

OPERATING MANUAL



KK SYSTEMS LTD

01001011 01001011 00100000 01010011 01011001 01010011 01010100 01000101 01001101 01010011

Copyright

© 1992–2000 KK Systems Ltd. No reproduction of any part of this document, in any form, is allowed without prior written permission from KK Systems Ltd. All other copyrights and trademarks acknowledged.

The PPC is protected by patents.

Extract from Conditions of Sale

Any electronic device or system can fail, possibly resulting in the loss of valuable programs or data. It is your responsibility to ensure that all such valuable material is backed-up at all times. We are not liable for any direct, indirect or consequential loss caused directly or indirectly through the use of this product. All our software is sold on an “as is” basis without a warranty of any kind. We do not claim that this product is suitable for all potential applications. It is your responsibility to verify that the product works in its intended application. In the interest of progress, we reserve the right to alter prices and specifications without prior notice.

Product versions

This manual covers the following versions:

PPC firmware	v1.07 dated 5/FEB/97 or later
KTERM.EXE	v1.01E

Table of Contents

PPC Overview	1-1
What is the PPC ?	1-1
The Front Panel	1-2
SW2+SW3 – PPC Reset	1-2
SW1 – Entry into the Executive	1-2
LED Indicators	1-2
Options	1-3
PPC-4	1-3
PPC-E	1-3
Get Started	2-1
Step 1: Software Installation	2-1
Step 2: Connections	2-1
Step 3: Enter the Executive	2-1
Step 4: Executive operations	2-1
Step 5: BASIC	2-2
Step 6: PASCAL and the Editor	2-2
Step 7: Autoexec operation	2-3
The PPC Executive	3-1
PPC Executive Fundamentals	3-2
Basic Principles	3-2
The Program Directory	3-2
Program Name Rules	3-2
PPC Executive functions and facilities	3-3
1. Interactive Basic	3-3
2. Create program	3-3
3. Edit program	3-3
4. Compile only	3-3
5. Compile and Run	3-4
6. File utilities	3-4
6.1 Display program head	3-4
6.2 Print program	3-4
6.3 Copy Program	3-5
6.4 Delete program	3-5
6.5 Delete all	3-5
6.6 Set password	3-5
6.7 Encrypt program	3-5
6.8 Decrypt program	3-5
7. Define power-up action	3-6
8. Transfer PPC ← PC	3-6
9. Transfer PPC → PC	3-6
10. Config & diagnostics	3-6
10.1 Runtime port config	3-7
10.2 Config dump	3-7
10.3 Input hex dump	3-8
10.4 Serial port tests	3-8
10.5 Printer config	3-8
10.6 Executive Mode Port 1 config	3-8
10.7 Set real-time clock (option)	3-8
10.8 Analog Port tests (-A option)	3-9

11. Reboot the PPC	3-9
PPC EDITOR	4-1
The Look of the Editor	4-1
Typing Modes	4-1
Editor Commands	4-2
Cursor-Moving Commands	4-2
Text Deleting Commands	4-3
Block Commands	4-3
Quick Cursor Movement	4-4
Find and Substitute	4-4
Leaving the Editor	4-5
Toggles	4-5
Miscellaneous	4-6
Prompts and Messages	4-7
Prompts	4-7
Editor Error Messages	4-7
Editor limits	4-8
Non-Printing Characters	4-8
PPC BASIC	5-1
What is PPC BASIC ?	5-1
Performance	5-1
Example	5-1
How to create a PPC BASIC program	5-1
How to execute (run) a PPC BASIC program	5-1
PPC BASIC language description	5-1
Numbers	5-2
Variables	5-2
Functions	5-2
Arithmetic and Compare Operators	5-2
Expressions	5-2
Statements	5-2
Program	5-2
BASIC Functions	5-4
ABS	5-4
RND	5-4
GET	5-4
WGET	5-4
ADC	5-4
SWI	5-4
BASIC Commands	5-5
REM	5-5
LET	5-5
PRINT	5-5
IF	5-5
GOTO	5-6
GOSUB	
RETURN	5-6
FOR	
NEXT	5-6
REDIR	5-6
INPUT	5-7
DAC	5-7
SEED	5-7

STOP	5-7
Direct Commands	5-8
HELP	5-8
RUN	5-8
LIST	5-8
NEW	5-8
SAVE	5-8
LOAD	5-8
DIR	5-8
KILL	5-8
BYE or SYSTEM	5-8
Programming Examples	5-9
Speed tip	5-9
PPC PASCAL	6-1
What is PPC PASCAL ?	6-1
Example	6-1
How am I going to learn PASCAL ?	6-1
How to create, edit and compile a PASCAL program	6-1
PPC PASCAL - Introduction	6-2
A Quick Program	6-2
Special features of PPC PASCAL	6-3
Compiler Listing	6-3
PPC PASCAL Syntax and Semantics	6-4
IDENTIFIER	6-4
UNSIGNED INTEGER	6-4
UNSIGNED NUMBER	6-4
UNSIGNED CONSTANT	6-5
CONSTANT	6-5
CHARACTER STRINGS	6-5
SIMPLE TYPE	6-6
TYPE	6-6
ARRAYs and SETs	6-7
Pointers	6-7
FILES	6-7
RECORDs	6-9
FIELD LIST	6-9
VARIABLE	6-10
FACTOR	6-10
TERM	6-11
SIMPLE EXPRESSION	6-11
EXPRESSION	6-11
PARAMETER LIST	6-11
STATEMENT	6-12
Assignment statements	6-12
CASE statements	6-12
FOR statements	6-13
GOTO statements	6-13
WITH statements	6-13
BLOCK	6-14
Forward References	6-14
PROGRAM	6-15
Strong TYPEing	6-15
Semicolons	6-15
Predefined Identifiers	6-16

Constants	6-16
Types	6-16
Variables	6-16
PROCEDURES AND FUNCTIONS	6-16
File Handling Procedures	6-16
WRITE	6-16
WRITELN	6-17
BWRITE	6-17
READ	6-18
READLN	6-19
BREAD	6-19
File Handling Functions	6-20
EOLN	6-20
Transfer Functions	6-20
TRUNC	6-20
ROUND	6-20
ENTIER	6-20
ORD	6-20
CHR	6-20
Arithmetic Functions	6-21
ABS	6-21
SQR	6-21
SQRT	6-21
FRAC	6-21
SIN	6-21
COS	6-21
TAN	6-21
ARCTAN	6-21
EXP	6-21
LN	6-22
HALT	6-22
RANSEED	6-22
RANDOM	6-22
SUCC(X)	6-22
PRED(X)	6-22
ODD	6-22
ADDR(V)	6-22
SIZE(V)	6-22
RECAST(e, typ)	6-23
MEMAVAIL	6-23
PASCAL Dynamic Memory Allocation	6-24
NEW	6-24
DISPOSE	6-24
MARK	6-24
RELEASE	6-24
Examples	6-25
PPC PASCAL special extensions	6-26
SETPORT - configure a serial port	6-26
READPORT - get serial port configuration	6-27
SETHSK - control handshake outputs	6-27
READHSK - read handshake inputs	6-28
IPQCOUNT - get #bytes in input queue	6-28
IPQCLEAR - clear input queue	6-28
OPQSPACE - test output queue space	6-28

