Disk/DOS error

Disk full, for example.

The third type of error is because a librarian restriction has been violated. The only message of this type is:

Absolute ORG not supported

The fourth group of errors should never occur. The most probable cause is that an input file is not in the appropriate relocatable format at all. The error messages are:

Cannot find section

Cannot find subsection

End of input file encountered

Illegal directive encountered

Illegal id encountered

Input file relocatable format incorrect

Name in input file exceeds 32 characters

SECTION/Common inconsistency

The remaining error should again never occur, and may indicate a librarian malfunction:

Librarian internal error

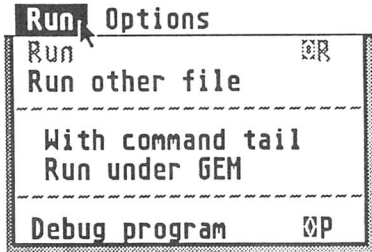
8 THE SYMBOLIC DEBUGGER

A source level debugger PROBE is supplied with Prospero C. This provides display of source program lines as they execute, together with inspection and modification of program variables. It is also possible to watch variables as they change, or halt the program when particular conditions hold.

Probe is simple in operation, and includes an on-line help facility. Since Probe operates in source language terms, no additional knowledge of assembler code or operating system is required. Probe is useful not only for finding programming errors, but provides general insight into the run-time operation of programs. This is particularly useful for programmers new to compiled languages. On the other hand, if machine level access is required, then Probe can be used in conjunction with SID or any other machine level debugger.

8.1 General description

Probe can be called by selecting **Debug program** from the Workbench Run menu :



or by use of the command ♦P. The File Selector form is presented from which the program to be debugged can be selected. The system then returns to a GEM or TOS type screen and control passes to Probe. Probe executes the program selected; this program must have been compiled with the compiler N option and preferably with L as well, in order to provide source line information to Probe.

Probe does not itself use GEM, but can be used to debug programs which do. When using Probe on a GEM program, screen switching can be used to direct the output from the program to a different memory area from Probe's output. As Probe is a stand-alone program, and not integrated into the Workbench, it can be run from the GEM Desktop when memory is short.

Probe uses the supplementary files produced by the compiler when the compilation options L and/or N are specified. These are the source listing (.PRN) file (L option), and the name (.NAM) and symbol (.SYM) files (N option). Probe will make use of any of the three supplementary files which are present at run time, although their presence is not required for program execution. Only modules compiled with the N option are “visible” to Probe. If no part of a program was compiled with the N option, then the entire program becomes invisible to Probe. Probe can operate with any number of separately compiled source files.

Probe operates by checking the state of the object program at the start of each statement. Only executable (rather than declarative) statements are checked. Source lines which have such checkpoints are marked in the listing file with an asterisk. Whenever Probe stops a program at a checkpoint, the source file name, function name and source line are displayed on the screen, and a debug command is requested with the Probe prompt '>>'. Probe always interrupts the program at the start of the very first executed line of the program. The program may also be stopped, at any time when not itself requesting console input, by striking a key on the keyboard.

After the prompt, commands are available to display the last few lines executed (ROUTE command), the nest of active lines (CALLS command), or any region of the source program (LIST command). Any variables active in the program can be displayed (DISPLAY command), or modified (ASSIGN command). Execution can be resumed for a number of source lines (STEP command), or indefinitely (GO command). Execution can also be halted at particular source lines, or when a variable meets a given condition (BREAK command). These circumstances can alternatively simply cause display of the source line and variable value with execution continuing (WATCH command). Provided that the supplied PROBE.HLP file is present, help is available for any of these commands (HELP command). The dialogue with Probe can be recorded in a disk file (ECHO command), and execution can be terminated with the QUIT command. Screen switching can be enabled or disabled by means of the OUTPUT command, and the alternate screen image to which the program's output has been sent can be examined using the VIEW command. The X (execute) command allows any other program to be executed from within Probe, without disturbing the program under test.

Note that the monitoring of several conditions can substantially reduce the execution speed of a program. If extensive computation occurs before the program part to be tested, it is better not to enable monitoring until the uninteresting part has executed.

Probe is independent of the normal run time error handling, which will proceed exactly as it would if the program were not being Probed.

8.2 Sample session

The following sample session shows Probe in use to test a program. Familiarity with Probe is best gained by hands-on use, and a session similar to that below can be used. The supplied example program PRIME is used for the example. Input is shown in lower case.

```

Probe Symbolic Debugger version mg 1.2
Copyright (C) 1987 Prospero Software
(Type H for Help)
main (source file: PRIME)
PRIME.PRN' not found. Which directory is it in? b:
  189 { unsigned int factor, maxfactor;
>>d factor
      0
>>go factor

Input a number up to a thousand million: 37

Smallest factor of 37 is :
  203 *      factor = 1;
  204 +      do
          'g factor'          1
  205 *      factor += 2;
>>route
Last 7 starred lines:
      main
  189 * { unsigned int factor, maxfactor;
  194 *      printf ("\nInput a number up to a thousand
million: ");
  195 *      while (scanf ("%lu", &number) < 1);
  196 *      if (number)
  197 *      { printf ("\nSmallest factor of %lu is : ",
number);
  198 *      maxfactor = (unsigned int) sqrt ((double)
number);
  201 *      if (number % 2)
  203 *      factor = 1;
>>display
      factor      (unsigned)      1
      maxfactor   (unsigned)      6
      number      (uns long)     37
>>go
Prime.

Input a number up to a thousand million:
..

```

8.3 General guidance on using Probe

8.3.1 GEMDOS aspects

Probe uses the GEMDOS “execute program” function to initiate programs under test. C programs compiled with the N compilation option search for Probe when they first start up, and link to Probe when it is found. This means that programs under test do not necessarily have to be initiated by Probe (although this is the normal method). It is possible to test, for instance, a number of C programs linked by `spawn . . .` calls, or even the operation of programs under the SID debugger program.

The process of searching for Probe is quick, and takes place only once. There is no significant effect on the performance of a program when Probe is absent.

GEMDOS is not a multitasking operating system, and therefore operations such as opening files, performed by Probe while a program is being tested, are considered by GEMDOS to be performed by the program under test. Files are automatically closed by GEMDOS when a program terminates. Probe is programmed to be resilient to this unexpected closing of files that it thinks of as its own, but errors are possible in complex situations involving `spawn`, for example.

In order to use Probe, there must be enough memory for both Probe and the program under test – Probe requires about 100K bytes. If Probe is executed from the Workbench, this limits the memory available to the program under test – however, Probe can be operated from the GEM Desktop as well as from the Workbench, which may be useful for debugging larger programs. It is quite possible to run other programs from within Probe, including text editors and even the Workbench, subject to there being sufficient memory and capacity for open files available.

If Probe is started using the Workbench **Debug program** menu selection, the default folder will be the one from which the program under test was loaded.

8.3.2 The start-up file

When a user program first enters Probe, a check is made in the current directory for a text file `PROBE.SYS`. If this file is present, it is read as a sequence of Probe commands, before taking any other action. All commands are available, so other uses are possible, such as switching on disk logging of the debugging session, or setting standard break or watch conditions.

8.3.3 Entry after runtime errors

When Probe is present, runtime errors are handled in exactly the same way as normal. However, immediately after the standard error report has been given, Probe is entered. The circumstances of the error can then be investigated. If the program is resumed after the error, the standard error handling will complete. Thus non-recoverable errors will result in immediate termination of the program.

On entry to Probe after a runtime error (an error producing a C runtime error report), the circumstances are slightly different from a normal entry, since the error does not necessarily occur in a source module compiled with option N, and even if it does, the error may not be at the start of the line. Therefore the source line shown at such an error is part executed, rather than in the usual unstarted state. It is also not very useful to enter a debugger using the “T” command, because the machine instruction revealed will usually be in some part of the runtime library, rather than the compiled program code.

If the error was an arithmetic error in an assignment, bear in mind that the assignment will be completed after the program resumes. To alter the value of the variable being assigned, you must stop on the next statement (using for example the “S” command), and correct the variable after the erroneous assignment is complete.

8.3.4 Working with SID and other programs

When starting Probe from the Workbench, the file selector form is used to obtain the name of the file that Probe will execute. This name is then passed to Probe as its command tail, and the program is invoked by Probe as a child process. It is possible to put any filename in Probe’s command tail – the simplest ways to do this from the Workbench are either to supply an empty filename when selecting **Debug program**, then type the command tail using Probe’s X command, or to execute Probe (the file PROBE.PRG) using **Run other file**, and supply a command tail to it in the usual way. When starting Probe from the GEM Desktop, similar considerations apply. For example, the command tail MYPROG can be replaced by SID MYPROG.PRG. In this case, the machine level debugger SID will start up, and the SID “G” command will start up MYPROG, the program under test. At any time when the program under test is subsequently halted in Probe, the “T” command will re-enter SID at the start of the compiled code for the current statement. SID can be used to step through the compiled code instruction by instruction, or to trace execution into assembler coded routines inaccessible to Probe.

8.4 Probe command parameters

The syntax notation used here is as described in appendix A.1.

8.4.1 Character set and source tokens

Any printing character can be used in Probe commands, although only a subset is used outside character strings. Except in command names, upper and lower case letters are significant. Each command is considered as a sequence of lexical “tokens” interspersed with “separators”, and the syntax of commands is defined in these terms. The tokens are of four kinds:

token :
identifier
number-constant
string-constant
special-symbol

8.4.2 Identifiers

Identifiers are formed from letters, underscores and digits. An identifier begins with a letter or underscore. Identifiers are used for variables and enumerated constants of the source program, and for function names.

8.4.3 Number constants

Number constants comprise integer or floating constants preceded by an optional minus sign. The syntax for number constants is as for the source language. See Part III of this manual and section 8.6 below for a formal specification. Character type constants containing one character can also be used in Probe to denote the ASCII code of the character. Examples of the various kinds of constants are:

1	integer
-12345678	integer
1.7	double
123.56E12	double
123.56E12f	float
0xFF	integer
'A'	integer

8.4.4 String constants

String constants define null-terminated strings which can be assigned or compared to variables of type `char *`. Their syntax is the same as in C, i.e.

```

string-constant :
    " s-char-sequenceopt"
s-char-sequence:
    s-char
    s-char s-char-sequence
s-char:
    Any character except double-quote, backslash or new-line
    escape-sequence
  
```

8.4.5 Special symbols

The following character sequences form special symbols used in the command syntax:

```

special-symbol : one of
    [
    ==  !=  <  >=  >  <=  =
    ->  *   :   ..  -   ?   &
  
```

NULL is not reserved, but is recognized as a valid value for a C pointer variable.

8.4.6 Identifiers and qualifiers

Identifiers used in Probe are taken from the source program, and may refer to functions or variables. Identifiers and line numbers in commands are interpreted at a particular point in the source. This is normally the current execution point, but may be modified within a command using a “qualifier”, which takes the form:

```
qualifier:  
  file-name :  
  function-name :  
file-name:  
  ' identifier '  
function-name:  
  identifier
```

The “:” may be preceded and/or followed by spaces. A *function-name* specifies an active function where identifiers and line numbers are to be interpreted. A *file-name* specifies a source listing (.PRN) file. (The directory name and .PRN extension are supplied by Probe.)

examples

```
main:  
'PRIME':
```

In this way, it is possible to access any active variable of a program under test. In the case of recursive functions, the latest activation is identified when the name is used as a qualifier.

8.4.7 Break and watch specifiers

Changes in variables, and the flow of execution of a program under test, can be monitored using the BREAK, WATCH and GO commands. The conditions to be monitored are defined using a “*break-watch-specifier*”:

```
break-watch-specifier:
    qualifieropt break-watch-conditionopt
break-watch-condition:
    lines
    variable
    variable op value
lines:
    integer-constant
    integer-constant . . integer-constant
```

The *break-watch-specifier* identifies a region of source text whose execution is to be monitored, or a variable whose changes are to be monitored. The syntax covers a number of cases, and corresponding monitoring conditions:

(nothing at all)	– execution of every source line
file-name qualifier	– execution of any line in that file
function-name qualifier	– execution of any line in a function of that name
lines	– restrict monitoring to given lines
variable	– all changes to the variable
variable op value	– all changes satisfying the condition

examples

```
main:
factor
year > 100
main:maxfactor
main:number = 0
20
20..30
'PRIME' : 20..30
```

8.5 Probe commands

In this section, the effect of each command is described. Note that in use, command names can conveniently be abbreviated to their first character, and can be given in upper or lower case.

8.5.1 assign

format

ASSIGN *qualifier_{opt} variable = value*

The assign command sets a variable to a specified value. Probe will attempt to make reasonable conversions where possible so that the value is compatible with the variable. Identifiers used in the command will be evaluated at the current point in the source, or in the context specified by a qualifier if one is given. Array and structure variables can only be altered element by element.

examples

```
a count=3
A buffer[i] = 'H'
Assign PlotArc:itemptr->radius[3] = 3.767
```

8.5.2 break

format

BREAK *break-watch-specifier_{opt}*

BREAK ?

BREAK -

break-watch-specifier:

qualifier_{opt} break-watch-condition_{opt}

break-watch-condition:

lines

variable

variable op value

The break command adds the given condition to the list of conditions which cause the program under test to be interrupted, and Probe's command interpreter to be entered. "Break -" removes all current break conditions. "Break ?" displays the currently active break conditions.

Conditions include:

- a single source line or range of lines being executed, or
- a variable changing in value or changing to satisfy a condition.

If no parameter is specified, then the execution of any source line (which has been compiled with the N option) will result in a break at the beginning of the line. Specifying just a *file-name* treats all lines in the file as break points. Where a *function-name* is specified, execution of any source line in a function of that name causes a break.

BREAK is identical to WATCH, except that whenever the condition is satisfied, execution halts and a Probe command is requested rather than continuing execution of the program. The information displayed is the same in either case, as follows:

- 1 The source line causing the condition.
- 2 For variables, up to twenty characters of the original command, and the value of the variable.

Since Probe displays the source line about to be executed whenever execution is interrupted for command input, an extra line is displayed with BREAK on a variable.

Any number of watch and break requests can be active at the same time. In this case, the overall display will reflect the combination of all the current requests. The source line displayed as causing a variable break or watch display is Probe's best estimate of the appropriate line. Note that the line displayed may be a direct assignment, an assignment via a pointer, or a pre- or post-increment or decrement operation.

When a variable is referenced in a break or watch command, its location is evaluated once only, at the time of the command. When watching, for example, an array element A[I], the element watched remains the same, even though the variable I may subsequently vary. If the object watched has automatic storage duration, or is in dynamically allocated memory, it may cease to exist while a break or watch command relating to it is still active. In this case the break or watch ceases to be meaningful.

examples

```
B INUM>100
B A[I]
b name = "Alice"
B ?
Break -
```

8.5.3 calls

format

CALLS

The calls command has no parameters. It displays each of the active source lines in turn, starting with the current source line. All but the current line will normally include a function call which is currently active. The active functions which are named by the calls command can be used as qualifiers in display commands to display their variables.

8.5.4 display

format

DISPLAY *qualifier_{opt} variable_{opt}*

The given variable is displayed, giving (the last component of) its name, (an indication of) its type, and its value. Where only a function is specified, the formal parameters and local variables are displayed. A warning is displayed if the function is not active. Where a source file is specified, the static variables that the file contains are displayed. Where no qualifier or variable is specified, the local variables of the current function are displayed. Structured variables are only displayed in full if specifically referenced in the command.

examples

```
D I
D ReadItem:
D sort: A[176]
```

The first of these displays the value of the variable I in the current function, the second displays the parameters and variables of function ReadItem, and the third displays the value of element 176 of the array A in function sort.

8.5.5 echo

format

ECHO

The echo command allows for all interaction with Probe to be logged to the disk file PROBE.LOG. Probe always opens this file at the start of a session. Alternate uses of the echo command switch on and off the logging process.

8.5.6 go

format

GO *break-watch-specifier*_{opt}

Execution of the program under test is resumed, with current Watch and Break conditions active. If a break condition (see 8.4.7 and 8.5.2 above) is specified, it becomes active until Probe is next entered (and is then removed from the list of break conditions). This is useful for getting quickly to an area of the program to be examined in further detail.

example

```
G
g sort:
```

8.5.7 help

format

HELP *command-name*_{opt}

command-name: one of

Assign	Break	Calls	Display
Echo	Go	Help	Key
List	Output	Profile	Quit
Route	Step	Trace	View
Watch	X	Z	

Information regarding the specified command is given, or if no command is specified, a general description of Probe facilities is displayed. Commands about which information is requested may be abbreviated when used as parameters to HELP, in the same way as when using the commands themselves. The help command requires the presence of the file PROBE.HLP during execution.

examples

```
HELP
h w
```

8.5.8 key

format

KEY

By default, Probe operates in a way which allows the user to interrupt the program under test at any time by striking any key. This command alternately disables and enables this facility. It is mainly intended for examining programs which themselves test the keyboard without executing an input statement. In this case, keyboard events must be handled by the program under test without interception by Probe. Note that programs under test run significantly slower when keyboard interruption is enabled.

example

```
K
```

8.5.9 list

format

LIST *qualifier_{opt} lines_{opt}*

The list command displays the specified source lines on the screen. If a qualifier is given, it specifies the source file whose lines are to be displayed. The qualifier must either be a source file name (as a character string) or a currently active function name. If no line number is given then lines in the vicinity of the current source line will be displayed. Subsequent lines displayed by LIST with no parameter follow on from the preceding display. If 'profiling' has been enabled (see 8.5.11), the execution counts are displayed.

examples

```
L
list 23..40
L 'SOURCE': 1..20
```

8.5.10 output

format

OUTPUT

The screen switching used to separate Probe's output from that of the program under test is switched on or off. A message indicates the new setting (enabled or disabled). The alternate screen is used for all output by the program under test, and can be viewed using the VIEW command.

example

O

8.5.11 profile

format

PROFILE

The profiling option is switched on or off. A message indicates the new setting (on or off). A 'profile' of a program is a listing of the frequency of execution of each part of the program. When profiling is selected, Probe maintains counts of the number of times each source line is executed. When a subsequent LIST command is executed, the counts are displayed, along with the line numbers and the source lines. The ECHO command can be used to retain a copy of the listing for subsequent printing or analysis.

A profile is generally used to determine which parts of a program are most frequently executed and therefore would best benefit from more efficient coding. It can also be very useful in determining if a particular part of the program has been executed at all.

example

P

8.5.12 quit

format

QUIT

Probe and the program under test are terminated immediately. This command must be entered in full in order to take effect: abbreviations will be rejected. Note that normal program termination does not take place; any files opened by the program under test will be left open, and information buffered within the runtime library will not be written to disk. Use of this command can be dangerous when debugging certain programs, for example when a GEM application has entered but not yet left 'wind_update' mode, which will prevent the parent program (e.g. the Workbench or the GEM Desktop) from functioning.

example

QUIT

8.5.13 route

format

ROUTE

The route command displays the most recently executed few source lines. These include both lines which have been completely executed, and those containing an active function call. Those whose execution has not completed will also feature in the output of the CALLS command (see 8.5.3).

example

R

8.5.14 step

format

STEP *line-count*_{opt}

Execution of the program under test is resumed for the specified number of lines. If no line-count is specified, a single line is executed. Note that only lines on which an executable statement starts are counted. These are the lines indicated by an asterisk on the compiler source listing (.PRN) file.

examples

```
S 5
s
```

8.5.15 trace

format

TRACE

The trace command is for use with a machine language debugger, such as SID. It resumes execution of the program under test for one line (like the Probe “S” command), but if SID (or another debugger) has been loaded, a breakpoint is automatically inserted at the first instruction of the compiled code for the current source line. Thus SID will be entered immediately, and a SID “L” command will display the compiled machine code for the current source line. SID can be used to view the compiled code, or step through the program, instruction by instruction.

A frequent use of the trace command is to test a program including assembler coded parts. A Probe breakpoint is set on the source line which calls the assembler routine. When the program stops at this line, a TRACE command is used to enter SID. The SID “T” command can be used to trace execution through the assembler coded routine, or “G” can be used to re-enter Probe at the start of the next line.

Refer to the documentation supplied with SID or any other debugger for details of its use. See also section 8.3.4 above.

example

```
X SID MYPROG.PRG
G
G 18
T /* Enter SID at start of line 18 */
```

8.5.16 view

format

VIEW

The VIEW command may be used to observe the alternate screen (to which output from the program under test is directed) when screen switching is in use. Pressing any key returns to the Probe screen. See the description of the OUTPUT command in 8.5.10 above.

example

v

8.5.17 watch

format

```
WATCH break-watch-specifieropt  
WATCH ?  
WATCH -
```

The watch command sets up a condition to be monitored whereby information is displayed whenever the specified circumstance holds during execution of the program under test. The display consists of the source line, and, if a variable was specified, its value and up to twenty characters of the original watch command. If no parameter is specified to the watch command, then every source line is displayed as it executes. "WATCH -" removes all current watch conditions. "WATCH ?" displays all current watch conditions. Possible conditions include execution of lines in a particular function or source file, changes in a variable, or variable changes satisfying a condition.

Any number of watch and break requests can be active at the same time. In this case, the overall display reflects the combination of all the current requests. See 8.5.2 above for other points applicable to both WATCH and BREAK.

examples

```
W  
W SAMPLE: 120..150  
W i  
w index >= 99  
W -
```

8.5.18 X (execute)

format

x *filename command-tail_{opt}*

The execute command allows any other program to be executed from Probe (provided sufficient memory and other resources are available). On completion of the command, the Probe prompt (>>) will reappear. The “X” command is useful for calling up editors and so on, without disturbing the state of the program being tested. If the command is used to initiate another program compiled with the N option, Probe facilities will only be available within that program if there is no program currently under test. If no filename extension is given, Probe supplies the extension .PRG.

examples

```
X GEMDEMO
x C-BENCH
```

8.5.19 Z (hexadecimal display)

format

z *value*

The hexadecimal display command allows any value to be displayed in hexadecimal. This is useful for determining the values of variables with bit-significant values, where the decimal value is hard to decode. It can also be useful for displaying the addresses of variables, for example when a pointer is supposed to be pointing to an array.

examples

```
Z &buffer
Z bufptr
```

8.6 Command syntax summary

Each command to Probe consists of a command name, possibly followed by a parameter or parameters. In fact, only the first letter of the command name is significant, and the complete name is only of use to help remember the function of the command. The only exception to this is the QUIT command, which must be typed in full to take effect. The parameters of some commands may be preceded by a “qualifier”, which names a source program file or function name to which the command applies. The syntax for references to variables or constants follows that of the source language quite closely.

The syntax of commands to Probe is:

command: one of

<i>assign-command</i>	<i>break-command</i>	<i>calls-command</i>
<i>display-command</i>	<i>echo-command</i>	<i>go-command</i>
<i>help-command</i>	<i>key-command</i>	<i>list-command</i>
<i>output-command</i>	<i>profile-command</i>	<i>quit-command</i>
<i>route-command</i>	<i>step-command</i>	<i>trace-command</i>
<i>view-command</i>	<i>watch-command</i>	<i>execute-command</i>
<i>hexadecimal-command</i>		

assign-command: **ASSIGN** *qualifier*_{opt} *variable* = *value*

break-command: **BREAK** *break-watch-specifier*_{opt}

BREAK ?

BREAK -

calls-command: **CALLS**

display-command: **DISPLAY** *qualifier*_{opt} *variable*_{opt}

echo-command: **ECHO**

go-command: **GO** *break-watch-specifier*_{opt}

help-command: **HELP** *command-name*_{opt}

key-command: **KEY**

list-command: **LIST** *qualifier*_{opt} *lines*_{opt}

output-command: **OUTPUT**

profile-command: **PROFILE**

quit-command: **QUIT**

route-command: **ROUTE**

step-command: **STEP** *line-count*_{opt}

trace-command: **TRACE**

view-command: **VIEW**

watch-command: **WATCH** *break-watch-specifier*_{opt}

WATCH ?

WATCH -

execute-command: **X** *filename* *command-tail*_{opt}

hexadecimal-command: **Z** *value*

break-watch-specifier:
*qualifier*_{opt} *break-watch-condition*_{opt}
break-watch-condition:
lines
variable
variable op value
qualifier:
file-name :
function-name :
variable:
identifier
variable [value]
variable.fieldname
variable ->
** variable*
value:
variable
constant
& variable
lines:
integer-constant
integer-constant . . integer-constant
line-count:
integer-constant
constant :
integer-constant
floating-constant
string-constant
op: one of
 == != < > >= <=
function-name:
identifier
file-name:
 ' identifier '
fieldname:
identifier
string-constant :
 " s-char-sequence_{opt}"
command-name: one of

Assign	Break	Calls	Display
Echo	Go	Help	Key
List	Output	Profile	Quit
Route	Step	Trace	View
Watch	X	Z	

9 THE CROSS-REFERENCE GENERATOR

A cross-reference generator CXREF is provided as part of the Prospero C package. It is a very useful facility when developing or maintaining programs of any size, and is tailored to the Prospero C syntax. All source identifiers are listed in alphabetical order, together with the source line(s) on which they are referenced. If header files are included, they are allocated sequential numbers starting at 1, and the report uses these numbers to refer to the included files.

9.1 Operation from the Workbench

The CXREF program is selected by the **Cross reference** option at the bottom of the **Compile** menu. Upon selection, the File Selector is used to obtain the name of the source file to be cross-referenced. The cross reference options are then specified using the following form :

Cross Reference

Source File :- GEMDEMO.C

Line Width :- 100

Output to :- PRINTER

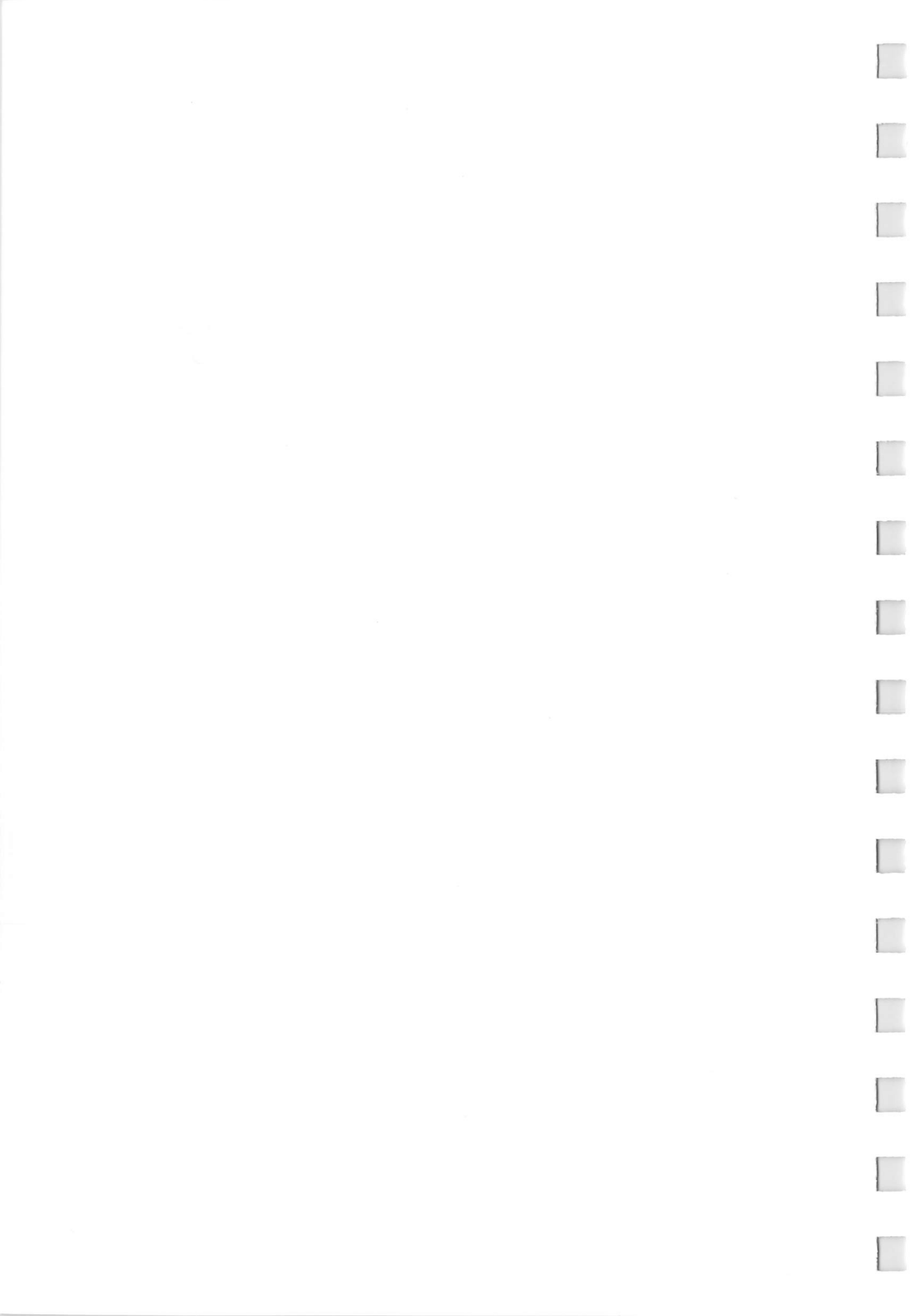
Two options may be altered – the line width for printout, and the destination of the cross-reference listing. The cross reference listing will be sent either directly to the printer, or to a disk file whose name is the same as the source file, with the extension .XRF, or the listing file may optionally be automatically loaded into a Workbench window on completion, if a window is available. Clicking on the word **PRINTER** toggles between the three options, and the text changes (in the above example) to **GEMDEMO.XRF** or **WINDOW** as appropriate.

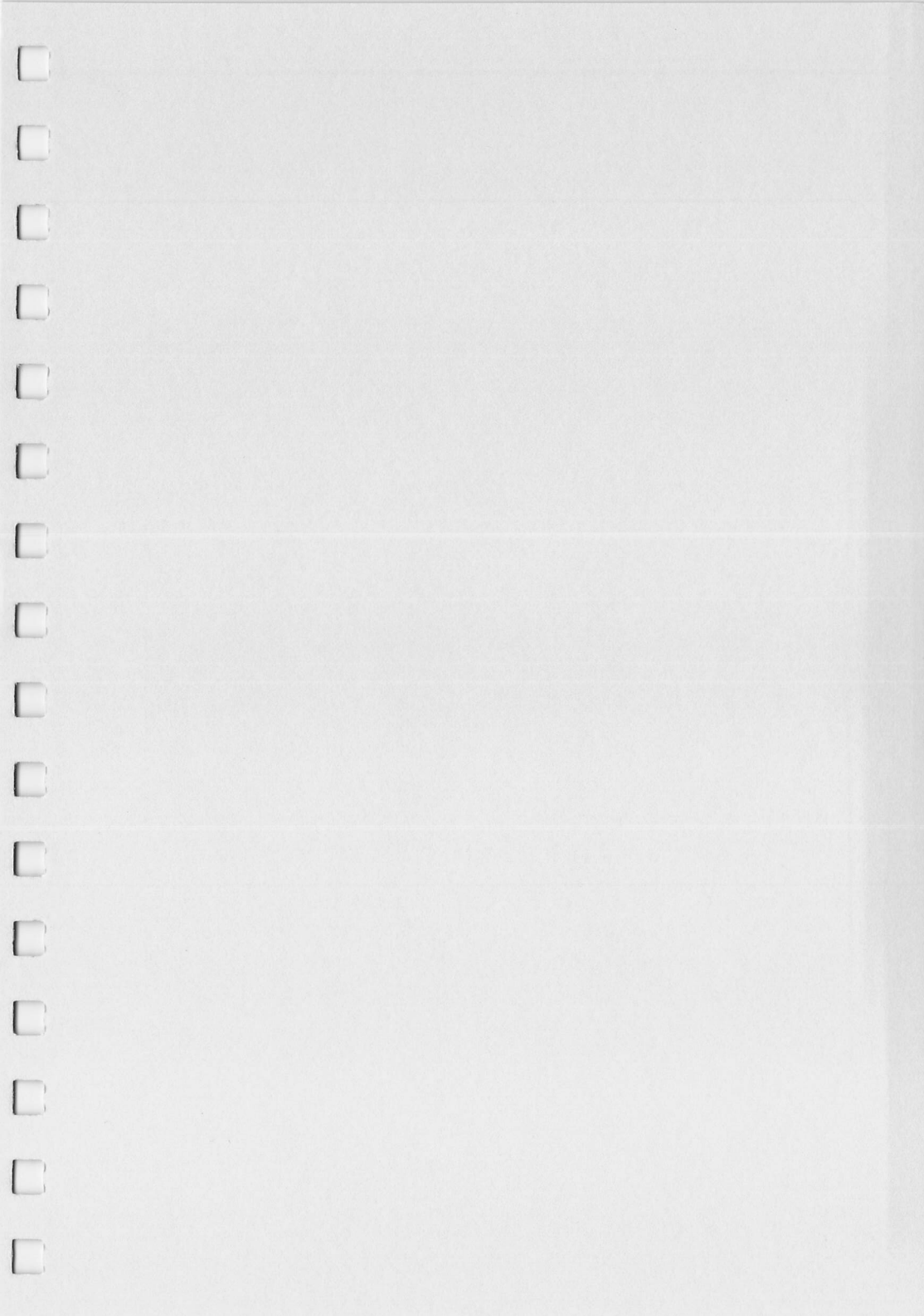
9.2 Operation from the command-line

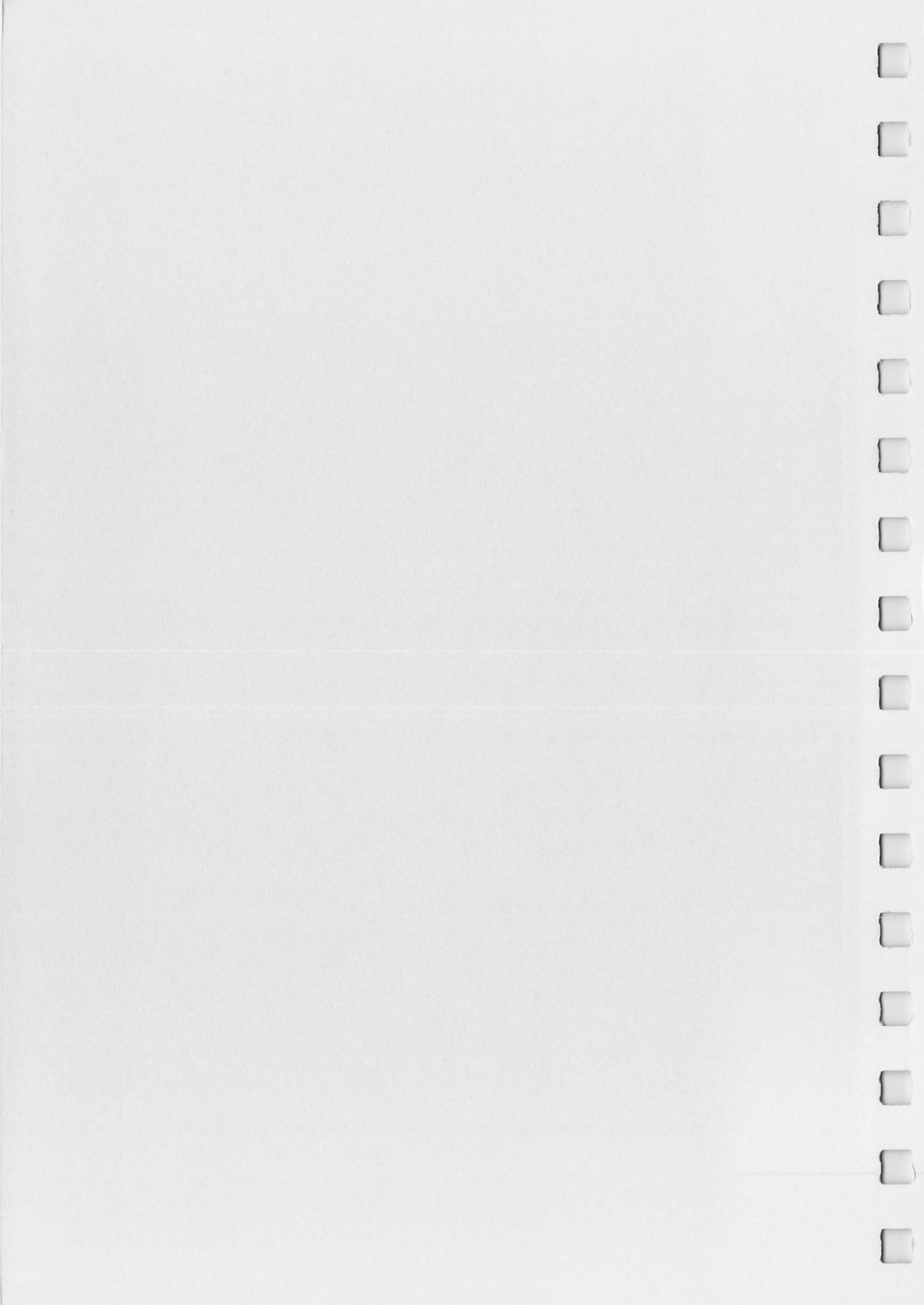
The cross-reference program CXREF.TTP may also be operated from the command-line. In use, the CXREF is followed by the source filename (the extension .C is added if no extension is specified), and optionally by a printer line width and filename for output, separated by commas. If no command-tail is specified, the user is prompted for each field. The default printer width is 100, and the default output file is CON: (the screen). The output file may also be a printer (PRN:).

