ELECTRONICS WORLD
+ WIRELESS WORLD

Interfacing with C

Howard Hutchings

TP312
/76

Interfacing with C

Howard Hutchings

江苏工业学院图书馆
藏书章

BUTTERWORTH
HEINEMANN

Butterworth-Heinemann Ltd
Linacre House, Jordan Hill, Oxford OX2 8DP

A member of the Reed Elsevier plc group

OXFORD LONDON BOSTON
MUNICH NEW DELHI SINGAPORE SYDNEY
TOKYO TORONTO WELLINGTON

First published 1995

© *Electronics World + Wireless World* 1995

All rights reserved. No part of this publication may be reproduced in any material form (including photocopying or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication) without the written permission of the copyright holder except in accordance with the provisions of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's written permission to reproduce any part of this publication should be addressed to the publishers

British Library Cataloguing in Publication Data
Hutchings, Howard
Interfacing with C
I. Title
004.62

ISBN 0 7506 2228 8

Library of Congress Cataloguing in Publication Data
Hutchings, Howard.
Interfacing with C / Howard Hutchings.
p. cm.
Includes bibliographical references and index.
ISBN 0 7506 2228 8 (pb)
1. C (Computer program language) I. Title.
QA76.73.C15H88 1995
005.7' 1262—dc20 94-44832
CIP

Typeset by Vision Typesetting, Manchester
Printed and bound in Great Britain by
Biddles Ltd, Guildford and King's Lynn

Preface

This book is about interfacing a personal computer using C, currently the language of choice for many real-time applications. Anyone who is interested in ports, transducer interfacing, analogue-to-digital conversion, convolution, digital filters, Fourier transforms, Kalman filtering or digital-to-analogue conversion will benefit from reading the book or selectively dipping into its pages. My intention is to provide a practical introduction to real-time programming with a generous collection of tried and tested programs. In most cases, the principles precede the applications in an attempt to provide genuine understanding and encourage further development.

Rather than design and build interface circuits, I chose to use the Blue Chip Technology data acquisition and control cards, which are port mapped and may be driven by any language – simplifying the task of interfacing. I recommend readers to buy good quality purpose-built hardware for this application; there are numerous manufacturers. Remember, the I/O card is where your computer and the vendor's board meet the outside world. Features to look for usually include a multi-channel A-to-D converter with one or more D-to-A converters and probably a collection of ports.

The pace of the book is gentle and intended to be user-friendly, the text encouraging you to run the programs and experiment with C. This form of motivation works; engineering students gain much from hands-on experience and are able to climb steep learning curves with the minimum amount of assistance. Inevitably, some of the programs become rather lengthy and, wherever possible, the fundamental construction is presented separately, an accompanying set of source code being available on disk. Most programs exist to be rewritten and if, after working through the examples, you cannot do better, I'll be disappointed. Mathematics is always a problem, but engineering mathematics is usually just a shorthand for the physics; successful signal processing using a PC will only come after genuine understanding of the reality behind the abstraction.

Throughout, the emphasis is on applications, all the programs in this book having been tried and tested on a Compaq 386, compiled using Microsoft C version 5.1. No single book can possibly raise all the questions or supply all the answers; at best, it should provoke further thought and indicate suggested study paths. For this reason, I have scattered references liberally throughout the text to assist the serious reader.

Above all, interfacing and programming with C is fun. C allows you to write efficient and readable code which gets very close to the target machine. Try your hand at the examples and enjoy making your computer solve the problems without picking up a soldering iron.

This book could never have been completed without the cooperation of many people. I am particularly indebted to the technical staff at Humberside University, who built many of the practical circuits, and to those students whose curiosity forced me to write with clarity so that I might explain what was possible. It's always a pleasure to thank the editorial staff at *Electronics World + Wireless World*, who accepted my eleventh hour fixes with tolerance and good humour. Finally my special thanks go to my wife, Margaret – a constant source of encouragement, who accepted with equanimity the separation brought about by this project.

Howard Hutchings

Listings in this book are available on disk from the editorial offices of *Electronics World + Wireless World*, Tel. 0181 652 3614.