OPQCOUNT - get #bytes in output queue	6-29
SETRTC - set real-time clock	6-29
GETRTC - read real-time clock	6-30
LOADTIMER - load a timer	6-30
READTIMER - read a timer	6-30
ULED - control user LED	6-31
READSWI - read front panel switches	6-31
VAL - convert a number	6-31
INIT - initialise data	6-32
SUMBUF - compute a checksum over a buffer	6-33
XORBUF - compute a XOR over a buffer	6-33
CRCBUF - compute a CRC over a buffer	6-33
_BIT - integer bit test	6-34
_AND - integer bit-wise AND	6-34
_OR - integer bit-wise OR	6-34
_XOR - integer bit-wise XOR	6-34
_CPL - integer bit-wise complement	6-34
_SHL - integer shift left	6-34
_SHR - integer shift right	6-34
_ROL - integer rotate left	6-35
_ROR - integer rotate right	6-35
_BITC - char bit test	6-35
_ANDC - char bit-wise AND	6-35
_ORC - char bit-wise OR	6-35
_XORC - char bit-wise XOR	6-35
_CPLC - char bit-wise complement	6-35
_SHLC - char shift left	6-36
_SHRC - char shift right	6-36
_ROLC - char rotate left	6-36
_RORC - char rotate right	6-36
STRCMP - compare strings	6-36
STRNCMP - compare strings, n chars	6-36
MEMCPY - memory copy	6-36
TOUPPER - convert to uppercase	6-37
CBRK - check for a break character	
OBRK - output a break character	6-37
PPC break condition checking	6-37
Analog Subsystem PASCAL functions	6-38
ADCINI - initialise ADC	6-38
ADC - read analog-digital converter	
ADCC - read analog-digital converter (calibrated)	6-38
ADCGAIN - set $\pm 100\text{mV} / \pm 1\text{V}$ gain	6-38
DAC - drive digital-analog converter	
DACC - drive digital-analog converter (calibrated)	6-38
Execution times	6-39
ANEERD - read analog subsystem calibration EEPROM	
ANEEWR - write analog subsystem calibration EEPROM	6-39
DIN1 - read TTL input #1	
DIN2 - read TTL input #2	6-39
DOUT1 - drive TTL output #1	
DOUT2 - drive TTL output #2	6-39
DG1 Parallel Digital I/O Card Pascal functions	6-40
PDIN - read parallel digital inputs	6-40
PDOUT - write parallel digital outputs	6-40
EEPROM non-volatile access functions	6-41
NVWRITEC - write a char	6-42

NVREADC - read a char	6-42
NVWRITEI - write an integer	6-42
NVREADI - read an integer	6-42
NVWRBLK - write a block	6-43
NVRDBLK - read a block	6-43
Comments and Compiler Options	6-44
COMMENTS	6-44
COMPILER OPTIONS	6-44
General Options	6-44
Option L	6-44
Option S	6-44
Option A	6-44
Option I	6-44
Option P	6-45
Top-of-File Options	6-45
Option R	6-45
Option O	6-45
Option C	6-45
Option U	6-46
Program and Data limits	6-47
Error messages	6-48
Compilation error messages	6-48
Runtime error messages	6-49
Reserved Words and Predefined Identifiers	6-50
Reserved Words	6-50
Special Symbols	6-50
Predefined Identifiers	6-50
Programming Examples	6-52
Robust reading of input data	6-52
Character data	6-52
ASCII numbers	6-52
Multi-byte binary numbers	6-53
Character translation	6-53
String translation	6-54
Performance hints	6-55
Program too large	6-56
Data Representation and Storage	6-57
Data Representation	6-57
Integers	6-57
Characters, Booleans and other Scalars	6-57
Characters:	6-57
Booleans:	6-57
Reals	6-57
Records and Arrays	6-57
Sets	6-58
Pointers	6-58
Recommended Books	6-59
Tutorial Style Books	6-59
Reference Books	6-59
KTERM Terminal Emulator	7-1
What is KTERM ?	7-1
Why use KTERM ?	7-1
KTERM installation	7-1
KTERM operation	7-1

KTERM File Transfer	7-1
1. To transfer a file from the PPC to your PC	7-2
2. To transfer a file from your PC to the PPC	7-2
PPC - PC filename relationships	7-2
PC filename directory control	7-2
File transfer security	7-2
DOS shell	7-3
KTERM command line options	7-3
KTERM examples	7-4
KTERM under Windows 3.x, Windows 95, Windows NT 4	7-5
Running KTERM over a modem link	7-6
Cables	7-6
PPC configuration	7-6
Modem configuration	7-6
KTERM usage	7-7
Operation	7-7
KTERM Troubleshooting	7-9
Executive menu does not appear at all	7-9
Garbled data appears	7-9
Editor display is incorrect, or other display faults	7-9
Control-c does not stop program execution	7-9
KTERM Error Messages	7-9
PPC Serial Ports & Cables	8-1
Hardware Interface - RS232	8-1
Pin Functions	8-1
Hardware Interface - RS422/RS485	8-2
RS422/RS485 grounding	8-2
RS422/RS485 Termination	8-3
RS232 to RS422/RS485 port conversion	8-3
PPC-E DIN41612 connections	8-4
RS485 driver control	8-4
I/O Queues	8-5
Transmit Queues	8-5
Receive Queues	8-5
Handshaking	8-6
TX Handshakes	8-6
RX Handshakes	8-6
Binary Data Handshaking	8-6
Modem Control	8-7
Port Parameters	8-8
Executive Mode configuration	8-8
Cables	8-9
Modems	8-9
PPC Loopback Adaptors	8-9
PC/Terminal ↔ PPC Cable	8-9
PC ↔ PPC cable	8-10
PPC → printer/plotter cable	8-10
PPC → CalComp plotter cable ("DTE Host")	8-11
I/O Expansion Options	9-1
16-Bit Analog Subsystem	9-2
Facilities and Operating Modes	9-2
Hardware Requirements	9-3
Accuracy and Performance	9-3