For example,

```
CXREF  
CXREF prime,,CON:  
CXREF prime,80,PRN:
```

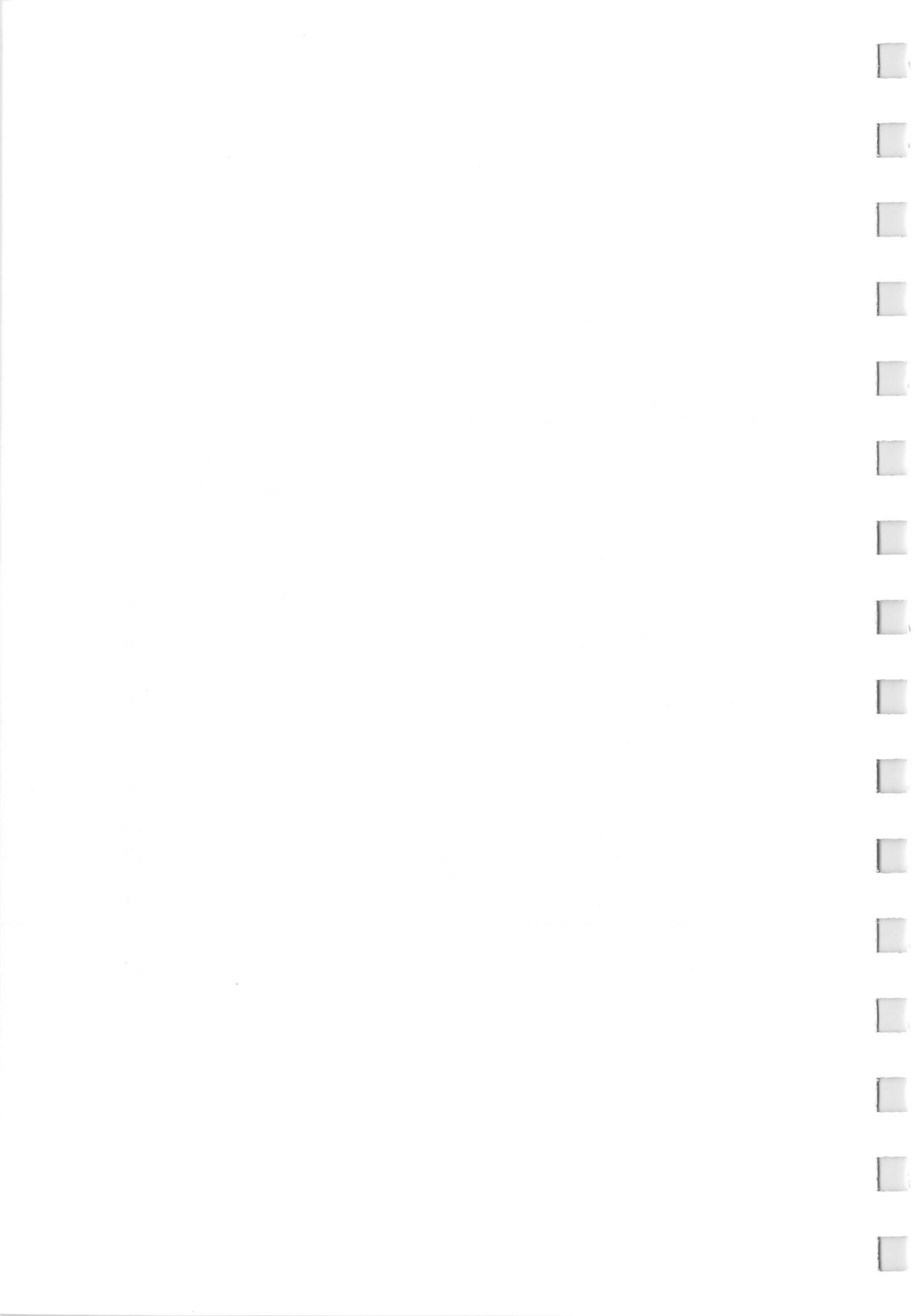






PART II - IMPLEMENTATION DETAILS

1	Introduction	1
1.1	Motorola processors	1
1.2	Operating system	1
2	Standard features	2
2.1	Data types	2
2.2	Statements	2
2.3	Expressions	2
2.4	Translation limits	3
2.5	Numerical limits	4
2.5.1	Sizes of integral types <limits.h>	4
2.5.2	Characteristics of floating types <float.h>	5
3	Input and output	6
3.1	Declaration of streams	6
3.2	File formats	6
3.2.1	Text files	6
3.2.2	Non-text files	6
3.3	Buffering	6
4	Storage allocation	8
5	Interfacing to assembler	10
5.1	Use of assembly language	10
5.2	Choice of assembler	10
5.3	XDEF/XREF linkage	10
5.3.1	Calling assembler from C	10
5.3.2	Calling C from assembler	11
5.4	External data	13
5.5	Preservation of registers	14
5.6	Parameters	14
5.7	Function results	15
5.8	Reserved section names	15



1 INTRODUCTION

The purpose of this section is to bring together a number of points relating to the implementation of Prospero C on different hardware and/or operating system environments.

1.1 Motorola processors

This implementation is for a microcomputer based on one of the Motorola family of processors: MC68020, MC68010, MC68000 or MC68008.

The Workbench, compiler, linker and other supplied programs make use only of instructions common to all processors in the family, and will therefore run on any such microcomputer.

1.2 Operating system

This implementation is for Atari ST machines using the standard GEMDOS (also known as TOS) operating system. Both the Workbench and object programs can make use of the hierarchical file structure, so that a full pathname can be specified anywhere that a filename is required.

2 STANDARD FEATURES

Prospero C is a complete implementation of the August 1987 draft of the forthcoming ANSI C Standard. This section summarizes the standard features and how they are implemented. Appendix H gives details of how Prospero C implements various aspects of the language not mandated by the Standard.

2.1 Data types

Type	Storage	Range of values
signed char	1 byte	-128..127
signed short	2 bytes	-32768 .. 32767
signed int	2 bytes	-32768 .. 32767
signed long	4 bytes	-2147483648 .. 2147483647
unsigned char	1 byte	0 .. 255
unsigned short int	2 bytes	0 .. 65535
unsigned int	2 bytes	0 .. 65535
unsigned long int	4 bytes	0 .. 4294967295
float	4 bytes	0, ± 1.17549435e-38 .. 3.40282347e+38
double	8 bytes	0, ± 2.225073858507201e-308 .. 1.797693134862316e+308
long double	8 bytes	same as double
void		

All pointers occupy four bytes. There are no limitations (other than the available memory) on the sizes of arrays or structures.

2.2 Statements

All the statements of Standard C are implemented:

expression-statement, compound-statement,
goto, **continue**, **break**, **return**,
if, **switch**, **while**, **do**, **for**.

2.3 Expressions

The following operators are available:

```
[ ] ( ) . ->
++ -- & * + - ~ ! sizeof
/ % << >> < > <= >= == !=
^ | && || ? :
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
```

2.4 Translation limits

Given sufficient memory Prospero C is capable of translating a single input source file containing all of the following:

At least 15 nesting levels of compound statements, iteration control structures, and selection control structures.

At least 10 nesting levels in conditional compilation (`#if`).

Greater than 12 pointer, array and function declarators modifying a basic type.

At least 127 expressions nested by parentheses.

Arbitrary number of significant characters in an internal identifier or macro name.

Macro bodies of any size

32 significant characters in an external name, case significant.

More than 511 external identifiers.

More than 127 identifiers with block scope in one block.

More than 1024 macro identifiers simultaneously defined.

More than 31 parameters in one function definition and call.

More than 31 parameters in one macro definition and invocation.

Up to 32767 characters in a logical source line.

Up to 32767 characters in a string literal.

No limit to the number of bytes in an object.

At least 10 nesting levels for `#include` files.

No limit to the number of case labels in a switch statement.

2.5 Numerical limits

The Prospero implementation of C has the following numerical limits.

2.5.1 Sizes of integral types <limits.h>

<code>CHAR_BIT</code>	8	max bits in smallest non-bit field object
<code>SCHAR_MIN</code>	-128	min value of type <code>signed char</code>
<code>SCHAR_MAX</code>	+127	max value of type <code>signed char</code>
<code>UCHAR_MAX</code>	255U	max value of type <code>unsigned char</code>
<code>CHAR_MIN</code>	see below	min value of type <code>char</code>
<code>CHAR_MAX</code>	see below	max value of type <code>char</code>
<code>SHRT_MIN</code>	-32768	min value of type <code>short int</code>
<code>SHRT_MAX</code>	+32767	max value of type <code>short int</code>
<code>USHRT_MAX</code>	65535U	max value of type <code>unsigned short</code>
<code>INT_MIN</code>	-32768	min value of type <code>int</code>
<code>INT_MAX</code>	+32767	max value of type <code>int</code>
<code>UINT_MAX</code>	65535U	max value of type <code>unsigned int</code>
<code>LONG_MIN</code>	-2147483648	min value of type <code>long int</code>
<code>LONG_MAX</code>	+2147483647	max value of type <code>long int</code>
<code>ULONG_MAX</code>	4294967295U	max value of type <code>unsigned long</code>

When the U option is in force, the value of an object of type `char` does not sign-extend when used in an expression, and the value of `CHAR_MIN` will be zero, and `CHAR_MAX` will be the same as `UCHAR_MAX`. Otherwise, the value of an object of type `char` does sign-extend when used in an expression, and the value of `CHAR_MIN` and `CHAR_MAX` will be the same as those of `SCHAR_MIN` and `SCHAR_MAX`.

2.5.2 Characteristics of floating types <float.h>

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.

The following describes the Prospero C floating-point representation, which also meets the requirements for single-precision and double-precision normalized numbers in the "IEEE Standard for Binary Floating-Point Arithmetic"(ANSI/IEEE Std 754-1985).

Sizes of floating types:

<code>FLT_RADIX</code>	2	Radix of exponent representation
<code>FLT_ROUNDS</code>	1	Rounding mode (to nearest)
<code>FLT_MANT_DIG</code>	24	binary digits in mantissa
<code>DBL_MANT_DIG</code>	53	
<code>FLT_EPSILON</code>	1.19209290E-07F	minimum x such that $1.0+x > 1.0$
<code>DBL_EPSILON</code>	2.2204460492503131E-16	
<code>FLT_DIG</code>	6	decimal digits in mantissa
<code>DBL_DIG</code>	15	
<code>FLT_MIN_EXP</code>	-125	minimum x such that 2^{x-1} is
<code>DBL_MIN_EXP</code>	-1021	normalized.
<code>FLT_MIN</code>	1.17549435E-38F	mimimum normalized +ve no.
<code>DBL_MIN</code>	2.225073858507201E-308	
<code>FLT_MIN_10_EXP</code>	-37	minimum x such that 10^x is
<code>DBL_MIN_10_EXP</code>	-307	normalized.
<code>FLT_MAX_EXP</code>	128	maximum x such that 2^{x-1} is
<code>DBL_MAX_EXP</code>	1024	representable.
<code>FLT_MAX</code>	3.40282347E+38F	maximum representable no.
<code>DBL_MAX</code>	1.797693134862316E+308	
<code>FLT_MAX_10_EXP</code>	+38	maximum x such that 10^x is
<code>DBL_MAX_10_EXP</code>	+308	representable.

As **long double** and **double** share the same representation, the limits and sizes for **long double** are as described for **double** above.

3 INPUT AND OUTPUT

3.1 Declaration of streams

The standard predeclared streams `stdin`, `stdout` and `stderr` are always available. There is no provision for a standard error handle under GEMDOS, and therefore `stderr` is equivalent to `stdout`. In addition, Prospero C defines two additional pre-opened streams `stdaux` (for input and output to the serial port) and `stdprn` (for output to the printer). If using the unbuffered file functions defined in `io.h`, handles 0, 1, 2 or 3 can be used to refer to the standard input, output, auxiliary and printer handles respectively.

3.2 File formats

3.2.1 Text files

A text file (on disk) follows the standard GEMDOS convention that lines are terminated by carriage-return / line-feed. Prospero C provides facilities on input and output for translating between such files and the standard C convention that lines are terminated by line-feed only. These are described further in Volume 2 (Prospero C Library).

3.2.2 Non-text files

Non-text files can be read using the `fread` and `fwrite` functions, or using the lower level unbuffered `read` and `write` functions. There is no limit to the size of a file or of an object which can be read from a file.

3.3 Buffering

The functions in the C library for file handling fall into two distinct categories. The header file `stdio.h` defines a set of functions for dealing with buffered files, known as streams, which are specified in the draft ANSI standard. The functions in `io.h` are used for unbuffered access to files, so that the access is only a little above the operating system level (or not at all, in the case of some of the functions whose names begin with an underscore). The unbuffered file i/o library and the header file `io.h` are not part of the draft ANSI standard.

When streams are opened for input, output or update, by default they are fully buffered. Due to limitations of the Atari operating system, it is not possible to distinguish an interactive device from a file stored on disk, so where the stream in fact refers to an interactive device where full buffering may not be appropriate, the program will have to alter the buffering mode after opening the file (e.g., using `setvbuf`). The standard predefined streams are an exception to this rule – standard output is always assumed to be a device and is unbuffered, while standard input is assumed to refer to the console if the file size returned by GEMDOS is zero. In this case, input will be line buffered, and a line-feed will be echoed to the screen after each line is read.

4 STORAGE ALLOCATION

Object programs can in general contain requirements for the following kinds of storage.

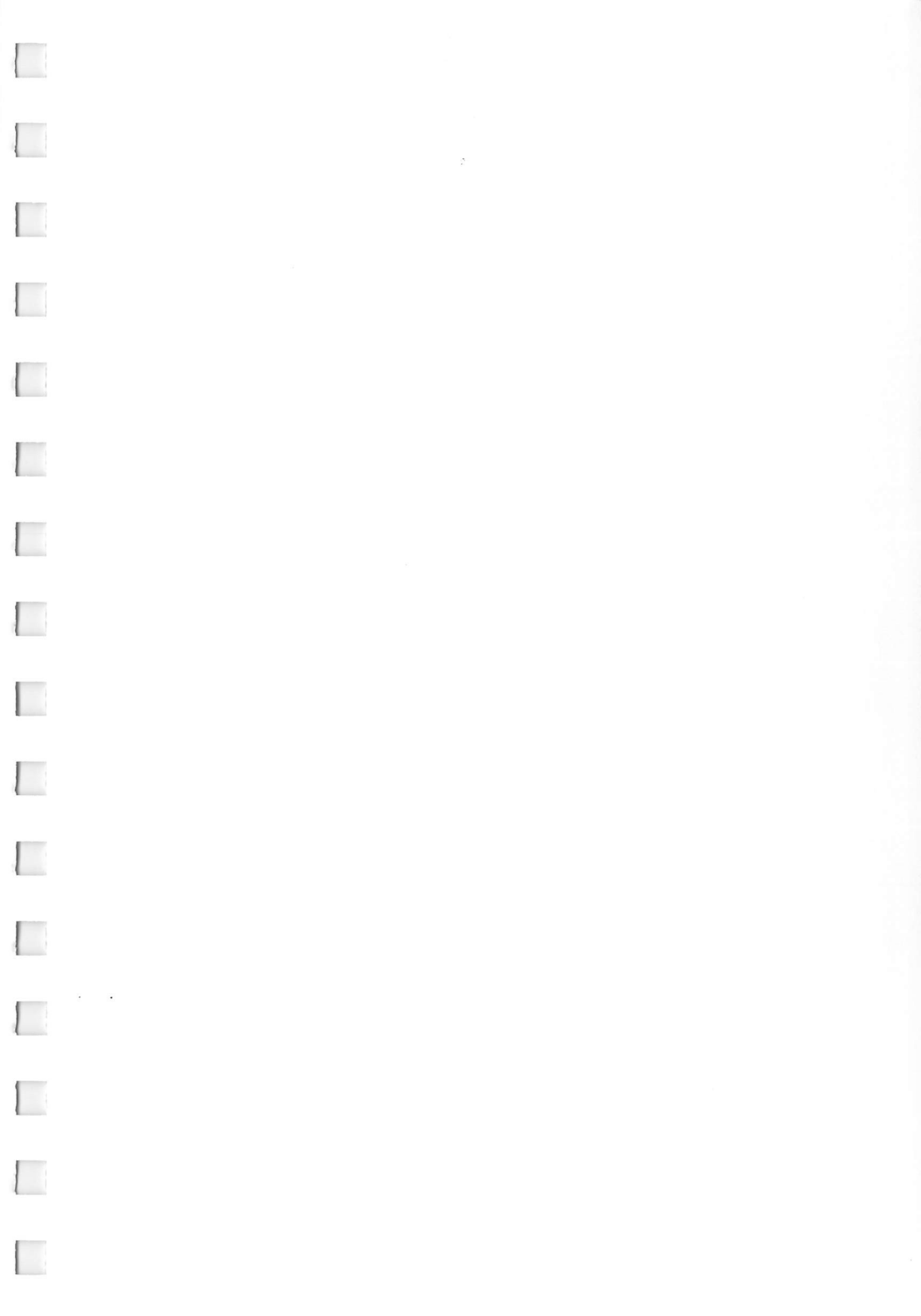
- Program code.
- Constants (literals).
- Static data areas.
- Stack/work area.

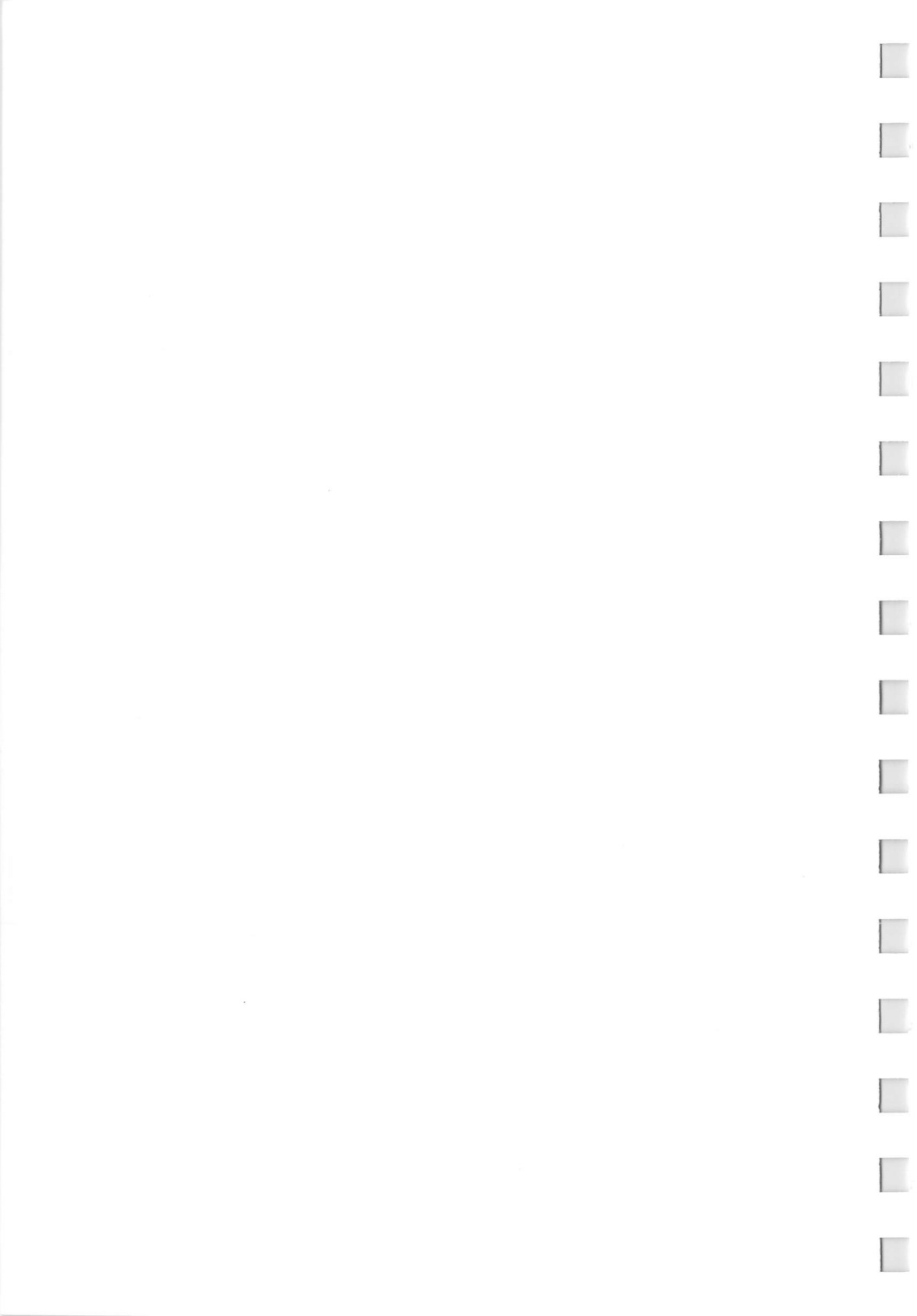
All objects declared as `static` or `extern`, or declared outside the scope of any function are allocated static data space.

In the object code from the compiler, the static data for each module is located on a word boundary. The stack is kept word-aligned throughout execution of the program.

The result of compiling a C translation unit is a single module in relocatable object format which consists of a number of “sections”. There are up to five sections generated:–

- (1) `.CODE`, which contains object code and constants (integers, doubles, strings, etc.). The code generated for the body of any one function cannot exceed 32K bytes.
- (2) `.ATAB`, which contains control information enabling run-time access to external functions and data.
- (3) `.INIT`, which contains control information (used by Prospero Fortran modules) enabling the run-time initialization of `COMMON` blocks.
- (4) `.NAME`, which by default contains just the main program’s name. If the `N` compile-time option is selected, it also contains the names of files and functions compiled, for utilization in the event of a run-time error when producing a function call trace-back, or by the symbolic debugger Probe (see Part I, section 8).
- (5) Each module’s static data forms a section whose name is the same as the source file name, without extension, preceded by the characters `D$`.



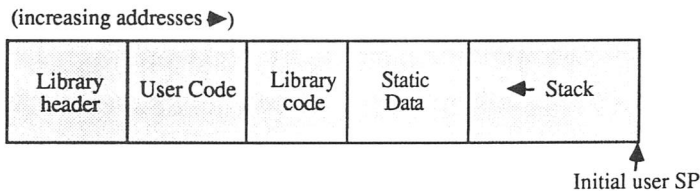


The user stack pointer SP is set to the highest address in the workspace area plus one, and the stack “grows” from higher to lower addresses. Dynamically allocated memory is taken from an area known as the heap, which grows from lower to higher addresses. If at run-time the library finds that the stack and the heap are about to collide, execution of the user program is terminated. The program must be re-linked and re-run with a larger amount of workspace.

Depending on a program’s requirements, extra memory areas may be allocated dynamically at run-time by the library heap manager.

The detailed layout of a program in memory is as follows:–

The executable code image and the static data area are both in the text section of the program file, and are loaded in to memory when the program is executed. The memory immediately above the program file image is allocated to the stack, the stack size being determined by a value in the program file header, controlled by a linker option (see Part 1, section 5.2). The space above this is released back to the operating system, but may be re-allocated by requests for dynamic memory allocation using `malloc` and equivalent functions.



In addition, there is a further area allocated dynamically by the library, containing global run-time information shared by the highest-level program and its dependent child programs. The size of this area is a few hundred bytes.

5 INTERFACING TO ASSEMBLER

5.1 Use of assembly language

To use machine features not available through the C language, routines may be written in assembly language and combined with the generated code during the link-edit process.

5.2 Choice of assembler

The C compiler generates relocatable object code. Assembler language modules may be processed by any assembler which generates the same format, and linked with the other components of the program. In particular, the GST Macro Assembler will be found satisfactory.

5.3 XDEF/XREF linkage

5.3.1 Calling assembler from C

An assembler-coded routine can be called from C in the normal way by a function call. In the assembler module, the name is quoted in an XDEF directive, or made global in some equivalent way. More than one routine can be in the module. Return is made by an RTS instruction or equivalent.

To make these remarks more specific, consider the C fragment:

```
..
extern int funca(double x, double y);
..
main()
{
    ..
    funca(xcoord, ycoord);
    ..
}
```

The assembler code for a routine funca or FUNCA (the linkage is case-insensitive) which is called in this way should be structured as shown below.


```

*.....
* C calling assembler
*.....

        XDEF      FUNCA

        SECTION  .CODE

FUNCA
        MOVEA.L  4(SP),A0    Get address of argument Y
        MOVEA.L  8(SP),A1    Get address of argument X
        ..
        RTS                  Return to C

```

5.3.2 Calling C from assembler

A C function not declared as `static` can be called from assembler code. The function name must be quoted in an XREF directive (or equivalent), and called by a BSR.L instruction (this assumes that the distance between the BSR and the called routine is expressible as a long BSR operand; if this is not the case, another technique must be used which is explained below). If the function requires arguments (see 5.6) they must be pushed on the stack before the call, and removed after it.

To make these remarks more specific, consider the C fragment:

```

int funcc(int *i, int *j)
{ int result;
  ...
  return result;
}

```

The assembler code for a routine which calls function `funcc` should be structured as shown below. In particular, in the large program example, the `.ATAB` section name must be used, so that code base addresses will be relocated correctly.

```
*.....
* Assembler calling C (small program example)
*.....
```

```
      XREF      FUNCC      ; Note that linker ignores case
                          ; (as do most assemblers)
```

```
SECTION .CODE
```

```
      ..
      ..                Set up func's arguments on stack
      PEA      I
      PEA      J
      BSR      FUNCC      Direct call to C function
```

```
*                (func must be within 32K bytes of this BSR)
```

```
      ..
      ..
```

```
I      DS.L    1
J      DS.L    1
```

```
      ..
```

```
*.....
* Assembler calling C (large program example)
*.....
```

```
* This linkage can always be used, but must be used if the
* distance between the calling and target routines exceeds 32K
* bytes.
```

```
      XREF.L    FUNCC      (The .L is essential!)
      XREF      .ATABS     .ATABS is a reserved public symbol
```

```
SECTION .ATAB      .ATAB is a reserved section name
```

```
JMPFNCC JMP      FUNCC      Direct 6-byte jump to FUNCC
```

```
SECTION .CODE
```

```
      ..
      ..                Set up FUNCC's arguments on stack
      PEA      I
      PEA      J
      JSR      JMPFNCC-.ATABS(A4) Indirect call to FUNCC
```

```
*                (A4 contains the address of .ATAB at run-time)
```

```
      ..
```

```
I      DS.L    1
J      DS.L    1
```

5.4 External data

C objects which have been declared as `extern` can be referenced from assembler code as shown below. The use of section `.ATAB` is necessary in order that program initialization can relocate the external data address at run-time.

In the general case, the assembler declarations must describe the layout of the C external object. Section 2.1 gives details of storage layout for different data types.

As an example, the following might appear in a C program:

```
extern struct time { char hours, mins, secs; } timer;

main()
{ ...
  printf("%d:%d:%d \n",
         timer.hours, timer.mins, timer.secs);
```

The method of accessing the object `timer` is as follows

```
* .....
* Accessing an external object
* .....

        XREF      .ATABS

* (.ATABS is a reserved public symbol defining
*  the start of section .ATAB)

*          Layout of struct time

HOURS   EQU      0
MINS    EQU      1
SECS    EQU      2

ATIMER  SECTION  .ATAB      .ATAB is a reserved section name
        DC.L     TIMER + $4000000  Address of TIMER at run-time

        SECTION  .CODE
        MOVEA.L  ATIMER-.ATABS(A4),A0  Get base addr of TIMER

*          (A4 contains the address of .ATAB at run-time)

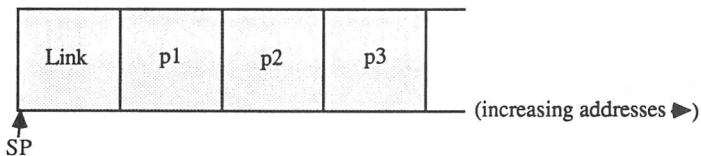
        MOVE.B  HOURS(A0),D0      Get contents of HOURS
        ..
```

5.5 Preservation of registers

The generated C code depends upon the contents of registers A3 to A6 being unchanged on return from a function. Assembler-coded functions must conform with these requirements. On return, the parameters must not have been removed from the stack.

5.6 Parameters

When a function has arguments, these are pushed onto the stack prior to the call. The first argument is pushed last, and so is nearest to the return link (and at the lowest address) on entry to the function.



On return, the link must have been removed, but not the parameters.

Values of scalar types (see Part III, section 3) occupy 1, 2, 4 or 8 bytes (see section 2 above for details). The corresponding number of bytes is pushed onto the stack, except that a 1-byte value is passed by pushing a pair of bytes (with the value in the low-addressed byte of the pair). Note however that if no prototype is in force when the function is called, the default argument promotions will have been made (see Part III, section 7.3.2.1). In the case of 4- or 8-byte values, the high-addressed word is pushed first followed by the lower-addressed word(s).

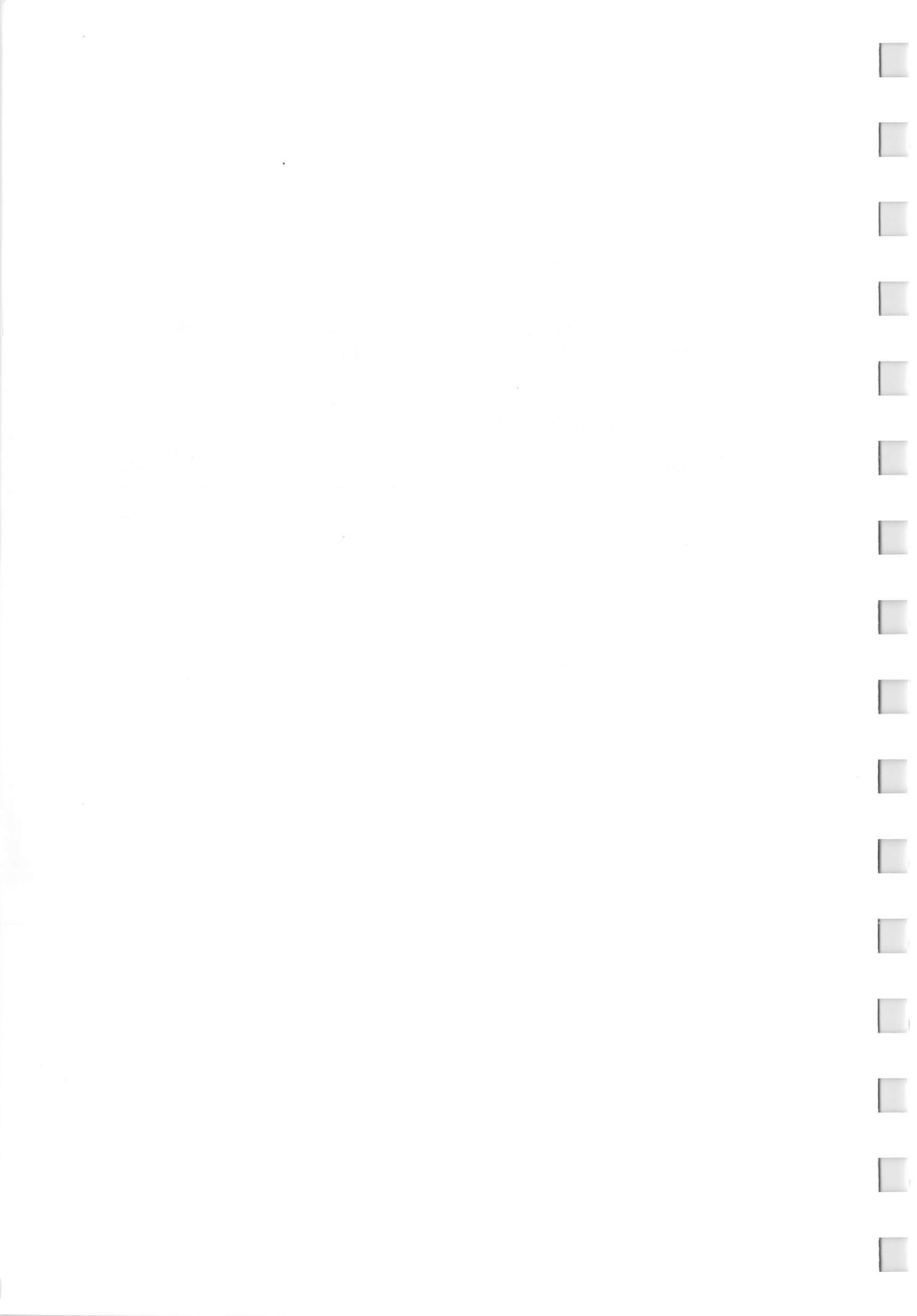
Structure value parameters are passed by making a copy of the structure on the stack. Note that an object of type array or function will be converted to a pointer to array or function before being passed, and so a four-byte pointer will be pushed onto the stack.