此为试读，需要完整PDF请访问： www.ertongbook.co

Contents

1 An introduction to C

C is a medium-level programming language developed by Dennis Ritchie of Bell Laboratories and implemented there on a PDP-11 in 1972. Historically, C was preceded by B, a language written by Ken Thompson in 1970 for the first UNIX system run on the PDP-7. B, in turn evolved from the Basic Cambridge Programming Language BCPL, developed by Martin Richards at Cambridge in 1967 as a systems programming language.

The *C Programming Language* (1978), by Kernighan and Ritchie is the definitive text. Although it is not adopted as an international standard, it is generally accepted as standard C. This original and enigmatic text is not an introductory programming manual; it assumes familiarity with basic programming concepts such as variables, assignment statements, loops and functions – and is probably best read once you have mastered C. The increasing popularity of the language has encouraged numerous less esoteric works, many attempting to simplify the original Kernighan and Ritchie text. Each of these introductions has its relative merits; no doubt you will make your own choice and find what suits you best. I have included a short bibliography of texts which I found particularly useful.

Properties and background

The versatility of C allows it to be run on personal 8-bit computers or the Cray-1, one of the worlds fastest computers. Designed to make programs fast and compact, this portable assembly language was used to program the remarkable computer-animated sequences in *Return of the Jedi* and *Star Trek II*. In many cases programs written in assembly language for 'efficiency' have been outperformed by comparable programs written in C. Despite being a medium-level language it still embodies advanced structural programming features normally associated with high-level languages such as Pascal. C is a concise language and small can be beautiful when programming. It has a particularly rich set of operators, ideal for configuring programmable input–output devices and flag testing.

The purpose of this book is to teach those aspects of the C language you will require to interface effectively. Our strategy is to teach C program constructions as we go along, presenting the information in 'byte' sized packets in an attempt to make it more attractive and digestible. We have tried to organize the programs in a progression of complexity, so that each program presents a new feature of C or an alternative program construction. Where possible the construction is illustrated with a flowchart and the program liberally littered with comments to aid comprehension.

All the program examples presented have been tried and tested on an IBM PC clone using a Microsoft C compiler version 5.1. The emphasis is on effective interfacing rather than elegant programming. Where possible I have included alternative program constructions in an attempt to demonstrate the flexibility of this remarkable language. The text encourages you to run the programs and experiment with C. Inevitably some of the programs become lengthy, which tends to discourage even experienced programmers! To maintain interest the fundamental construction is presented separately. Most programs exist to be rewritten; and if after working through the examples you cannot do better, I'll be disappointed.

Rather than design and build our own interface circuits we chose to use the Blue Chip Technology data acquisition and control cards. These plug-in cards are port mapped and may be driven by any language, simplifying the task of interfacing – allowing us to concentrate more effort and attention on the programming aspect of the problem.

Fundamental interfacing

The primitive concept of sending bit patterns to the outside world can produce remarkably sophisticated electronic projects with the minimum of hardware, principally because much of the problem is solved using creative software. Imagine an Exocet missile skimming low over the waves as it homes in on its target. On board, the computer receives data from the missile's transducers through the input port. The data is processed in real time and the result used to control the trajectory in anticipation of a successful strike. Despite the complexity of the task the fundamental problem can be reduced to that of reading ones and zeros from a peripheral connected to an input port – processing the data and finally writing the manipulated data to the outside world through an output port.

Here's the catch: how do you find the available I/O space; format the control word; control the I/O card; process the data? Unfortunately, unless you are an experienced assembly language programmer these objectives represent formidable assignments. Instead of aiming for 'the best possible design', we will be content with the 'best design possible' and use C to get very close to the target machine.

Programmable input–output devices

Communication between the real world and a personal computer is through the ports of the peripheral interface adapter (PIA) or versatile interface adapter (VIA). These relatively complex and specialized chips can be programmed to behave as input–output devices and effectively buffer the data bus from the controlled peripheral, thereby protecting the system. Employing memory mapped input–output ensures that the CPU 'sees' the ports simply as a collection of addresses, indistinguishable from any address in memory. Provided the input–output device is configured carefully, data transfer can be made almost routine. In effect the operation of these devices are analogous to constructing electronic circuits without ever using a soldering iron, simply because the necessary connections are made by placing the required bit patterns in the appropriate control registers.