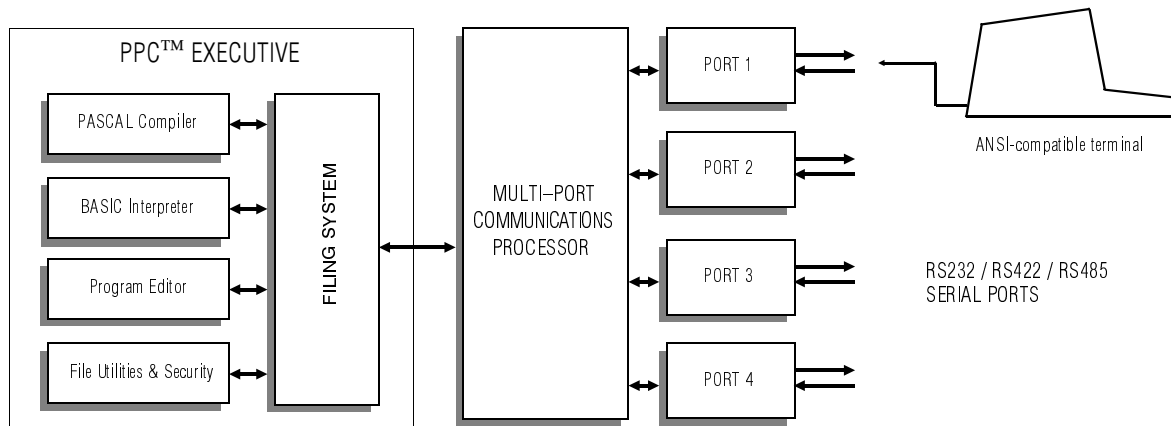
Installation	9-3
Protection	9-3
Isolation	9-3
DG1 Digital I/O Card	9-4
Hardware Requirements	9-4
Installation	9-4
Protection	9-4
Isolation	9-4
Troubleshooting	10-1
Cannot enter the Executive	10-1
The PPC “hangs” on entry to Runtime Port Configuration menu	10-1
A program terminates for no apparent reason	10-1
A program terminates while reading data	10-1
A program hangs during communications	10-2
Loopback tests hang-up the PPC	10-2
RS422/485 Problems	10-2
C Introduction	11-1
Step 1: Get to know the PPC	11-1
Step 2: Compiler Installation	11-1
Step 3: PPC C Software Utilities Installation	11-2
Step 4: C Option installation	11-2
Step 5: Reboot your PC	11-2
Step 6: Create a program	11-2
Creating larger programs	11-3
C Reference	12-1
The memory map	12-1
C Extensions	12-2
I/O Read/Write Functions	12-2
I/O Queue Management Functions	12-3
I/O Port Configuration Functions	12-3
I/O Port Direct Handshake Control	12-4
“Break” Sequences	12-4
Timing Functions	12-4
Real-Time Clock Functions	12-4
Checksums and CRC	12-4
Bit Operations and Rotation	12-5
Extended Memory Functions	12-5
Miscellaneous Functions	12-5
DES Encryption Functions	12-6
Miscellaneous	12-7
PPCEXTRA.LIB Extra Library	12-7
Checking of “printf”-type Function Parameters	12-7
Performance Hints	12-7
Technical Support	12-7
ASCII Character Codes	13-1
Other Products	1-1
Inline RS232/RS4xx Converters	1-1
KD485 DIN Rail Converter	1-1

PPC Overview

What is the PPC ?

The PPC is a user-programmable protocol converter. Data received on a communications port can be modified as required and output on another communications port. The modification is done by a BASIC, PASCAL or ANSI C program written by the user.

In a compact self-contained unit, the PPC integrates all functions required to develop the required BASIC or PASCAL programs, to execute them and, finally, to produce a standalone protocol converter which can be left in place.



An ANSI-compatible “dumb” terminal is the only additional equipment that is required, and it can be disconnected once the program is finished. All PPC configuration is done via the terminal; there are no internal switches.

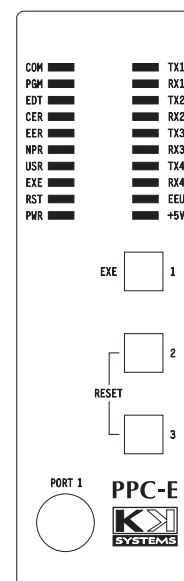
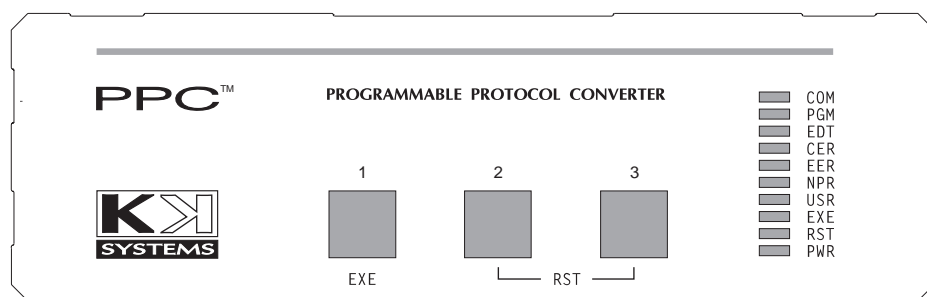
The high level languages provided are integrated into the PPC in a way which eliminates any need for directly accessing the PPC hardware and greatly simplifies the creation of datacomms programs. The programmer does not need any knowledge of the hardware and low-level software involved in the PPC.

An ANSI C cross-compiler is also available, with a runtime library specially modified for the PPC.

Each of the serial ports supports the full set of asynchronous modem handshake and control lines and can be connected to most computers, printers, plotters, modems and other devices. Each port can be configured for RS232, RS422, or 2-wire or 4-wire RS485 operation.

The filing system stores up to 20 BASIC, PASCAL or binary (compiled C) programs in non-volatile semiconductor EEPROM memory which uses no batteries and has a typical retention time of 100 years. Any one of the programs can be designated “autoexec” and execute automatically at every power-up.

The Front Panel



The PPC has three front panel switches, SW1, SW2 & SW3, which are readable from within BASIC, PASCAL and C programs. They also perform the following functions:

SW2+SW3 – PPC Reset

Depressing SW2 and SW3 simultaneously for approximately 1 second causes the PPC to be reset. This is equivalent to interrupting power. The reset state is indicated by the RST LED.

SW1 – Entry into the Executive

If SW1 is depressed when the PPC exits from reset, the PPC enters the Executive, even if it had been previously configured to "autoexec" a program at power-up. The Executive is a menu-driven system for PPC configuration and for editing user programs. See the Executive chapter for details.

LED Indicators

The ten LED indicators on the main LED bar display have the following functions:

- COM** A compilation is in progress (PASCAL only).
- PGM** A program (PASCAL, C or BASIC) is running.
- EDT** The PPC editor is running.
- CER** A compilation error (PASCAL only).
- EER** An execution error (run-time error).
- NPR** The PPC was configured to "autoexec" a program but the program is missing.
- USR** User-accessible LED. Can be switched on/off from PASCAL and C programs.
- EXE** Flashes when the PPC is in the Executive.
- RST** The PPC is being reset. ON when SW2+SW3 depressed together for 1 second.
- PWR** Power input. Always ON.

When the PPC is running a BASIC program, COM & PGM are ON together because BASIC is an interpreter and is therefore effectively both compiling and executing the program.

The EER indicator is latched. If a runtime error occurs in a program, it remains on until the program terminates. In PASCAL, runtime error trapping can be disabled (see the PPC PASCAL chapter) and doing so also disables this front panel indication.

The PPC-E has a second LED bar display. The top eight LEDs indicate TX,RX activity on each of the four ports. These eight LEDs can also be accessed directly from C programs.

Options

PPC-4

The standard PPC-4 is a four-port product which replaces the previously offered PPC-2 and the -4 option. A number of factory-fitted options are available, including the following:

-R suffix	A lithium-battery-powered real time clock. Allows PASCAL and C programs access to date and time
-H2 suffix	Higher-speed version. 100% increase in CPU speed.
-H3 suffix	Higher-speed version. 200% increase in CPU speed.
-C suffix	The ROM includes the runtime library for the ANSI C cross-compiler kit.
-96 suffix	Additional RAM, accessible from C as a 96k random-access file. This memory can also be battery-backed using a Dallas "Smart Socket".
-DG1 suffix	Digital Parallel I/O Card. 16 TTL-level inputs plus 16 high-current open-collector outputs. Accessible from Pascal and C. See the DG1 I/O Card chapter.
-A suffix	Analog Subsystem. 16-bit 8-channel analog-digital converter (ADC). The Analog Subsystem itself consists of a number of options which include optional D-A converters and other features. See the Analog Subsystem chapter.

The model and any options are marked on the underside, together with the serial number.