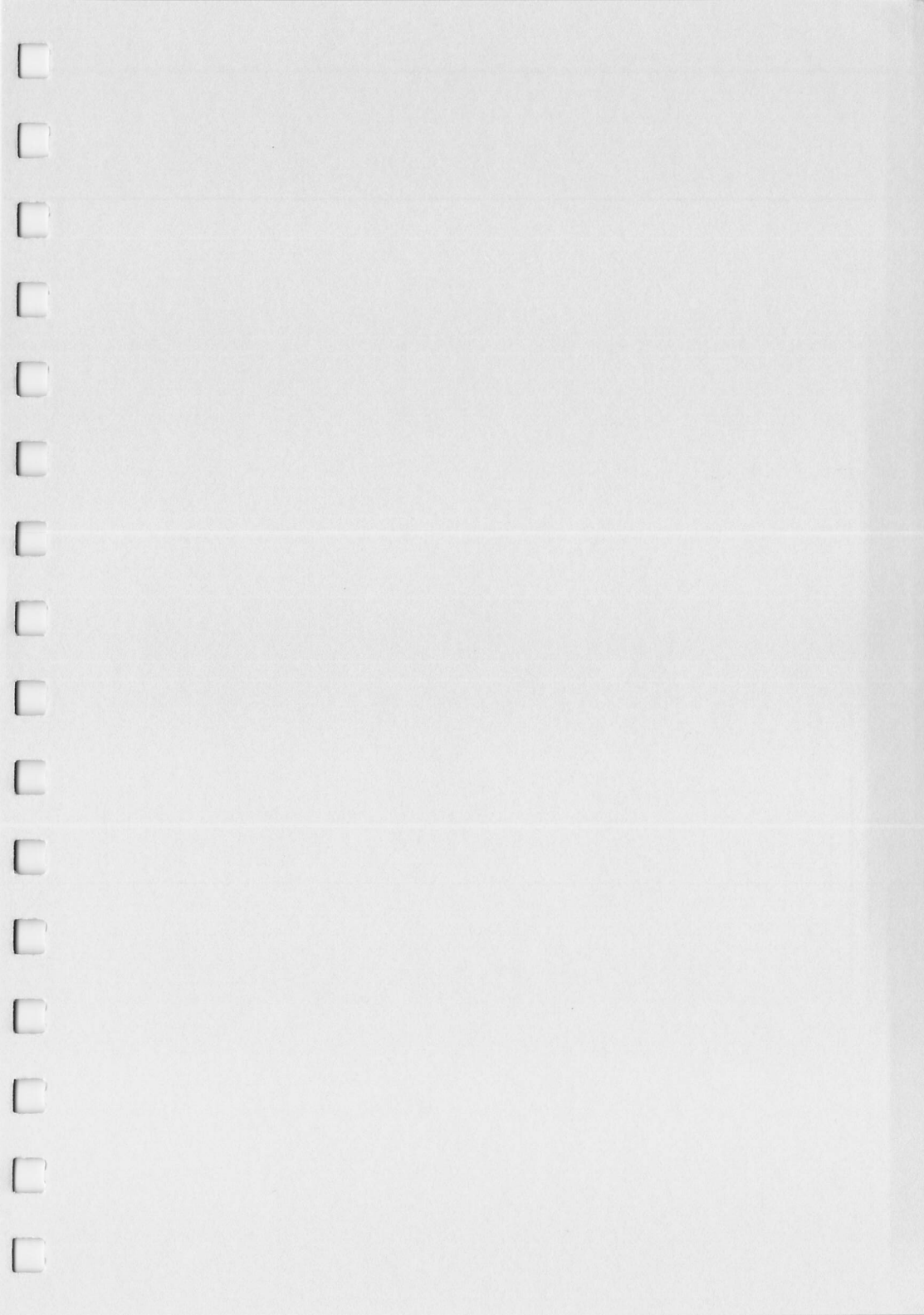
5.7 Function results

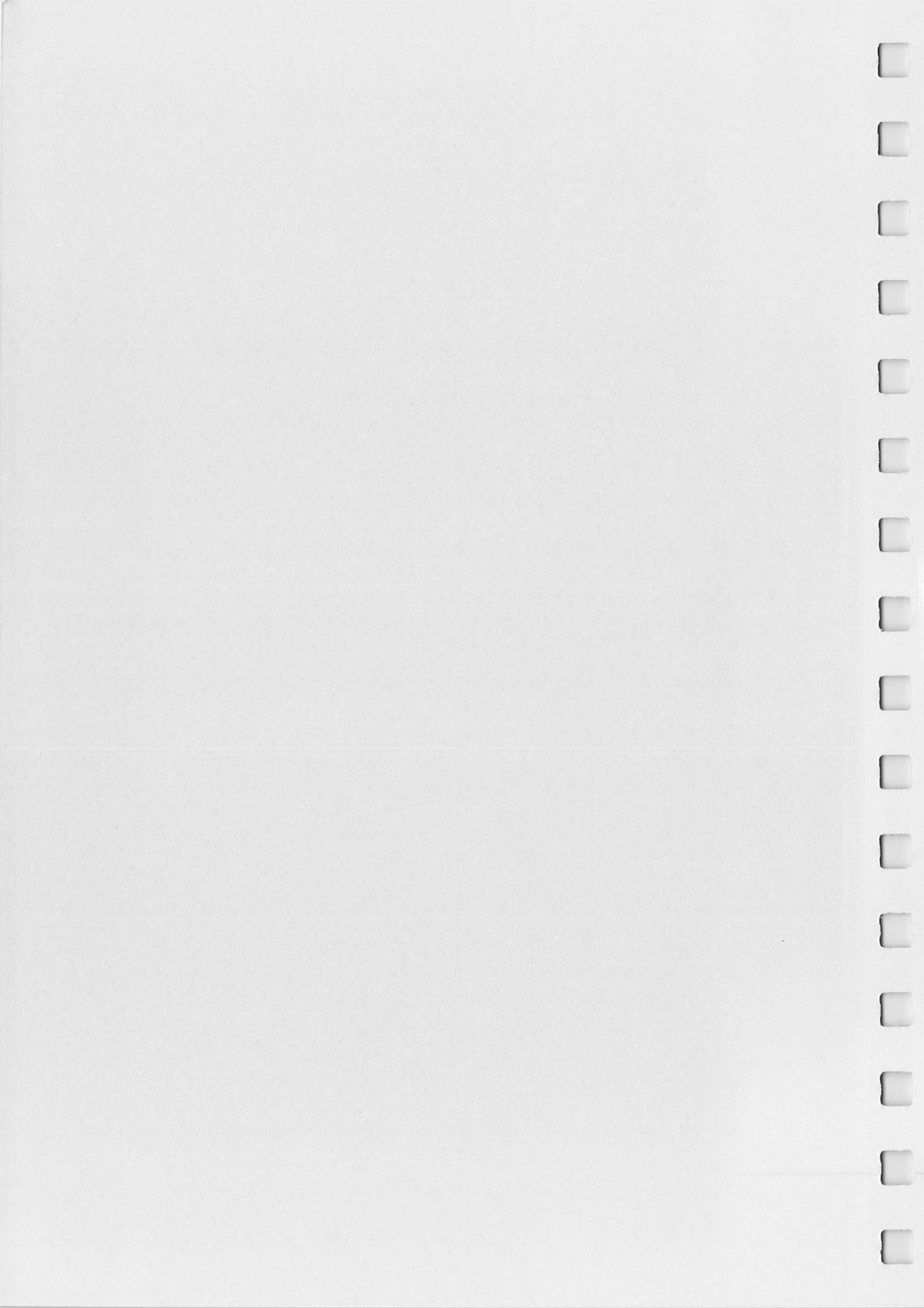
Functions returning integer types return the result in register D0. Functions returning pointer types return the result in register A0. Functions returning floating point types return the result in an area of library workspace known as the floating point accumulator. If a function returns an aggregate type, the caller passes an additional “hidden” parameter on the stack before the actual parameters are passed. The extra parameter is the address of a location in the caller’s data space that is to receive the function result.

5.8 Reserved section names

The section names `.CODE` and `.ATAB` and public symbol `.ATABS` are reserved and should only be used as shown by the examples above. The section names `.INIT`, `.NAME`, `.ENTRY` and `.LWT` are also reserved and must not be used by any assembler routine under any circumstances.







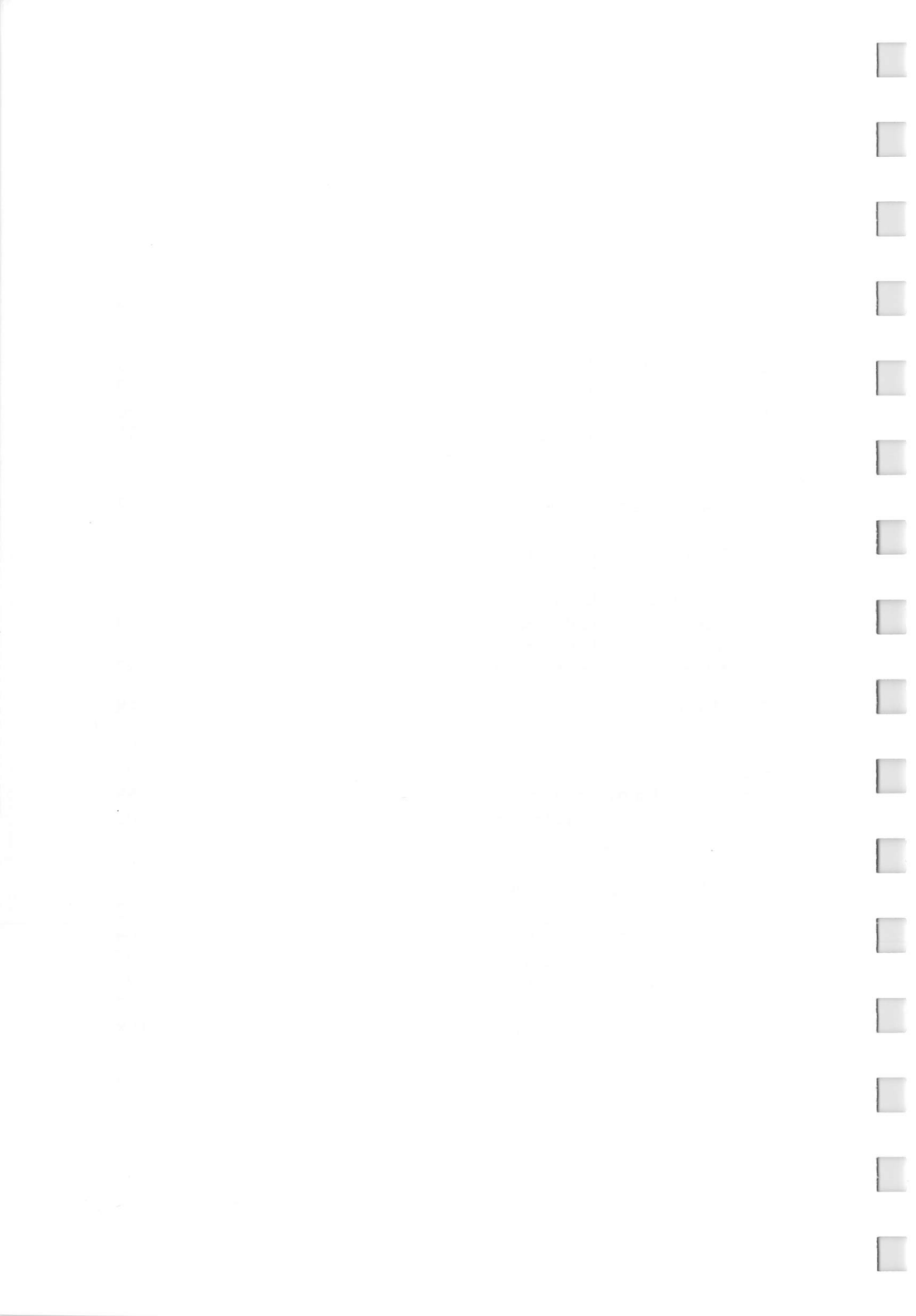
PART III – LANGUAGE DEFINITION

1	Introduction	1
1.1	Notation	1
2	Environment	2
2.1	Program structure	2
2.2	Translation phases	2
2.3	Character sets	4
2.3.1	Source and target character sets	5
2.3.2	Trigraph sequences	5
3	Object types and conversions	6
3.1	Object Types	6
3.2	Conversions	8
3.2.1	Characters and integers	8
3.2.2	Signed and unsigned integers	10
3.2.3	Floating and integral	12
3.2.4	Floating types	12
3.3	Usual arithmetic conversions	13
3.4	Other operands	14
3.4.1	Lvalues and function designators	14
3.4.2	void	14
3.4.3	Pointers	15
4	Object names and scopes	16
4.1	Scopes of identifiers	16
4.2	Linkages of identifiers	18
4.3	Name spaces of identifiers	19
4.4	Storage durations of objects	20
5	Lexical elements	21
5.1	Keywords	22
5.2	Identifiers	22
5.3	Constants	23
5.3.1	Floating constants	24
5.3.2	Integer constants	25
5.3.3	Enumeration constants	28
5.3.4	Character constants	28
5.3.5	Escape sequences	29
5.4	String literals	31
5.5	Operators	32
5.6	Punctuators	32
5.7	Comments	32

6	Declarations	33
6.1	Type specifiers	34
6.2	const and volatile	36
6.3	struct and union specifiers	37
6.3.1	Structure and union tags	40
6.4	Enumeration specifiers	42
6.5	Storage-class specifiers	43
6.6	Declarators	45
6.6.1	Pointer declarators	46
6.6.2	Array declarators	47
6.6.3	Function declarators	48
6.6.4	Type names	50
6.6.5	Type definitions and type equivalence	51
6.7	Function definitions	53
6.8	Initialization	55
6.8.1	Initializing arrays, structs and unions	56
6.9	External definitions	59
6.9.1	External object definitions	60
7	Expressions	61
7.1	Precedence of operators	62
7.2	Primary expressions	63
7.3	Postfix operators	64
7.3.1	Array subscripting	64
7.3.2	Function calls	65
7.3.4	Postfix increment and decrement operators	69
7.4	Unary operators	70
7.4.1	Prefix increment and decrement operators	70
7.4.2	Address and indirection operators	71
7.4.3	Unary arithmetic operators	72
7.4.4	The sizeof operator	73
7.5	Cast operators	74
7.6	Multiplicative operators	75
7.7	Additive operators	76
7.8	Bitwise shift operators	78
7.9	Relational operators	79
7.10	Equality operators	80
7.11	Bitwise AND operator	81
7.12	Bitwise exclusive OR operator	81
7.13	Bitwise inclusive OR operator	81
7.14	Logical AND operator	82
7.15	Logical OR operator	82
7.16	Conditional operator	83

Contents

7.17	Assignment operators	84
7.17.1	Simple assignment	84
7.17.2	Compound assignment	85
7.18	Comma operator	86
7.19	Constant expressions	86
8	Statements	88
8.1	Labelled statements	88
8.2	Compound statement, or block	88
8.3	Expression and null statements	90
8.4	Jump statements	90
8.4.1	The goto statement	91
8.4.2	The continue statement	91
8.4.3	The break statement	92
8.4.4	The return statement	92
8.5	Selection statements	94
8.5.1	The if statement	94
8.5.2	The switch statement	94
8.6	Iteration statements	96
8.6.1	The while statement	96
8.6.2	The do statement	96
8.6.3	The for statement	97
9	The Preprocessor	98
9.1	Introduction	98
9.2	Preprocessing directives	98
9.3	Defining a macro	99
9.3.1	Redefining a macro name	100
9.3.2	Scope of macro definitions	100
9.3.3	Macro replacement	101
9.3.4	Argument substitution	102
9.3.5	Rescanning and further replacement	102
9.3.6	The # operator	104
9.3.7	The ## operator	105
9.4	Conditional inclusion	106
9.4.1	Evaluating constant expressions	107
9.5	Source file inclusion	108
9.6	Line control	110
9.7	Error directive	110
9.8	Pragma directive	111
9.9	Null directive	111
9.10	Predefined macro names	111
9.11	Preprocessor syntax summary	113
10	Bibliography	115
11	Index	116



1 INTRODUCTION

This manual documents the C language as implemented in Prospero C. The implementation adheres to the draft ANSI C standard (3 August 87).

The C language is a general purpose programming language. It has been designed to allow both efficient, perhaps machine specific, and portable programs to be written.

When creating the draft ANSI C standard, the standardization committee aimed to preserve the traditional spirit of C. The rationale to the draft standard enumerates the following facets of this spirit:

Trust the programmer.

Don't prevent the programmer from doing what needs to be done.

Keep the language small and simple.

Provide only one way to do an operation.

Make it fast, even if it is not guaranteed to be portable.

This manual is intended as a guide to knowledgeable programmers and is not intended to act as an introduction or tutorial to the language.

This manual specifies:

The representation of C programs.

The syntax of the C language.

The semantic rules for interpreting C programs.

The restrictions and limits imposed by the language standard.

The restrictions and limits imposed by this implementation.

1.1 Notation

The following notational conventions are used throughout this manual:

Bold courier font	for C keywords and program fragments.
<i>Italic font</i>	for C syntax
Courier font	for C program examples.
Thing _{opt}	indicates that 'Thing' is optional.

2 ENVIRONMENT

2.1 Program structure

A C program consists of one or more source files. Each source file must be independently processed by the compiler.

A source file together with all the headers and source files included via the preprocessing directive **#include**, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a *translation unit*.

2.2 Translation phases

The process of turning C source text into an executable file can be considered to occur in a series of phases. Note that these stages are in general purely conceptual, and do not take the form of separate ‘passes’ of the compiler – they simply serve to define the order in which characters are converted into tokens, and therefore clarify such points as whether adjacent string literals separated by a comment would be concatenated (yes, as the comment would be replaced by a space character in phase 4, and the literals would therefore be adjacent in phase 6).

The phases are as follows :-

1. Physical source file characters are mapped to the source character set (introducing new-line characters for end-of-line indicators). Trigraph sequences (see section 2.3.2) are replaced by their corresponding single-character internal representation.
2. Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines. A source file that is not empty must end in a new-line character, which must not be immediately preceded by a backslash character (thus lines may not be continued across include files and the last character in a non-empty file must be a newline).
3. The source file is decomposed into preprocessing tokens and sequences of white-space characters (including comments). A source file must not end in a partial preprocessing token or comment. Each comment is replaced by one space character. New-line characters are retained. Multiple white-space characters are replaced by a single white-space character (the process of dividing a source file’s characters into preprocessing tokens is context-dependent, in the case of < on a **#include** preprocessing directive line).
4. Preprocessing directives are executed and macro invocations are expanded. A **#include** preprocessing directive causes the named

header or source file to be processed from phase 1 through phase 4, recursively.

5. Escape sequences in character constants and string literals are converted to single characters in the execution character set.
6. Adjacent string literals are concatenated.
7. White-space characters separating tokens are no longer significant. Preprocessing tokens are converted into (normal) tokens. Unsuccessful conversion of a preprocessing token generates a syntax error. The resulting tokens are syntactically and semantically analyzed and translated.
8. All external object and function references are resolved, i.e., the object code is linked to produce an executable program.

Phases 1 to 6 represent the behaviour of the preprocessor, described in greater detail in section 9. Phase 7 is the responsibility of the remainder of the first pass of the compiler, while the second pass of the compiler and the linker complete phases 7 and 8.

2.3 Character sets

The following characters are in the source and target character sets:

The 52 lower-case and upper-case letters of the English alphabet:

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

The 10 decimal digits:

```
0 1 2 3 4 5 6 7 8 9
```

The 29 graphic characters:

!	exclamation point	;	semi-colon
"	double quote	<	less than
#	hash, or number sign	>	greater than
%	percent	=	equal
&	ampersand	?	question mark
'	single quote	[left bracket
(left parenthesis]	right bracket
)	right parenthesis	/	slash
*	star	^	circumflex
+	plus	_	underscore
-	minus	{	left brace
,	comma	}	right brace
.	period		vertical bar
\	backslash	~	tilde
:	colon		

The space character, and control characters representing horizontal tab (HT), vertical tab (VT), and form feed (FF). The end-of-line indicator (new-line character) has the value ASCII 10. The carriage return character has the value ASCII 13.

Any other characters encountered in a source file (except in a preprocessing token, a character constant, a string literal, or a comment), produce illegal tokens.

The space character, horizontal tab (HT), vertical tab (VT), and form feed (FF), are collectively known as white-space characters.

2.3.1 Source and target character sets

C allows a distinction to be made between the character set used to write the program (source character set) and the character set used in the execution environment that runs the program (target character set). It is possible that the internal encoding of the character sets be different in the two cases. This might occur if a cross compiler were used to compile programs for another processor. In Prospero C this does not occur.

2.3.2 Trigraph sequences

The trigraph sequences enable the input of characters that may not appear on some keyboards. These characters are also defined in the "ISO 646-1983" Invariant Code Set, which is a subset of the seven-bit ASCII code set.

All occurrences in a source file of the following sequences of three characters are replaced with the corresponding single character.

Trigraph	Replaced by
??=	#
??([
??)]
??/	\
??<	{
??>	}
??!	
??'	^
??-	~

There are no other trigraph sequences. Each ? that does not begin one of the trigraphs listed above is not altered.

The following source line:

```
printf("??'??/?/?/");
```

becomes (after replacement of the trigraph sequences ??' and ??/)

```
printf("^?\\"");
```

3 OBJECT TYPES AND CONVERSIONS

A type defines the set of values that an object may take. Every type has a set of operations that may be performed on its values. There are three classes of type:

- Types that designate objects (object types).

- Types that designate functions (function types).

- Types that designate objects but lack the information needed to determine their contents (incomplete types).

3.1 Object types

Characters, integers, and floating-point numbers are collectively called the *basic types*.

character An object large enough to store any member of the execution character set.

integer There are four types of signed integers, **signed char**, **short int**, **int** and **long int**.

A **signed char** occupies the same amount of storage as a *plain char*. A *plain int* has the natural size suggested by the architecture of the execution environment for efficient execution. The other types meet specific minimum limits as given in `<limits.h>`. See appendix G.

For **signed char** and each type of **int**, there is a corresponding **unsigned** type that utilizes the same amount of storage (including the sign bit). The range of non-negative values of a signed type is a subrange of its corresponding unsigned type, and the representation of the same value in each type is the same. A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned type. Unsigned types follow the same alignment as signed types.

floating point There are three floating-point types, called **float**, **double** and **long double**. The set of values of a **float** is a subset of the set of values of a **double**; the set of values of a **double** is a subset of the set of values of a **long double**. In Prospero C, **double** and **long double** are implemented as the same representation.

enumeration	An enumeration comprises a set of named integer constant values. Each distinct enumeration comprises a different enumerated type.
void	The void type specifies an empty set of values.
aggregates	An aggregate is constructed from the basic, enumerated, and incomplete types. The resulting type is called a derived type.
arrays	comprise a contiguously allocated set of members of any one type of object.
structures	comprise a sequentially allocated set of named members of various types of object.
unions	comprise an overlapping set of named members of various types of object.
functions	accept arguments of various types and have a return type of any one incomplete or object type except array or function.
pointers	to functions, to objects of any type, and to incomplete types.

These methods of constructing derived types can be applied recursively.

Types **char** and **int** (of all sizes), both signed and unsigned, and enumerations are collectively called *integral types*.

Types **float**, **double** and **long double** are collectively called *floating types*.

Integral and floating types are collectively called *arithmetic types*.

Arithmetic types and pointers are collectively called *scalar types*.

Arrays and structures are collectively called *aggregate types*.

An array of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage).

A structure, union, or enumeration of unknown content is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure, union, or enumeration tag with its defining content later in the same scope.

Function types do not exist in themselves. A definition of a function creates an identifier of function type. When referred to in an expression, this identifier may cause a sequence of statements to be executed, or the address of the code for those statements to be assigned or compared.

See section 6 for fuller details of type declarations.

3.2 Conversions

Several operators convert operand values from one type to another automatically (implicit conversions). This section specifies the result obtained from such an implicit conversion, as well as those obtained from an explicit conversion.

3.2.1 Characters and integers

A `char`, a `short int` or an `int bit-field`, or their signed or unsigned varieties, or an object that has enumeration type, may be used in an expression wherever an `int` may be used. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise it is converted to an `unsigned int`. These are called the *integral promotions*.

Integral Promotions

Original type	Promoted type
<code>signed char</code>	<code>int</code>
<code>unsigned char</code>	<code>int</code>
<code>short</code>	<code>int</code>
<code>unsigned short</code>	<code>int</code> , if <code>short</code> is narrower than <code>int</code> <code>unsigned</code> , if <code>short</code> is as wide as <code>int</code>
<code>int : N</code>	<code>int</code>
<code>unsigned : N</code>	<code>int</code> , if $N < \text{no. bits in int}$ <code>unsigned</code> , if $N = \text{no. bits in int}$

The integral promotions preserve value including sign. Whether a *plain char* is treated as signed is dependent on the compiler U option.

Effects of conversions

From	To	Effect
signed char	short	sign extend
signed char	long	sign extend
signed char	unsigned char	no representation change -128..-1 maps to 127..255
signed char	unsigned short	sign extend to short , change to unsigned
signed char	unsigned long	sign extend to long , change to unsigned long
short	signed char	preserve low order byte (possible truncation)
short	unsigned char	preserve low order byte (possible truncation)
long	signed char	preserve low order byte (possible truncation)
long	unsigned char	preserve low order byte (possible truncation)
unsigned char	signed char	no representation change 127.. 255 maps to -128 .. -1
unsigned char	short	zero extend
unsigned char	long	zero extend
unsigned char	unsigned short	zero extend
unsigned char	unsigned long	zero extend
unsigned short	signed char	preserve low order byte (possible truncation)
unsigned short	unsigned char	preserve low order byte (possible truncation)
unsigned long	signed char	preserve low order byte (possible truncation)
unsigned long	unsigned char	preserve low order byte (possible truncation)

3.2.2 Signed and unsigned integers

When an unsigned integer is converted to another integral type, if the value can be represented by the new type, its value is unchanged.

When a signed integer is converted to an unsigned integer of equal or greater length, if the value of the signed integer is non-negative, its value is unchanged. Otherwise:

If the unsigned integer is longer, the signed integer is first promoted to a signed integer of the same length as the unsigned integer.

The value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.

In a two's-complement representation, as used by Prospero C, there is no actual change in the bit pattern except filling the high-order bits with copies of the sign bit if the unsigned integer is longer.

When an integer is demoted to a shorter unsigned integer, the result is the non-negative remainder on division by the number one greater than the largest unsigned number that can be represented in the shorter type. When an integer is demoted to a shorter signed integer, or an unsigned integer is converted to a signed integer of equal length, if the value is outside the representable range of the target type, the least significant bits of the value are used to produce the result:

e.g., `0x3fffffff` (**signed long**) converted to **signed int** yields `0xffff` or `-1`, converted to **unsigned int** yields `0xffff` or `65535`.



5 INTERFACING TO ASSEMBLER

5.1 Use of assembly language

To use machine features not available through the C language, routines may be written in assembly language and combined with the generated code during the link-edit process.

5.2 Choice of assembler

The C compiler generates relocatable object code. Assembler language modules may be processed by any assembler which generates the same format, and linked with the other components of the program. In particular, the GST Macro Assembler will be found satisfactory.

5.3 XDEF/XREF linkage

5.3.1 Calling assembler from C

An assembler-coded routine can be called from C in the normal way by a function call. In the assembler module, the name is quoted in an XDEF directive, or made global in some equivalent way. More than one routine can be in the module. Return is made by an RTS instruction or equivalent.

To make these remarks more specific, consider the C fragment:

```
..
extern int funca(double x, double y);
..
main()
{
    ..
    funca(xcoord, ycoord);
    ..
}
```

The assembler code for a routine `funca` or `FUNCA` (the linkage is case-insensitive) which is called in this way should be structured as shown below.

3.2.3 Floating and integral

When a value of floating type is converted to integral type, the fractional part is discarded. If the value of the integral part cannot be represented in the space provided, the behavior is undefined.

When a value of integral type is converted to floating type, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value.

Effects of conversions

From	To	Effect
any floating point type	any integral type	discard fractional part and change to appropriate integral representation (as if converted to unsigned long first)
any integral type	any floating point type	represent as floating point value nearest to original value

3.2.4 Floating types

When a **float** is promoted to **double** or **long double**, or a **double** is promoted to **long double**, its value is unchanged.

When a **double** or **long double** is demoted to **float** (in Prospero C a **long double** is the same as a **double**, so there is no change between these two), if the value being converted is outside the range of values that can be represented, the behavior is undefined. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value.

Effects of conversions

From	To	Effect
float	double	no loss of precision
float	long double	no loss of precision
double	float	possible loss of precision and/or range
double	long double	no change of representation
long double	float	possible loss of precision and/or range
long double	double	no change of representation

3.3 Usual arithmetic conversions

The term *usual arithmetic conversions* refers to the process of converting the operands of an operator to the same, defined type. The purpose of these conversions is to reduce the possible combinations of operands that an operator would actually have to operate on. The given type is also the type of the resulting value (for most operators; see sections 7.7 to 7.18 for some exceptions to this rule).

```
IF      either operand has type long double,
        the other operand is converted to long double.
ELIF    either operand has type double,
        the other operand is converted to double.
ELIF    either operand has type float,
        the other operand is converted to float.
ELIF    either operand has type unsigned long int,
        the other operand is converted to unsigned long int.
ELIF    either operand has type long int,
        the other operand is converted to long int.
ELSE    the integral promotions are performed (see section 3.2.1),
        after which
        IF      either operand has type unsigned int,
                the other operand is converted to unsigned int.
        ELSE    both operands have type int.
```

3.4 Other operands

3.4.1 Lvalues and function designators

An lvalue is an expression (with an object type or an incomplete type other than `void`) that designates an object.

The name *lvalue* comes originally from the assignment expression `E1 = E2`, in which the left operand `E1` must be a (modifiable) lvalue.

When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A modifiable lvalue is an lvalue that:

- Does not have array type.

- Does not have an incomplete type.

- Does not have a type declared with the `const` type specifier.

- For a `struct` or `union`, does not have any member (including, recursively, any member of all contained structures or unions) declared with the `const` type specifier.

An lvalue that has type *array of type* is converted to an expression that has type *pointer to type* that points to the initial member of the array object and is not an lvalue. This conversion does not occur when the lvalue is the operand of the `sizeof` operator or the unary `&` operator, or is a string literal used to initialize an array of characters.

A function designator is an expression that has function type. Except when it is the operand of the `sizeof` operator or the unary `&` operator, a function designator with type *function returning type* is converted to an expression that has type *pointer to function returning type*.

3.4.2 `void`

The (nonexistent) value of a *void expression* `void` may not be used in any way, and implicit or explicit conversions may not be applied to such an expression. If an expression of any other type occurs in a context where a *void expression* is required, its value is discarded.

3.4.3 Pointers

A pointer to **void** may be converted to a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a *pointer to void* and back again; the result will compare equal to the original pointer.

An integral constant expression with the value 0, or such an expression cast to type **void ***, is called a null pointer constant. If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. The null pointer constant is guaranteed to compare unequal to a pointer to any object or function.

4 OBJECT NAMES AND SCOPES

In C, names may be given to objects (storage locations), types (an interpretation of the operations that may be performed on an object), functions (a collection of declarations and executable statements), macro names (a mechanism for giving a meaningful name to a sequence of tokens), and tags (a sort of second class type concept).

There are rules for deciding which, if any, of these names are visible at a point in the translation unit, whether it is possible to temporarily reuse a name (scope and name space), how identical names may connect together (linkage), and the period of time (relative to program execution) that storage is reserved for names (of objects).

4.1 Scopes of identifiers

An identifier is visible (i.e., can be used) only within a region of program text called its *scope*. There are four kinds of scope:

Block	If a declaration or type name appears inside a block, or within the list of parameter identifiers in a function definition, the identifier has <i>block scope</i> . This scope terminates at the } that closes the associated block.
File	If a declaration or type name appears outside any block or list of parameters, it has <i>file scope</i> .
Function	A label name is the only kind of identifier that has <i>function scope</i> . It can be used (in a goto statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance. Label names must be unique within a function.
Function prototype	A function prototype is a declaration of a function that declares the types of its parameters. If a declaration or type name appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has <i>function prototype scope</i> , which terminates at the end of the function declarator. Any outer declaration of an identical identifier in the same name space is hidden until the current scope terminates.

struct, **union** and **enum** tags have scope that begins just after the appearance of the tag in a type specifier. Each enumeration constant has scope that begins just after the appearance of the enumeration constant in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.

e.g.,

```
int i;                /* global i */

void fs(i)            /* introduces a new i */
long i;
{ /* ... */
  { float i;          /* introduces another i */
    /* ... */
  }
}
```

4.2 Linkages of identifiers

Linkage is the name given to the process whereby two or more lexically identical identifiers are made to refer to the same object or function.

There are three kinds of linkage:

External If the declaration of an identifier for an object or a function contains the storage-class specifier **extern**, the identifier has the same linkage as any in-scope declaration of the identifier with file scope. If there is no in-scope declaration with file scope, the identifier has external linkage. Also if the lexically first declaration of an identifier (declared as an object or function) with file scope in the translation unit does not have a storage-class specifier given, it has external linkage. In the set of translation units that constitutes a complete program, each instance of a particular identifier with external linkage denotes the same object or function.

Internal If the lexically first declaration of an identifier (declared as an object or function) with file scope in the translation unit contains the storage-class specifier **static**, the identifier has internal linkage. Within one translation unit, each instance of an identifier with internal linkage denotes the same object or function.

None Identifiers with no linkage denote unique entities. The following identifiers have no linkage:

An identifier declared to be anything other than an object or a function.

An identifier declared to be a function parameter.

An identifier declared to be an object inside a block without the storage-class specifier **extern**.

If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** operator, which is evaluated at compile time), somewhere in the entire program there must be exactly one external definition for the identifier (a declaration that has external linkage and for which storage is allocated).

If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

4.3 Name spaces of identifiers

In C the same identifier may be associated with up to 4 program entities at the same time. This overloading of an identifier is resolved by examining the syntactic context in which it occurs. Thus, there are separate name spaces for various categories of identifiers:

Label names (differentiated by the syntax of the label declaration and `goto` keyword).

Tags of structures, unions, and enumerations. They are differentiated by the preceding `struct`, `union` or `enum` keyword. There is only one name space for all tags.

Members of structures or unions. Each structure or union has a separate name space for its members (differentiated by the type of the expression used to access the member via the `.` or `->` operator).

All other identifiers, called ordinary identifiers (i.e., variable and function names, enumeration constants, typedef names and function parameters).

Macro names. These are handled at the lexical level and are not part of the C syntax.

It is illegal to have more than one identically named identifier in the same name space, in any one scope. The one exception to this rule is the *forward declaration* of tags – see section 6.3.1.

An example illustrating these points is :-

```
struct dup {                /* tag                */
    int dup;                /* member          */
} dup;                      /* ordinary identifier */

if (dup.dup == 1)
    goto dup;              /* label          */
{                          /* a different scope */
    int dup = 1;
    struct dup {
        float dup;
    } x;
dup :                      /* ...           */
}
```

4.4 Storage durations of objects

An object has a storage duration that determines its lifetime. There are two classes of storage duration:

static An object declared with **static** storage duration is created and initialized only once, prior to program startup. It exists and retains its last-stored value throughout the execution of the entire program. All objects with external linkage have static storage duration.

auto A new instance of an object declared with automatic storage duration is created, in theory, on each normal entry into the block in which it is declared. In practice Prospero C allocates sufficient space on entry to the function to accommodate the largest storage requirement. If an initialization is specified, it is performed on each normal entry, but not if the block is entered by a jump to a label. The object is discarded when execution of the block ends in any way.

5 LEXICAL ELEMENTS

A token is the minimal lexical element of the language. The categories of tokens are:

keywords

identifiers

constants

“string literals”

operators

punctuators

White-space, and comments are ignored except as they separate tokens. White space may appear within a token only as part of a character constant or string literal, i.e., it is significant.

Each keyword, identifier, or constant must be separated by some white space from any otherwise adjacent keyword, identifier, or constant.

The program fragment `0xG` is parsed as an invalid token, even though a valid parse might be obtained if the identifier `x`, or `G`, previously defined as a macro name, were replaced by its macro definition (for example, if `x` were defined as `+`). Similarly, the program fragment `0xF` is parsed as a (valid) hex constant token, whether or not `x`, or `F`, is a macro name.

If the input stream has been parsed into tokens up to a given character, the next token is the longest sequence of characters (not separated by white-space) that could constitute a token.

The program fragment `x+=++y` is parsed as `x += ++ y`.

token:

keyword

identifier

constant

string-literal

operator

punctuator

5.1 Keywords

The following tokens (entirely in lower-case) are reserved (in translation phases 5 through 8) for use as keywords, and may not be redeclared as objects:

<code>auto</code>	<code>double</code>	<code>int</code>	<code>struct</code>
<code>break</code>	<code>else</code>	<code>long</code>	<code>switch</code>
<code>case</code>	<code>enum</code>	<code>register</code>	<code>typedef</code>
<code>char</code>	<code>extern</code>	<code>return</code>	<code>union</code>
<code>const</code>	<code>float</code>	<code>short</code>	<code>unsigned</code>
<code>continue</code>	<code>for</code>	<code>signed</code>	<code>void</code>
<code>default</code>	<code>goto</code>	<code>sizeof</code>	<code>volatile</code>
<code>do</code>	<code>if</code>	<code>static</code>	<code>while</code>

A Prospero C extension reserves the words (when the S option is off):

`fortran pascal`

5.2 Identifiers

An identifier denotes an object, a function, or one of the following entities that will be described later:

a tag or a member of a structure, union, or enumeration.

a typedef name.

a label name.

a macro name.