Unfortunately each microprocessor manufacturer appears to have adopted a particular programmable input–output device to suit their own system. As with microprocessor instruction sets, familiarity with a particular device tends to make the user patriotic and reluctant to change. Despite the unique features of many of these devices, certain characteristics remain common, making the transition from the comparatively primitive programmable peripheral interface PPI such as the Intel 8255 to the complex and complicated MOS Technologies 6522 VIA relatively painless.

To explain the adopted interfacing protocols with any clarity it was necessary to be chip specific. Unfortunately this dates the text, although the fundamental concepts will remain current for some time to come.

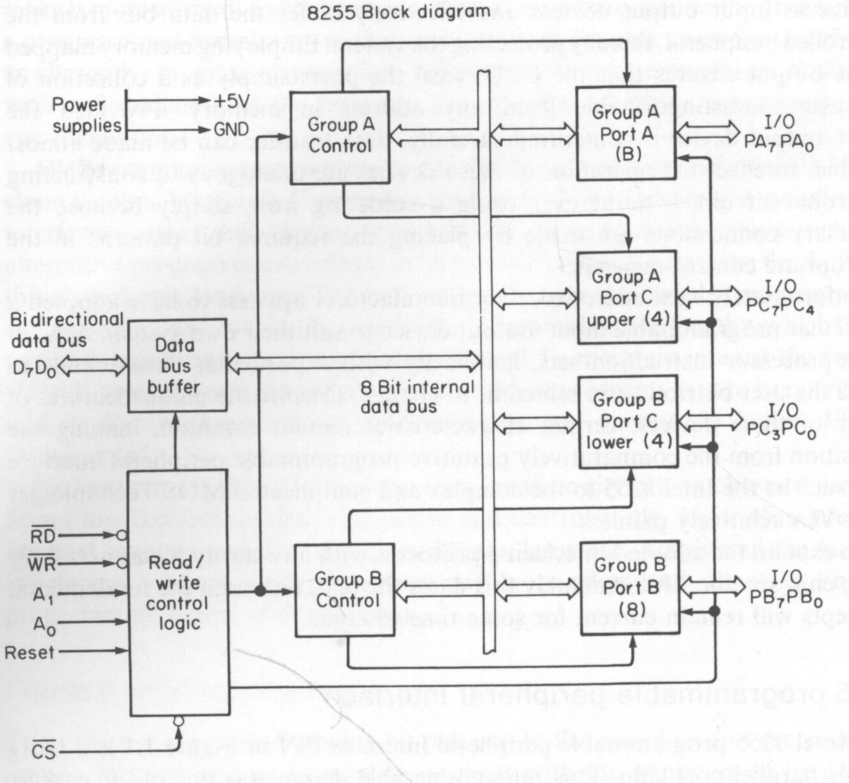
8255 programmable peripheral interface

The Intel 8255 programmable peripheral interface PPI in Figure 1.1 is a fairly simple parallel port chip. This rather venerable design was one of the earliest interface adapters on the market, originally intended for use in the 8008 and 8080A systems, but now enjoying a resurgence of popularity on account of the ease with which it interfaces to the IBM PC bus, which is effectively the bus for the 8088 processor operated in the 'maximum mode'. Large-scale integration ensures that parallel input–output operation can be concentrated into a single 40-pin package. Making the chip software configurable offers the flexibility of deferred design – the values placed in the control register determine which groups of lines are inputs and outputs.

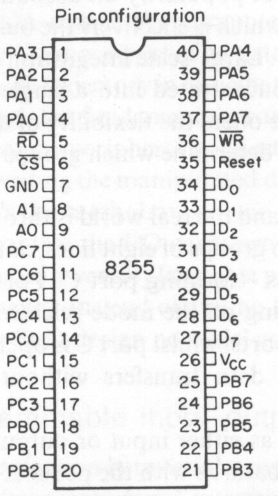
Communication between the microcomputer and the real world is through the 24 input–output lines. These are divided into two groups of eight lines, data ports A and B, together with two groups of four lines – forming port C. Port C can either be a data port or a control port depending on the mode selected.

In mode 0 ports A and B operate as two 8-bit ports, whilst port C is operated as two 4-bit ports. This mode supports simple data transfers without handshaking.