Customer-specific options can be implemented, subject to a reasonable quantity. These can be in the form of special PPC Pascal and C runtime functions, or as an "option board" such as the Analog Subsystem or the DG1 Digital Parallel I/O. Examples of customer-produced designs are 1MB FLASH cards and I²C cards.

PPC-E

This is the Eurocard variant. Due to the surface-mount construction most of the above PPC-4 options are fitted as standard, namely the -R -H2 -C -96. The only presently applicable option is -H3.

Get Started

This chapter is a brief tour around the PPC. You will create a simple BASIC program, use the Editor, create a simple PASCAL program, use some of the Executive functions and, finally, create a simple standalone protocol conversion product.

Step 1: Software Installation

The supplied diskettes contain executables which are relevant only if you are accessing the PPC from an IBM-compatible PC.

To install, simply COPY everything in the root or \utils directory of the supplied diskette to any suitable directory on your hard disk, and edit your AUTOEXEC.BAT file to include that directory in the PATH.

If your PC has floppy drives only, you need only copy the KTERM.EXE program to your working diskette. The working diskette must have COMMAND.COM on it, otherwise you will not be able to shell-out of KTERM.

Step 2: Connections

With the PC-PPC cable detailed in the Serial Ports and Cables chapter, connect an ANSI-compatible terminal or a PC to Port 1 of the PPC (see the PPC Executive chapter for details) and connect the PPC power unit.

Step 3: Enter the Executive

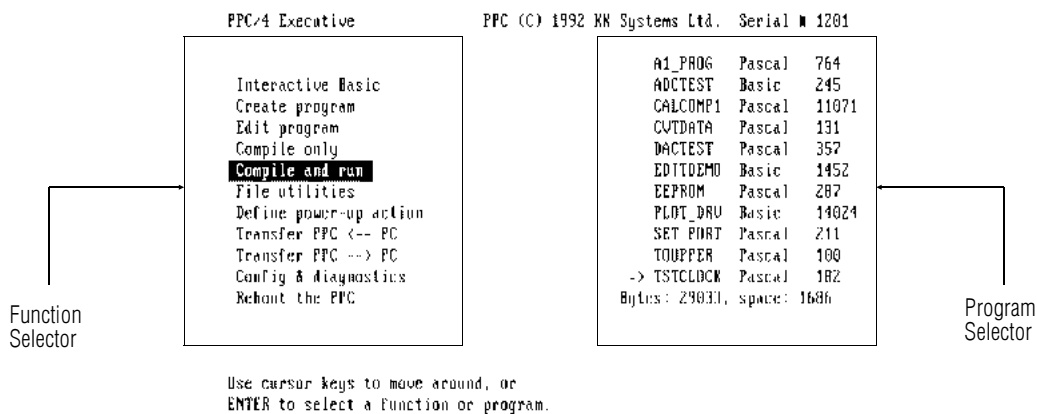
Following power-up, the PPC will run through a quick power-up test which is visible on the front panel LEDs, after which the EXE LED remains flashing.

If the EXE LED does not start flashing then the PPC has probably been previously configured to automatically execute a program; see the Troubleshooting chapter on how to enter the Executive.

If you are using a terminal, press the ESC key, several times if necessary. If you are using an IBM-compatible PC, run the supplied KTERM.EXE program by typing

KTERM ↵

The Executive menu (see the PPC Executive chapter) should now appear:



Step 4: Executive operations

With the ↑↓ cursor keys, move the highlighted bar up and down. With the ←→ cursor keys, move the bar between the function selector (on the left) and the program selector (on the right).

Pressing ↵ while the highlighted bar is on the right merely selects that program for future operations. Pressing ↵ while the highlighted bar is on the left selects that function which, with some of the functions, can descend into another function menu. You are free to examine the facilities provided; pressing ESC always returns to the main Executive menu.

Step 5: BASIC

Highlight the Interactive Basic item and press ↵. You are now in PPC BASIC and you will create a new program. Type-in the following simple program (the comments on the right are not part of the program):

```
10 PRINT "READY"
20 C=WGET(1)           wait for a character on Port 1 and load it into C
30 IF C=66 C=67       if the char is 'B' then make it 'C'
40 PRINT CHR$(C),    output it to the default port (Port 1)
50 GOTO 10           and repeat forever
```

Run the program by typing

```
RUN ↵
```

The program starts with the word READY and expects some data. If you press various keys on the keyboard, you will see that they are echoed unchanged, except that the letter 'B' is changed to 'C'.

To terminate the program, press CTRL-C (press C while holding-down the CTRL key). You may have to press it more than once. Save the program in the PPC's filespace under the name BTOCB with the command

```
SAVE BTOCB ↵
```

Upper/lowercase is irrelevant. Confirm that it has been saved with the command

```
DIR ↵
```

The program is still in memory and can be RUN again if desired. Finally, exit BASIC with

```
BYE ↵
```

Step 6: PASCAL and the Editor

PASCAL programs cannot be interactively typed-in as we did with BASIC. Instead, an empty program is created, and the PPC editor is used to type-in the text. Highlight the **Create Program** item and press ↵. At the program name prompt, type

```
BTOCP ↵
```

(upper/lowercase is irrelevant) and, when the language selector appears, highlight the **Pascal** selection and press ↵. When the program has been successfully created, press ESC to exit to the Executive main menu.

In the program directory on the right, you will see the newly-created program BTOCP, with a Pascal language attribute, a size of zero bytes, and a => marker displayed next to it. If the => marker is *not* next to the program, correct this by highlighting the correct program and pressing ↵.

Highlight the **Edit Program** item and press ↵. This enters the PPC editor which displays a blank screen with the program name at top left. Full details of all editor commands are in the PPC Editor chapter but here we will use only a small subset. The editor commands are a subset of those used in Wordstar-4.

Type-in the following simple program. Upper/lowercase is unimportant. The indenting (done with the TAB key) is also unimportant but is common in Pascal as it helps to make the logic of a program more obvious.

```
program btocp;
var c:char;
begin
  writeln(port1,'READY');
  repeat
    read(port1,c);
    if c='B' then c:='C';
    write(port1,c);
  until false;
end.
```

Save the program and exit the editor with CTRL-K X (first, type K while holding-down the CTRL key and, second, type X just by itself). Back in the Executive main menu, you will again see the program BTOCP in the directory but this time with a size of around 100+ bytes.

Highlight the **Compile Only** item and press ↵ to test-compile the program. No errors should be reported. If any are reported, select the **E** option to re-enter the editor and correct the error.

When the **Compile Only** function completes with no errors, proceed to **Compile and Run**. The program is now running and, as with the BASIC example earlier, every character you type is echoed except 'B' which is changed to 'C'.

To terminate the program, press CTRL-C. You may have to press it more than once.

Step 7: Autoexec operation

This is the final step in producing a standalone protocol converter.

In the **Config & Diagnostics** function, select the **Runtime Port Configuration**. This displays the runtime configurations for all the ports. These configurations apply only when a Basic, Pascal or binary (compiled C) program is running. Furthermore, the Port 1 configuration applies only to programs running in autoexec mode (as ours will be in a moment). Ensure that Port 1 is set to 9600 baud, 8 bits/word, no parity and 1 stop bit. All six handshakes should be in their factory settings: all ON except XON/XOFFs.

Return to the main Executive menu with two ESC keys.