Macro names are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the token sequences that constitute their macro definitions. Refer to section 9 for further details.

The case of letters is significant in determining if two identifiers are the same, e.g., `abc`, `ABC`, `AbC` are different from each other.

Names may be both internal and external. There is no limit on the length of internal names. The maximum number of significant characters in an internal name is 31. External names, i.e., those names imported or exported to other translation units, have a limit of 31 significant characters, and the linker does not treat case as significant.

identifier:

non-digit

identifier non-digit

identifier digit

non-digit: one of

–	a	b	c	d	e	f	g	h	i	j	k	l	m
	n	o	p	q	r	s	t	u	v	w	x	y	z
	A	B	C	D	E	F	G	H	I	J	K	L	M
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

\$ (unless S option is specified)

digit: one of

0 1 2 3 4 5 6 7 8 9

An identifier is a sequence of non-digit characters (including the underscore `_` and the lower-case and upper-case letters) and digits. The first character must be a non-digit character.

An identifier occurring in translation phases 7 and 8 may not consist of the same sequence of characters as a keyword. It is possible to define a macro name that has the same spelling as a keyword.

The following are all valid identifiers:

```
L001      _bufptr      WHILE      array      integer
Number_Of_Elements
```

5.3 Constants

Each constant has a type, determined by its form and value, see sections 5.3.1 and 5.3.2

constant:

floating-constant

integer-constant

enumeration-constant

character-constant

5.3.1 Floating constants

A floating constant has a value part that may be followed by an exponent part and a suffix that specifies its type.

The digit sequences are interpreted as decimal integers. The exponent indicates the power of 10 by which the value part is to be scaled.

An unsuffixed floating constant has type **double**. If suffixed by the letter **f** or **F**, it has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**.

The following are all valid floating point constants:

```
1.0e36  12e6L  .618  .01e-9  3.14159f
42.     012e4
```

floating-constant:

fractional-constant *exponent*_{opt} *floating-suffix*_{opt}
digit-sequence *exponent* *floating-suffix*_{opt}

fractional-constant:

*digit-sequence*_{opt} . *digit-sequence*
digit-sequence .

exponent:

e *sign*_{opt} *digit-sequence*
E *sign*_{opt} *digit-sequence*

sign: one of

+ -

digit-sequence:

digit
digit-sequence digit

floating-suffix: one of

f l F L

5.3.2 Integer constants

An integer constant begins with a digit, and does not contain a period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.

Decimal constant	Starts with a non-zero digit and consists of a sequence of decimal digits.
Octal constant	Starts with the prefix <code>0</code> optionally followed by a sequence of the digits <code>0</code> through <code>7</code> only.
Hexadecimal constant	Starts with the prefix <code>0x</code> or <code>0X</code> followed by a sequence of the decimal digits and the letters <code>a</code> (or <code>A</code>) through <code>f</code> (or <code>F</code>) representing values 10 through 15 respectively.

The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16.

The type of an integer constant is the first of the following list in which its value can be represented.

Unsuffix decimal:

`int, long int, unsigned long int`

e.g.,

<code>1</code>	<code>int</code>
<code>80000</code>	<code>long int</code>
<code>300000000</code>	<code>unsigned long int</code>

Unsuffix octal or hexadecimal:

`int, unsigned int, long int, unsigned long int`

e.g.,

<code>07</code>	<code>int</code>
<code>040000</code>	<code>unsigned int</code>
<code>0100000</code>	<code>long int</code>
<code>0260000000</code>	<code>unsigned long int</code>
<code>0x0</code>	<code>int</code>
<code>0xfeed</code>	<code>unsigned int</code>
<code>0x10000</code>	<code>long int</code>
<code>0xf000ffff</code>	<code>unsigned long int</code>

Suffixed by the letter `u` or `U`:

`unsigned int`, `unsigned long int`

e.g.,

<code>2u</code>	<code>unsigned int</code>
<code>80000u</code>	<code>unsigned long int</code>
<code>066u</code>	<code>unsigned int</code>
<code>0660000u</code>	<code>unsigned long int</code>
<code>0xcu</code>	<code>unsigned int</code>
<code>0xfacedU</code>	<code>unsigned long int</code>

Suffixed by the letter `l` or `L`:

`long int`, `unsigned long int`

e.g.,

<code>911</code>	<code>long int</code>
<code>30000000001</code>	<code>unsigned long int</code>
<code>0341</code>	<code>long int</code>
<code>0333333333331</code>	<code>unsigned long int</code>
<code>0xab01</code>	<code>long int</code>
<code>0xace0140f1</code>	<code>unsigned long int</code>

Suffixed by both the letters `u` or `U` and `l` or `L`:

`unsigned long int`.

e.g.,

<code>101ul</code>	<code>unsigned long int</code>
<code>0lu</code>	<code>unsigned long int</code>
<code>0x3ful</code>	<code>unsigned long int</code>

Note: Negative values such as `-38` actually consist of the unary minus applied to a positive constant.. The compiler front end actually does the calculation, no code is generated. However, the type of the constant is decided before the unary minus is applied. Thus `-32768` is actually a `long int` because `32768` is a `long int` (see unsuffixed decimal constants above). However, `-0x8000` is an `unsigned int` because `0x8000` is `unsigned int`.

integer-constant:

decimal-constant integer-suffix_{opt}

octal-constant integer-suffix_{opt}

hexadecimal-constant integer-suffix_{opt}

decimal-constant:

non-zero-digit

decimal-constant digit

octal-constant:

0

octal-constant octal-digit

hexadecimal-constant:

0x hexadecimal-digit

0X hexadecimal-digit

hexadecimal-constant hexadecimal-digit

non-zero-digit: one of

1 2 3 4 5 6 7 8 9

octal-digit: one of

0 1 2 3 4 5 6 7

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

integer-suffix:

unsigned-suffix long-suffix_{opt}

long-suffix unsigned-suffix_{opt}

unsigned-suffix: one of

u U

long-suffix: one of

l L

5.3.3 Enumeration constants

An identifier declared as an enumeration constant has type **int**.

Numeric values are assigned to the **enum** constants starting at 0 and incrementing by 1. As described later this sequence may be interrupted and the numeric values need not be unique.

enumeration-constant:
identifier

5.3.4 Character constants

A character constant is a sequence of one or more characters enclosed in single-quotes, as in '**z**' or '**ws**'. The characters may be any characters in the source character set, or specified by an escape sequence, as described in section 5.3.5. They are mapped in one character per byte to characters in the target character set.

The double-quote " and question-mark ? are representable either by themselves or by the escape sequences \" and \? respectively, but the single-quote ' can only be represented by the escape sequence \'.

A character constant has type **int**. The value of a character constant containing a single character is the numerical value of the representation of the character interpreted as an integer. Prospero C places each character in a byte, left to right in the word. As many characters are allowed in a character constant as bytes that occupy an **int**, i.e., 2. By default the type **char** is treated the same as **signed char**, the high-order bit position of a single-character character constant is treated as a sign bit. If the U (unsigned) compiler option is enabled then **char** is treated as **unsigned char**.

e.g., 'a' '\ ' ' ' '\7' '\n' '\x20'

character-constant:
'c-char-sequence'

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any character in the source character set except
the single-quote ', backslash \, or new-line
escape-sequence

5.3.5 Escape sequences

Escape sequences or characters are used in character and string constants to represent characters that might otherwise be impossible to enter directly into the source text.

Alphabetic escape sequences representing non-graphic characters in the target character set produce the following effects on display devices:

<code>\a</code> ("alert")	Produces an audible or visible alert. The active position is not changed. It is equivalent to an ASCII value of 7.
<code>\b</code> ("backspace")	Moves the active position one character to the left. If the active position was at the beginning of a line, the behavior is unspecified. It is equivalent to an ASCII value of 8.
<code>\f</code> ("form feed")	Moves the active position to the initial position at the start of the next logical page. It is equivalent to an ASCII value of 12.
<code>\n</code> ("new line")	Moves the active position to the initial position of the next line. It is equivalent to an ASCII value of 10.
<code>\r</code> ("carriage return")	Moves the active position to the initial position of the current line. It is equivalent to an ASCII value of 13.
<code>\t</code> ("horizontal tab")	Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is equivalent to an ASCII value of 9.
<code>\v</code> ("vertical tab")	Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position, the behavior is unspecified.
<code>\'</code>	Output a single quote '
<code>\"</code>	Output a double quote "
<code>\?</code>	Output a question mark ?
<code>\\</code>	Output a backslash \

It is also possible to represent characters by specifying their bit pattern; either in octal or hexadecimal.

Up to three octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character. The numerical value of the octal integer specifies the value of the desired character. E.g., the escape sequence `\033` represents the character whose code is 27, and the construction `\0` is commonly used to represent the null character.

The hexadecimal digits that follow the backslash and the letter `x`, or `X` in a hexadecimal escape sequence are taken to be part of the construction of a single character. The numerical value of the hexadecimal integer so formed specifies the value of the desired character. E.g., the escape sequence `\0x1a` represents the character whose code is 26.

If any other escape sequence is encountered a warning is given and the backslash character returned.

Even though eight bits are used for objects that have type `char`, the construction `\x123` specifies a character constant containing only one character. To specify a character constant containing the two characters whose values are `0x12` and `'3'`, the construction `\0223` may be used, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character.

See section 5.3.4 for a full discussion of character constants.

escape-sequence:

simple-escape-sequence

octal-escape-sequence

hexadecimal-escape-sequence

simple-escape-sequence: one of

`\' \\" \? \\ \a \b \f \n \r \t \v`

octal-escape-sequence:

`\ octal-digit`

`\ octal-digit octal-digit`

`\ octal-digit octal-digit octal-digit`

hexadecimal-escape-sequence:

`\x hex-digit`

`hexadecimal-escape-sequence hex-digit`

5.4 String literals

A string literal is a sequence of zero or more characters enclosed in double-quotes, as in `"axc"`.

The same considerations apply to each character in a string literal as if it were in a character constant, except that the single-quote `'` is representable either by itself or by the escape sequence `\'`, but the double-quote `"` can only be represented by the escape sequence `\"`.

A string literal has static storage duration and type *array of char*. It is initialized with the given characters. String literals that are adjacent tokens are concatenated into a single string literal (in translation phase 6). A null character is appended onto the end of all string literals, so that `sizeof("Jimmy")` has value 6.

This pair of adjacent string literals:

```
"\x01" "2"
```

produces a single string literal containing the two characters whose values are `'\x01'` and `'2'`. This is because escape sequences are converted into single characters in the execution character set just prior to adjacent string literal concatenation.

e.g.,

```
"Hello\n"           "The bell\a tolled"
"\0\1\2"          "\"
"a string\
occupying more than one line"
```

string-literal:

```
"s-char-sequenceopt"
```

s-char-sequence:

```
s-char
```

```
s-char-sequence s-char
```

s-char:

```
any character in the source character set except the double-quote ",
backslash \, or new-line
```

```
escape-sequence
```

5.5 Operators

An operator specifies an operation to be performed that yields a value (an evaluation). An operand is an entity on which the operator acts. See section 7 for full details.

operator: one of

```
[ ] ( ) . ->
++ -- & * + - ~ ! sizeof
/ % << >> < > <= >= == !=
^ | && ||
? :
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
```

5.6 Punctuators

A punctuator is a symbol that has independent syntactic and semantic significance but does not specify an operation to be performed that yields a value. Depending on context, the same symbol may also represent an operator or part of an operator.

punctuator: one of

```
[ ] ( ) { } * , : = ; ... #
```

5.7 Comments

Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. In addition, if the characters `/*` occur within a header name preprocessing token, the behavior is undefined. The contents of a comment are examined only to find the characters `*/` that terminate it.

Thus comments do not nest.

6 DECLARATIONS

Declaring an identifier in C causes it to be associated with some object. The object could be a type, a variable or a function. A declaration specifies how the identifier should be interpreted in an expression and also gives it a set of attributes. A declaration that also causes storage to be reserved for an object or function named by an identifier is a definition.

The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. A declaration with an *init-declarator-list* is a defining occurrence and specifies an initial value. The declarators give the identifiers being declared.

Although allowed by the syntax, it is an error to write an empty declarator.

declaration:

declaration-specifiers init-declarator-list_{opt};

declaration-specifiers:

storage-class-specifier declaration-specifiers_{opt}

type-specifier declaration-specifiers_{opt}

init-declarator-list:

init-declarator

init-declarator-list , init-declarator

init-declarator:

declarator

declarator = initializer

6.1 Type specifiers

C provides a set of basic types and a method of creating aggregates of one or more of these types.

The type specifier provides the basic types. Objects may be declared to have one of these basic types, e.g., `int i;`, and aggregate types built from them, e.g., `char s[3];`

long or short	Only one of these may be given in a type-specifier, optionally in conjunction with an int . long may also be specified in conjunction with double .
signed or unsigned	Only one of these may be given in a type-specifier (in conjunction with int , long , or short). They may also be specified alone in which case the presence of int is implied.
const and volatile	These may be specified alone (in which case the presence of int is implied), or in conjunction with other type specifiers. They are also the only specifiers that may be used in conjunction with a struct , union , or enumeration specifier, or with a typedef name.

If there are no type specifiers in the declaration specifiers in a declaration, the type is taken to be **int**.

Basic types	Abbreviations
integer types	
signed char	char
signed short	short
signed int	int
signed long	long
unsigned char	char
unsigned short int	unsigned short
unsigned int	unsigned
unsigned long int	unsigned long
floating types	
float	
double	
long double	
other types	
void	

Type	Storage	Range of values
signed char	1 byte	-128..127
signed short	2 bytes	-32768 .. 32767
signed int	2 bytes	-32768 .. 32767
signed long	4 bytes	-2147483648 .. 2147483647
unsigned char	1 byte	0 .. 255
unsigned short int	2 bytes	0 .. 65535
unsigned int	2 bytes	0 .. 65535
unsigned long int	4 bytes	0 .. 4294967295
float	4 bytes	0, $\pm 1.17549435e-38$.. $3.40282347e+38$
double	8 bytes	0, $\pm 2.225073858507201e-308$.. $1.797693134862316e+308$
long double	8 bytes	same as double
void		

type-specifier:

void

char

short

int

long

float

double

signed

unsigned

const

volatile

struct-or-union-specifier

enum-specifier

typedef-name

6.2 `const` and `volatile`

The idea behind the `const` type specifier is to provide the programmer with a method of telling the compiler that objects may be regarded as “read-only”. Objects declared with the `const` specifier may only be given a value via an initializer or, if the object is a parameter variable, by the value being passed as an argument to the function of which the object is a parameter.

The `volatile` type specifier is a method of telling the compiler that the values of certain objects may be modified through external means, e.g., memory mapped I/O.

If a *pointer to a `const`* object is converted (by an explicit cast) to a pointer to a type without the `const` attribute, modifying the object by means of the *pointer to non-`const`* will succeed, but may lead to undefined behavior.

An object whose type includes the `volatile` type specifier may be modified in ways unknown to Prospero C, or have unknown side effects. Therefore any expression referring to such an object is evaluated strictly according to the sequence rules of the abstract machine. Furthermore, at every sequence point the value of the object in storage agrees with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.

If a pointer to a `volatile` object is converted (by an explicit cast) to a pointer to a type without the `volatile` attribute, and the object is referred to by means of the *pointer to non-`volatile`*, the behavior is undefined.

If a declaration includes `const` or `volatile` modifying the type of a declarator that has aggregate type or `union` type, the type of each member of the aggregate or `union` inherits the modifying type specifier.

An object declared:

```
extern volatile const real_time_clock;
```

may be modified by hardware, but cannot be modified by the program.

Although syntactically `const` and `volatile` are two type specifiers, they actually modify declarators.

e.g.,

```
const struct s { int f1; } a;
struct s b;
```

here **a** is a `const` object, but **b** is not.

```
typedef const struct s { int f1; } cs;
cs a;
cs b;
```

here both **a** and **b** are `const` objects.

6.3 struct and union specifiers

A **struct** is a type consisting of an ordered sequence of named members.

e.g.,

```
struct {
    int    first;
    char   second;
    float  *third[5];
}
```

A **union** is a type consisting of an overlapping sequence of named members. A **union** may be thought of as a **struct** all of whose members begin at the same address. The value of at most one of the members can be stored in a **union** object at any time.

e.g.,

```
union {
    int    ival;
    char   cval;
    float  *apf[5];
} x;

x.ival = 0;
/* ... */ x.ival /* ... */

x.cval = 'A';
/* ... */ x.cval /* ... */

*x.apf[3] = 1.2;
/* .. */ *x.apf[3] /* ... */
```

The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type. The type is incomplete until after the } that terminates the list.

A **struct** or **union** may not contain a member with incomplete or function type. Hence it must not contain an instance of itself (but may contain a pointer to an instance of itself).

A member of a **struct** or **union** may have any object type. In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a bit-field. Bit-fields provide a method of storing more than one object in a word of storage. This is done by specifying the number of bits occupied by the object, its width.

The constant expression that specifies the width of a bit-field must have integral type and non-negative value that must not exceed the number of bits in an `int`. If the width is zero, the declaration may not have a declarator.

The unary `&` address (address-of) operator may not be applied to a bit-field object, thus there are no pointers to bit-fields.

A bit-field may have type `int`, `unsigned int`, or `signed int`. The high-order bit position of a plain `int` bit-field is treated as a sign bit. A bit-field is interpreted as an integral type.

Prospero C will allocate sufficient storage to hold a bit-field. If enough space remains, a bit-field that follows another bit-field will be packed into adjacent bits of the word. If insufficient space remains, space is allocated from the next word.

The first bit field encountered is the least significant in the byte (or word) it is allocated.

e.g.,

```
struct {
    int aa : 2;
    int bbb : 3;
    int : 7;
    int c : 1;
    int ddd : 3;
}
```

will pack as `dddxxxxxxxxbbbaa` in a 16 bit `int`. (Bit fields will be considered as byte streams with bits numbered as follows: [7..0] [15..8] [23..16] etc., so the above example would be considered as `[xxxxxxxx]` [`dddcbbaa`]) Consecutive bit-fields that would all pack into one byte will only be allocated one byte.

Bit-fields can straddle byte boundaries, but not word boundaries.

A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.

e.g.,

```
struct {
    int rdy_flag : 2;
    int : 4;
    int rcv_flag : 1;
    int snd_flag : 1;
}
```

As a special case of this, a bit-field with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

Within a **struct** object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a **struct** object, suitably cast, points to its initial member or, if it is a bit-field, to the unit in which it resides. Pointers to subsequent fields compare greater than earlier fields.

There may be unnamed holes within a **struct** object. These are added by the compiler to achieve the appropriate alignment. Any object whose size is greater than one byte will be allocated space starting on a word boundary, and an even number of bytes. If the size of a **struct** is more than two bytes, there may also be unnamed padding at the end to make its size even, in order to achieve the appropriate alignment were the **struct** to be a member of an array.

e.g.,

```
struct {
    char f1;
    int  f2;
    char f3;
} s1;
```

There is one padding byte between the first and second field and at the end of the structure; **sizeof(s1)** is equal to 6.

```
struct {
    char f1;
    char f2;
    char f3;
} s2;
```

There is a padding byte at the end of the structure; **sizeof(s2)** is equal to 4.

struct-or-union-specifier:

*struct-or-union identifier*_{opt} { *struct-declaration-list* }
struct-or-union identifier

struct-or-union:

struct
union

```

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    type-specifier-list struct-declarator-list ;

type-specifier-list:
    type-specifier
    type-specifier-list type-specifier

struct-declarator-list:
    struct-declarator
    struct-declarator-list , struct-declarator

struct-declarator:
    declarator
    declaratoropt : constant-expression

```

6.3.1 Structure and union tags

A **struct** or **union** specifier of the form:

```
struct-or-union identifier { struct-declaration-list }
```

declares the identifier to be the tag of the **struct** or **union** specified by the list. A subsequent declaration in the same scope may then use the tag, but the bracketed declaration list must be omitted.

A specifier of the form:

```
struct-or-union identifier
```

is an incomplete type. It declares a tag that may be used only when the size of an object of the specified type is not needed. Such a construct is needed when defining mutually referencing structs.

e.g.,

```

struct s2;
struct s1 { struct s2 *s2p; /*...*/ }; /* D1 */
struct s2 { struct s1 *s1p; /*...*/ }; /* D2 */

```

This declares a new tag **s2** in the inner scope; the declaration **D2** then completes the declaration of the new type.

If the type is to be completed, another declaration of the tag in the same scope (but not in an enclosed block, which would declare a new type known only within that block) must define the members.

struct-or-union identifier

The declaration supersedes any prior declaration of the **struct** or **union** tag in an enclosing scope.

A **struct** or **union** specifier of the form:

struct-or-union { struct-declaration-list }

specifies a distinct **struct** or **union** type that can only be referred to by the declaration of which it is a part (it is anonymous).

A **struct** or **union** specifier of the form:

struct-or-union identifier { struct-declaration-list }

specifies a **struct** or **union** type that can be referred to by the declared tag, and which is a complete type.

```
struct tree_node {
    int count;
    struct tree_node *left, *right;
};
```

declares a **struct** that contains an integer and two pointers to objects of the same type. Once this declaration has been given, the declaration:

```
struct tree_node s, *sp;
```

defines **s** to be an object of the given type and **sp** to be a pointer to an object of the given type. With these declarations, the expression **sp->left** refers to the left **struct tree_node** pointer of the object to which **sp** points; the expression **s.right->count** refers to the **count** member of the right **struct tree_node** pointed to from **s**.

The following alternative formulation uses the **typedef** mechanism:

```
typedef struct tree_node TNODE;
struct tree_node {
    int count;
    TNODE *left, *right;
};
TNODE s, *sp;
```

6.4 Enumeration specifiers

An **enum** specifier is a method of associating a named list of identifiers with an object or type.

In the default case the **enum** constants are assigned values from zero in increments of 1. It is possible to interrupt this sequence by explicitly assigning an integral constant to one or more of the **enum** constants. The values of the constants need not be ordered or unique within the specifier.

The identifiers in an enumeration list are declared as constants that have type **int** and may appear wherever **int** is permitted. Thus, the identifiers of enumeration constants in the same scope must all be distinct from each other and from other identifiers declared in ordinary declarators.

The role of the identifier in the enum-specifier is analogous to that of the tag in a struct-or-union-specifier: it names a particular enumeration. An enumeration is an incomplete type until after the **}** that terminates the enumeration list.

The addressable storage unit allocated for an object that has an enumeration type is an **int**.

e.g.,

```
enum numbers {
    zero, one, two, three
};
enum roman_enum {
    I=1, V=5, X=V+V, L=V*X, C=100, D=500, M=1000
} date;
```

The values of the enumeration constants declared above are as follows: **zero** is 0, **one** is 1, etc.; **X** is 10, **L** is 50 etc.

enum-specifier:

```
enum identifieropt { enumeration-list }
enum identifier
```

enumeration-list:

```
enumeration
enumeration-list , enumeration
```

enumeration:

```
enumeration-constant
enumeration-constant = constant-expression
```

6.5 Storage-class specifiers

The storage-class specifier determines the lifetime of the declared object. At most one storage-class specifier may be given in the declaration specifiers in a declaration.

auto The object has local (within a function extent). This specifier is not allowed outside function definitions.

extern The object has static storage duration. This specifier may appear on an object in a declaration at any point in a program. See section 6.9.1 for a discussion of defining versus reference occurrences.

register This specifier follows the same rules as **auto**. Its purpose is to give a hint from the programmer to the code generator that the object being declared is frequently used and an attempt ought to be made to keep it in a register for as long as possible. Prospero C does its own register allocation.

static The object has static storage duration. When applied to a function it also means that the function name is not visible outside of the translation unit. Objects declared with the **static** storage class are defining occurrences.

typedef Not actually a storage class in the semantic sense. Used to indicate that a synonymous data type is being declared.

pascal This is a Prospero C extension that allows interfacing with Prospero Pascal. It is essentially the same as **extern** but the object it declares may be accessed from, or defined in, a program written in Pascal.

Functions in Prospero C have their parameters in the opposite order to Pascal. A function declared with the **pascal** specifier is called according to the Pascal conventions rather than those of 'C'.

The /S (strict) option disables this keyword.

fortran This is a Prospero C extension, similar to **pascal** above, which allows interfacing with Prospero Fortran.

The /S (strict) option disables this keyword.

The following rules apply to a declaration without a storage-class specifier:

For a function, the meaning is the same as if the storage-class specifier were **extern**.

For an object declared inside a function or among its parameters, the declaration specifies automatic storage duration.

For an object declared outside a function, the declaration is an external object definition or tentative definition.

storage-class-specifier:

```
typedef  
extern  
static  
auto  
register  
pascal  
fortran
```


6.6 Declarators

A declarator is an identifier used to declare an array, pointer or function returning type. The identifier is augmented by asterisks (*), brackets ([]) and parentheses. A declarator is not a complete declaration. A type specifier is also required to provide the base type.

Each declarator declares one identifier. When a construction of the same form as the declarator appears in an expression, it constitutes an operand of the scope, storage duration, and type indicated by the declaration specifiers.

declarator:

pointer_{opt} direct-declarator

direct-declarator:

identifier

(declarator)

direct-declarator [constant-expression_{opt}]

direct-declarator (parameter-type-list)

direct-declarator (identifier-list_{opt})

pointer:

** type-specifier-list_{opt}*

** type-specifier-list_{opt} pointer*

parameter-type-list:

parameter-list

parameter-list ,

parameter-list:

parameter-declaration

parameter-list , parameter-declaration

parameter-declaration:

declaration-specifiers declarator

declaration-specifiers abstract-declarator_{opt}

identifier-list:

identifier

identifier-list , identifier

6.6.1 Pointer declarators

In C, pointers into storage tend to be used in preference to array indexing. A pointer is declared to point to a declared type.

If **P** has the form:

** type-specifier-list_{opt} P*

the type of the contained identifier is type-specifier *pointer* to **T**. If the type specifier list includes **const**, the identifier is a constant pointer. If the type specifier list includes **volatile**, the identifier is a volatile pointer.

The following pair of declarations demonstrates the difference between a variable pointer to a constant value and a constant pointer to a variable value.

e.g.,

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of the **const int** pointed to by **ptr_to_constant** may not be modified, but **ptr_to_constant** itself may be changed to point to another **const int**. Similarly, the contents of the **int** pointed to by **constant_ptr** may be modified, but **constant_ptr** itself must always point to the same location.

The declaration of the constant pointer **constant_ptr** may be clarified by including a definition for the type pointer to **int**.

e.g.,

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

declares **constant_ptr** as a **const** object that has type *pointer to int*.

No type specifier other than **const** and **volatile** (or both) may appear in the list of type specifiers.

6.6.2 Array declarators

An array is a method of grouping together one or more objects of the same type. In C, arrays tend to be rarely used in expressions, they have been supplanted by pointers. However, arrays occur in declarations as a method of obtaining a defined amount of storage.

The constant expression specifying the size of an array must have integral type and be a constant value greater than zero.

In the following contexts an array bound may be omitted:

In a multi-dimensional array declaration the first size may be omitted.

An array is being declared as a parameter of a function.

When the array declaration has storage-class specifier **extern** and the definition that actually allocates storage is given elsewhere.

When the declarator is followed by initialization; in this case, the size is determined by the number of initializers supplied.

If the size is not present, the array type is an incomplete type.

If **A** has the form:

$$A[\textit{constant-expression}_{opt}]$$

the contained identifier has type type-specifier *array of T*. When several *array of* specifications are adjacent, a multi-dimensional array is declared.

e.g.,

```
float fa[11], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers.

Note the distinction between the declarations:

```
extern int *x;  
extern int y[];
```

The first declares **x** to be a pointer to **int**; the second declares **y** to be an array of **int** of unspecified size (an incomplete type), the storage for which is defined elsewhere.

6.6.3 Function declarators

A function declaration is a method of specifying an identifier to be a function (or pointer to function, or array of functions), also giving the type of its parameters and its return type.

As well as being able to call a function there are a few operations that may be performed on function expressions (assignment and comparison).

If **F** has the form:

F(*parameter-type-list*)

or

F(*identifier-list*_{opt})

the contained identifier has the type *type-specifier*function returning **T**.

A parameter type list declares the types of, and may declare identifiers for, the parameters of the function. Terminating the list with an ellipsis (*, . . .*), implies that zero or more arguments of unknown type may follow.

To specify a function having no parameters the specifier **void** is given, i.e., **int f(void)**.

Note: The declaration **int f1()**; does not declare a function having no parameters. For compatibility with pre-standard C usage this specifies a function returning **int** with unknown parameters.

In a parameter declaration that is not part of a function definition, i.e., in a function prototype declarator, the storage-class specifier **register** is ignored.

The declaration:

```
double frexp(double value, int *exp);
```

declares a function **frexp** returning a **double** that takes two arguments, a **double** and a *pointer to int*. The parameter names serve no purpose here other than documentation; they are required in a function *definition*.

Here are two more intricate examples:

```
FILE *freopen(const char *, const char *, FILE *);
```

declares **freopen** as a function returning a *pointer to FILE*, that takes three arguments: two constant strings (**const char ***) and a *pointer to FILE*. (**FILE** is a typedef-name used in the standard input/output library, see <stdio.h>)

The declaration:

```
void (*signal(int, void (*)(int)))(int);
```

declares **signal** as a function that returns a *pointer to function with one int parameter returning void*. **signal** itself has two parameters, an **int** and a *pointer to function with one int parameter returning void*. This can be seen more clearly from the following:

```
typedef void (*ptr_to_fn_void)(int);  
ptr_to_fn_void signal(int, ptr_to_fn_void);
```

which is an equivalent declaration.

All declarators, in the same scope, of a particular function must declare the same return type. Each parameter type list, if present, must agree in the number of parameters and in the use of the ellipsis terminator; corresponding parameters must have the same types. Also if the declarator in a function definition contains an identifier list, the type of a parameter identifier shall also be deemed to agree with its corresponding prototype parameter if, after applying the default argument promotion to the identifier's type, that type is the same as the corresponding parameter type. (For each parameter declared with function or array type, its type for these comparisons is the one resulting from conversion to pointer type).

Although the semantic constraints on parameter matching between declarator and definition allow differences (up to promotion) in type, there is no guarantee that the generated code will be correct.

The following will work in Prospero C because **int** and **char** parameters occupy the same amount of stack space:

```
void absf1(int);  
  
void absf1(a)  
char a;  
{  
}
```

Mixing **float** and **double** declarations will cause undefined results since the objects use different amounts of space on the stack when passed as parameters.

The only storage-class specifier allowed in a parameter declaration is **register**.

An identifier list in a function declarator that is not part of a function definition must be empty.

A function declarator may not declare a return type of array or function.

6.6.4 Type names

In two situations in C the name of a type is required (cast expressions and when applying **sizeof** to a type).

Note: A type name is not a **typedef** name. It is basically a declaration that omits the object being declared.

As indicated by the syntax, empty parentheses in a type name are interpreted as *function with no parameter specification*, rather than redundant parentheses around the omitted identifier.

The constructions:

```
char
unsigned *
int *[2]
int (*)[6]
int *()
void (*)(int)
int (*const [])(unsigned int, ...)
```

name respectively the types:

```
char,
pointer to unsigned int,
array of two pointers to int,
pointer to an array of six ints,
function with no parameter specification returning a pointer to int,
pointer to function with int parameter and no return value,
```

array of an unspecified number of constant pointers to functions, each with one parameter that has type **unsigned int** and an unspecified number of other parameters, returning an **int**.

type-name:

type-specifier-list abstract-declarator_{opt}

abstract-declarator:

pointer

pointer_{opt} direct-abstract-declarator

direct-abstract-declarator:

(abstract-declarator)

direct-abstract-declarator_{opt} [constant-expression_{opt}]

direct-abstract-declarator_{opt} (parameter-type-list_{opt})

6.6.5 Type definitions and type equivalence

A **typedef** declaration does not introduce a new type, only a synonym for the type. A typedef name shares the same name space as other identifiers declared in ordinary declarators. An identifier declared as a typedef may be redeclared in an inner block, but the type specifiers may not be omitted in the inner declaration and may not consist only of **const** or **volatile** or both.

Two type specifier lists are the same if they contain the same set of type specifiers (including tags).

For this purpose, two **structs**, **unions**, or enumerations are considered to have different tags if either or both do not have a tag.

Two types are the same if:

They have the same ordered set of type specifier lists and abstract declarators (including the types of function parameters), either directly or via **typedefs**.

Two arrays of members that have the same type are treated as having the same type if either array declarator has no size specification;

Two functions that return the same type are treated as having the same type if one function declarator has no parameter specification and the other specifies a fixed number of parameters, none of which is affected by the default argument promotions.

e.g., after:

```
typedef int range, domain(char);
typedef struct { double re, im; } complex;
```

the constructions:

```
range step;
extern domain *set;
complex z, *zp, za[3];
```

are all valid declarations. The type of **step** is **int**, that of **set** is *pointer to function with one char parameter returning int*, and that of **z** is the specified **struct**; **zp** is a pointer to such a **struct**; **za** is an array with three elements of **struct** type. The object **step** is considered to have exactly the same type as any other **int** object.

After the declarations:

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type **t1** and the type pointed to by **tp1** are equivalent to each other and to the type **struct s1**, but different from the types **struct s2** and **t2** and the type pointed to by **tp2** and from type **int**. The C standard has also introduced the concept of type equivalence between translation units. This equivalence is based on identical storage layout. The compiler cannot, and is not required to, enforce this equivalence.

If an identifier is declared more than once in the same scope (by declarations of an object with internal or external linkage or of a function), or if the same identifier is declared with external linkage in separate translation units or in disjoint scopes within the same translation unit, all the declarations must specify the same type. Otherwise, the behavior is undefined.

typedef-name:
identifier

6.7 Function definitions

A function definition contains a block of zero or more declarations and statements in addition to the definition of arguments and return type.

The identifier declared in a function definition (which is the name of the function) must have a function type, as specified by the declarator portion of the function definition.

There are two methods of specifying the names and types of the parameters:

```
f(a,b)
int a; char b;
```

Here a list of identifiers appears in the function parameter list. The types of these parameters then follow. If the type of a parameter is not given, `int` is assumed.