In mode 1 ports A and B may be configured as either input or output. They cannot be defined individually on a line by line basis as with the ports of certain other programmable I/O devices. Six bits of port C are set aside for handshaking and interrupt control.



(a)



(b)

Pin names

D_7-D_0	Data bus (Bi-directional)
Reset	Reset input
$\overline{CS}$	Chip select
$\overline{RD}$	Read input
$\overline{WR}$	Write input
A0, A1	Port address
PA7-PA0	Port A (bit)
PB7-PB0	Port B (bit)
PC7-PC0	Port C (bit)
V_{cc}	+5 Volts
GND	0 Volts

(c)

Figure 1.1 (a) Intel 8255 programmable port used as a programming model in the early stages of this series. (b) Pin configuration. (c) Pin names

Mode 2 uses the eight lines of port A for bi-directional data transfer. Handshaking is provided by the five most significant bits of port C.

Programming the 8255

The programming model of the 8255 consists of four 8-bit registers, ports A, B and C and a control register. Depending upon where you locate the device in the available I/O space, the register model appears as four contiguous addresses as shown in Figure 1.2.

The operation of the I/O ports is controlled by the format of the 8-bit word written to the control register, located at address (Base + 3). The control word format is shown in Figure 1.3. Simple input–output operations, without handshaking, require the control word to be configured in mode 0. Table 1.1 represents the mode 0 – port definition chart; which should be an effective source of reference when consulting the example programs.

The Blue Chip data acquisition system, used here for the purpose of description, provides 48 I/O lines by port mapping two 8255s on the same plug-in card. The ports are terminated in a single 50 way connector at the rear of the IBM PC. Bus contention is avoided by making the base address selectable in the range 300H to 3FFH, the prototyping region.

IBM PC bus

As shown in Figure 1.4 the PC-XT bus is an 8-bit data bus implemented in a 62-pin edge connector. Many of the bus signals are used for direct memory access and interrupt handling and may be ignored at first reading.

The address bus lines A0–A19 can address up to 1 Mbyte of address space. Although the 8088 processor can use all 16 lines A0–A15 to access 64 Kbyte of I/O space, only 10 lines A0–A9 are actually decoded restricting the number of available ports to 1024. Many of the available I/O locations have been adopted by IBM for their own purposes, these assigned locations being shown in Table 1.2. Despite the crowded nature of the I/O space there are several ports available, particularly in the prototype region 300H–31FH. However, certain peripheral