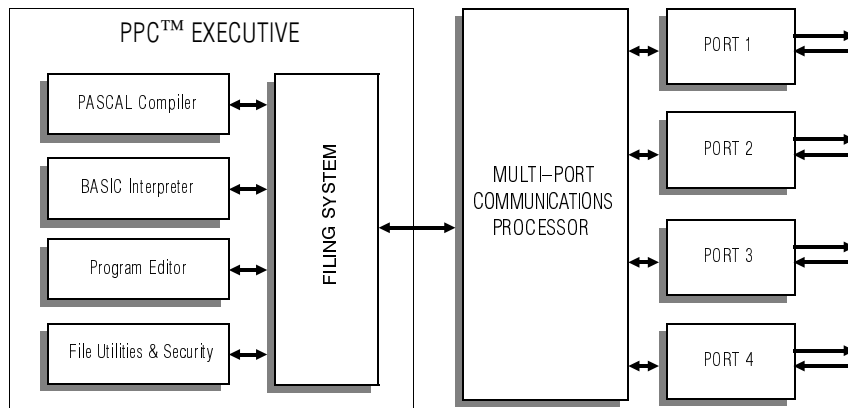
Ensure the => directory marker is displayed next to your BTOCP program. Select the **Define Power-Up Action** function, highlight the **Run Selected Program** item and press ↵. Your program is now autoexec.

Return to the main Executive menu with ESC. Next time the PPC powers-up, it will automatically compile and run your program. To test it, select the **Reboot the PPC** function. After a few seconds, the READY prompt will appear and the program runs as before.

Note that the program cannot now be terminated with CTRL-C, or with any other key. To return to the Executive, press all three switches SW1+SW2+SW3 (on the front panel) and, when the RST LED illuminates, release SW2+SW3 only. Maintain SW1 pressed until the Executive menu appears.

Back in the Executive, return to the **Define Power-Up Action** function and select the **Enter Executive** mode. The brief tutorial is now complete.

The PPC Executive

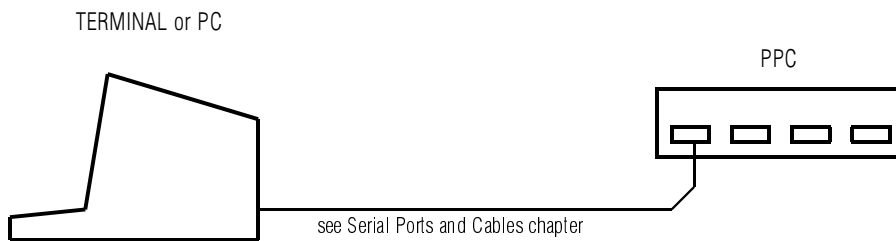


The Executive is the PPC's user interface. It integrates all the functions required for creation, editing, compilation and execution of user programs.



Please read through this chapter before embarking on any serious work with the PPC. It contains important information which you will need in order to use the PPC effectively.

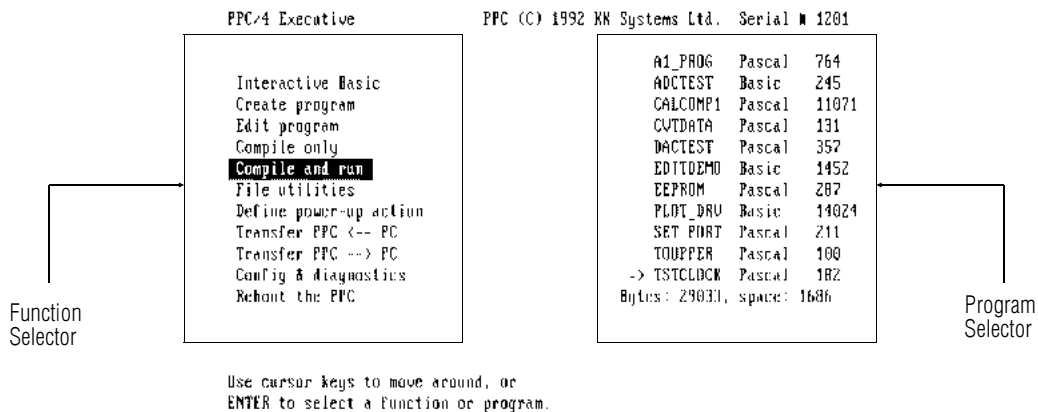
To enter the Executive, connect a **25-line ANSI terminal** (configured to 9600 baud, 8 bits/word, no parity, 1 stop bit, XON/XOFF handshake) to the PPC with the cable detailed in Serial Ports and Cables chapter, as shown below:



Power-up the PPC and, a few seconds later, the Executive menu will appear on the terminal.

Alternatively, if you are using an IBM or IBM-compatible PC with the KTERM.EXE terminal emulator program, connect your PC's COM1 port to the PPC with the cable specified in the Serial Ports and Cables chapter.

Power-up the PPC and wait until the EXE indicator flashes. This indicates that the PPC is in the Executive. *Then* invoke KTERM on the PC. The Executive main menu will appear and should look similar to the following:



If the Executive menu does not appear:

With a terminal: press ESC. If this fails, refer to the Troubleshooting information later in this chapter. **With KTERM:** exit KTERM (with alt-x) and re-invoke KTERM. Repeat several times if necessary. If this fails, refer to the Troubleshooting information in the KTERM chapter.

Basic Principles

The Executive is built around a Function Selector at the left-hand part of the screen, and a Program (file) Selector at the right-hand part of the screen. To select a particular *program* (for subsequent operations such as editing or compilation), use the → key to place the highlighted bar into the program directory, then use the ↑↓ keys to highlight the desired program which is finally selected with ↵ (the ENTER key). The selected program is now marked with a ⇒ marker.


Similarly, to select a particular Executive *function*, use the ← key to place the highlighted bar into the function selector, then use the ↑↓ keys to highlight the desired function which is finally invoked with ↵ (the ENTER key).

When you have descended into a sub-menu, you can return to the previous menu with ESC. The ESC key can be used to exit from most Executive functions.

 Do not press **CTRL-S** while you are in any of the Executive functions (these include the editor, the compiler, BASIC, or a PASCAL program running from **Compile & Run**). **CTRL-S** is an XOFF which will stop all output to the terminal until a **CTRL-Q** (XON) is pressed.

The Program Directory

This shows all programs currently stored in the filing system. Up to 20 programs may be stored.

 The terms *programs* and *files* are used interchangeably in this Manual. In the PPC, every file should be a BASIC or PASCAL program, otherwise it serves no purpose.

All programs are stored in a non-volatile EEPROM device. “Non-volatile” means that the data is not lost when power is disconnected – just like a e.g. a hard disk. An EEPROM device is highly secure, highly reliable and does not require any batteries, either.

 Do not disconnect power to the PPC when file operations are taking place. Doing so will probably result in loss of data.

The total program space available is 30719 bytes. A maximum of 20 programs can be stored. Each program reduces the remaining free space only by its exact size, plus 19 bytes which are used to store the program name and other information.

Two markers can appear next to the program directory:

⇒ Identifies the program selected for subsequent operations.

a Identifies the program which will be **automatically** executed at power-up. For details, see the “how to make an autoexec program” information later in this chapter.

Program Name Rules

Each program name must be 1–8 characters. Any character may be used in the name, with the exception of characters which fall within the following three groups:

- 1 ! # \$ % & () * + , - . / : ; < = > ? @ [\] ^ ' "
- 2 character codes below 033 decimal (21 hex)
- 3 character codes above 240 decimal (F0 hex)

Note that the above *does* allow most international (accented) characters, such as üöäå...