```
f(int a, char b)
```

Here the names and types of the parameters is given in the parameter list.

It is illegal to mix the two styles of declaration.

The difference between these two definitions is that the second form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the first form does not.

The return type of a function must be `void` or an object type other than array.

An identifier declared as a typedef name may not be redeclared as a parameter. The declarations in the declaration list may only contain the `register` storage class specifier. Also, parameters may not be initialized.

On entry to the function the value of the argument expression is converted to the type of the parameter, as if by assignment to the parameter. Because array expressions and function designators as arguments are converted to pointers before the call, a declaration of a parameter as *array of type* will be adjusted to *pointer to type*, and a declaration of a parameter as *function* will be adjusted to *pointer to function*, as in lvalues and function designators, see section 3.4.1.

Each parameter is treated as having automatic storage duration. Its identifier is an lvalue. A parameter is in effect declared at the head of the compound statement that constitutes the function body, and therefore may not be redeclared in the function body (except in an enclosed block).

To pass one function to another, one might say:

```
int f(void);
/*...*/
g(f);
```

Note that f must be declared explicitly in the calling function, as its appearance in the expression $g(f)$ was not followed by $($. Then the definition of g might read:

```
g(int (*funcp)(void))
{
    /*...*/
    (*funcp)()
    /* or funcp() ... */
}
```

or, equivalently,

```
g(int func(void))
{
    /*...*/
    func()
    /* or (*func)() ... */
}
```

function-definition

declaration-specifiers_{opt} declarator declaration-list_{opt} compound-statement_{opt}

6.8 Initialization

Initialization on declarations provides a method of giving an initial value to an object. The object is assigned the value immediately after storage is allocated to it (at program execution time). This means that objects of **static** storage class are initialized once at program startup. Objects of non-**static** storage class are initialized every time they start a new lifetime.

Thus **static** objects are assigned their value prior to executing the first statement in **main**. The initialization of **auto** objects can be thought of as if an assignment statement was executed as the declarations were ‘flowed through’ on entering a block.

If an object that has **static** storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

The value of an **auto** object declared without initialization is undefined until explicitly assigned.

If a **goto** to a label in a compound statement containing declarations with initializers is made; the initialization will not occur.

The initializer for a scalar must be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression; the same type constraints and conversions as for simple assignment apply.

A brace-enclosed initializer for a **union** object initializes the member that appears first in the declaration list of the **union** type.

Constraints on initialization:

There must be no more initializers in an initializer list than there are objects to be initialized.

The type of the entity to be initialized must be an object type or an array of unknown size.

All the expressions in an initializer for an object that has **static** storage duration, or in an initializer list for an object that has aggregate or **union** type must be constant expressions.

The initializer for a **struct** or **union** object that has automatic storage duration must either be an initializer list as described below, or a single expression that has the same **struct** or **union** type. In the latter case, the initial value of the object is that of the expression.

e.g.,

```
static i = 0;
float f = 1.23;

void vf()
{ long l = i + 1;
  int s_f = sizeof(f);
  struct s_t vs = f_ret_s_t();
  static unsigned ls = 14;
  int i;

  for (i = 0; i < 10; i++)
  { char af = 'z';
    /* some code */
  }
  /* some code */
}
```

6.8.1 Initializing arrays, structs and unions

Entire arrays, structs and unions may be initialized. This is achieved by specifying a list of constant expressions between braces.

If there are fewer initializers in a list than there are members of an aggregate, the remainder of the aggregate will be initialized implicitly. The value used will be the same as that assigned to objects that have **static** duration, defined without any initial value.

An array of characters may be initialized by a string literal, optionally enclosed in braces. Successive characters of the string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the members of the array.

If the aggregate contains members that are aggregates or **unions**, or if the first member of a **union** is an aggregate or **union**, the rules apply recursively to the subaggregates or contained **unions**. If the initializer of a subaggregate or contained **union** begins with a left brace, the succeeding initializers initialize the members of the subaggregate or the first member of the contained **union**. Otherwise, only enough initializers from the list are taken to account for the members of the first subaggregate or the first member of the contained **union**; any remaining initializers are left to initialize the next member of the aggregate of which the current subaggregate or contained **union** is a part.

If an incomplete array is initialized, its size is determined by the number of initializers provided for its elements. At the end of its initializer list, the array is no longer an incomplete type.

The declaration:

```
int x[] = { 1, 3, 5 };
```

defines and initializes **x** as a one-dimensional array object that has three members, and

```
float y[4][3] = {
    { 1, 2, 3 },
    { 1, 4, 9 },
    { 1, 8, 81 }
};
```

is a definition with a fully bracketed initialization: the values 1, 2, and 3 initialize the first row of the array object **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise the next two lines initialize **y[1]** and **y[2]**. The initializer ends early, so **y[3]** is initialized with zeroes. Precisely the same effect could have been achieved by

```
float y[4][3] = {
    1, 2, 3, 1, 4, 8, 1, 9, 81
};
```

The initializer for **y[0]** does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for **y[1]** and **y[2]**.

The declaration:

```
float z[4][3] = {
    { 3 }, { 5 }, { 7 }, { 11 }
};
```

initializes the first column of **z** as specified and initializes the rest with zeros.

The declaration:

```
struct{float f; int i;} fi[]=
    { 1.2, 3, { 7.9 }, 0.4, 14 };
```

is inconsistently bracketed. The array **fi** has 3 elements. Field **fi[2].i** has value 0, since the initialization for this element is incomplete.

The declaration:

```
short q[4][3][2] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 }
};
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: `q[0][0][0]` is 1, `q[1][0][0]` is 2, `q[1][0][1]` is 3, and 4, 5, and 6 initialize `q[2][0][0]`, `q[2][0][1]`, and `q[2][1][0]`, respectively; all the rest are zero. The initializer for `q[0][0][0]` does not begin with a left brace, so up to six items from the current list may be used. There is only one, so the values for the remaining five members are initialized with zero. Likewise, the initializers for `q[1][0][0]` and `q[2][0][0]` do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional sub-aggregates. If there had been more than six items in any of the lists, an error message would be given. The same initialization result could have been achieved by:

```
short q[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 0, 0, 0, 0,
    4, 5, 6
};
```

or by:

```
short q[4][3][2] = {
    {
        { 1 },
    },
    {
        { 2, 3 },
    },
    {
        { 4, 5 },
        { 6 },
    }
};
```

in a fully-bracketed form.

The declaration:

```
char s[] = "abc", t[3] = "abc";
```

defines character array objects **s** and **t** whose members are initialized with string literals. This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\\0' },  
t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines a character pointer **p** that is initialized to point to a character array object whose members are initialized with a string literal (four elements including a trailing '\\0'). If an attempt is made to use **p** to modify the contents of the array, the behavior is undefined.

```
initializer  
  assignment-expression  
  { initializer-list }  
  { initializer-list , }
```

```
initializer-list  
  initializer  
  initializer-list , initializer
```

6.9 External definitions

The unit of program text after preprocessing is a translation unit. This consists of a sequence of external definitions of functions and objects (and other declarations, such as type declarations), described as *external* because they appear outside of any function.

An external definition implicitly declares its identifier to have file scope and static storage duration. If there is no storage-class specifier, the identifier has external linkage. As with other declarations, if there are no type specifiers the type is taken to be **int**.

The storage-class specifiers **auto** and **register** may not appear in an external definition.

translation-unit:
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

6.9.1 External object definitions

C allows identifiers to be declared, in a translation unit, whose storage is allocated elsewhere. The question thus arises, when looking at a declaration: is this a definition (storage is actually allocated), or a reference?

The following rules are used to decide the situation.

A declaration of an identifier of an object outside of any function that includes an initializer constitutes the definition of the object.

A declaration of an identifier of an object outside of any function without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. A definition (tentative or otherwise) for the same identifier with the same linkage may be encountered elsewhere in the translation unit. All such tentative definitions are taken to be declarations of the same object (subject to the linkage rules in section 4.2). The first tentative definition is taken to be a definition with initializer equal to 0.

E.g.,

```
int i1 = 1;          /* definition, external linkage */
static int i2 = 2;  /* definition, internal linkage */
extern int i3 = 3;  /* definition, external linkage */
int i4;             /* tentative definition, external linkage */
static int i5;     /* tentative definition, internal linkage */

int i1;            /* valid tentative definition, refers to previous */
int i2;            /* invalid, linkage disagreement */
int i3;            /* valid tentative definition, refers to previous */
int i4;            /* valid tentative definition, refers to previous */
int i5;            /* invalid, linkage disagreement */

extern int i1;     /* refers to previous, whose linkage is external */
extern int i2;     /* refers to previous, whose linkage is internal */
extern int i3;     /* refers to previous, whose linkage is external */
extern int i4;     /* refers to previous, whose linkage is external */
extern int i5;     /* refers to previous, whose linkage is internal */
```


7 EXPRESSIONS

An expression is a sequence of operators and operands that specifies how to compute a value, or how to generate side effects, or both.

The order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

The precedence of operators within an expression is specified by the syntax. Parentheses may be used to regroup subexpressions containing operators of different precedence.

There are some operators that have operands that are of integral type. Their return values depend on the internal representations of integers, which is two's complement in Prospero C.

All lvalues designating an object (whether or not the object is a member of an aggregate or union) must have one of the following types:

- the declared type of the object;

- a type that differs from the declared type of the object only in the presence or absence of the **unsigned** type specifier;

- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a sub-aggregate or contained union);

- a character type.

A *full expression* is an expression that is not part of another expression. Each of the following is a full expression:

- an initializer;

- the expression in an expression statement;

- the controlling expression of a selection statement (**if** or **switch**);

- the controlling expression of an iteration statement (**while**, **do**, or **for**);

- the expression in a **return** statement.

The end of a full expression is a sequence point.

7.1 Precedence of operators

primary expressions

16	literals names	simple tokens
16	a[i]	subscripting
16	f()	function call
16	.	direct selection
16	->	indirect selection

unary operators

15	++ --	postfix increment/decrement
14	++ --	prefix increment/decrement
14	sizeof	size
14	(type-name)	cast
14	~	bitwise not
14	!	logical not
14	-	arithmetic negation
14	&	address of
14	*	contents of

binary operators

13L	* / %	multiplicative
12L	+ -	additive
11L	<< >>	shift
10L	< > <= >=	relational
9L	== !=	equality
8L	&	bitwise and
7L	^	bitwise xor
6L		bitwise or
5L	&&	logical and
4L		logical or
3R	? :	conditional
2R	= += -= *= /= %=	assignment
2R	<<= >>= &= ^= =	assignment
1L	,	comma

L, indicates left associative operators; R, right associative operators.

7.2 Primary expressions

An identifier naming an object or function is a primary expression. The value of an identifier depends on the type given when it was declared.

In some circumstances identifiers of particular types have implicit conversions performed on them, see section 3.4.1 for details.

A constant is a primary expression whose type depends on its form, as described in section 5.3

A string literal is a primary expression. It is an lvalue of type *array of char*.

A parenthesized expression is a primary expression whose type and value are identical to those of the unparenthesized expression. It is an lvalue, a function designator, or a **void** expression if the unparenthesized expression is, respectively, an lvalue, a function designator, or a **void** expression.

e.g.,

```
Count          2.6          301u
"fred"         (a+b* c) (0)
```

primary-ex:
identifier
constant
string-literal
(expression)

7.3 Postfix operators

postfix-ex:

primary-ex

postfix-ex [*expression*]

postfix-ex (*argument-expression-list*_{opt})

postfix-ex . *identifier*

postfix-ex -> *identifier*

postfix-ex ++

postfix-ex --

argument-expression-list:

assignment-ex

argument-expression-list , *assignment-ex*

7.3.1 Array subscripting

$\mathbf{a}[\mathbf{i}]$ and $\mathbf{i}[\mathbf{a}]$ are defined to be identical to $(*(\mathbf{a})+(\mathbf{i}))$.

Successive subscript operators designate a member of a multi-dimensional array object. If \mathbf{a} is an n -dimensional array with dimensions $i * j * \dots * k$, then \mathbf{a} (used as other than an lvalue) is converted to a pointer to an $(n-1)$ -dimensional array with dimensions $j * \dots * k$. If the unary $*$ operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the pointed-to $(n-1)$ -dimensional array, which is itself converted into a pointer if used as other than an lvalue. Thus arrays are stored in row-major order (last subscript varies fastest).

One of the expressions must have type pointer to T , the other expression must have integral type, and the result has type T .

Consider the array object defined by the declaration:

```
int x[3][5];
```

Here \mathbf{x} is a $3 * 5$ array of `ints`; more precisely, \mathbf{x} is an array of three member objects, each of which is an array of five `ints`. In the expression $\mathbf{x}[\mathbf{i}]$, \mathbf{x} is first converted to a pointer to the initial array of five `ints`. Then \mathbf{i} is adjusted according to the type of \mathbf{x} , which conceptually entails multiplying \mathbf{i} by the size of the object to which the pointer points, namely an array of five `int` objects. The results are added and indirection is applied to yield an array of five `ints`. When used in the expression $\mathbf{x}[\mathbf{i}][\mathbf{j}]$, that in turn is converted to a pointer to the first of the `ints`, so $\mathbf{x}[\mathbf{i}][\mathbf{j}]$ yields an `int`.

Note: $\mathbf{x}[\mathbf{i}, \mathbf{j}]$ is not a two dimensional reference. It is equivalent to $(*(\mathbf{x})+(\mathbf{i}, \mathbf{j}))$ i.e., the subscript is a comma expression.

7.3.2 Function calls

A postfix expression followed by parentheses () containing a, possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the function called. The list of expressions specifies the arguments to the function.

e.g.,

```
i = getchar();
putchar('\n');
printf("%d items\n", Count);
```

If the expression that precedes the parenthesized argument list in a function call is solely an identifier, and if no declaration is in scope for this identifier, the identifier is implicitly declared. This declaration occurs as if, in the innermost block containing the function call, the following declaration appeared:

e.g.,

```
extern int identifier();
```

An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument. Parameters are passed by value, except arrays whose address is implicitly taken and passed. Passing the address of an object can be achieved by using the address-of operator &. In the case of complete arrays being passed, an implicit address-of occurs. Note: when handling complex function declarations, it is useful to remember that a function is called in the same form as it was declared:

```
long (* fp)(char, int);    /* declaration */
(* fp)('a', 73);          /* call */
```

If no function prototype declarator is in scope at the function call, the default argument promotions are performed on each parameter. Traditionally C programmers have made use of knowledge regarding parameter layout. If the number of arguments does not agree with the number of parameters, the behavior is undefined.

The order of evaluation in Prospero C is such that arguments are evaluated from right to left, followed by the function designator. There is a sequence point just before the actual call.

If a function prototype declarator is in scope at the function call, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters. The ellipsis notation (`, . . .`) in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments. If a parameter is declared with a type that is affected by the default argument promotions, and no semantically equivalent function prototype is in scope where the function is defined, and a call is executed, the behavior is undefined.

It is an error for the arguments in the call to differ in number or type from the prototype.

Note: The arguments in a function call are only compared against the parameters of a function prototype declarator, if one exists. No comparison is made against a function definition.

Recursive function calls are permitted. The maximum depth of recursion is only limited by the memory available.

The expression that denotes the function called must have type pointer to function returning `void` or returning an object type other than array.

e.g.,

```
void max_and_min(int [], int, int *, int *);
char contains(int [], int, int);
short process(int [], int, short (int));
short print_item(int);

int list[LIST_SIZE],
    max_val,
    min_val,
    search_val;
short err;

if (contains(list, LIST_SIZE, search_val))
    { /* some processing */
    }

max_and_min(list, LIST_SIZE, &max_val, &min_val);
err = process(list, LIST_SIZE, print_item);
```

7.3.2.1 Default argument promotions

The default argument promotions consist of the *integer promotions* (see section 3.2.1) and also converting `float` to `double`. These conversions are performed on expressions passed as arguments to functions where

No prototype declarator is in scope at the point of call.

A prototype declarator with the `(, ...)` notation is in scope and the argument corresponds to one of the unspecified parameters.

7.3.3 Structure and union members

A postfix expression followed by a dot `.` and an identifier selects a member of the structure or union object preceding the dot. The value is that of the named member, and is an lvalue if the first expression is an lvalue.

A postfix expression followed by an arrow `->` and an identifier selects a member of a structure or union object pointed to by the expression to the left of the arrow. The value is that of the named member of the object to which the first expression points, and is an lvalue.

If `&S` is a valid pointer expression (where `&` is the *address-of* operator, which generates a pointer to its operand) the expression `(&S)->field` is equivalent to `S.field`.

If a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is not defined. In one special case the results are defined. If a union contains several structures that share a common initial sequence, and if the union object currently contains one of these structures, it is possible to inspect the common initial part of any of them to obtain the expected results.

e.g.,

```
enum lisp_type { LIST, INTEGER, STRING };
union lisp {
    struct {
        enum lisp_type cell_is;
        union lisp *head, *tail;
    } list;
    struct {
        enum lisp_type cell_is;
        int value;
    } integer;
    struct {
        enum lisp_type cell_is;
        char *string;
    } string;
};
void print_cell(union lisp *x)
{
    switch (x->list.cell_type) {
        case LIST : {
            do {
                print_cell(x->list.head);
                x = x->list.tail;
            } while (x); /*x != NULL */
            break;
        }
        case INTEGER : {
            printf("%d", x->integer.value);
            break;
        }
        case STRING : {
            printf("%s", x->string.string);
        }
    }
}
```

If **f** is a function returning a **struct** or **union**, and **x** is a member of that **struct** or **union**, **f().x** is a valid postfix expression but is not an lvalue.

The following is a valid fragment:

```
struct cell {
    int value;
    struct cell *next;
};
struct cell find(struct cell *, int),
    *new(void),
    *head,
    *tail,
    temp;

(head = new())->next = new();
head->value = 3;
head->next->value = 5;

temp = find(head, 3);
tail = find(head, 6).next;
```

7.3.4 Postfix increment and decrement operators

The result of the postfix `++` operator is the value of the operand. After the result is noted, the value of the operand is incremented (that is, the value 1 of the appropriate type is added to it). The side effect of updating the stored value of the operand occurs between the previous and the next sequence point.

The postfix `--` operator is analogous to the postfix `++` operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

The operand of the postfix increment or decrement operator must have scalar type and must be a modifiable lvalue.

e.g.,

```
i = 0;
while (*s) /* find end of string and length */
    i++, s++;
s--;
while (i) {
    i--; /* reverse string s into string p */
    *p++ = *s--;
}
*p = 0;
```

7.4 Unary operators

unary-ex:

postfix-ex

++ *unary-ex*

-- *unary-ex*

unary-operator cast-ex

sizeof *unary-ex*

sizeof (*type-name*)

unary-operator: one of

& * + - ~ !

7.4.1 Prefix increment and decrement operators

The value of the operand of the prefix **++** operator is incremented. The result is the new value of the operand after being incremented. The expression **++E** is equivalent to **(E+=1)**.

The prefix **--** operator is analogous to the prefix **++** operator, except that the value of the operand is decremented. The expression **--E** is equivalent to **(E-=1)**.

The operand of the prefix increment or decrement operator must have scalar type and must be a modifiable lvalue.

e.g.,

```
i = 0;
while (*s) /* find end of string and length */
    ++i, ++s;
while (i) {
    --i; /* reverse string s into string p */
    *p++ = *--s;
    /* post-increment p! */
}
*p = 0;
```

7.4.2 Address and indirection operators

The result of the unary `&` (address-of) operator is a pointer to the object or function designated by its operand. If the operand has type *T*, the result has type pointer to *T*.

The unary `*` operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type pointer to *T*, the result has type *T*. If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined.

If `*P` is an lvalue and `T` is the name of an object of pointer type, the cast expression `*(T)P` is an lvalue that has the same type as that to which `T` points. Among the invalid values for dereferencing a pointer by the unary `*` operator are:

- A null pointer constant;

- an address inappropriately aligned for the type of object pointed to;

- the address of an object that has automatic storage duration when execution of the block in which the object is declared has terminated.

The operand of the unary `&` operator must be either:

- A function designator, or

- an lvalue that designates an object that is not a bit-field and is not declared with the `register` storage-class specifier.

The operand of the unary `*` operator must have pointer type, other than pointer to `void`.

e.g.,

```
void check_bounds(int *val_addr, int lo, int hi)
/* force object into given range lo..hi */
{ if (*val_addr < lo)
    *val_addr = lo;
  else if (*val_addr > hi)
    *val_addr = hi;
}
```

```
main()
{ int value = getvalue();
  check_bounds(&value, 0, 100);
}
```

7.4.3 Unary arithmetic operators

The result of the unary `+` operator is the value of its operand. The integral promotions are performed on the operand, and the result has the promoted type. The expression `+E` is equivalent to `(0+E)`.

The result of the unary `-` operator is the negation of its operand. The integral promotions are performed on the operand, and the result has the promoted type. The expression `-E` is equivalent to `(0-E)`.

The result of the `~` operator is the bitwise complement of its operand. The integral promotions are performed on the operand, and the result has the promoted type. The expression `~E` is equivalent to `(ULONG_MAX-E)` if `E` has type `unsigned long`, and `(UINT_MAX-E)` if `E` has any other unsigned type.

The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type `int`. The expression `!E` is equivalent to `(0==E)`.

The operand of the unary `+` operator must have scalar type; of the unary `-` operator, arithmetic type; of the `~` operator, integral type; of the `!` operator, scalar type.

e.g.,

```
#define ERR_FLAG    0x0400
short FLAGS;
/* ... */

FLAGS &= ~ERR_FLAG;    /* clear error flag */

if (!FLAGS)            /* FLAGS is zero    */
    FLAGS = -1;
/* the expression -(1), not the constant (-1) */
```

7.4.4 The sizeof operator

The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type.

When applied to an operand that has type **char**, **unsigned char** or **signed char**, the result is 1.

When applied to an operand that has array type, the result is the total number of bytes in the array.

When applied to a parameter declared to have array or function type, the **sizeof** operator yields the size of the pointer obtained by conversion.

When applied to a **short int** the result is 2.

When applied to an **int** the result is 2.

When applied to a **long int** the result is 4.

When applied to a **float** the result is 4.

When applied to a **double**, or **long double** the result is 8.

When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. Trailing padding occurs if the last field of a **struct**, or **union** only uses 1 byte.

The size is determined from the type of the operand, which is not itself evaluated. The result is an integer constant. The result has type **size_t** defined in the `<stddef.h>` header.

Prosero C will issue a warning if the operand has side effects.

The **sizeof** operator may not be applied to:

- An expression that has function type;
- an incomplete type;
- to the parenthesized name of the above two;
- to a bit-field object.

e.g.,

```
int table[TABSIZE],
    table_length,
    no_of_elements;

table_length = sizeof table;
                /* == TABSIZE*sizeof(int) */
no_of_elements = table_length / sizeof table[0];
```

7.5 Cast operators

Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a *cast*.

A cast changes the type of its operand. This conversion may be quiet in the sense that no change of representation is involved (but the new type may cause the code generator to act differently), or a change of representation may occur.

If the type name specifies the `void` type, the operand must not have an incomplete type. Otherwise, the type name must specify scalar type and the operand must have scalar type.

A cast does not yield an lvalue.

Conversions that involve pointers (other than a *pointer to void* converted to or from a pointer to an object type or an incomplete type) must be specified by means of an explicit cast. A pointer may be converted to an integral type. In Prospero C, `long` is the only integral type guaranteed to hold a complete pointer. Zero (`0`) converts to the null pointer constant.

A pointer to an object or incomplete type may be converted to a pointer to a different object type or a different incomplete type. The resulting pointer might not be valid if it is improperly aligned for the type pointed to. It is guaranteed, however, that a pointer to an object of a given alignment may be converted to a pointer to an object of a less strict alignment and back again; the result will compare equal to the original pointer. (An object that has type `char` has the least strict alignment.) For example, given `int *p`; the expression `(int *) (char *) p` will always compare equal to `p`.

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result will compare equal to the original pointer. If a converted pointer is used to call a function of other than the original type, the behavior is undefined.

e.g.,

```
void *malloc();
struct pair { int value;
              char name[10]; } *ptr;

ptr = (struct pair *) malloc(sizeof *ptr);
*(int *) ptr = 0;
strncpy((char *) (1 + (int *) ptr), "ZERO", 10);
```

cast-ex:

```
unary-ex
( type-name ) cast-ex
```

7.6 Multiplicative operators

The result of the binary `*` operator is the product of the operands. The binary `*` operator is commutative and associative.

The result of the `/` operator is the quotient from the division of the first operand by the second; the result of the `%` operator is the remainder.

When integers are divided and the division is inexact, if both operands are positive the result of the `/` operator is the largest integer less than the algebraic quotient and the result of the `%` operator is positive. If either operand is negative, the result of the `/` (divide) operator is the smallest integer greater than the algebraic quotient. The magnitude of `a/b` is the integer given by `abs(a) / abs(b)`. If the two operands have the same sign then the result is positive, otherwise it is negative. If one operand is negative the sign of the result of the `%` operator is negative. If the quotient `a/b` is representable, the following expression is true:

$$(a/b) * b + a \% b == a$$

Each of the operands must have arithmetic type. The operands of the `%` operator must have integral type.

The usual arithmetic conversions are performed on the operands.

multiplicative-ex :

cast-ex

*multiplicative-ex * cast-ex*

multiplicative-ex / cast-ex

multiplicative-ex % cast-ex

7.7 Additive operators

The result of the binary `+` operator is the sum of the operands. The binary `+` operator is commutative and associative.

The result of the binary `-` operator is the difference resulting from the subtraction of the second operand from the first.

When an expression that has integral type is added to or subtracted from a pointer, the integral value is first multiplied by the size of the object pointed to. The result is a pointer of the same type as the original pointer. If the original pointer points to a member of an array object, and the array is large enough, the result points to another member of the same array object, appropriately offset from the original member. Thus if `P` points to a member of an array object, the expression `P+1` points to the next member of the array object. Unless both the pointer operand and the result point to a member of the same array object, the behavior if the result is used as the operand of a unary `*` operator is undefined.

When two pointers to members of the same array object are subtracted, the difference is divided by the size of a member. The result represents the difference of the subscripts of the two array members. The result has type `ptrdiff_t` defined in the `<stddef.h>` header. If two pointers that do not point to members of the same array object are subtracted, the behavior is undefined. However, if `P` points to the last member of an array object, the expression `(P+1) - P` has the value 1, even though `P+1` does not point to a member of the array object.

For addition, either both operands must have arithmetic type, or one operand must be a pointer to an object and the other must have integral type (incrementing is equivalent to adding 1).

For subtraction, one of the following must hold:

- Both operands have arithmetic type.

- Both operands are pointers to objects that have the same type.

- The left operand is a pointer to an object and the right operand has integral type (decrementing is equivalent to subtracting 1).

If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

e.g.,

```

int table[TABSIZE],
    pos, ix;
struct { int *start, end;
        } index[INDEXSIZE];

#define NEXT      0
#define LENGTH    1
#define DATA     2
/*
table represents a linked list of variable length
data blocks held as an array of pseudo structures:
    struct {
        int offset;      of next entry
        int length;     of this data block
        int data[];     incomplete array!!
    }
    BASELOC             is first entry to be indexed
    table[NEXT]         retrieves the offset field
    table[LENGTH]       retrieves the length field
    table[DATA]         is 0th element of data field
*/
pos = BASELOC;
for (ix = 0; pos > 0; ix++) {
    index[ix].start = &table[pos + DATA];
    /* zeroth data element */

    index[ix].end = index[ix] + table[pos + LENGTH];
    /* all data before that address */
    pos = table[pos + NEXT];
    /* advance to next logical block */
}

```

additive-ex:

multiplicative-ex

additive-ex + multiplicative-ex

additive-ex - multiplicative-ex

7.8 Bitwise shift operators

The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has an unsigned type, the value of the result is `E1` multiplied by (2 raised to the power `E2`), which is then reduced modulo `ULONG_MAX+1` if `E1` has type `unsigned long`, and `UINT_MAX+1` otherwise. (The constants `ULONG_MAX` and `UINT_MAX` are defined in the header `<limits.h>` – see Appendix G.)

The result of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a non-negative value, the value of the result is the integral part of the quotient of `E1` divided by (2 raised to the power `E2`). If `E1` has a signed type and a negative value, the vacated bits fill with the same bit value as the original top most bit.

The integral promotions are performed on each of the operands. Then the right operand is converted to `int`; the type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behavior is undefined.

Each of the operands must have integral type.

e.g.,

```
unsigned long date_time;
/* year (base 1980), month, day, hours, mins, secs:
   YYYYYYMMMMDDDDDDhhhhhhmmmmmmsssss */

date_time =
    (hours << 12) | (minutes << 6) | seconds |
    ((unsigned long)
     ((year-1980 << 9) | (month << 5) | day) << 17 );
```

shift-ex:

additive-ex

shift-ex << additive-ex

shift-ex >> additive-ex

7.9 Relational operators

When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If the objects pointed to are members of the same aggregate object:

Pointers to structure members declared later compare higher than pointers to members declared earlier in the structure.

Pointers to array elements with larger subscript values compare higher than pointers to elements of the same array with lower subscript values.

All pointers to members of the same **union** object compare equal.

If the objects pointed to are not members of the same aggregate or **union** object, the result is undefined, with the following exception. If **P** points to the last member of an array object, the pointer expression **P+1** compares higher than **P**, even though **P+1** does not point to a member of the array object.

Each of the operators **<** (less than), **>** (greater than), **<=** (less than or equal to), and **>=** (greater than or equal to) yield 1 if the specified relation is true and 0 if it is false. The result has type **int**.

Either both operands must have arithmetic type, or both must be pointers to objects that have the same type.

If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

Note: Relational comparisons with the null pointer constant are illegal. In order to provide compatibility with existing code, Prospero C will only treat this case as a warning if the **S** option is off.

relational-ex:

shift-ex

relational-ex < shift-ex

relational-ex > shift-ex

relational-ex <= shift-ex

relational-ex >= shift-ex

7.10 Equality operators

The `==` (equal to) and the `!=` (not equal to) operators are analogous to the relational operators except for their lower precedence.

If two pointers to objects or pointers to incomplete types compare equal, they point to the same object. If two pointers to functions compare equal, they point to the same function. If one of the operands is a pointer to an object (or pointer to an incomplete type) and the other has type pointer to `void`, the pointer to an object (or pointer to an incomplete type) is converted to type *pointer to void*.

One of the following must hold:

Both operands have arithmetic type.

Both operands are pointers to the same type.

One operand is a pointer to an object and the other is a pointer to `void`.

One operand is a pointer and the other is a null pointer constant.

e.g.,

```
void *malloc(),
    *mem_base;
struct list *head; /* fields value, next */

mem_base = malloc(sizeof *head);
head = (struct list *) mem_base;

/* ... */

while (head != mem_base &&
       head != 0 && head->value != search_val)
    head = head->next;
```

equality-ex:

relational-ex

equality-ex == relational-ex

equality-ex != relational-ex

7.11 Bitwise AND operator

The result of the binary $\&$ operator is the bitwise *AND* of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands are set). The binary $\&$ operator is commutative and associative.

Each of the operands must have integral type.

The usual arithmetic conversions are performed on the operands.

AND-ex:
equality-ex
AND-ex $\&$ *equality-ex*

7.12 Bitwise exclusive OR operator

The result of the \wedge operator is the bitwise exclusive *OR* of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set). The \wedge operator is commutative and associative.

Each of the operands must have integral type.

The usual arithmetic conversions are performed on the operands.

exclusive-OR-ex:
AND-ex
exclusive-OR-ex \wedge *AND-ex*

7.13 Bitwise inclusive OR operator

The result of the \mid operator is the bitwise inclusive *OR* of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set). The \mid operator is commutative and associative, and an expression involving several \mid operations at the same level may be regrouped.

Each of the operands must have integral type.

The usual arithmetic conversions are performed on the operands.

inclusive-OR-ex:
exclusive-OR-ex
inclusive-OR-ex \mid *exclusive-OR-ex*

7.14 Logical AND operator

The `&&` operator yields 1 if both of its operands compare unequal to 0, otherwise it yields 0. The result has type `int`. Unlike the bitwise binary `&` operator, the `&&` operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares equal to 0, the second operand is not evaluated. Each of the operands must have scalar type.

e.g.,

```
i = (m == n) && (o == p);
```

is equivalent to:

```
if (m == n)
    if (o == p)
        i = 1;
    else
        i = 0;
else i = 0;
```

logical-AND-ex:

inclusive-OR-ex

logical-AND-ex && inclusive-OR-ex

7.15 Logical OR operator

The `||` operator will yield 1 if either of its operands compare unequal to 0, otherwise it yields 0. The result has type `int`. Unlike the bitwise `|` operator, the `||` operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares unequal to 0, the second operand is not evaluated. Each operand must have scalar type.

e.g.,

```
i = (m == n) || (o == p);
```

is equivalent to:

```
if (m == n)
    i = 1;
else
    if (o == p)
        i = 1;
    else
        i = 0;
```

logical-OR-ex:

logical-AND-ex

logical-OR-ex || logical-AND-ex

7.16 Conditional operator

The condition operator causes either its second or third operand to be evaluated, depending on the value of its first operand. The first operand is evaluated; there is a sequence point after its evaluation. If its value compares unequal to 0, the second operand is evaluated and its value is the result; otherwise the third operand is evaluated and its value is the result.

A conditional expression does not yield an lvalue.

If both the second and third operands have arithmetic type, the usual arithmetic conversions are performed to bring them to the same type and the result has that type. If both the operands have structure, union, or pointer type, the result has that type. If both the operands are **void** expressions, the result is a **void** expression. If one of the operands is a pointer to an object or incomplete type and the other is a *pointer to void*, the pointer to an object or incomplete type is converted to type pointer to **void**, and the result has that type. If one operand is a pointer and the other operand is a null pointer constant, the result has the type of the pointer.

The first operand must have scalar type.

One of the following must hold for the second and third operands:

Both operands have arithmetic type.

Both operands are pointers to the same type.

Both operands have the same structure or union type.

Both operands are void expressions.

One operand is a pointer to an object or incomplete type, and the other is a pointer to **void**.

One operand is a pointer and the other is a null pointer constant.

e.g.,

```
max = a > b ? a : b;  
select = (y ? s1 : s2).f;  
ptr = count > 0 ? src_ptr : NULL;
```

conditional-ex:

logical-OR-ex

logical-OR-ex ? ex : conditional-ex

7.17 Assignment operators

An assignment operator stores a value in the object designated by the left operand. An assignment expression has the type of the left operand and the value of the left operand after the assignment. The side effect of updating the stored value of the left operand must occur between the previous and the next sequence point.

In an assignment statement, Prospero C evaluates the left operand first, followed by the right operand. In an assignment expression the right operand is evaluated first then the left operand.

The left operand must be a modifiable lvalue.

The result is not an lvalue.

assignment-ex:

conditional-ex

unary-ex assignment-operator assignment-ex

assignment-operator: one of

`= * = / = % = + = - = << = >> = & = ^ = | =`

7.17.1 Simple assignment

In the simple assignment with `=`, the value of the right operand is converted to the type of the left operand and replaces the value stored in the object designated by the left operand.

If an object is assigned to another object that overlaps in storage with any part of the object whose value is being assigned, the behavior is undefined.

One of the following must hold:

Both operands have arithmetic type.

Both operands have the same structure or union type.

Both operands are pointers to the same type.

One operand is a pointer to an object or incomplete type and the other is a pointer to `void`.

The left operand is a pointer and the right is a null pointer constant.

Both operands are pointers to types that differ only in the presence or absence in the right operand of the type specifiers `const` or `volatile` or both.

The rules for **const** objects and **volatile** objects must be obeyed when the object is referred to by means of the pointer to **const** or **volatile** (or both).

e.g.,

```
i = 0;
a[i] = ++i - 1;          /* a[0] = 0, i = 1 */
i = (a[i] = ++i) - 1;   /* a[2] = 2, i = 1 */
a[i] = (a[i] = ++i) - 1; /* a[1] = 1, a[2] = 2,
                        i = 2 */
```

In the program fragment:

```
int i;
char c;
/*...*/
/*...*/ ((c = i) == -1) /*...*/
```

the **int** value **i** may be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison.

7.17.2 Compound assignment

A compound assignment of the form **E1 op= E2** differs from the simple assignment expression **E1 = E1 op +(E2)** only in that the lvalue **E1** is evaluated only once. For the operators **+=** and **-=** only, either the left operand must be a pointer to an object type and the right must have integral type, or the operands must have arithmetic type.

For the other operators, each operand must have arithmetic type consistent with those allowed by the corresponding binary operator.

Note: The unary increment operator (**++**) and decrement (**--**) operators are also assignment operators, i.e., **--i** is equivalent to **(i-=1)**.

e.g.,

```
int *p;
long count;
short FLAG;

while (*p == ENTRY)
    p += ENTRY_LEN;
count /= 4;
FLAG |= err_bits;
```

7.18 Comma operator

The left operand of a comma operator is evaluated as a **void** expression; there is a sequence point after its evaluation. Then the right operand is evaluated; the result has its type and value.

A comma operator does not yield an lvalue.

As indicated by the syntax, in contexts where a comma is a punctuator (in lists of arguments to functions and lists of initializers) the comma operator as described here may appear only in parentheses. In the function call:

```
f(a, (t=3, t+2), c)
```

the function has three arguments, the second of which has the value 5.

e.g.,

```
val = (index += offset,  
       table[index] + table[index+1]);
```

expression:

*assignment-ex
expression , assignment-ex*

7.19 Constant expressions

A constant expression is any integral expression that can be evaluated to a single value in the translation environment.