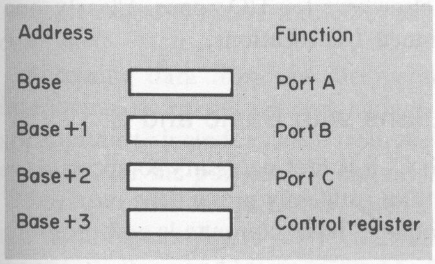


Figure 1.2 8255 programming model

此为试读，需要完整PDF请访问：www.ertongbook.co

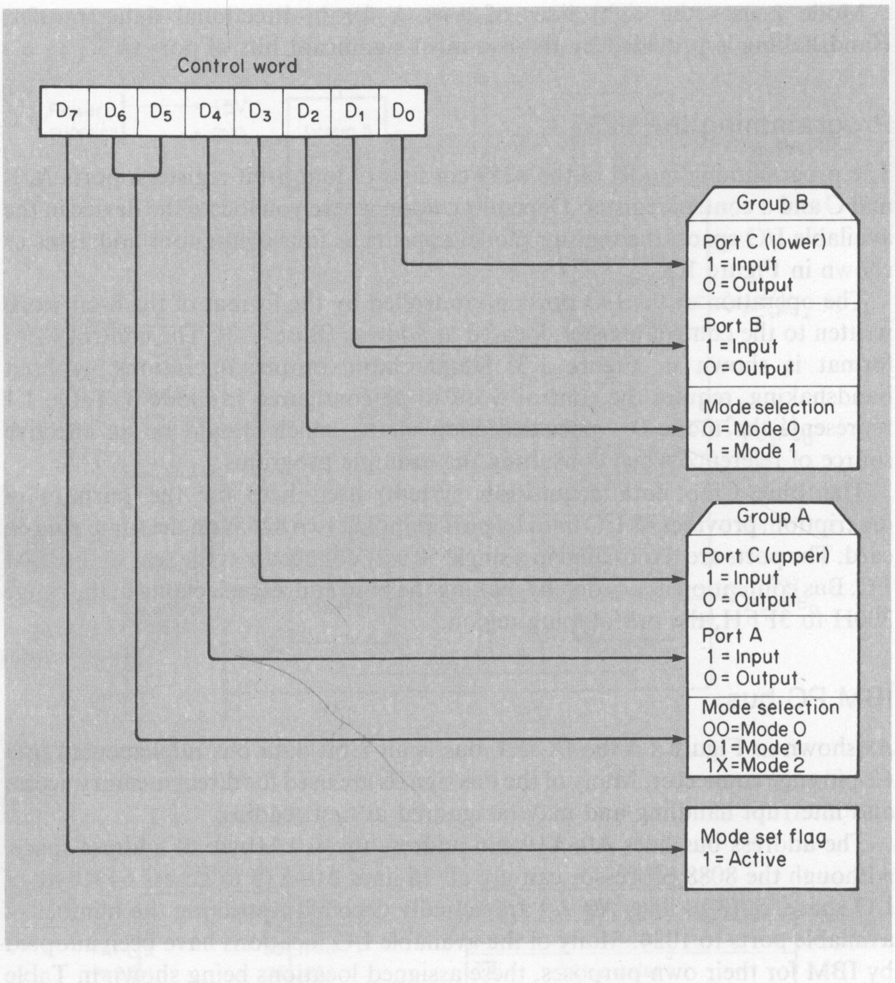


Figure 1.3 Control word bit function for 8255

board manufacturers have monopolized some of these addresses for their own products, which means you must look elsewhere for I/O space. Clearly one simple solution is to use unoccupied assigned I/O locations.

Accessing specific memory locations with Basic and C

Before you can interface successfully with C, it is first necessary to access data from specific memory addresses. Rather than ruthlessly present the required C constructions, we prefer to reiterate the familiar Basic commands and program structures for the purpose of comparison.

GW-Basic run on the IBM PC supports both memory-mapped I/O using Peek

Table 1.1 Mode 0 port definition chart. In this mode, simple input and output operations for each of the three ports are provided. No 'handshaking' is required; data is simply written to or read from a specified port

	Control word bits								Group A		Group B	
No.	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Port A	Port C (upper)	Port B	Port C (lower)
0	1	0	0	0	0	0	0	0	Output	Output	Output	Output
1	1	0	0	0	0	0	0	1	Output	Output	Output	Input
2	1	0	0	0	0	0	1	0	Output	Output	Input	Output
3	1	0	0	0	0	0	1	1	Output	Output	Input	Input
4	1	0	0	0	1	0	0	0	Output	Input	Output	Output
5	1	0	0	0	1	0	0	1	Output	Input	Output	Input
6	1	0	0	0	1	0	1	0	Output	Input	Input	Output
7	1	0	0	0	1	0	1	1	Output	Input	Input	Input
8	1	0	0	1	0	0	0	0	Input	Output	Output	Output
9	1	0	0	1	0	0	0	1	Input	Output	Output	Input
10	1	0	0	1	0	0	1	0	Input	Output	Input	Output
11	1	0	0	1	0	0	1	1	Input	Output	Input	Input
12	1	0	0	1	1	0	0	0	Input	Input	Output	Output
13	1	0	0	1	1	0	0	1	Input	Input	Output	Input
14	1	0	0	1	1	0	1	0	Input	Input	Input	Output
15	1	0	0	1	1	0	1	1	Input	Input	Input	Input

and Poke, together with port-mapped I/O using the Inp and Out commands. C provides a similar construction, the former requiring the use of pointers – which are tricky until you get used to them. However, pointers cannot be used to access I/O devices in computer systems such as the IBM PC which have a separate address space for I/O devices. IBM has allocated addresses in the range 768 to 799 (denary) specifically for I/O prototyping. See Table 1.2.