Each program has a “language” attribute which can be BASIC or PASCAL. Additional attributes may be supported in the future. The “language” of a program is determined when the program is created with the **Create Program** function or, alternatively, it is generated automatically when the program is transferred over from a PC using KTERM, according to the following rules:

MS-DOS filetype		PPC language attribute
.BAS	→	BASIC
.PAS	→	PASCAL
.BIN	→	BINARY

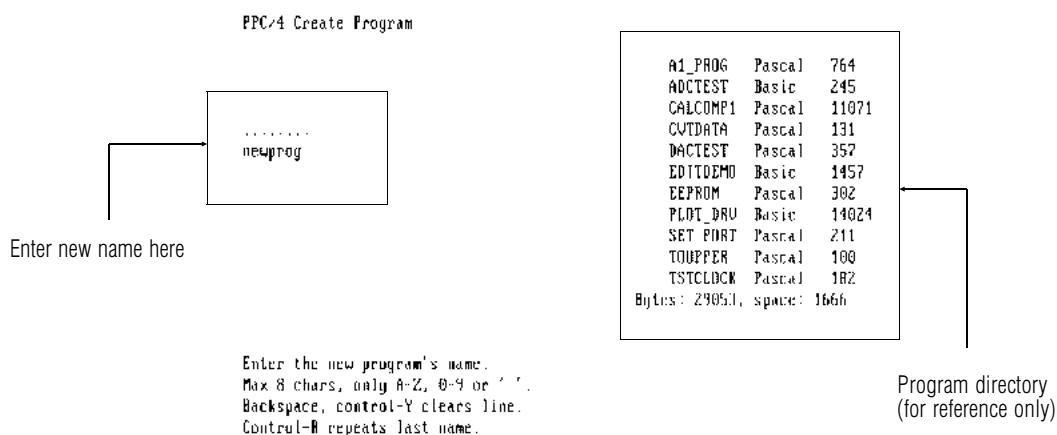
PPC Executive functions and facilities

1. Interactive Basic

Pressing **↵** when this item is highlighted enters the PPC BASIC interpreter. You can simply type-in a program and **RUN** it. For a list of commands and functions, type **HELP**. For full details see the **PPC BASIC** chapter.

2. Create program

Pressing **↵** when this item is highlighted enters the PPC function which is normally used to create all PPC programs:



When you have entered the program name, you will be prompted for the language: Basic or Pascal. The actual create operation can take up to 10 seconds and, following this, the newly-created program will appear in the directory with a filesize of zero bytes. You can now use the PPC Editor to enter text into the program. You will remain in the **Create Program** menu for convenience, in case you wish to create more programs. To return to the main menu, press ESC.

The PPC has no facility for changing the language attribute of a program once it has been created. The only way to change it is to transfer the program to a PC, rename its filetype, and transfer it back to the PPC.

The PPC rigidly enforces its language rules: only a BASIC program can be loaded into the BASIC Interpreter; only a PASCAL program can be compiled by the PPC PASCAL compiler.

Note that there are two other ways of creating PPC programs:

- 1 In PPC BASIC, the SAVE command will create a program with the Basic attribute.
- 2 When a .BAS or .PAS file is transferred from a PC to the PPC with KTERM, a program with a Basic or Pascal attribute respectively is created.

3. Edit program

Pressing **↵** when this item is highlighted enters the PPC Editor and opens-up for editing the program which is currently selected in the directory. The editor command set is partly Wordstar-4-compatible: use **CTRL-K Q Y** to exit the editor without saving any changes. For full details see the PPC Editor chapter.

4. Compile only

Pressing **↵** when this item is highlighted invokes the PPC PASCAL compiler and compiles the program which is currently selected in the directory; it must be a PASCAL program. For full details of the PASCAL compiler please see the PPC PASCAL chapter.

At the end of a successful compilation, the message

Press any key to return to Executive

appears. If errors occurred during compilation, the compiler error message is followed by:

ESC -> EXECUTIVE, C -> continue compilation, E -> Editor:

and you have the option to directly enter the editor, where the cursor will be automatically placed at the point where the compiler found the error. The **Compile** function's purpose includes:

- 1 It ensures that the program compiles without compilation errors. This is essential; a program which compiles with errors will not generate working machine code.
- 2 It will detect the case where a program's binary output (the machine code) is too large to run in the PPC. This is unlikely to happen except on programs whose PASCAL source program is itself close to filling-up the PPC's filesystem, or when you have declared a too-large item of data. The total amount of memory in which the machine code runs is approximately 25k and any static data items must fit within this figure also.
- 3 It will report compiler symbol table overflow during compilation. As above, this is unlikely to occur except on large programs.


 Do not proceed to **Compile and Run** until your program compiles without any errors.


- In PPC firmware v1.07+ the Compile function can also be used to execute a **binary** (i.e. compiled C) program. Previously, the only way to run those was to mark them "autoexec" and reboot the PPC.

5. Compile and Run

This is similar to the **Compile only** function described above, except that following a successful compilation the binary output is executed. When the program's execution ends, control returns to the Executive. If runtime break checks have been enabled with the C+ compiler option, you can interrupt the program with ctrl-c. For full details of the PASCAL compiler please see the PPC PASCAL chapter.

 To be useful, most practical data stream processing PPC programs must loop forever and never terminate.

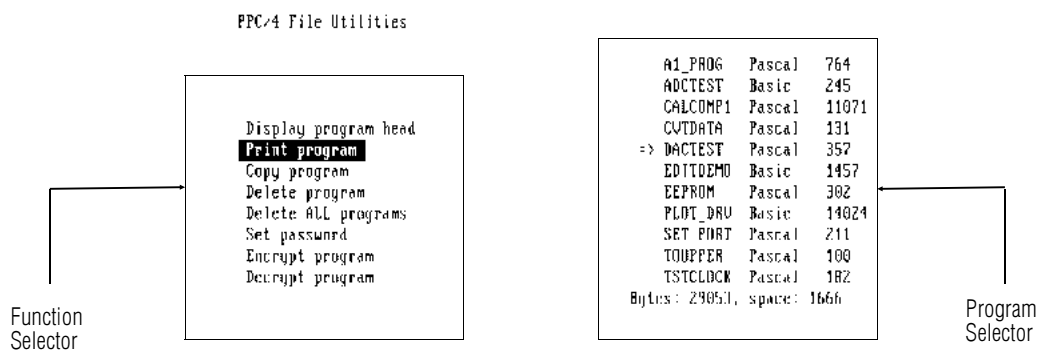
 For maximum performance, your finished program should have runtime break checks disabled. See PPC PASCAL chapter for details of compiler options.

 **KTERM users:** if you reset (or power-down) the PPC, you should exit KTERM (with alt-x) and re-invoke KTERM *after* the PPC's EXE LED starts flashing again.

- In PPC firmware v1.07+ the Compile and Run function can also be used to execute a **binary** (i.e. compiled C) program. Previously, the only way to run those was to mark them "autoexec" and reboot.

6. File utilities

Pressing \leftarrow when this item is highlighted enters the following menu:



6.1 Display program head

This function can be used to display the first 24 lines (or 42/49 lines if using KTERM with **/VL** option with EGA/VGA) of a program. It allows a quick identification of what the program does, in case this is not self-evident from the program's name.