A constant expression may not cause side-effects.

Note: Because the operands to **sizeof** are not evaluated, it is possible to use non-constant expressions in this context. These include: the function-call operator **()**, the increment or decrement operator **++** or **--**, an assignment operator, a comma operator, the array-subscript **[]** and member-access **.** and **->** operators, the address **&** and indirection ***** unary operators, and arbitrary casts.

The operands in an integral constant expression may consist only of integer, enumeration, and character constants, **sizeof** expressions, and casts of arithmetic types to integral types.

More latitude is permitted for constant expressions in initializers. Floating constants and arbitrary casts may also be used. Lvalues designating objects that have static storage duration or function designators may also be used to specify addresses, explicitly with the unary `&` operator, or implicitly, for expressions of array or function type.

e.g.,

```
static int dbl_size = sizeof(double),  
        a[4],  
        *p = &a[2];
```

Further constraints that apply to the integral constant expressions used in conditional-inclusion preprocessing directives are discussed in section 9.4

constant-expression:
conditional-expression

8 STATEMENTS

There are two forms a statement may take:

Simple statement. Because the '=' token is an operator in C, assignment can be viewed as an expression with side effects.

Compound statements. These provide such constructs as loops and multi-way branches.

statement:

labelled-statement
compound-statement
expression-statement
jump-statement
selection-statement
iteration-statement

8.1 Labelled statements

Any statement may be preceded by a prefix that declares an identifier as a label name.

Prospero C will give a warning if a label is not the destination of a **goto** statement within the function.

The **case** and **default** labels may only appear within the body of a **switch** statement (see 8.5.2).

labelled-statement:

identifier : statement
case *constant-ex : statement*
default : *statement*

8.2 Compound statement, or block

A “compound statement” (also called a “block”) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializations. The initializers of objects that have automatic storage duration are evaluated and the objects are initialized in the order they appear in the translation unit.

A compound statement causes a new scope to be opened. It is possible to declare new variables, typedefs and tags in this new scope. A compound statement may also include a series of statements. Prospero C calculates the maximum amount of storage required by all blocks (nested blocks adding to the outer enclosing block). This storage is allocated when the function is entered.

An object declared with the keyword **extern** inside a block may not be initialized in the declaration. This is because storage for it is defined elsewhere.

e.g.,

```
for (i = 0; i < NROWS; i++) {
    int row_total = 0;
    for (j = 0; j < NCOLS; j++)
        row_total += matrix[i][j];
    printf("row %d total is %d\n", i, row_total);
}
```

compound-statement:

{ declaration-list_{opt} statement-list_{opt} }

declaration-list:

declaration

declaration-list declaration

statement-list:

statement

statement-list statement

8.3 Expression and null statements

The expression in an expression statement is evaluated as a `void` expression for its side effects.

A null statement (consisting of just a semicolon) performs no operations.

If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit (it is not mandatory) by converting the expression to a `void` expression by means of a cast:

e.g.,

```
int p(int);
/*...*/
(void) p(0);
```

In the program fragment:

```
char *s;
/*...*/
while (*s++ != '\n')
    ;
```

a null statement is used to supply an empty loop body to the iteration statement.

expression-statement:
expression_{opt} ;

8.4 Jump statements

A jump statement causes an unconditional jump to another place.

Because Prospero C performs syntax and semantic analysis in a single pass any missing labels are not detected until the end of the function in which they are referenced.

jump-statement:
`goto identifier ;`
`continue ;`
`break ;`
`return expressionopt ;`

8.4.1 The goto statement

A **goto** statement causes an unconditional jump to the named label in the current function.

8.4.2 The continue statement

A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body.

e.g.,

```
for (i=1; i++; i<10)
{
    if (a[i+1]==1)
        continue;
    a[i+1]=0;
    /* continue arrives here */
}

while (i)
{
    switch (i)
    {
        case 1: if (a[i+2]==2)
                continue;
                a[i+2]=0;

        case 2: if (a[i+3]==3)
                continue;
                a[i+3]=0;

        default :
    }
    a[i+4]=0;
    /* continue arrives here */
}
```

8.4.3 The break statement

A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

e.g.,

```
switch (k)
{
  case 0: for (i=1; i++; i<10)
    { for (j=1; j++; j<10)
      if (j==5)
        break; /* terminate j loop */
      else
        a[i+j]=0;
      if (a[i+5]==6)
        break; /* terminate i loop */
    }
    /* fall through */

  case 1: switch (l)
    {
      case 0: a[1]=0; /* fall through */
      case 1: a[2]=0;
              break; /* terminate l switch */
      default :
    }
    break; /* terminate k switch */
}
```

8.4.4 The return statement

A **return** statement terminates execution of the current function and returns control to its caller. A function may have any number of **return** statements, with and without expressions.

If a **return** statement with an expression is executed, the value of the expression is returned to the caller. If the expression has a type different from that of the function in which it appears, it is converted as if it were assigned to an object of that type.

If a **return** statement without an expression is executed, and the value of the function call is used by the caller, the behavior is undefined. Reaching the **}** that terminates a function is equivalent to executing a **return** statement without an expression.

In Prospero C, the storage (if any) for the return value is allocated in the calling function.

e.g.,

```
int f1()
{ int i;
  /* code */
  return i;
}
```

```
struct s f2()
{ struct s v1;
  /* code */
  if (a[6]==9)
    return v1;

  /* change v1 */

  return v1;
}
```

8.5 Selection statements

A selection statement selects among a set of statements depending on the value of a controlling expression.

selection-statement:

```
if ( expression ) statement  
if ( expression ) statement else statement  
switch ( expression ) statement
```

8.5.1 The if statement

In both forms, the first substatement is executed if the expression compares unequal to 0. In the **else** form, the second substatement is executed if the expression compares equal to 0. If the first substatement is reached via a label, the second substatement is not executed.

The controlling expression of an **if** statement must have scalar type.

An **else** is associated with the lexically immediately preceding **else-less if** that is in the same block (but not in an enclosed block).

Note: The controlling expression must be enclosed in parentheses.

e.g.,

```
if (index < TABSIZE)  
{ if (table[index])  
  zero++;  
}  
else  
  puts("Error: index out of bounds");
```

8.5.2 The switch statement

A **switch** statement causes control to jump into the statement that is the “switch body”, depending on the value of a controlling expression and the values of any **case** prefixes in the switch body. A **case** or **default** label (cf. section 8.1) is accessible only within the closest enclosing **switch** statement.

The integral promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the type of the promoted controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** prefix. Otherwise, if there is a **default** label, control jumps to the labelled statement. If no converted **case** constant matches and there is no **default** label, none of the statements in the switch body is executed.

In Prospero C the number of **case** labels in a **switch** statement is limited only by the amount of memory available.

The controlling expression of a **switch** statement and the constant expression of each **case** label must have integral type. No two of the **case** constants in the same **switch** statement may have the same value after conversion. There may be at most one **default** label in a **switch** statement.

e.g.,

```
switch (error_level)
{
  case 0 :
  case 1 : warnings++;
          /* fall through */

  case 2 : errors++;
          printf("Error (level %d) : %s\n",
                error_level, error_msg);
          break;

  default : puts("FATAL error");
           exit();
}

switch (value)
  default :
    if (value > 5)
      case 0 :
      case 1 : value = 0;
    else
      case 9 : value = 1;

/* value is set to 1 if it was originally
   2, 3, 4, 5 or 9 (or negative)
   or set to 0 if it was originally
   0, 1, 6, 7, 8 or greater than 9
*/
```

8.6 Iteration statements

An iteration statement causes a statement called the “loop body” to be executed repeatedly until the controlling expression compares equal to 0.

The controlling expression of an iteration statement must have scalar type.

iteration-statement:

```
while ( expression ) statement  
do statement while ( expression ) ;  
for ( expropt ; expropt ; expropt ) statement
```

8.6.1 The while statement

The evaluation of the controlling expression takes place before each execution of the loop body.

e.g.,

```
while (index < TABLE_SIZE)  
    if (table[index] = key)  
        break;  
    else  
        index++;
```

8.6.2 The do statement

The evaluation of the controlling expression takes place after each execution of the loop body.

e.g.,

```
do {  
    ch = getchar();  
    switch (ch)  
    { /* ... */  
    }  
} while (ch != 'Q');
```

8.6.3 The for statement

Except for the behavior of a `continue` statement in the loop body, the statement

```
for ( ex-1 ; ex-2 ; ex-3 ) statement
```

and the sequence of statements:

```
ex-1 ;  
while (ex-2) {  
    statement  
    /* continue in for-loop would arrive here... */  
    ex-3 ;  
    /* ...but continue in while-loop arrives here */  
}
```

are equivalent.

Thus `ex-1` specifies initialization for the loop; `ex-2`, the controlling expression, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; `ex-3` specifies an operation (such as incrementing) that is performed after each iteration.

Both `ex-1` and `ex-3` may be omitted, or may have any type, or may be void expressions. If `ex-2` is omitted, it is treated as if a non-zero constant had been written, i.e., the loop condition is always true.

9 THE PREPROCESSOR

9.1 Introduction

The C preprocessor is a macro preprocessor that processes tokens from the source text file before passing them onto the syntax phase of the compiler proper. The preprocessor is responsible for translation phases 1 to 7 (see section 2.2), and yields a stream of preprocessed tokens to the syntax analyzer. The listing file produced when the L option is specified represents the token stream emitted from the preprocessor.

Although the Prospero C preprocessor is an integral part of the compiler Pass 1 it should conceptually be thought of as a separate entity.

e.g., the following sequence of characters:

```
01< <h3/1.2>=x+++b
#include <2/1.3x>
#define struct.field $
```

forms the following sequence of preprocessing tokens (each individual preprocessing token is delimited by a { on the left and a } on the right).

```
{01} {<} {<} {h3} {/} {1.2} {>} {=} {x} {++} {+} {b}
{#} {include} {<2/1.3x>}
{#} {define} {struct} {.} {field} {$}
```

9.2 Preprocessing directives

A preprocessing directive is a command to the preprocessor. It is begun by a # preprocessing token that is the first non-white-space character on the source line.

The # is optionally followed by one of the following preprocessor directives:

```
define
undef
if
ifdef
ifndef
elif
else
endif
include
line
error
pragma
```

9.3 Defining a macro

The `#define` directive is used to associate a meaningful name with a number, identifier, or expression. It may also be used to aid portability and enhance program readability without sacrificing performance.

There are two types of macros: those without parameters, object-like macros; and those with parameters, function-like macros.

Object-like macro Defines an object-like macro that causes each subsequent instance of the macro name to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

```
#define BUFLen 512
#define TRUE 1
#define forever while (TRUE)
```

define *identifier replacement-list new-line*

Function-like macro Defines a function-like macro with arguments, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends until the new-line character that terminates the `#define` preprocessing directive.

```
#define square(x) ((x) * (x))
#define setbit(x, y) ((x) |= 1 << (y))
```

define *ident (ident-list_{opt}) replace-list new-line*

The identifier immediately following the `define` is called the “macro name”. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

The number of arguments in an invocation of a function-like macro must agree with the number of parameters in the macro definition, and be followed by a) preprocessing token that terminates the invocation.

9.3.1 Redefining a macro name

It is an error to redefine a name already defined as a macro name unless the replacement lists are identical. In the case of a function-like macro the parameters must also be identical in spelling and number.

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation (all white-space separations are considered identical).

e.g., the following sequence is valid:

```
#define BUFLLEN          512
#define max(a, b)        ((a)>(b) ? (a) : (b))
#define BUFLLEN          512
#define max(a,b)         ((a)>(b) ? (a) : (b))
```

but the following redefinitions are invalid:

```
#define BUFLLEN          256
#define max               100
#define max(x,y)          ((x)>(y) ? (x) : (y))
#define max(a,b)          ((a)>(b) ? (a) : (b))
```

9.3.2 Scope of macro definitions

A macro definition lasts (independent of block structure) until a corresponding **#undef** directive is encountered or (if none is encountered) until the end of the translation unit.

A preprocessing directive of the form:

```
# undef identifier new-line
```

causes the specified identifier to be no longer defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name. If the S (strict) flag is enabled Prospero C will give a warning message if an attempt is made to **undef** an identifier that is not defined as a macro.

9.3.3 Macro replacement

During translation phase 4 each token is checked to see if it is identical to a defined name. If it does match a macro name it is replaced by the body of that macro name as follows:

Object-like macros.

The token is replaced by the token list making up the body of the macro name.

Function-like macros.

If the macro name is followed by a (token, the tokens between the matching opening and closing parenthesis constitute the parameters of the macro invocation. Individual arguments between the outermost parenthesis are separated by comma tokens (comma preprocessing tokens bounded by nested parenthesis do not separate arguments). Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a white-space character.

Prospero C gives a warning (or an error if the S option is in force) if more arguments are given in an occurrence than its definition. The extra arguments are discarded. Prospero C gives an error if insufficient arguments are supplied. A function-like macro name not followed by a (token is treated as an identifier, and left unchanged by the preprocessor.

Given the previous legal macro definitions, the following fragments:

```
char buffer[BUFLLEN];
forever { do_next_char(); }
num = square(num);
setbit(flag, 4);
```

expand to the following:

```
char buffer[512];
while (1) {do_next_char(); }
num = ((num) * (num));
((flag) |= 1 << (4));
```

9.3.4 Argument substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see sections 9.3.6 and 9.3.7), is replaced by the corresponding argument after all macros contained therein have been expanded. The argument's preprocessing tokens are completely macro replaced as if they formed the rest of the source file.

9.3.5 Rescanning and further replacement

After all parameters in the replacement list have been substituted, the resulting preprocessing token sequence is rescanned with the rest of the source file's preprocessing tokens for more macro names to replace.

If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Further, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These non-replaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

e.g.,

```
#define A      A B C
#define B      B C A
#define C      C A B

A
/* expands to... */
A B C
/* rescan: A is not expanded... */
A { B C A } { C A B }
/* rescan of B: A and B not expanded... */
A { B { C A B } A } { C A B }
/* no further expansion possible in B */
/* rescan of C: A and C not expanded... */
A B C A B A { C A B C A }
/* no further expansion possible in C */
A B C A B A C A B C A
/* ...which is the result */
```

The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and of generating more code than a function if invoked several times.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

To illustrate the rules for substitution and reexamination, the sequence:

```
#define pair(x,y)      (x->next = y)
#define last(q)       (q->next ? last(q->next) : q)

pair(a,b);
pair(a, pair(b, c));
a = last(c);
a = last(last(b));
pair(a, last(c));
b = last(pair(a, c));
```

results in:

```
(a->next = b);
(a->next = (b->next = c));
a = (c->next ? last(c->next) : c);
a = ((b->next ? last(b->next) : b)->next ?
last((b->next ? last(b->next) : b)->next) :
(b->next ? last(b->next) : b));
(a->next = (c->next ? last(c->next) : c));
b = ((a->next = c)->next ? last((a->next = c)
->next) : (a->next = c));
```

The name following the # preprocessing token that appears to begin a preprocessing directive is not subject to macro replacement, even if it has been defined as a macro name.

e.g.,

```
#define include error
#include "inc.h" /* still includes inc.h */
```

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one.

9.3.6 The # operator

If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the string literal. Special handling is required to produce the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters).

Each # preprocessing token in the replacement list for a function-like macro must be followed by a parameter as the next preprocessing token in the replacement list.

9.3.7 The ## operator

If, in the replacement list, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence.

For both object-like and function-like macro invocations, before the replacement list is re-examined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. If the result is not a valid token, the behavior is undefined. The resulting token is available for further macro replacement.

To illustrate the rules for creating string literals and concatenating tokens, the sequence:

```
#define str(x)      #x
#define xstr(x)    str(x)

#define VERSION    2
#define header(v)  xstr(incv ## v)
#define incfile(v) header(v.h)

str(hello)
str(!)
str(str(?))
xstr(str(?))
#include incfile(VERSION)
```

results in:

```
"hello"
"! "
"str(?)"
"\ "?\ ""
#include "incv2.h"
```

See section 9.5 for a full description of the lexical rules governing the characters in a filename in a #include directive.

Space around the # and ## tokens in the macro definition is optional.

A ## preprocessing token may not occur at the beginning or at the end of a replacement list for either form of macro definition.

9.4 Conditional inclusion

The preprocessor can be used to selectively include/exclude lines of input source from further processing by the compiler.

```
#if const-expr
group-of-lines1
#else
group-of-lines2
#endif
```

If the *const-expr* is zero the text *group-of-lines1* is skipped; the text *group-of-lines2* is processed and passed on to the compiler. If the *const-expr* has a non-zero value *group-of-lines1* is processed while *group-of-lines2* is skipped.

It is possible to nest **#if** directives. The preprocessor matches **#elses**, **#elifs** and **#endifs**.

For nested **#if**, **#else** and **#endif** directives the **#elif** may be used:

```
# if constant-ex new-line groupopt
# else
# if constant-ex new-line groupopt
```

can be replaced by:

```
# if constant-ex new-line groupopt
# elif constant-ex new-line groupopt
```

Preprocessing directives of the forms:

```
# ifdef identifier new-line groupopt
# ifndef identifier new-line groupopt
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to **#if defined** *identifier* and **#if !defined** *identifier* respectively.

Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only as far as the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (non-zero) is processed. If none of the conditions evaluates to true, and there is a **#else** directive, the group controlled by the **#else** is processed; lacking a **#else** directive, all the groups until the **#endif** are skipped.

The expression must be an integral constant expression that must not contain a **sizeof** operator, a cast, or an enumeration constant. However, it may contain unary expressions of the form:

defined *identifier*

or

defined (*identifier*)

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive), 0 if it is not.

Note: Superficially the folding of constant expressions is always done in **long** or **unsigned long** mode. However, some operators produce **int** results i.e., relational operators. Combinations of these can result in some expression evaluation being done in **int** mode.

9.4.1 Evaluating constant expressions

Prior to evaluation, identifiers currently defined as macro names are replaced (except for those identifiers modified by **defined**) in the list of preprocessing tokens just as in normal text. After all replacements are finished and just before the evaluation of the controlling constant expression, all remaining identifiers are replaced with **0L** and each integer constant not already suffixed with **l** or **L** is considered to be additionally suffixed with **L**. During the evaluation of the expression the usual arithmetic conversions apply. This includes interpreting character constants, which may involve converting escape sequences into characters. The numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression that is not constant folded.

9.5 Source file inclusion

A preprocessing directive of the form:

```
# include <h-char-sequence> new-line
```

where *h-char-sequence* is a sequence of characters other than > or newline, causes the replacement of that directive by the entire contents of the source file identified by the specified character sequence between the < and > delimiters and a set of pathnames and causes the replacement of that directive by the entire contents of the header. If the characters \ , " , or * occur in the character sequence, the behavior is undefined.

Prospero C allows #include files to be nested to a depth limited only by the amount of memory available. Recursive #includes (*a.h* includes *b.h* includes *a.h* ...) are allowed. The recursion will be broken, and a warning message given, if the same file is included more than four times.

If the characters /*, or */ occur in the character sequence, the behavior is undefined.

A preprocessing directive of the form:

```
# include "q-char-sequence" new-line
```

where *q-char-sequence* is a sequence of characters other than " or newline, causes the replacement of that directive by the entire contents of the source file identified by the specified character sequence between the " delimiters. The named source file is searched for in association with the original source file. If the search fails, the directive is reprocessed as if it read:

```
# include <h-char-sequence> new-line
```

with the identical contained character sequence (including > characters, if any) from the original directive.

A preprocessing directive of the form:

```
# include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after include in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements must match one of the two previous forms.

The sequence of preprocessing tokens between a < and a > preprocessing token pair is combined into a single header name preprocessing token by replacing each sequence of white space between pairs of preprocessing tokens in the sequence by a single space character, deleting any white space between the < and the next preprocessing token and between the > and the previous preprocessing token, and replacing each contained preprocessing token by its spelling. i.e., to form a single token between the < and > delimiters, the glue operator ## must be used as tokens are *not* implicitly concatenated.

e.g.,

```
#if __STDC__
    #define header stdio ## . ## h
#else
    #define header extio ## . ## h
#endif
#define incname < header >
#include incname
```

For more complex examples of macro-replaced #includes, see the sections on # and ## operators above (sections 9.3.6 and 9.3.7).

9.6 Line control

The “line number” of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (while processing the source file to the current token)

A preprocessing directive of the form:

```
# line digit-sequence new-line
```

causes Prospero C to behave as if the line number of the next source line is specified by the digit sequence (interpreted as a decimal integer).

A preprocessing directive of the form:

```
# line digit-sequence string-literal new-line
```

sets the line number similarly and changes the presumed name of the source file to be the characters contained within the string literal.

A preprocessing directive of the form:

```
# line pp-tokens new-line
```

allows the digit-sequence and string-literal to be generated via macro names. The preprocessing tokens after **line** on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after all replacements must match one of the two previous forms.

9.7 Error directive

A preprocessing directive of the form:

```
# error pp-tokensopt new-line
```

causes the list of pp-tokens to be sent to the listing and log file. Compilation then stops.

9.8 Pragma directive

A preprocessing directive of the form:

```
# pragma pp-tokensopt new-line
```

provides a mechanism for passing information to the compiler. Prospero C does not currently make use of the pragma tokens. They are simply discarded.

9.9 Null directive

A preprocessing directive of the form:

```
# new-line
```

has no effect.

9.10 Predefined macro names

The following macro names are predefined by Prospero C.

<code>__LINE__</code>	The line number of the current source line (a decimal constant).
<code>__FILE__</code>	The presumed name of the source file (a string literal).
<code>__DATE__</code>	The date of translation of the source file (a string literal of the form " Mmm dd yyyy ", where the names of the months are the same as those generated by the asctime function, and the first character of dd is a space character if the value is less than 10).
<code>__TIME__</code>	The time of translation of the source file (a string literal of the form " hh:mm:ss " as in the time generated by the asctime function).
<code>__STDC__</code>	The decimal constant 1 if the S (strict) option is specified, else 0.

None of these macro names, nor the identifier **defined**, may be the subject of a **#define** or a **#undef** preprocessing directive.

In addition, Prospero C will predefine the following macros with an empty body if the corresponding compile-time option is invoked. These macros may be undefined by the program.

_NINFO	Defined if the N (include source line information) option is specified.
_ICHECK	Defined if the I (check array indexes) option is specified.
_ACHECK	Defined if the A (check assignments) option is specified.
_PCHECK	Defined if the P (check pointers) option is specified.
_UCHAR	Defined if the U (char is unsigned) option is specified.
_COMPACT	Defined if the C (compact code) option is specified.

9.11 Preprocessor syntax summary

preprocessing-file:

group

group:

group-part

group group-part

group-part:

pp-tokens_{opt} new-line

if-section

control-line

if-section:

if-group elif-groups_{opt} else-group_{opt} endif-line

if-group:

if *constant-ex new-line group_{opt}*

ifdef *identifier new-line group_{opt}*

ifndef *identifier new-line group_{opt}*

elif-groups:

elif-group

elif-groups elif-group

elif-group:

elif *constant-ex new-line group_{opt}*

else-group:

else *new-line group_{opt}*

endif-line:

endif *new-line*

control-line:

include *pp-tokens new-line*

define *identifier replacement-list new-line*

define *ident lparen ident-list_{opt}) replace-list new-line*

undef *identifier new-line*

line *pp-tokens new-line*

error *pp-tokens_{opt} new-line*

pragma *pp-tokens_{opt} new-line*

new-line

lparen:

the left-parenthesis character without preceding white-space

replacement-list:

*pp-tokens*_{opt}

pp-tokens:

preprocessing-token

pp-tokens preprocessing-token

preprocessing-token:

header-name

(only within a **#include** directive)

identifier

(no keyword distinction)

pp-number

character-constant

string-literal

operator

punctuator

each non-white-space character that cannot be one of the above

header-name:

<h-char-sequence>

"q-char-sequence"

h-char-sequence:

h-char

h-char-sequence h-char

h-char:

any character in the source character set
except the new-line character and >

q-char-sequence:

q-char

q-char-sequence q-char

q-char:

any character in the source character set
except the new-line character and "

pp-number:

digit

. digit

pp-number digit

pp-number nondigit

pp-number e sign

pp-number E sign

pp-number .

new-line:

the new-line character

10 BIBLIOGRAPHY

“Draft Proposed American National Standard for Information Systems - Programming Language C”, 3 August 1987

“Rationale for Draft Proposed American National Standard for Information Systems – Programming Language C”, 3 August 1987

“The C Reference Manual” by Dennis M. Ritchie, a version of which was published in “The C Programming Language” by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., (1978).

“1984 /usr/group Standard” by the /usr/group Standards Committee, Santa Clara, California, USA (November, 1984).

“American National Dictionary for Information Processing Systems”, Information Processing Systems Technical Report ANSI X3/TR-1-82 (1982).

“ISO 646-1983 Invariant Code Set”.

“IEEE Standard for Binary Floating-Point Arithmetic” (ANSI/IEEE Std 754-1985).

11 INDEX

- # directive 111
- # operator 104
- ## operator 105
- #define 99
- #elif 106
- #endif 106
- #else 106
- #error 110
- #if 106
- #ifdef 106
- #ifndef 106
- #include 108-109
- #line 110
- #pragma 111
- #undef 100
- _DATE_ 111
- _FILE_ 111
- _LINE_ 111
- _STDC_ 111
- _TIME_ 111
- _ACHECK 112
- _COMPACT 113
- _ICHECK 112
- _NINFO 112
- _PCHECK 112
- _UCHAR 112
- abstract declarator 51
- aggregate
 - initialization 56
- aggregate type 7
- alignment 39
 - bit-field 38
 - restriction 74
- alignment rules 6
- arithmetic
 - pointer 76
 - type 7
- array
 - bounds 47
 - conversion of 14, 49
 - declaration 47
 - declarator 47
 - default size 47
 - element address comparison 79
 - expression 87
 - extern 47
 - function argument 53
 - function returning 50
 - incomplete initializer 56
 - initialization 47
 - multi-dimensional 47
 - order of storage 64
 - padding 39
 - pointer to 76
 - same type 51
 - sizeof 47
 - subscript 64, 86
- assignment
 - compound 85
 - expression 84
 - simple 84
- auto 20, 53, 55, 59
 - initialization 55
- basic type 7
- bit-field 8
 - alignment 38
 - declaration 37
 - type 38
- block 90
 - structure 16, 88
- break statement 93
- C
 - pre-standard 48
 - Prospero C 20, 22, 28, 36, 38, 43, 49, 73, 79, 84, 98, 100, 101, 108, 111
 - standard 52
- call by value 66
- case label 88
- char 35
- character
 - code set 5
 - constant 28, 30, 107
 - set 4
 - string 31
 - type 6
- comment 21, 32
- compound
 - statement 55, 89
- conditional
 - compilation 106
- const 14, 34, 35, 36, 46, 51

- constant 21, 23, 63
 - character 30
 - decimal 27
 - double 24
 - enum 42
 - evaluation 107
 - expression 38, 47, 86, 87, 107
 - floating 24, 87
 - folding 107
 - hexadecimal 25
 - integer 25
 - integral 74
 - null pointer 15, 55, 74, 79, 80, 83, 84
 - octal 25
 - pointer 46
 - unsigned 27
- continue statement 91
- conversion
 - argument type 65
 - arithmetic 13
 - by assignment 84
 - by return 92
 - character-integer 8
 - double-float 12
 - float-double 12
 - floating-integer 12
 - function name 14
 - implicit 63
 - integer-character 8
 - integer-floating 12
 - integer-long 8
 - integer-pointer 15
 - integer-unsigned 10
 - long-integer 10
 - long-unsigned 10
 - of array 14
 - pointer 36, 74
 - pointer-integer 74
 - pointer-pointer 15
 - quiet 74
 - type rules 13
 - unsigned-integer 10
 - usual arithmetic 75, 76, 79, 81, 83, 107
- declarations 33
- declarator 45
 - abstract 51
 - default label 88
 - defined 107
 - derived type 7
 - do' statement 96
 - double 35
 - empty statement 90
 - enum
 - constant 42
 - specifier 42
 - storage 42
 - type 42
 - enumerated type 6
 - enumeration constant 28
 - escape sequence 3, 29, 31, 107
 - expression 61
 - ? : conditional 83
 - full 61
 - order of evaluation 61
 - parenthesized 63
 - primary 63
 - statement 90
 - extensions 22, 43
 - external definition 18, 59
 - float 7, 35, 50
 - for statement 97
 - fortran 22
 - Fortran-77 43
 - function 33, 37, 44
 - argument 65
 - auto 43
 - call 64, 66
 - declaration of 48
 - declarator 48
 - definition 53
 - designator 14
 - expression 48
 - function argument 53
 - implicit declaration of 66
 - name argument 53
 - order of parameters 43
 - parameter 47
 - parameter type 48
 - pointer to 53
 - prototype 16
 - same type 51
 - static 43

- type 6
- void parameter 48
- goto statement 91
- hexadecimal constant 25
- if-else statement 94
- incomplete type 6, 7, 15, 47, 80
- initialization 20, 55
 - aggregate 56
 - auto 55
 - character array 56
 - constraints 55
 - implicit 55
 - in blocks 88
 - permitted form of initializer 87
 - scalar 55
 - static 55
 - struct 55
- int 35
 - implied 34
 - long 34
 - short 34
- integral type 7
- keyword 22
 - macro names 23
- label 19, 22, 89
 - case 89
 - default 89
- labelled statement 88
- linkage 60
 - external 18
 - internal 18
 - none 18
- long 35
- loop body 96
- lvalue 14, 53, 61, 71, 83-87
 - modifiable 84
- macro
 - argument 101
 - argument substitution 102
 - body 99
 - definition scope 100
 - function-like 99, 101
 - identical 100
 - invocation 101
 - name 22, 99
 - object-like 99, 101
 - parameters 99
 - predefined 111, 112
 - preprocessor 98
 - rescanning and replacement 102
- name spaces 19
 - label 19
 - struct members 19
 - tag 19
 - union members 19
- new-line character 2
- null statement 90
- object
 - types 6
- object definitions 60
- octal constant 25
- operand 32
- operator 21, 32
 - <!=> inequality 80
 - <!> logical negation 70
 - <##> glue 105
 - <#> stringify 104
 - <&&> logical AND 82
 - <&> 38
 - <&> bitwise AND 81
 - <%> modulus 75
 - <*> indirection 70
 - <*> multiplication 75
 - <+> increment 64, 70
 - <+> addition 76, 77
 - <+> unary plus 72
 - <,> comma 86
 - <--> decrement 64, 69, 70
 - <-> > member select indirect 64
 - <-> subtraction 76, 77
 - <.> member select direct 64
 - </> division 75
 - <<<> left shift 78
 - <<=> less than or equal 79
 - <<> less than 79
 - <==> equality 80
 - <>=> greater than or equal 79
 - <>> greater than 79
 - <>>> right shift 78
 - <[> subscript 64
 - <^> bitwise exclusive OR 81
 - <|> bitwise inclusive OR 81
 - <||> logical OR 82
 - <~> bitwise complement 72

- <--> unary minus 72
- arithmetic 75
- assignment 84
- bitwise shift 78
- bitwise shift 78
- cast 74
- commutative 75,76,80,81
- explicit conversion 74
- multiplicative 75
- postfix <+>{increment} and <-->{decrement} 69
- prefix <+>{increment} and <-->{decrement} 70
- relational 79
- sizeof 14, 18, 70, 86, 107
- option
 - A 112
 - C 112
 - I 112
 - N 112
 - P 112
 - S 22, 43, 79, 100, 111
 - U 8, 28, 112
- parameter
 - default argument promotion 49
 - ellipsis 49, 67
 - passing 65
 - register 50
 - specification 53
 - storage duration 53
 - typedef 53
- pascal 22, 43
- phases of translation 2
- pointer
 - arithmetic 76
 - comparison 79
 - constant 46
 - declaration of 46
- precedence 62
- preprocessing directives 98
- promotion
 - arithmetic 13
 - default argument 67
 - integral 8
- punctuator 21, 32
- recursion 66
- register 43, 53, 59
 - allocation 43
 - reserved words 22
 - return statement 92
 - scalar type 7
 - scope 16, 49
 - block scope 16
 - disjoint 52
 - file scope 16, 59
 - function scope 16
 - of externals 60
 - tag scope 16
 - self-referential struct 40
 - separate compilation 2
 - sequence point 69, 86
 - short 35
 - signed 35
 - source character set 5
 - statements 88
 - static 20, 43
 - tentative definition 60
 - storage
 - defined elsewhere 47
 - overlapping 84
 - static 43, 87
 - storage duration 20
 - automatic 20
 - static 20
 - storage-class
 - auto 44
 - declaration 43
 - extern 44
 - omitted specifier 44
 - register 44, 48
 - specifier 43
 - static 44
 - string
 - concatenation 31
 - literal 31
 - type of 63
 - struct
 - initialization 55
 - member name 19
 - structure
 - declaration 37
 - reference 67
 - tag 40
 - switch 94

- syntax error 3
- tag
 - enum 19
 - scope 16
 - struct 19
 - union 19
- target character set 5
- tentative definition 60
- tokens 21
- translation phases 2
- translation unit 2
- trigraph sequences 5
- type 6
 - aggregate 7
 - arithmetic 7
 - basic 7
 - char 6
 - conversion by return 92
 - conversion rules 13
 - declaration 45
 - derived 7
 - equivalence 51
 - function 6
 - incomplete 6, 7, 15, 80
 - integral 7, 76, 78
 - names 50
 - scalar 7
 - tags 51
- type specifier 34
 - omitted 34
- typedef 22, 43
 - declaration 44
 - declaration 51
- unary expression 70
- underscore character 23
- union
 - declaration 37
 - reference 67
 - member name 19
 - tag 40
- unsigned 35
- void 7, 14, 35, 74, 80
 - expression 14
- volatile 34, 35, 36, 46, 51
- while statement 96
- white-space 2, 21, 99, 101, 104





A SOURCE LANGUAGE SYNTAX

Throughout this manual and the appendix, the formal syntax definitions follow the notation described below. Non-terminal symbols (syntactic categories) are in *italic* type, while terminal symbols (literal words and characters) are in **bold courier** type. Note that the upper- and lower-case versions of a letter are not equivalent. Optional symbols are suffixed by the subscript 'opt'.

The definition of a syntactic category consists of the name of the non-terminal symbol followed by a colon on one line, followed by the definition or definitions indented on the following lines, one line per alternative (except where 'one of' is specified). A blank line indicates the end of a definition.

For increased readability, the non-terminal symbols are often hyphenated.

In this appendix, the nature of the source file which is input to the compiler (the 'compilation-unit') is viewed from two complementary aspects: the lexical (or bottom-up) and syntactic (or top-down). These views merge at about the level of the expression. The division of the remainder of this appendix into two subsections is designed to mirror these two viewpoints.

Taken together, subsections A.2 and A.3 contain one, and only one, definition for every non-terminal symbol.

Subsection A.4 defines the syntax of the preprocessor.