C compilers for machines with this type of I/O system usually include library functions, which allow direct access to the port-mapped I/O space. For example, the Microsoft C compiler version 5.1 provides the functions inp() and outp() defined in the header file conio.h. By incorporating the additional compiler directive #include<conio.h> these functions can be made part of the program as it is compiled.

Accessing data from specific memory locations is central to the task of interfacing, no matter which language is employed. For this reason we feel it is appropriate to include extracts from the GW-Basic and Microsoft C programmers manuals, for the purpose of comparison.

Believing that one picture is worth a thousand words we intentionally include the lighthearted Figure 1.5 as a reminder of how to access port-mapped data using GW-Basic and Microsoft C. These illustrative programs do not include any initialization protocols.

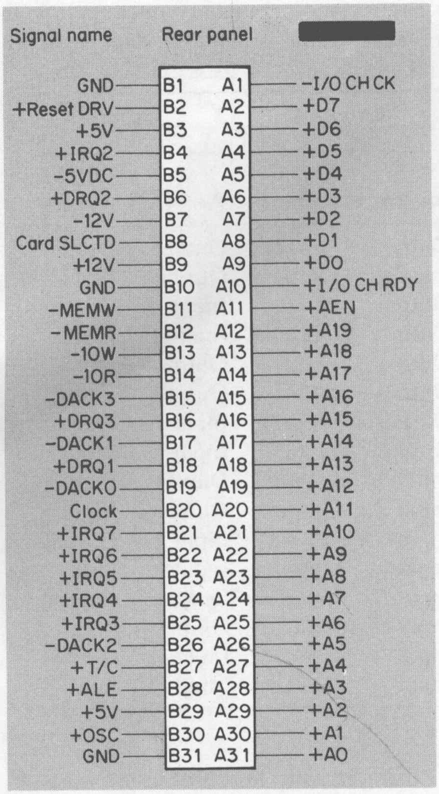


Figure 1.4 IBM PC bus structure

Reading the contents of I/O space using pointers

My initial objective was to demonstrate how to write a C program to read and display the contents of the I/O address space shown in Table 1.2. Two options were available: either I pedantically advertise the necessary C constructions before presenting the program; or I present the program and encourage you to run it and then demonstrate the appropriate constructions. I chose the latter approach in the belief that evidence of a successful program will provide a sense of direction and encourage you to read the subsequent text more critically. Beware, Listing 1.1 is not an elementary program, it contains many advanced features. Examine the fine detail after reading Chapter 1 – then improve upon it!

Run Listing 1.1 after first linking and compiling. The program responds by asking you to input the denary base address from the keyboard. Consult Table 1.2 for suitable values of I/O addresses. The result will be the contents of 16 contiguous addresses starting with the Base, displayed on the monitor. Now

Table 1.2 Address space for IBM I/O devices

Hex address range	Use
000–00F	DMA chip 8237A-5
020–021	Interrupt 8259A
040–043	Timer 8253-5
060–063	PPI 8255A-5
080–083	DMA page registers
0AX	NMI mask register
0CX	Reserved
0EX	Reserved
200–20F	Game control
210–217	Expansion unit
220–24F	Reserved
278–27F	Reserved
2F0–2F7	Reserved
2F8–2FF	Asynchronous communications (secondary)
300–31F	Prototype card
320–32F	Fixed disk
378–37F	Printer
380–38C*	SDLC communications
380–389*	Binary synchronous communications (secondary)
3A0–3A9	Binary synchronous communications (primary)
3B0–3BF	IBM monochrome display/printer
3C0–3CF	Reserved
3D0–3DF	Colour/graphics
3E0–3E7	Reserved
3F0–3F7	Disk
3F8–3FF	Asynchronous communications (primary)

*Since addresses overlap, you cannot use both communications options at once.

enter a new Base address and watch the program repeat the procedure. Unoccupied I/O locations may be identified by the contents being set to zero.

C program development

The C language is compiled. Program statements, i.e. the source code, are not executed directly as with interpreted languages. Instead they are written to a file called the source program, using a text editor or word processor. The source program is then processed by the C compiler. The output from the compiler is the machine code equivalent of the source program: the object program. Incorporating certain external modules using the link program results in an executable program. The flowchart for the compilation/link process is shown in Figure 1.7.