If the program is binary, a 16-line hex dump is displayed of the first 256 bytes of the binary image.

6.2 Print program

This function generates a hard copy of your program on a printer attached to any PPC port.

When this function is selected with \leftarrow a menu of PPC output ports appears. This is followed by a menu of output devices; currently supported types are **ASCII** and **Postscript**.

Selecting **ASCII** outputs the currently selected program to a printer connected to the selected output port. Tabs in the program are expanded to spaces on 4-character tab stops and since the only control characters used are CRLFs, any ASCII printer will work. The printer can be initialised with a user-definable string specified in the **Printer Config** function (see 10.5); the default initialisation is for a HP Laserjet and selects a 16.6cpi line printer font.

Selecting **Postscript** is as for ASCII, except that the text is converted to Postscript and a Postscript printer must therefore be connected to the selected port. The Courier font resident in all Postscript printers is selected. The string defined in the above-mentioned **Printer Config** function has no effect when printing to Postscript.


6.3 Copy Program

This function duplicates an existing program but under a different name, retaining the language attribute.

When this function is selected with ↵ you are prompted to enter the new program's name, followed by ↵. The actual copy operation can take up to 10 seconds.

6.4 Delete program

When this function is selected with ↵ you are prompted to confirm with "Y". The actual delete operation can take up to 10 seconds. The Executive will not permit a delete if the **Power-Up Action** is set to "autoexec" a program; this prevents accidents such as deleting the autoexec program.

 The deleted program cannot be un-deleted and cannot be recovered. Don't rely on this for security though!

6.5 Delete all

When this function is selected with ↵ you are prompted to confirm with "Y". The operation can take up to 10 seconds.

This function does not affect any part of the PPC configuration, and it does not affect the 2000-byte Pascal-accessible nonvolatile storage area. It does, however, reset the **Power-up Action** setting to the "Enter Executive" state, to prevent the possible power-up execution of a nonexistent program.


 All programs stored in the PPC are irreversibly deleted and overwritten with zeroes.

6.6 Set password

When this function is selected with ↵ you are prompted to enter a password which will be used for all subsequent **Encrypt** and **Decrypt** operations. The password does not currently serve any other purpose. It is not permanently stored and is deleted when the Executive terminates, e.g. at power-down or when the **Reboot** function is invoked.

The password must be 8–16 characters in length and is case-sensitive. Any character between 33 decimal (21 hex) and 239 decimal (EF hex) inclusive may be used, but you should avoid non-displayable characters because of the potential for confusion between what you *think* you have entered and what you *actually* entered.

 Before encrypting a program, please **make a note of the password**.

 If you have accidentally encrypted a program and then realised that you have forgotten the password, proceed immediately to the **Decrypt** function and decrypt it.

 Remember the password is case-sensitive. **Password** is different from **PassWord**.

6.7 Encrypt program

When this function is selected with ↵ the currently selected program is encrypted with the currently active password. An error message is displayed if no password has been specified.

An encrypted program cannot be displayed, edited, printed, transferred to a PC, or compiled *from within the Executive*. However, it can be marked "autoexec" (see 7.) and compile/execute automatically at every power-up.

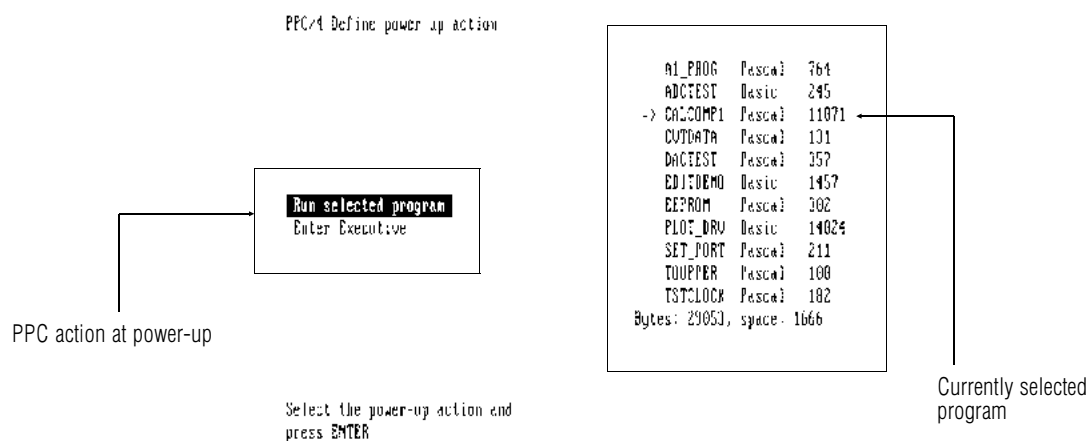
 **Do not use this function unless you have made a note of the password.**
An encrypted program cannot be recovered without the correct password.

6.8 Decrypt program

When this function is selected with ↵ the currently selected program is decrypted with the currently active password. An error message is displayed if no password has been specified.

7. Define power-up action

Pressing \leftarrow when this item is highlighted enters a menu:



You can instruct the PPC to either automatically compile+run (“autoexec”) the selected program, or to always enter the Executive. The selection is stored in non-volatile memory.

When the PPC is configured to autoexec a program, the only ways to enter the Executive are

- 1 If the program terminates, the PPC enters the Executive. However, a program which terminates will probably not be useful in most applications, so this method is useful during program development only.
- 2 Press all three front panel switches (SW1+SW2+SW3) until RST illuminates and then, while still holding SW1, release SW2+SW3. Wait until the EXE LED starts flashing again.

8. Transfer PPC \leftarrow PC

This function works only with the supplied KTERM.EXE terminal emulator. It will not work with any other terminal or terminal emulator program.

Pressing \leftarrow when this item is highlighted instructs KTERM to display a prompt for a .PAS, .BAS or .BIN filename, or a TAB key can be used to display a menu of such files. Pressing \leftarrow transfers the selected file to the PPC and stores it in the PPC’s filing system. See the KTERM chapter for full details of KTERM operation.

9. Transfer PPC \rightarrow PC

This function works only with the supplied KTERM.EXE terminal emulator. It will not work with any other terminal or terminal emulator program.

Pressing \leftarrow when this item is highlighted transfers the currently selected program from the PPC to the PC. At the PC the file is written either into the current directory, or as directed by the environment variable KTERMFILES=path.

PPC programs with the Pascal, Basic or binary language attribute are written at the PC as .PAS, .BAS or .BIN files, respectively. See the KTERM chapter for full details of KTERM operation.

10. Config & diagnostics

Pressing \leftarrow when this item is highlighted enters a sub-menu:

PPC/4 Configuration & diagnostics

```
Runtime port configuration
Config dump
Input hex dump
Serial Port tests
Printer config
Executive Mode Port 1 config
Set real-time clock (option)
Analytic Port tests (option)
```

10.1 Runtime port config

Pressing \leftarrow when this item is highlighted displays a screen which allows the power-up defaults for all PPC port parameters to be easily configured:

```

PPC/4 run-time serial port configuration

      baud  bits/      stop  RX handshake  TX handshake
      rate word  parity bits  RTS DTR XOFF  CTS DSR XOFF
Port 1:  9600   8   none   1    ON  ON  ON    ON  ON  ON
Port 2:  9600   8   even   1    ON  ON  ON    ON  ON  ON
Port 3: 115200  5   odd    1    ON  ON  OFF   ON  ON  OFF
Port 4: 38400   7   even   2    OFF OFF  ON    OFF OFF  ON

Use cursor keys to move around. SPACE or '-' or '+' to change settings.
ESC exits and saves any changes, '?' restores previously saved settings.
PORT 1 settings above apply only while a program is running.