A.2 Lexical aspects

A.2.1 Tokens

token:

keyword
identifier
constant
string-literal
operator
punctuator

A.2.2 Keywords

keyword: one of

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while
fortran	pascal	(when the S option is off)	

A.2.3 Identifiers

*identifier:**non-digit**identifier non-digit**identifier digit**non-digit: one of*

–	a	b	c	d	e	f	g	h	i	j	k	l	m
	n	o	p	q	r	s	t	u	v	w	x	y	z
	A	B	C	D	E	F	G	H	I	J	K	L	M
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

\$ (unless S option is specified)

digit: one of

0 1 2 3 4 5 6 7 8 9

A.2.4 Constants

*constant:**floating-constant**integer-constant**enumeration-constant**character-constant**floating-constant:**fractional-constant* *exponent*_{opt} *floating-suffix*_{opt}*digit-sequence* *exponent* *floating-suffix*_{opt}*fractional-constant:**digit-sequence*_{opt} . *digit-sequence**digit-sequence* .*exponent:***e** *sign*_{opt} *digit-sequence***E** *sign*_{opt} *digit-sequence**sign: one of*

+ -

*digit-sequence:**digit**digit-sequence* *digit**floating-suffix: one of***f** **l** **F** **L**

integer-constant:

decimal-constant integer-suffix_{opt}

octal-constant integer-suffix_{opt}

hexadecimal-constant integer-suffix_{opt}

decimal-constant:

non-zero-digit

decimal-constant digit

octal-constant:

0

octal-constant octal-digit

hexadecimal-constant:

0x hexadecimal-digit

0X hexadecimal-digit

hexadecimal-constant hexadecimal-digit

non-zero-digit: one of

1 2 3 4 5 6 7 8 9

octal-digit: one of

0 1 2 3 4 5 6 7

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

integer-suffix:

unsigned-suffix long-suffix_{opt}

long-suffix unsigned-suffix_{opt}

unsigned-suffix: one of

u U

long-suffix: one of

l L

enumeration-constant:

identifier

character-constant:

'c-char-sequence'

c-char-sequence:

c-char

c-char-sequence c-char

c-char:

any character in the source character set except
the single-quote ' , backslash \, or new-line

escape-sequence

escape-sequence:

simple-escape-sequence

octal-escape-sequence

hexadecimal-escape-sequence

simple-escape-sequence: one of

\' \\" \? \\ \a \b \f \n \r \t \v

octal-escape-sequence:

\ octal-digit

\ octal-digit octal-digit

\ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:

\x hex-digit

hexadecimal-escape-sequence hex-digit

A.2.5 String literals

string-literal:

"s-char-sequence_{opt}"

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

any character in the source character set except
the double-quote " , backslash \, or new-line

escape-sequence

A.2.6 Operators

operator: one of

```
[ ] ( ) . ->
++ -- & * + - ~ ! sizeof
/ % << >> < > <= >= == !=
^ | && ||
? :
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
```

A.2.7 Punctuators

punctuator: one of

```
[ ] ( ) { } * , : = ; ... #
```

A.3 Syntactic aspects

A.3.1 Expressions

primary-ex:

identifier
constant
string-literal
(expression)

postfix-ex:

primary-ex
postfix-ex [expression]
postfix-ex (argument-expression-list_{opt})
postfix-ex . identifier
postfix-ex -> identifier
postfix-ex ++
postfix-ex --

argument-expression-list:

assignment-ex
argument-expression-list , assignment-ex

unary-ex:

postfix-ex
++ unary-ex
-- unary-ex
unary-operator cast-ex
sizeof *unary-ex*
sizeof (*type-name*)

unary-operator: one of

& * + - ~ !

cast-ex:

unary-ex
(type-name) cast-ex

multiplicative-ex :

cast-ex
*multiplicative-ex * cast-ex*
multiplicative-ex / cast-ex
multiplicative-ex % cast-ex

additive-ex:

multiplicative-ex
additive-ex + multiplicative-ex
additive-ex - multiplicative-ex

shift-ex:

additive-ex
shift-ex << *additive-ex*
shift-ex >> *additive-ex*

relational-ex:

shift-ex
relational-ex < *shift-ex*
relational-ex > *shift-ex*
relational-ex <= *shift-ex*
relational-ex >= *shift-ex*

equality-ex:

relational-ex
equality-ex == *relational-ex*
equality-ex != *relational-ex*

AND-ex:

equality-ex
AND-ex & *equality-ex*

exclusive-OR-ex:

AND-ex
exclusive-OR-ex ^ *AND-ex*

inclusive-OR-ex:

exclusive-OR-ex
inclusive-OR-ex | *exclusive-OR-ex*

logical-AND-ex:

inclusive-OR-ex
logical-AND-ex && *inclusive-OR-ex*

logical-OR-ex:

logical-AND-ex
logical-OR-ex || *logical-AND-ex*

conditional-ex:

logical-OR-ex
logical-OR-ex ? *ex* : *conditional-ex*

assignment-ex:

conditional-ex
unary-ex *assignment-operator* *assignment-ex*

assignment-operator: one of

= * = / = % = + = - = << = >> = & = ^ = | =

expression:
 assignment-ex
 expression , assignment-ex

constant-expression:
 conditional-expression

A.3.2 Declarations

declaration:
 declaration-specifiers init-declarator-list_{opt};

declaration-specifiers:
 storage-class-specifier declaration-specifiers_{opt}
 type-specifier declaration-specifiers_{opt}

init-declarator-list:
 init-declarator
 init-declarator-list , init-declarator

init-declarator:
 declarator
 declarator = initializer

type-specifier:
 void
 char
 short
 int
 long
 float
 double
 signed
 unsigned
 const
 volatile
 struct-or-union-specifier
 enum-specifier
 typedef-name

struct-or-union-specifier:
*struct-or-union identifier*_{opt} { *struct-declaration-list* }
struct-or-union identifier

struct-or-union:
struct
union

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

struct-declaration:
type-specifier-list struct-declarator-list ;

type-specifier-list:
type-specifier
type-specifier-list type-specifier

struct-declarator-list:
struct-declarator
struct-declarator-list , struct-declarator

struct-declarator:
declarator
*declarator*_{opt} : *constant-expression*

enum-specifier:
enum *identifier*_{opt} { *enumeration-list* }
enum *identifier*

enumeration-list:
enumeration
enumeration-list , enumeration

enumeration:
enumeration-constant
enumeration-constant = constant-expression

storage-class-specifier:
typedef
extern
static
auto
register
pascal
fortran

declarator:
*pointer*_{opt} *direct-declarator*

direct-declarator:

identifier

(declarator)

direct-declarator [*constant-expression*_{opt}]

direct-declarator (*parameter-type-list*)

direct-declarator (*identifier-list*_{opt})

pointer:

* *type-specifier-list*_{opt}

* *type-specifier-list*_{opt} *pointer*

parameter-type-list:

parameter-list

parameter-list ,

parameter-list:

parameter-declaration

parameter-list , *parameter-declaration*

parameter-declaration:

declaration-specifiers declarator

*declaration-specifiers abstract-declarator*_{opt}

identifier-list:

identifier

identifier-list , *identifier*

type-name:

*type-specifier-list abstract-declarator*_{opt}

abstract-declarator:

pointer

*pointer*_{opt} *direct-abstract-declarator*

direct-abstract-declarator:

(abstract-declarator)

*direct-abstract-declarator*_{opt} [*constant-expression*_{opt}]

*direct-abstract-declarator*_{opt} (*parameter-type-list*_{opt})

typedef-name:

identifier

initializer

assignment-expression

{ *initializer-list* }

{ *initializer-list* , }

initializer-list

initializer

initializer-list , *initializer*

A.3.3 Statements

statement:

labelled-statement
compound-statement
expression-statement
jump-statement
selection-statement
iteration-statement

labelled-statement:

identifier : *statement*
case constant-ex : *statement*
default : *statement*

compound-statement:

{ *declaration-list*_{opt} *statement-list*_{opt} }

declaration-list:

declaration
declaration-list declaration

statement-list:

statement
statement-list statement

expression-statement:

*expression*_{opt} ;

jump-statement:

goto *identifier* ;
continue ;
break ;
return *expression*_{opt} ;

selection-statement:

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
switch (*expression*) *statement*

iteration-statement:

while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for (*expr*_{opt} ; *expr*_{opt} ; *expr*_{opt}) *statement*

A.3.4 External definitions

translation-unit:
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

function-definition
*declaration-specifiers*_{opt} *declarator declaration-list*_{opt} *compound-statement*_{opt}

A.4 Preprocessing directives

preprocessing-file:
group

group:
group-part
group group-part

group-part:
*pp-tokens*_{opt} *new-line*
if-section
control-line

if-section:
*if-group elif-groups*_{opt} *else-group*_{opt} *endif-line*

if-group:
if *constant-ex new-line group*_{opt}
ifdef *identifier new-line group*_{opt}
ifndef *identifier new-line group*_{opt}

elif-groups:
elif-group
elif-groups elif-group

elif-group:
elif *constant-ex new-line group*_{opt}

else-group:
else *new-line group*_{opt}

endif-line:

endif *new-line*

control-line:

include *pp-tokens new-line*
 # **define** *identifier replacement-list new-line*
 # **define** *ident lparen ident-list_{opt}) replace-list new-line*
 # **undef** *identifier new-line*
 # **line** *pp-tokens new-line*
 # **error** *pp-tokens_{opt} new-line*
 # **pragma** *pp-tokens_{opt} new-line*
 # *new-line*

lparen:

the left-parenthesis character without preceding white-space

replacement-list:

pp-tokens_{opt}

pp-tokens:

preprocessing-token
pp-tokens preprocessing-token

preprocessing-token:

header-name (only within a **#include** directive)
identifier (no keyword distinction)
pp-number
character-constant
string-literal
operator
punctuator
 each non-white-space character that cannot be one of the above

header-name:

<*h-char-sequence*>
 "*q-char-sequence*"

h-char-sequence:

h-char
h-char-sequence h-char

h-char:

any character in the source character set
 except the new-line character and >

q-char-sequence:

q-char

q-char-sequence q-char

q-char:

any character in the source character set
except the new-line character and "

pp-number:

digit

. digit

pp-number digit

pp-number nondigit

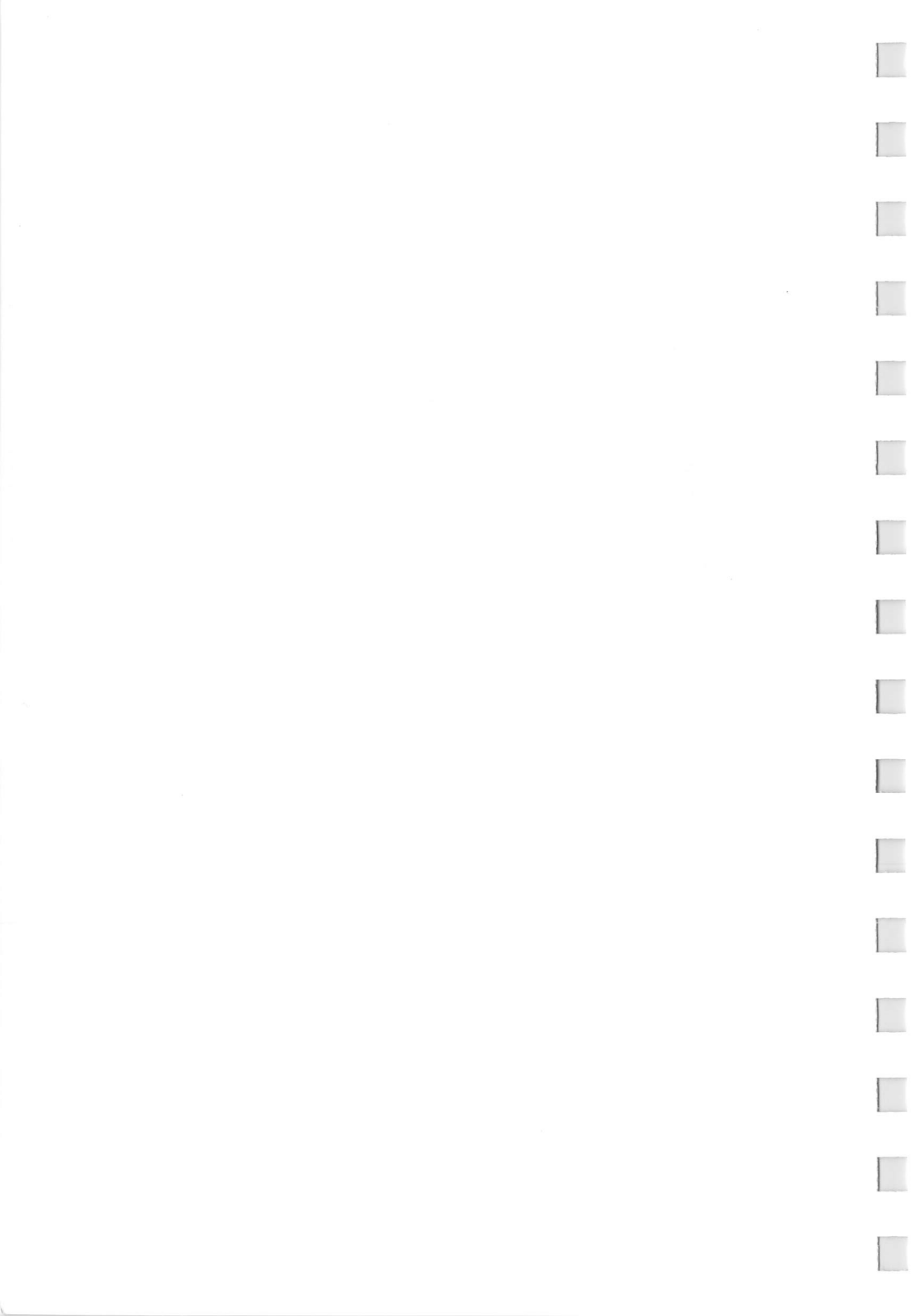
pp-number \ominus sign

pp-number \boxplus sign

pp-number .

new-line:

the new-line character



B COMPILE-TIME ERRORS

For each error or warning number, the text which is displayed by the compiler (provided C.ERR is present) is given.

Number Meaning

001	digit expected
002	number not followed by operator, punctuator or white space
003	illegal octal number
004	illegal hex constant or escape sequence
014	'float' constant overflow, assumed 'double'
015	'float' constant underflow, assumed 'double'
024	'double' constant overflow
025	'double' constant underflow
030	overflow in hex character constant, top bits ignored
031	overflow in octal character constant, top bits ignored
034	\x... escape sequence has no hex digits
035	illegal escape code
044	character constant contains too many characters
045	illegal character constant
049	character constant exceeds source line
059	string literal exceeds source line
060	comments do not nest
061	end of included file encountered inside comment
068	end of file encountered inside comment
069	premature end-of-file encountered
100	(implicit) cast to narrower type may lose information
101	constant truncated due to (implicit) cast
102	-ve constant cast to 'unsigned' type

103	non-NULL constant cast to pointer
104	implicit address of 'register' array
116	overflow casting floating point constant to 'signed char'
117	overflow casting floating point constant to 'signed long'
118	overflow casting floating point constant to 'signed short'
126	overflow casting floating point constant to 'unsigned char'
127	overflow casting floating point constant to 'unsigned long'
128	overflow casting floating point constant to 'unsigned short'
136	overflow casting 'double' to 'float'
204	attempt to assign to 'struct'/'union' with 'const' field
205	attempt to modify 'const' expression
206	attempt to use value of 'void' expression
211	identifier assumes previous (out of scope) extern declaration
212	parameter with function type converted to ptr-to-fn
213	parameter with array type converted to pointer type
214	incompatible pointer types
215	incompatible types in expression
220	default declaration of identifier as 'extern int ()'
224	too few arguments in function call
225	too many arguments in function call
226	argument not compatible with parameter type
227	object in function call not function or ptr-to-fn

234	identifier not declared - assumed 'extern int'
235	no such field in 'struct'/'union'
236	left operand of selection must be a 'struct'/'union'
244	operand of ++/-- may not be pointer to 'void'/function
246	operand of ++/-- not a scalar
247	operand of ++/-- is not an lvalue
254	address of 'register' variable illegal
255	dereferencing NULL pointer constant
256	operand of & is not an lvalue
257	address of bit field illegal
258	indirection requires pointer type
259	dereferencing pointer to 'void'
260	- 'unsigned long' could overflow
266	operand of + not a scalar
267	operand of ! not a scalar
268	operand of - not arithmetic
269	operand of ~ not integral
270	operand of 'sizeof' has side effects
276	'sizeof' bit field illegal
277	'sizeof' function illegal
278	'sizeof' incomplete type illegal
279	'sizeof void' illegal
286	cast of non-scalar illegal
287	cast to non-scalar illegal
288	cast of 'void' illegal
296	operands of * / not arithmetic
297	operands of % not integral
298	operands of + not both arithmetic or pointer/integer

- 298 operands of + not both arithmetic or
pointer/integer
- 299 operands of - not both arithmetic,
pointer/integer or pointer/pointer
- 300 right operand of << >> is negative, shift
ignored
- 301 right operand of << >> exceeds width of left
operand
- 306 operands of << >> not integral
- 307 constant out of range
- 314 incompatible types in comparison
- 316 operands of comparison not scalar
- 326 operands of & | ^ not integral
- 336 operands of && || not scalar
- 344 arms of ? : expression are not same type
- 346 control expression not scalar
- 350 assignment may lose information
- 354 'static' function not defined in translation
unit
- 355 incompatible types in assignment
- 356 lhs of assignment is not an lvalue
- 364 incompatible types in compound assignment
- 370 left operand of comma has no side effects
- 371 control branch displacement exceeds 32K
- This normally implies that the body of a compound statement
which requires a branch around it (e.g. the subject of a selection
or iteration statement) is too large.
- 400 Compiler internal error
- 403 Compiler stack size insufficient
- 404 Compiler workfile contents invalid
- 405 Compiler workspace insufficient
- 406 Disk/DOS error during compilation

- 410 constant truncated due to overflow
- 416 constant division by zero
- 417 constant remainder implies division by zero
- 418 integer constant expression expected
- 504 declaration appears after statement
- 505 empty declarator
- 506 local object too large
- 507 global object too large
- 508 local data size exceeds implementation limit
- 509 object may not have type 'void'
- 510 'register' has no effect on 'struct'/'union'
- 514 type specifiers may not be omitted in 'typedef' declaration
- 515 type specifiers missing - assumed 'int'
- 516 illegal combination of type/storage class specifiers
- 524 'enum' used before defined
- 525 redeclaration has different linkage
- 526 declaration has different type to previous (out of scope) declaration
- 527 redeclaration has different type
- 528 declaration has different type to previous extern declaration
- 529 duplicate defining occurrence
- 534 'struct'/'union' used before defined
- 535 bit field base type not 'int', assumed 'int'
- 536 'struct'/'union' has zero size
- 537 field may not be 'void', function or incomplete array type
- 539 storage class specifier illegal on field
- 539 duplicate field name in 'struct'/'union'

544 bit field width illegal, assumed 'int' width
545 named bit field may not have zero width
546 enumeration constant too large
548 function header missing (semicolon after
function header?)
549 parameter declarations illegal except after
identifier list
554 storage class specifier illegal on parameter
variable
555 identifier list illegal except in function
header
556 illegal function return type
557 identifier missing in parameter declaration
558 parameter may not be 'void'
559 variable in parameter declaration was not in
parameter list
564 'void' parameter list has extra parameters
565 pointer initializer is non-NULL integer
566 only 'const' or 'volatile' are legal in this
context
569 scalar initializer not compatible type
574 ptr-to-fn initializer is non-NULL integer
575 array of functions illegal, assumed array of
ptr-to-fn
576 array too large
577 illegal array size
578 only first dimension of array may be omitted
579 ptr-to-fn initializer not compatible type
580 'signed char' initializer truncated
581 'signed long' initializer truncated
582 'signed short' initializer truncated
584 too many initializers

585 initializer is not constant
586 constant pointer expression expected
587 constant ptr-to-fn expression expected
588 floating point constant expected
589 function cannot be initialized
590 'unsigned char' initializer truncated
591 'unsigned long' initializer truncated
592 'unsigned short' initializer truncated
594 'signed' bit field initializer truncated
595 'unsigned' bit field initializer truncated
596 illegal initialization
597 initializer is not aggregate
598 initializer is not scalar
599 pointer initializer not compatible type
600 label defined but not used in function
601 label is target of goto from outside
 initialized block
602 target of 'goto' is in nested initialized block
606 duplicate label declaration
608 label used but not defined in function
609 local 'extern' may not be initialized
610 expression statement has no side effect
620 control expression is an assignment
626 control expression is not a scalar
630 no cases in 'switch'
631 'switch' has constant control expression
634 duplicate 'case' label on same statement
635 'default' occurs more than once on same
 statement
636 duplicate 'case' label

637 'default' occurs more than once

638 'switch' control expression not integral

640 unreachable code

644 'return exp;' not found in function returning non-'void'

645 'return;' found in function returning non-'void'

646 'return exp;' found in function returning 'void'

654 function contains both 'return;' and 'return exp;'

655 'return' expression is not compatible with function result type

666 'continue' outside loop

667 'break' outside loop/'switch'

668 'case'/'default' outside 'switch'

704 'auto' illegal on global object

705 'register' illegal on global object

805 duplicate macro parameter

806 parameter list expected after function macro name

807 identifier expected in macro argument list

808 macro arguments not separated by commas

809 identifier missing in macro argument list

810 probable recursive include, include ignored.
A file has been included more than 4 times simultaneously

811 missing > after '#include <filename>

816 '#include' not followed by a filename

817 '#include' filename exceeds source line

818 could not find include file

819 could not open include file

820 unknown directive encountered while skipping

821 '#else'/'#elif' found in '#else' arm
824 '#else'/'#elif' not matched by '#if...'
825 '#endif' not matched by '#if...'
826 : not matched by ? in '#if' expression
827 ? not matched by : in '#if' expression
828 floating point illegal in '#if' expression
829 'defined' not followed by an identifier
834 attempt to redefine function macro as object
 macro
835 attempt to redefine macro with different
 arguments
836 attempt to redefine preprocessor keyword
 'defined'
844 attempt to redefine macro with different body
845 attempt to redefine object macro as function
 macro
846 '#define' not followed by an identifier
847 '#undef' not followed by an identifier
848 attempt to '#undef' a predefined macro
 The predefined macros beginning __ may not be undefined
849 #directive must be first token on line
850 null macro argument
851 too many arguments in macro call
854 attempt to redefine a predefined macro name
 The predefined macros beginning __ may not be redefined
856 missing argument(s) in macro call
866 '#line digit-sequence' not followed by
 "filename"
867 '#line' not followed by digit-sequence
876 illegal '#if' expression
877 illegal literal in '#if' expression
878 'sizeof' illegal in '#if' expression

879 illegal operator in '#if' expression
886 missing expression in '#if' expression
887 missing identifier in 'defined(identifier)'
888 missing) in 'defined(identifier)'
896 missing operand in '#if' expression
897 missing operator in '#if' expression
898 unmatched) or missing ? in '#if' expression
899 unknown directive
900 .. illegal, . assumed
901 source skipped to this point after error in top
level declaration
902 source skipped to this point after error in
parameter declaration
903 source skipped to this point after error in
field declaration
906 Prospero extension to C
When S option specified
907 syntax error: unexpected token
The offending token is printed
908 illegal token
The offending token is printed
910 mismatched }
911 error recovery inserted token
The inserted token is printed
912 discarding token
The discarded token is printed
913 source skipped to this point after error in
'case' label
917 unrecoverable syntax error
918 parser stack overflow
Expression too complicated
919 Compiler internal error
Should never occur - please contact Prospero

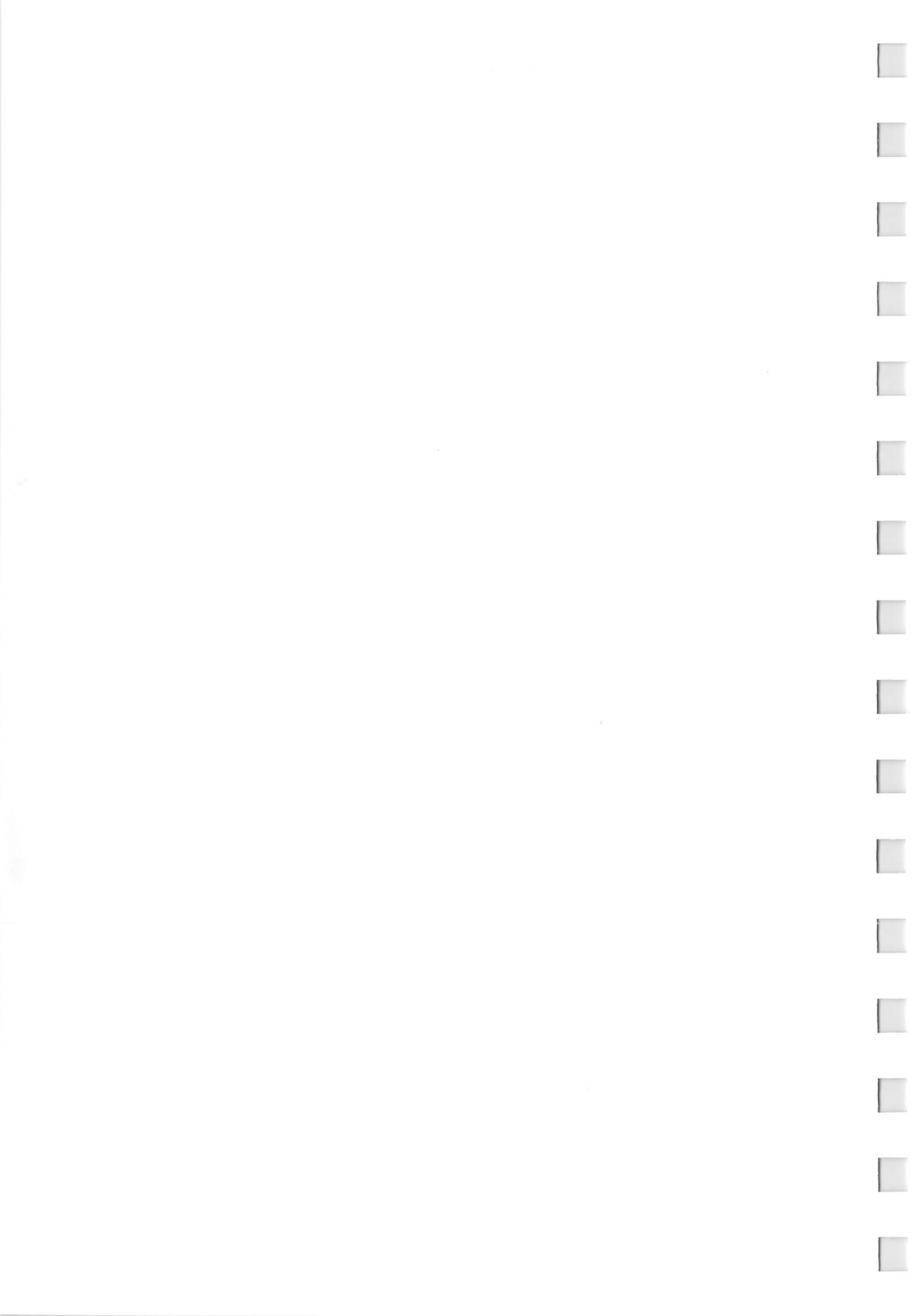
920 source skipped to this point after error in
expression

926 syntax error: expected

927 syntax error: unexpected right parenthesis

928 syntax error: missing comma

980-999 Compiler internal error
Should never occur - please contact Prospero



C RUN-TIME ERRORS

The format of the messages produced for run-time errors is given in Part I under “Operation of object programs”. This appendix lists the error codes, with significance and possible causes. Some errors can only occur if the relevant checking option was selected at compile time – see Part I, section 4.2.

Code Meaning

B Bounds exceeded.

An index bound has been exceeded (with option I selected) or a value is outside the range of the receiving field in an assignment (with option A selected).

C Switch error.

No case label corresponding to expression value (and no default label specified). Continuation is to the statement following the switch statement.

J Divide by zero (integers).

Continuation possible, but results not predictable.

K Overflow on floating to integer conversion.

Conversion of a floating point value to integer gives a value outside integer range.

O Overflow during signed integer arithmetic.

From 32-bit add or subtract (when option A or I invoked) or from 32-bit multiply (always checked). Continuation possible, truncated result used.

P Pointer not valid.

A pointer contains an invalid value. Attempting to “free” a memory block not allocated by malloc, or dereference a NULL pointer (with option P selected).

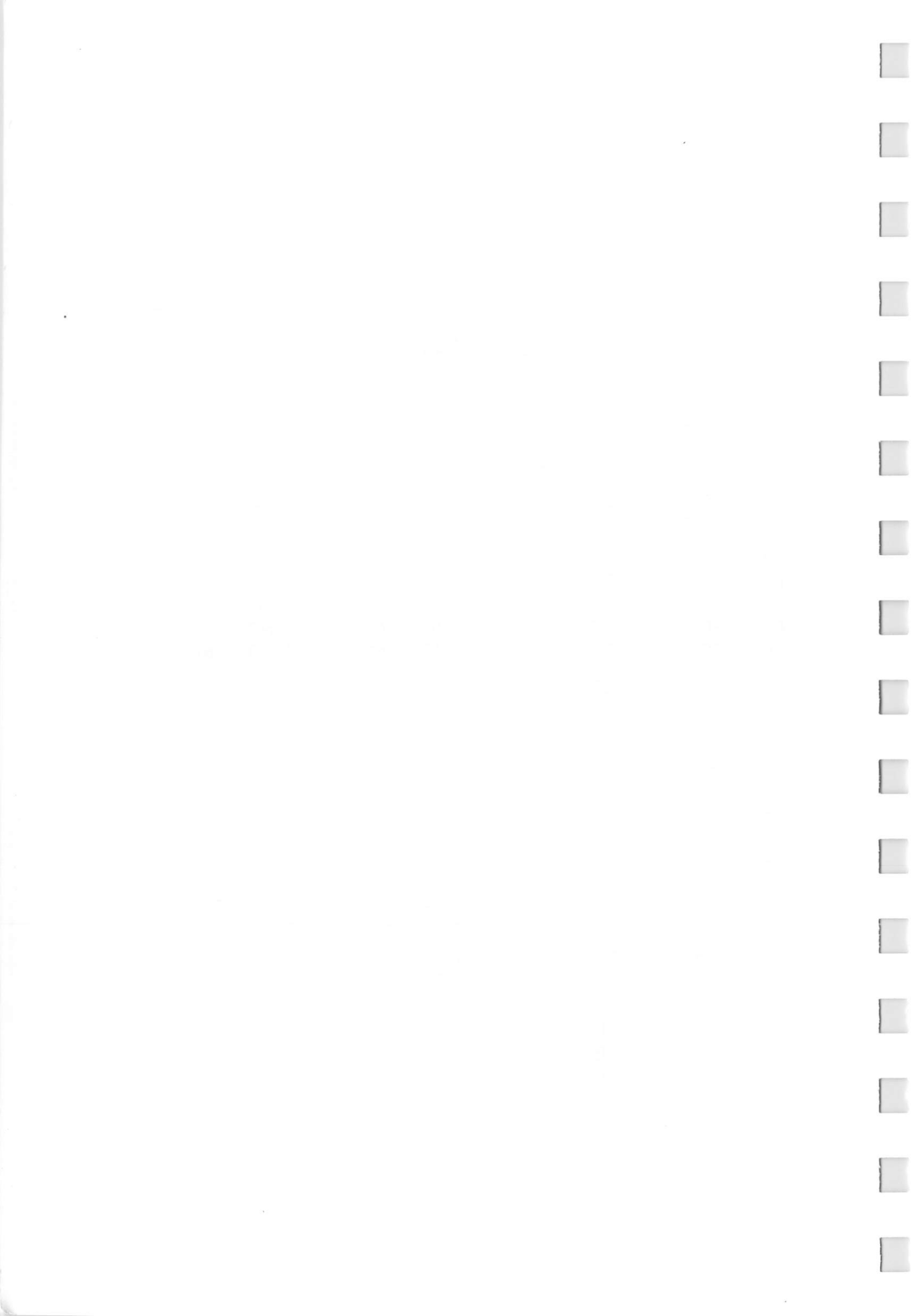
S Stack space insufficient.

The dynamic stack used for parameters and local variables of functions has exceeded the space available. (Only checked if option N specified.)

- X Overflow during floating point arithmetic.
Exponent out of range. Continuation possible but results not predictable.
- Y Missing function.
When linking, “Unsatisfied external” was reported (see Part I, section 5.3.1). At run-time, such an external has been referenced, by calling a non-existent function.
- Z Divide by zero (floating-point).
Continuation possible, but results not predictable.

D ASCII CHARACTER SET

Hex	Character	Hex	Character	Hex	Character	Hex	Character
00	NUL	20	space	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	\$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(48	H	68	h
09	HT	29)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[7B	{
1C	FS	3C	<	5C	\	7C	
1D	GS	3D	=	5D]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL



E MIXED LANGUAGE PROGRAMMING

Program construction

Mixed language programs may be constructed by amalgamating Prospero Fortran, Pascal and C components. One of the languages must supply the main program (or main function in the case of C), but can call other functions or procedures coded in any of the three languages.

Input/output may be performed in each language independently. There is, however, one exception to this: because standard input and output are initialized differently for the three languages, their use is restricted to the language of the main program. In the case of a Pascal or Fortran main program, this means that `stdin` and `stdout` should not be referred to in any C functions called (explicitly or by calling functions such as `printf` whose output is implicitly to `stdout`). In the case of a C main program, it means that the standard files `input` and `output` cannot be used in Pascal segments, unless they have been explicitly assigned to a file or device and `reset/rewrite` issued, and the standard unit `*` cannot be used in Fortran subprograms.

When a mixed language program terminates, either normally, or through a run-time error, all open files of the main language are closed. Pascal and Fortran will also cause each other's open files to be closed. However, C files will not be closed on termination of a Fortran or Pascal main program, and Pascal or Fortran files will not be closed on termination of a C main program.

After compilation by the appropriate compiler, the components are link-edited together (refer to Part I). In the linker command file, the `.BIN` files are listed in sequence (with the main program module typically coming first, after `FIRST.BIN`), followed by the names of the two libraries to be selectively scanned (`/S` option), with the library for the main program language coming first, followed by `LAST.BIN`. The file `FIRST.BIN` is the same for Pascal and Fortran, but not for C. If mixing C with Pascal or Fortran, the `FIRST.BIN` from Pascal or Fortran should be used. The file `LAST.BIN` is the same for all three languages.

Correspondence of data types

The table below shows the correspondence between Fortran, Pascal and C data types.

C	Fortran	Pascal
unsigned long int	no equivalent	no equivalent
long int	INTEGER	integer
int	INTEGER*2	integer2
unsigned int	no equivalent	word
signed char	INTEGER*1	integer1
unsigned char	no equivalent	byte
float	REAL	real
double	DOUBLE PRECISION	longreal
struct {	COMPLEX	RECORD
real re, im;		re, im: real;
}		END;

Pascal and Fortran COMMON variables cannot be accessed from within C, nor can C external objects (other than functions) be accessed from Pascal, except by passing their addresses as parameters.

Parameters

The name of a Pascal procedure at the outer level or a Fortran subroutine can be declared as **pascal** (or **fortran**) **extern void** functions within C and then referenced. Pascal and Fortran functions are similarly declared as functions returning non-void. The **pascal** or **fortran** type modifier indicates that the parameters are passed in the opposite order to that usually employed by C, that they are removed from the stack by the called (rather than the calling) function, and that function results are returned in accordance with the convention of the stated language. A C coded function declared with the **pascal** or **fortran** type modifier will also conform to these conventions, and can therefore be called from Pascal or Fortran.

A C function prototype with one of the type modifiers described above must be supplied for each non-C coded function, and the parameters must match in number, order, and type (see above) with those of the Pascal or Fortran code. All parameters passed to a Fortran coded function are passed by reference – a pointer to the relevant type should be employed.

Note that in the case of a CHARACTER argument, Prospero Fortran does not pass the address of the start of the data (as would be done in C for a `char` array), but the address of a 6-byte Character Variable Descriptor (CVD). This is structured like a C structure consisting of a 4-byte address field followed by a 2-byte length field.

A C function which has a function pointer argument can be called from Pascal or Fortran. Calling from Fortran presents no problem, but because Pascal passes two addresses in such cases (the entry address and the static link), the C prototype must include an extra dummy argument to match the latter. For example

```
Pascal  FUNCTION area (PROCEDURE calc): real; EXTERNAL;  
C       float area (long dummy, void (*calc) (void))
```

The Pascal procedure passed as an actual argument corresponding to `calc` must be declared at the outer level.