It is rewarding to examine the general structure of all C programs before becoming involved in the fine detail of interfacing.

Example 1.1

```
10 REM PEEKING I/O
20 P = INP (768):REM READ PORT A
30 PRINT P
```

Example 1.2

```
10 REM POKING I/O
20 OUT 769,50:REM WRITE PORT B
```

Example 1.3

```
/*************************
* PEEKING I/O ADDRESSES *
**************************/
#include <stdio.h>
#include <conio.h>
main()
{
unsigned char p;
p = inp(768);
/*----------
READ PORT A
----------*/
printf("%d_n",p);
}
```

Example 1.4

```
/************************
* POKING WITH C *
*************************/
#include<stdio.h>
#include <conio.h>
main()
{
outp(769,50)
/*----------
WRITE TO PORT B
----------*/
}
```

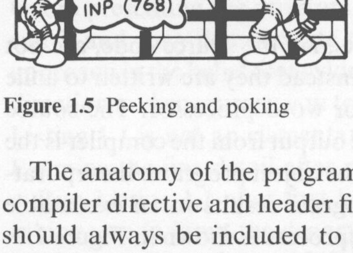
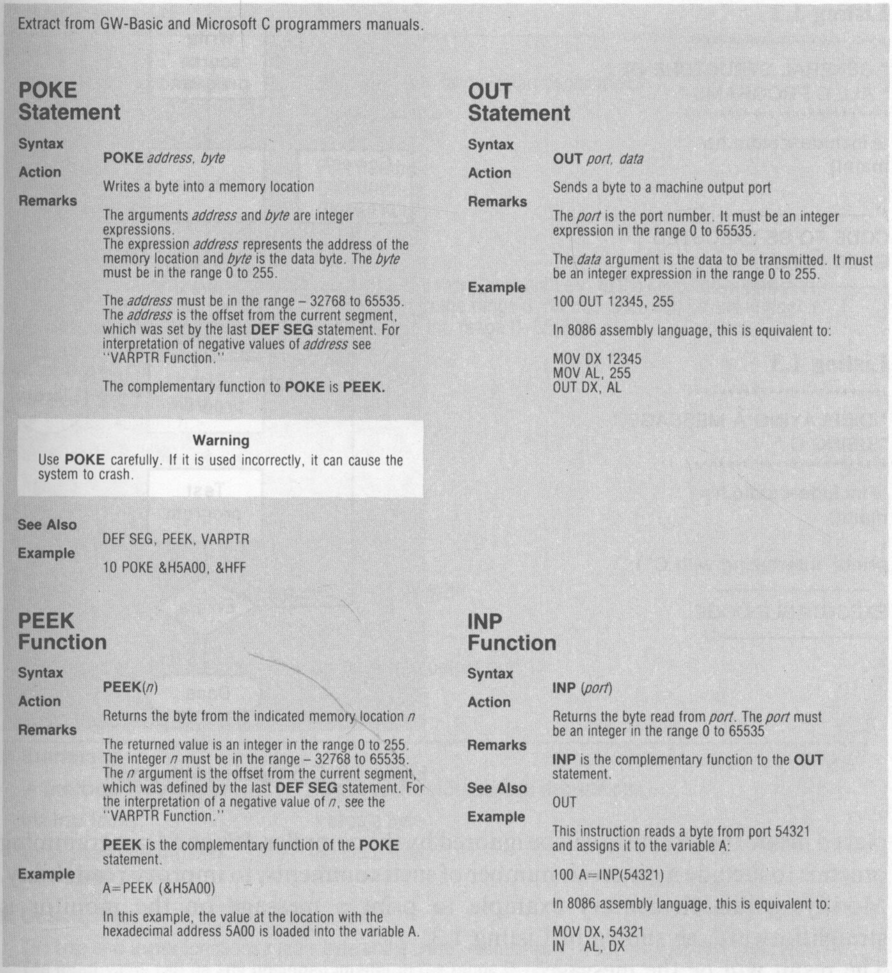


Figure 1.5 Peeking and poking

Listing 1.1

```
/************************
* READING I/O ADDRESSES *
* USING POINTERS *
*************************/
#include<stdio.h>
main()
{
int *port_x;
unsigned char contents;
int i,j,x;
/*---------------------------
*port_x IS A POINTER DECLARED AS
AN INTEGER. THE VARIABLE contents
IS AN UNSIGNED CHARACTER. THE
VARIABLES i,j AND x ARE INTEGERS
-----------------------------*/
start:printf("Input base address");
scanf("%d",&i);
/*---------------------------
ENTER A DENARY INTEGER i FROM THE
KEYBOARD
------------------------------*/
for(j = i;j <= 16 + i;j++)
{
x = j;
port_x = (int*)j;
/*---------------------------
THIS CONSTRUCTION ESTABLISHES THE
ADDRESS OF THE POINTER
-----------------------------*/
contents = *port_x;
/*---------------------------
WHEN * IS USED AS A PREFIX TO AN
INTEGER VARIABLE NAME WE RECOVER
THE VALUE AT THAT ADDRESS
                             */
printf("%d\n",contents);
/*---------------------------
PRINT THE DENARY CONTENTS OF THE
I/O ADDRESSES ON THE SCREEN
                             */
}
goto start;
}
```

The anatomy of the program is made up as follows: #include<stdio.h> is the compiler directive and header file. This file is provided with each C compiler and should always be included to guarantee successful compilation. Check the C compiler manual for the precise syntax for your particular system. Some compilers require #include"stdio.h" or #include<h.stdio>. Stdio is a contraction of



Extract from GW-Basic and Microsoft C programmers manuals.

POKE Statement

Syntax
POKE *address, byte*

Action
Writes a byte into a memory location

Remarks
The arguments *address* and *byte* are integer expressions.
The expression *address* represents the address of the memory location and *byte* is the data byte. The *byte* must be in the range 0 to 255.

The *address* must be in the range – 32768 to 65535. The *address* is the offset from the current segment, which was set by the last **DEF SEG** statement. For interpretation of negative values of *address* see "VARPTR Function."

The complementary function to **POKE** is **PEEK.**

Warning
Use **POKE** carefully. If it is used incorrectly, it can cause the system to crash.

See Also
DEF SEG, PEEK, VARPTR

Example
10 POKE &H5A00, &HFF

PEEK Function

Syntax
PEEK(*n*)

Action
Returns the byte from the indicated memory location *n*

Remarks
The returned value is an integer in the range 0 to 255. The integer *n* must be in the range – 32768 to 65535. The *n* argument is the offset from the current segment, which was defined by the last **DEF SEG** statement. For the interpretation of a negative value of *n*, see the "VARPTR Function."

PEEK is the complementary function of the **POKE** statement.

Example
A=PEEK (&H5A00)

In this example, the value at the location with the hexadecimal address 5A00 is loaded into the variable A.

OUT Statement

Syntax
OUT *port, data*

Action
Sends a byte to a machine output port

Remarks
The *port* is the port number. It must be an integer expression in the range 0 to 65535.

The *data* argument is the data to be transmitted. It must be an integer expression in the range 0 to 255.

Example
100 OUT 12345, 255

In 8086 assembly language, this is equivalent to:

```
MOV DX 12345
MOV AL, 255
OUT DX, AL
```

INP Function

Syntax
INP (*port*)

Action
Returns the byte read from *port*. The *port* must be an integer in the range 0 to 65535

Remarks
INP is the complementary function to the **OUT** statement.

See Also
OUT

Example
This instruction reads a byte from port 54321 and assigns it to the variable A:

100 A=INP(54321)

In 8086 assembly language, this is equivalent to:

```
MOV DX, 54321
IN   AL, DX
```

Figure 1.6

'standard input–output'. This particular header file provides the necessary system information to input data from the keyboard and display it on the monitor. Some programs require additional header files, the names in these header files containing system-related information that is made part of the program as it is compiled – ref inp() and outp().

All C programs are functions, usually made up of the principal function main() together with any nested functions. The example in Listing 1.2 is probably the most primitive C program imaginable, where the code located inside the braces is simply a non-executable comment, analogous to the Basic REM statement. Notice that the non-executable comment is preceded by a slash star /* and terminated by a star slash */. These comment delimiters ensure that any remark

此为试读，需要完整PDF请访问： www.ertongbook.co