Interchange of files

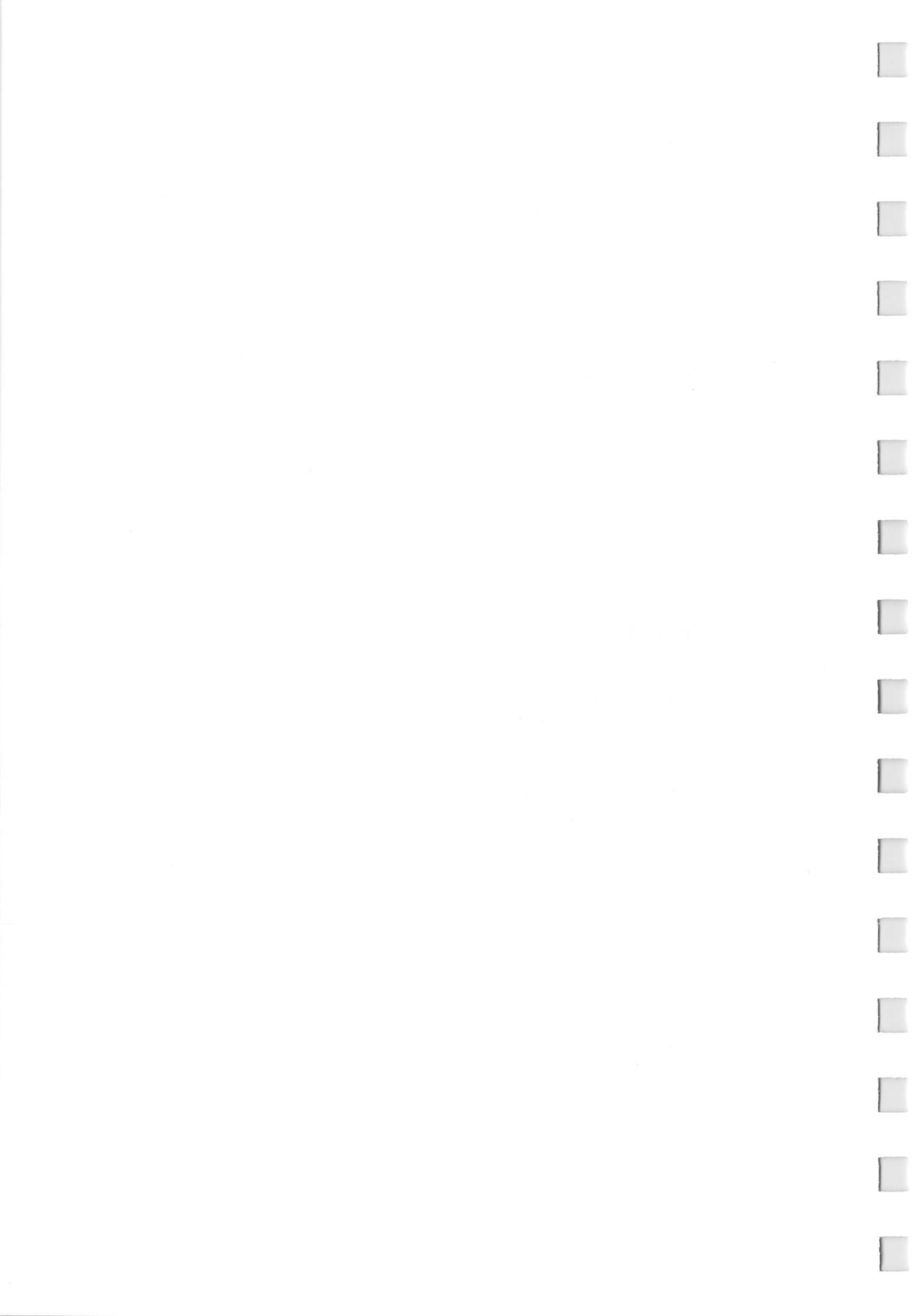
A Prospero Fortran file of variable-length formatted records and a Prospero Pascal file of type text are equivalent to a text mode file in Prospero C. (All, in fact, are normal GEMDOS text files.)

A Prospero Fortran file of fixed-length records, or a non-text file in Prospero Pascal, should be read or written in binary mode in C. The simplest way is to declare a structure whose layout corresponds to that of the Pascal file element type or the Fortran iolist and record length, then use `fread` or `fwrite` to transfer elements to or from an array of such structures in C. Thus for example a file written by a Pascal program as

```
FILE OF RECORD  
    item: integer;  
    vmax,vmin: real;  
END;
```

would be read in C by

```
typedef struct { long item;  
                float vmax, vmin;  
            } item;  
  
item buffer[N];  
  
s = fopen (filename, "rb");  
fread (buffer, sizeof (item), N, s);
```



F DEFINITION OF TERMS

- Alignment** A requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte or word address.
- Argument** An expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation. Also known as “actual argument” or “actual parameter”.
- Byte** The unit of data storage in the execution environment large enough to hold a single character in the character set of the execution environment. A byte is composed of a contiguous sequence of 8 bits. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. Except for bit-fields, objects are composed of contiguous sequences of one or more bytes.
- Bit** The unit of data storage in the execution environment large enough to hold an object that may have one of two values.
- Function** A body of executable code.
- Parameter** An object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition. Also known as “formal argument” or “formal parameter”.
- Implementation limits** Restrictions imposed upon programs by Prospero C .
- Null character** A character with all bits set to 0.
- Object** A region of data storage in the execution environment, the contents of which can represent values.
- Sequence point** A point in the source where all side effects of previous evaluations is complete and no side effects of subsequent evaluations has yet taken place.
- Side effect** The name given to the effect whereby a value is assigned to an object during the evaluation of an expression.

Undefined behavior

Behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately-valued objects, for which the C Standard imposes no requirements.

Unspecified behavior

Behavior, for a correct program construct and correct data, for which the C Standard imposes no requirements.

G IMPLEMENTATION LIMITS

This appendix contains the definitions of integral and floating point limits, as defined in the header files <limits.h> and <float.h>.

G.1 Integral limits <limits.h>

```
#define CHAR_BIT      8                /* No of bits in a char */
#define SCHAR_MIN    (-128)           /* Minimum signed char */
#define SCHAR_MAX    127              /* Maximum signed char */
#define UCHAR_MAX    255u             /* Maximum unsigned char */

#ifdef _UCHAR
#define CHAR_MIN     SCHAR_MIN        /* char is signed */
#define CHAR_MAX     SCHAR_MAX        /* Minimum plain char */
#define CHAR_MIN     SCHAR_MIN        /* Maximum plain char */
#else
#define CHAR_MIN     0                 /* char is unsigned */
#define CHAR_MAX     UCHAR_MAX        /* Minimum plain char */
#define CHAR_MIN     UCHAR_MAX        /* Maximum plain char */
#endif

#define SHRT_MIN     (-32768)          /* Minimum short */
#define SHRT_MAX     32767             /* Maximum short */
#define USHRT_MAX    65535u           /* Maximum unsigned short */
#define INT_MIN      (-32768)         /* Minimum int */
#define INT_MAX      32767            /* Maximum int */
#define UINT_MAX     65535u           /* Maximum unsigned int */
#define LONG_MIN     (-2147483648)    /* Minimum long */
#define LONG_MAX     2147483647       /* Maximum long */
#define ULONG_MAX    4294967295u     /* Maximum unsigned long */
```

G.2 Floating-point limits <float.h>

Macros whose names start with **FLT_** refer to the limits of type **float**, while those beginning **DBL_** and **LDBL_** refer to the limits of types **double** and **long double** respectively. In Prospero C, **double** and **long double** share the same representation, and therefore the same limits.

```
/* Radix of exponent */
#define FLT_RADIX    2

/* Rounding mode (to nearest) */
#define FLT_ROUNDS   1

/* number of binary digits in mantissa */
#define FLT_MANT_DIG 24
#define DBL_MANT_DIG 53
#define LDBL_MANT_DIG 53
```

```
/* Minimum  $x$  such that  $1.0+x > 1.0$  */
#define FLT_EPSILON      1.19209290E-07F
#define DBL_EPSILON      2.2204460492503131E-16
#define LDBL_EPSILON     2.2204460492503131E-16

/* number of decimal digits in mantissa */
#define FLT_DIG           6
#define DBL_DIG           15
#define LDBL_DIG         15

/* minimum  $x$  such that  $2^{x-1}$  is normalized */
#define FLT_MIN_EXP      -125
#define DBL_MIN_EXP      -1021
#define LDBL_MIN_EXP     -1021

/* minimum normalized positive number */
#define FLT_MIN          1.17549435E-38F
#define DBL_MIN          2.225073858507201E-308
#define LDBL_MIN         2.225073858507201E-308

/* minimum  $x$  such that  $10^x$  is normalized */
#define FLT_MIN_10_EXP   -37
#define DBL_MIN_10_EXP   -307
#define LDBL_MIN_10_EXP -307

/* maximum  $x$  such that  $2^{x-1}$  is representable */
#define FLT_MAX_EXP      128
#define DBL_MAX_EXP      1024
#define LDBL_MAX_EXP     1024

/* maximum representable floating point number */
#define FLT_MAX          3.40282347E+38F
#define DBL_MAX          1.797693134862316E+308
#define LDBL_MAX         1.797693134862316E+308

/* maximum  $x$  such that  $10^x$  is representable */
#define FLT_MAX_10_EXP   +38
#define DBL_MAX_10_EXP   +308
#define LDBL_MAX_10_EXP +308
```

H IMPLEMENTATION SPECIFIC BEHAVIOR

This appendix brings together a number of details of the behavior of Prospero C in areas where different implementations of standard C may differ, while still conforming to the ANSI Standard. Anyone wanting to write programs which will work on a number of C implementations and environments should not make assumptions about the behavior in any of the instances defined below.

H.1 Implementation defined behavior

This covers areas where the standard recognizes that different implementations will behave in different ways, for example to reflect the most efficient operations on a particular microprocessor or operating system, but states that each implementation must document the way in which they perform.

Environment

Arguments to `main`:

The command tail given when a program is invoked from the GEM Desktop, the Workbench, or any other program, is parsed to divide it into a number of strings delimited by whitespace, and pointers to these strings (the program arguments) are placed in the `argv` array passed to the `main` function. Whitespace can be included in a program argument by enclosing it in double quotes. To include a double quote in a program argument, it should be preceded by a backslash character. Any backslash character not followed by a double quote is included in the argument unchanged.

The value of `argv[0]` is a pointer to an empty string, as the program name is not made available under GEMDOS. The program arguments may be in mixed case. The parameters `argv` and `argc`, and the program arguments pointed to by `argv` may all be modified by the program.

Interactive devices:

There is no way to determine under GEMDOS whether a handle refers to an interactive device or to a file. All files opened by the program are therefore assumed to refer to disk files, and are fully buffered unless otherwise specified by `setvbuf`. Standard output is always line buffered. Standard input is line buffered if its size returned by GEMDOS is zero. In this case, a newline is echoed to the console after every line of input. If the file size is not zero, standard input is assumed to have been redirected, and is fully buffered.

Identifiers

All characters of an internal identifier are significant, no length restriction.

The linker will ignore all characters after the 32nd in external identifiers, and does not treat case as significant.

Characters

There are no extra characters in the source set.

The ASCII character set is assumed for the execution character set.

Two chars in a `short` or `int`, four in a `long`. Order is most significant byte first.

There are eight bits in `char`, in the order 76543210.

The source character set is assumed to be ASCII (i.e., a subset of the execution character set).

Character constants containing characters not in the execution character set are passed as-is.

The value of a multi-character constant is the `int` constructed from the sequence of bytes in the character constant. Character constants are defined to have the most significant byte first; the rightmost character in the constant being the least significant byte of the integral value. e.g., `'\1\0'` will have the value 256.

The type `char` is compile-time configurable as `signed` or `unsigned`.

Integers

Integers are two's complement, and plain `int` is 16 bits long.

Bitwise operations on `signed` integers are supported.

The sign of the remainder on integer division is the same as that of the quotient, and the value such that $((a/b) * b) + (a\%b) == a$.

Right shift of `signed` is arithmetic (sign extends).

Floating point

Floating point values are represented in the IEEE standard format – see Part II.

Converting floating-point to integer will truncate towards zero.

Converting a floating-point value to a narrower floating type will round to nearest.

Arrays and pointers

There is no limit (other than machine memory) on the size of an array.

The type of `size_t` is `unsigned long int`.

When casting a pointer to an integer, the receiving integer should be `long`, or the value will be truncated. Casting an integer to a pointer will behave as if the integer were first converted to `unsigned long`.

A pointer to function is identical in size, structure, and properties to a “normal” pointer.

`ptrdiff_t` is equivalent to type `signed long`.

Registers

Register allocation is performed by the code generator. Explicit uses of `register` therefore have no effect.

Structures, unions, and bit-fields

If a member of a union is accessed using a member of a different type, the result is undefined.

Each element of a struct will be aligned on a word boundary if its size is greater than 1 byte, with padding if necessary, otherwise on a byte boundary. A padding byte will be placed at the end of the structure if the size of the structure is greater than one byte.

A plain `int` bit-field is treated as `signed`.

Bit-fields never cross an `int` boundary

The first bit field encountered is least significant. e.g.,

```
int aa : 2;
int bbb : 3;
int : 7;
int c : 1;
int ddd : 3;
```

will pack as `dddcxxxxxxxxbbbaa` in a 16-bit `int`. Consecutive bit-fields that would all pack into one byte will only be allocated one byte (unless word alignment is enforced).

Bit-fields can straddle a storage-unit boundary (`char`).

Declarators

The maximum number of modifying declarators exceeds that in the C Standard (12) by a large margin.

Statements

There is no limit on the number of `case` values.

Preprocessing directives

Character constants will have their values calculated in the same manner as outside the preprocessor. See Part III, section 4.3.4 for more details.

Header file names in angle brackets will be searched for in the current directory, the directory specified by the Workbench path for include files, and all paths specified by the PATH environment variable.

Header file names in quotes should specify the path to be searched as part of the header file name. This will be interpreted relative to the current directory.

No `#pragma` directives are supported.

H.2 Unspecified behavior

The following represent behavior for correct programs and data that is not specified by the C Standard. Although it is not wise to rely on any particular behavior in any of these cases, especially when trying to write portable programs, the behavior of Prospero C is as described below.

Order of **static** initialization.

Block data information is written to the object code file. Initialization therefore conceptually takes place simultaneously, before program start-up.

Order of expression evaluation.

Depth first tree walk.

Evaluation of function designator and arguments.

Because of the order of arguments on the stack (see below), arguments are evaluated right to left, followed by the function designator.

For external functions declared with the **pascal** or **fortran** keyword, the arguments will be evaluated left to right.

Storage layout for formal parameters.

First argument at lowest address, last argument at highest address – this affects the order of evaluation (see above).

H.3 Undefined behavior

The following represent some aspects of behavior for incorrect programs or data where the C Standard imposes no requirements on the behavior. It is unwise to make assumptions about how a compiler will behave in the following circumstances, especially when trying to write portable programs. Prospero C behaves as described below.

Character encountered outside required character set.

The character becomes a separate preprocessing token. If, after preprocessing is complete, that token is still present, an 'Illegal token' error will be given.

An attempt is made to modify a string literal.

This will generally work, but is not recommended, since the implementation of string literals allows for future enhancements that would place them in protected, or read-only, memory areas. Note that identical string literals are not commoned up. Note also that extending a string literal will almost certainly overwrite code (rather than overwriting data).

Number of significant characters in identifiers.

All characters in an identifier are significant. External identifiers that differ only after 32 characters will cause an error at link time.

Unspecified escape sequences.

These are left unmodified, with a warning given.

Calling functions with too few arguments.

This will work, unless the missing arguments are accessed within the called function. Reading missing arguments will give undefined values; writing to them will overwrite the caller's local variables.

Calling function with too many arguments.

This will always work.

Type mismatch on parameters.

This will give undefined values when the parameters are referenced (and may overwrite the caller's local data).

Invalid memory references.

Accesses to certain memory areas are trapped by hardware. Others will return an undefined value.

Pointer to function cast to different function type.

Whether or not this works depends on what the different function types return (a function assumes that the result variable is large enough to hold its result value).

Arithmetic on pointers outside array objects.

This will behave as expected (as if all memory were a suitable array).

Subtracting pointers not pointing to the same array object.

This will behave as if all memory were a suitable array.

Comparing pointers to different aggregates.

This will not cause an error (pointers are compared by address value).

Assigning overlapping objects.

This will generally work only for objects of size 4 bytes and smaller.

Modifying **const** object via non-**const** pointer.

This will not be detected as an error. Similarly for **volatile**.

Type clashes on external identifiers.

These will not be detected. The compiler may be able to give a warning if the clash is within one source file.

Uninitialized automatic storage.

This has undefined contents.

Using a function result when no value is returned.

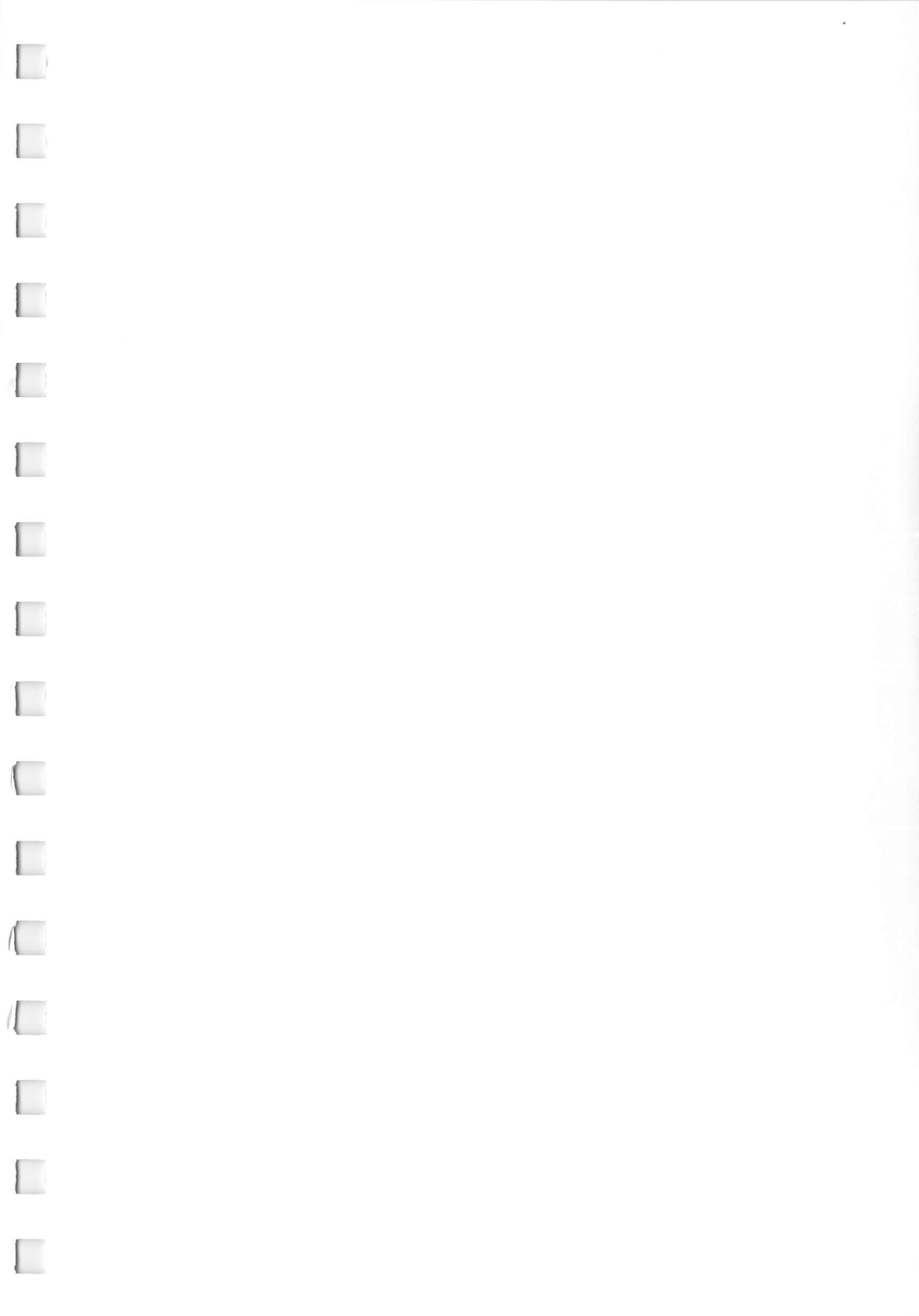
This yields an undefined value.

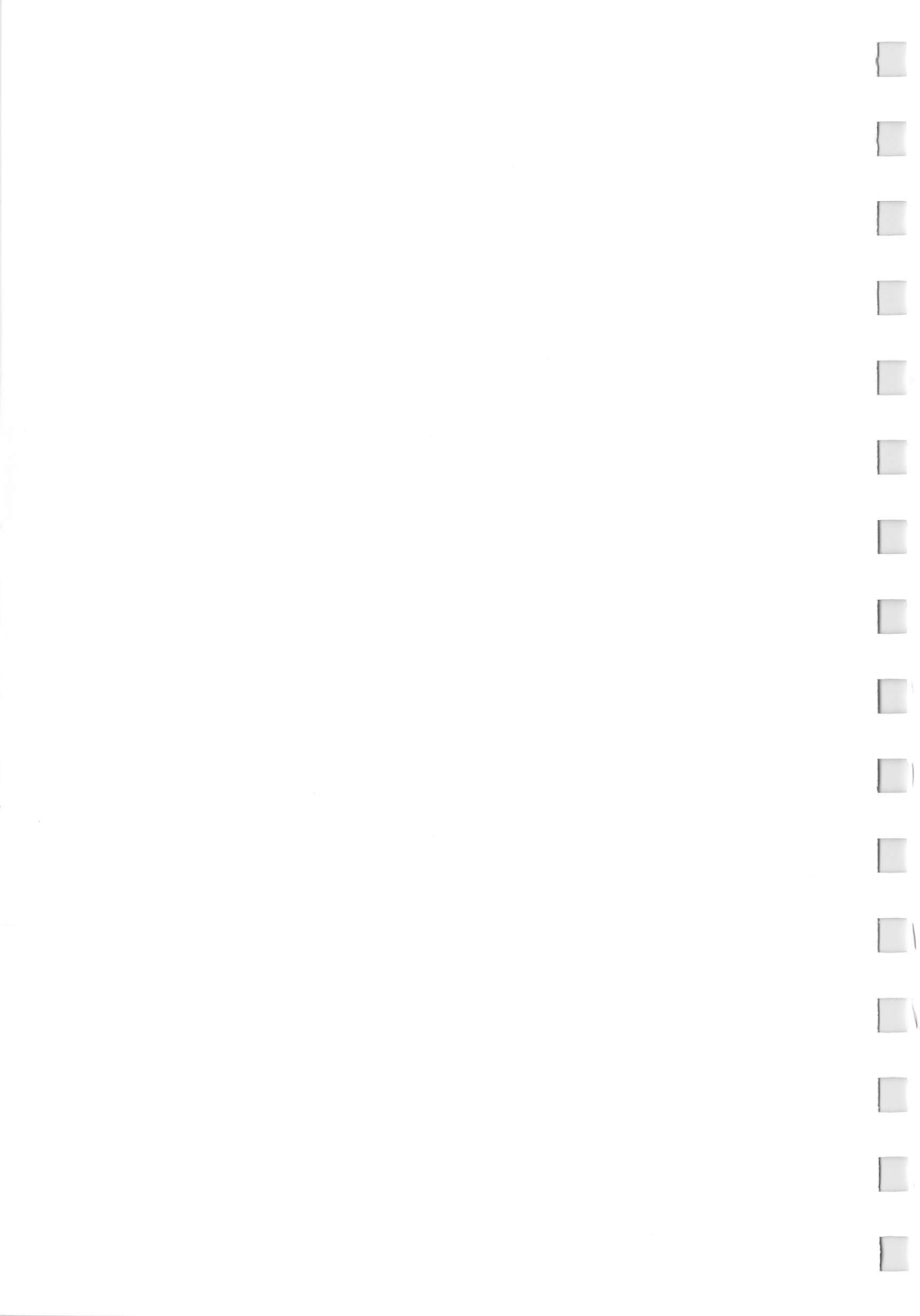
A macro argument consisting of no tokens.

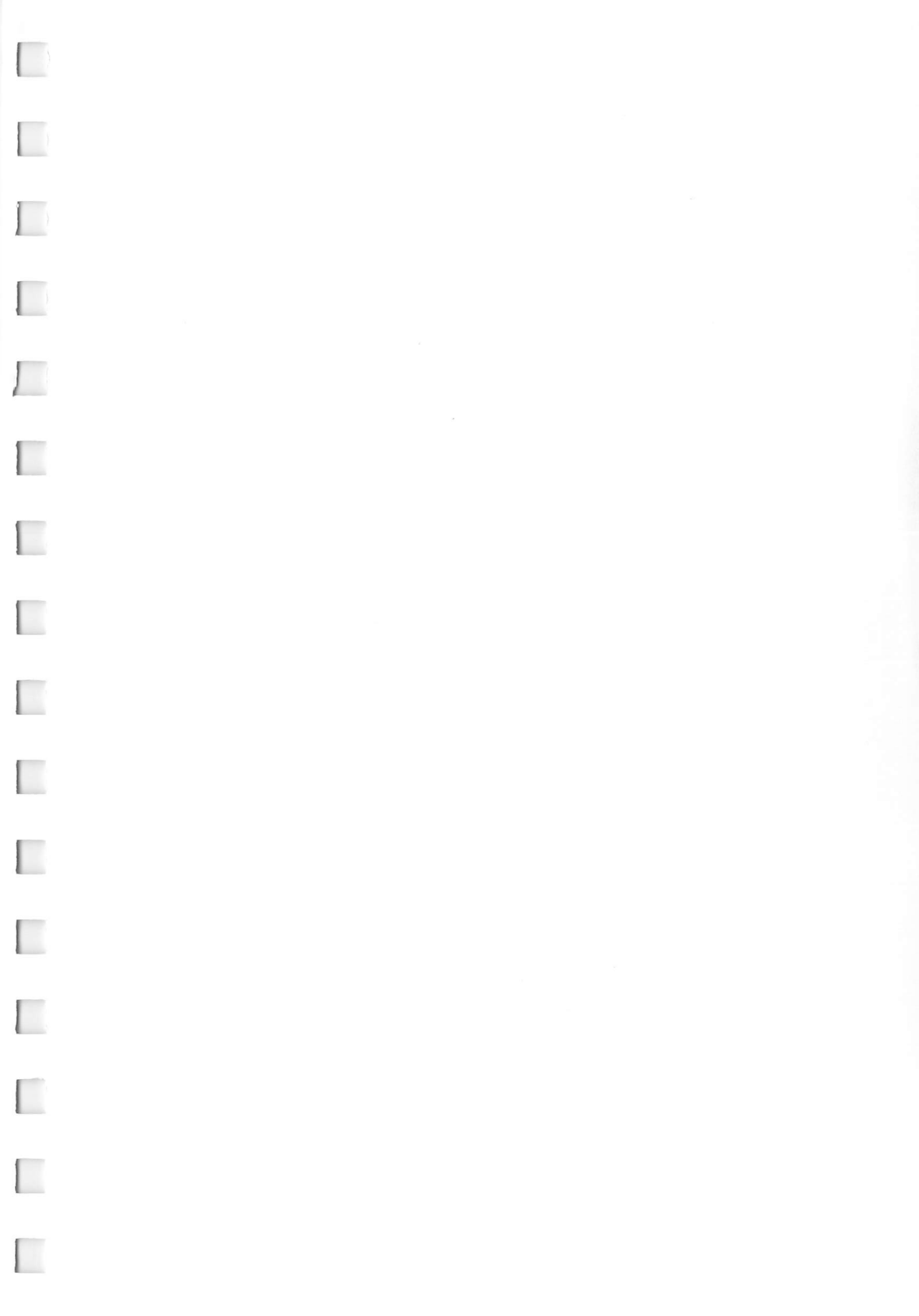
This will substitute no tokens into the macro body. A warning will be given.

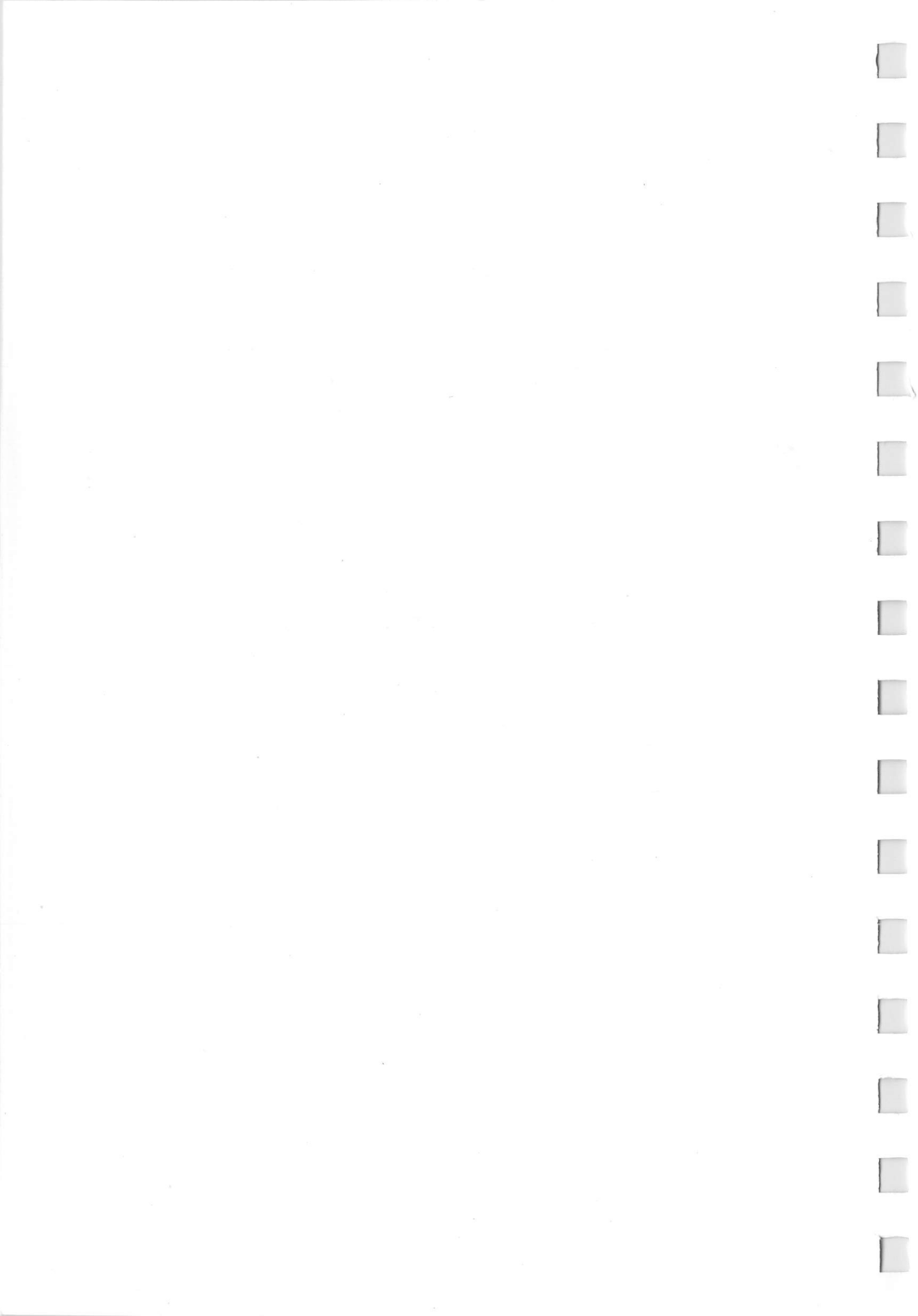
Macro arguments are not parsed for preprocessing directives.

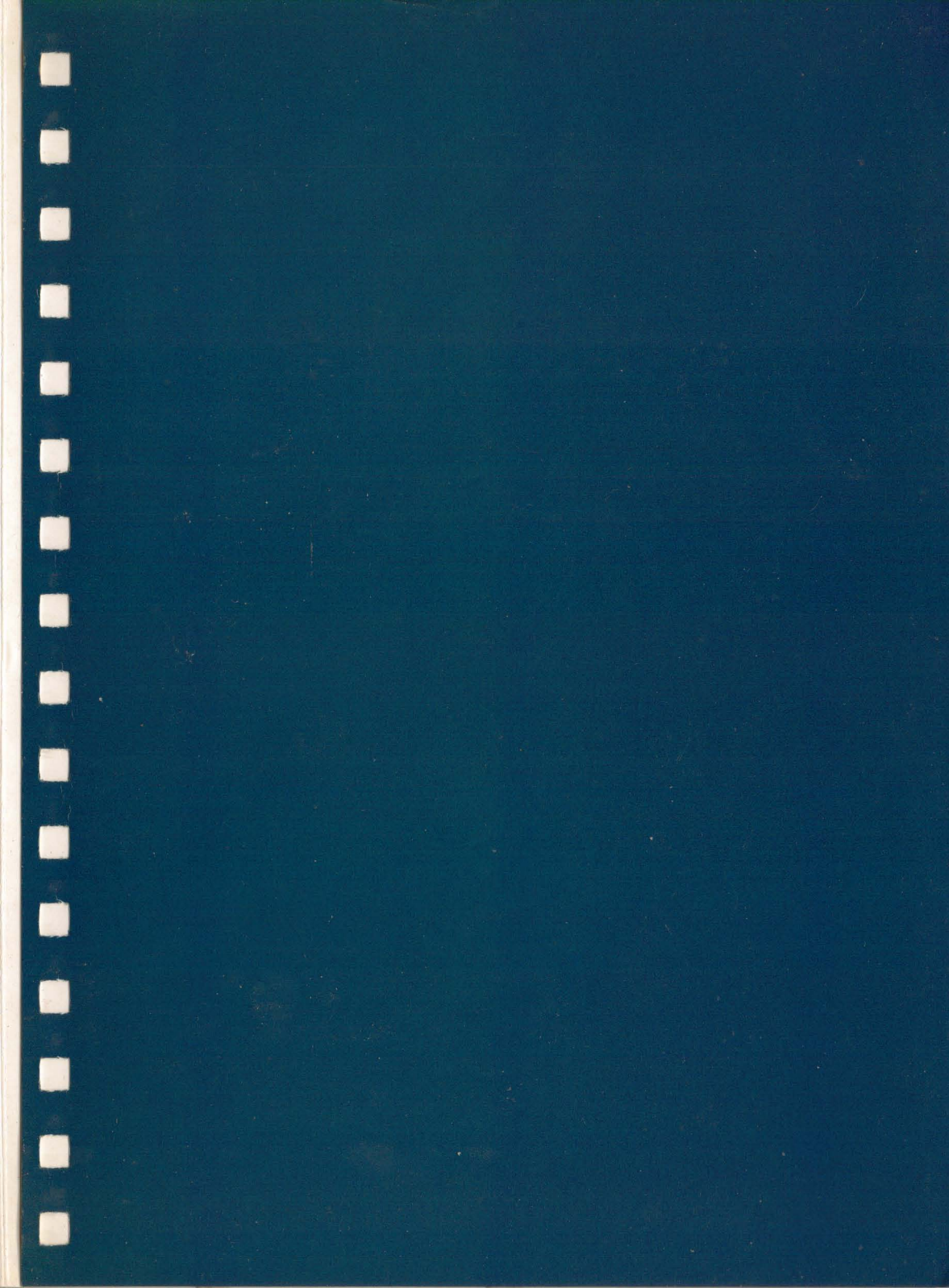
If the result of `##` is not a valid token, an error message 'Illegal token' will be given.











Prospero Software
LANGUAGES FOR MICROCOMPUTER PROFESSIONALS

Prospero Ciani
L'Invasione