```

As stated above, this function configures the **power-up** defaults. This means that if you run a PASCAL program which reconfigures e.g. a port's baud rate, then the new baud rate will remain until you either reboot the PPC, reset it, or power it down. (Ports cannot be reconfigured from BASIC).

If you have a two-port PPC (PPC/2) then the information for ports 3 and 4 will not be displayed.

Although all four ports function identically, there are some differences between their capabilities due to differences in internal hardware implementation. These are detailed in the PPC Serial Ports chapter, but in summary: Ports 3,4 support a wider range of baud rates etc; Port 1 cannot be set to 300 baud; the CTS handshake on ports 1,2 is permanently ON.



VERY IMPORTANT: the **Port 1** configuration defined in this function applies only when a program is running in the **autoexec** mode. If a program is running within the Executive (i.e. from **Compile & Run**) then the settings in the **Executive Mode Port 1 config** function apply instead.

10.2 Config dump

When this function is selected with \leftarrow a menu of PPC ports appears. This is followed by a menu of output devices; currently supported types are **ASCII**, **Postscript** and **HPGL**.

Pressing \leftarrow at this point outputs a text report of the PPC port configuration and other information to the selected output port. The report is in the following form:

```

PPC/4 run-time serial port configuration

      baud  bits/      stop  RX handshake  TX handshake
      rate word  parity bits  RTS DTR XOFF  CTS DSR XOFF
Port 1:  9600   8   none   1    ON  ON  ON    ON  ON  ON
Port 2:  9600   8   none   1    ON  ON  ON    ON  ON  ON
Port 3: 115200  8   none   1    ON  ON  OFF   ON  ON  OFF
Port 4: 38400   8   none   1    ON  ON  ON    ON  ON  ON

PPC/4 Port 1 Executive Mode baud rate config
Port 1:  9600   8   none   1    OFF OFF  ON    ON OFF  ON

ROM version: 1.02 01-JAN-93
Serial#: 1999
Options: none

```

The **ASCII** output mode outputs the report to an ASCII printer. Almost any printer can be used. The **Postscript** output mode requires any Postscript device to be connected to the selected port. The **HPGL** output mode works with most HPGL-compatible plotters and other output devices, regardless of paper size. The text of the report is plotted with Pen #1 which must be present.

10.3 Input hex dump

This function is very useful for determining exactly what characters are present in a data stream.

When this function is selected with \leftarrow a menu of PPC ports appears. Pressing \leftarrow at this point enters a mode in which any data which arrives at the specified port is displayed using a combination of hex and ASCII characters. The output is in the following form:

```
000000: 0D 0A 50 50 43 2F 34 20 72 75 6E 2D 74 69 6D 65  '..PPC/4 run-time'
000010: 20 73 65 72 69 61 6C 20 6F 72 74 20 63 6F 6E 66  ' serial port con'
000020: 69 67 75 72 61 74 69 6F 6E 0D 0A 0D 0A 20 20 20  'figuration.... '
000030: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20  ' '
```

To exit the input hex dump mode, press any key.



When using this function to view data flowing along a cable between two devices *neither of which is the PPC*, remember to connect only PPC's pins 3 (RX) and 1/7 (ground). Do not connect-up any PPC handshakes – doing so could cause contention. However, this means that only the first 255 bytes of the hex dump may be reliable.

10.4 Serial port tests

Pressing \leftarrow when this item is highlighted invokes a very thorough loopback test which checks all data, handshake and modem control lines on all two (PPC-2) or four (PPC-4) serial ports. This test requires the following:

- 1 The terminal–PPC cable must be the cable recommended in Serial Ports and Cables chapter. This already contains the required Port 1 loopback connections at the PPC end of the cable.
- 2 All PPC ports other than Port 1 must be fitted with special loopback adaptors. See the PPC Serial Ports chapter for wiring details of these adaptors.

If this test reports any errors, then either your loopback adaptors are faulty, or the PPC is faulty. The most likely cause is a port blown-up by overvoltage or by a powerful electrostatic discharge.

10.5 Printer config

Pressing \leftarrow when this item is highlighted displays a function which allows a printer initialisation string to be specified. This string is used only in the **Print Program (6.2)** function.

The string is displayed in hex only, but can be entered in a mixed format:

```
78          is a decimal number and gives "N"
#4E         is a hex number and also gives "N" (digits A-F must be UPPERCASE)
"N"         is a quoted character and also gives "N"
"AbC"      defines a 3-character string "AbC"
```

Note that with quoted characters or strings both single and double quotes may be used provided they are matched. This also facilitates the easy inclusion of a quote character *within* a string. The maximum length of the string is 20 bytes.

The following example defines the factory-set printer initialisation string, which is for the HP Laserjet II/III printer family:

```
#1B "E" #1B "(s16.66H" #1B "&18D"
```

10.6 Executive Mode Port 1 config

Pressing \leftarrow enters the configuration screen for **Port 1**. The settings herein apply **in EXECUTIVE mode only**. Only the baud rate may be altered: 1200–38400 baud. The other parameters are fixed at 8 bits/word, no parity, 1 stop bit, TX XON/XOFF and TX CTS enabled.



Very Important: Do not alter the baud rate away from **9600** baud unless you are using either a “dumb” terminal or a non-KTERM emulator. The supplied KTERM.EXE program is capable of changing the baud rate temporarily up to 38400 baud for the duration of the KTERM session, but it assumes a **9600** baud *initial* setting.

10.7 Set real-time clock (option)

Pressing \leftarrow when this item is highlighted enters a function which allows the time and date in the PPC real-time clock (RTC) option to be configured.

The suggested procedure is to set-up a particular time and, when that time is reached according to a reference clock, press **ESC** to load it into the RTC.



You must ensure that the day of week is set correctly for the date being entered. The PPC does not check it.



The lithium battery should last for around 20 years. If you have to replace it, or if it has been short-circuited for any length of time, run the reset (R) command. This re-initialises the RTC hardware.

If your PPC does not contain the factory-fitted RTC option, a message to that effect appears.

10.8 Analog Port tests (-A option)

Pressing \downarrow when this item is highlighted enters a function which displays the value returned by the eight analog-digital converter (ADC) channels AIN1-AIN9, plus the value returned by the on-board temperature sensor (channel 9), in the following format:

CH=1	CH=2	CH=3	CH=4	CH=5	CH=6	CH=7	CH=8	CH=9
-00005	+04726	-00001	-17756	-18188	-18144	-18284	-18543	+01978

In addition, the values read on ADC channels 1,2 are copied to the digital-analog converter (DAC) channels 1,2 where they can be measured with external equipment.

To interpret the above readings, use these formulae:

$$\text{volts} = \text{reading} / 32768$$
$$\text{temperature } (^{\circ}\text{C}) = \text{reading} * 2.5 / 2048$$

The calculated values are "uncalibrated" and will therefore not be precise to the full "16 bits". The formula for the temperature is subject to change.

If your PPC does not contain the analog subsystem option, a message to that effect appears.

11. Reboot the PPC

Pressing \downarrow when this item is highlighted has the equivalent effect to powering-down the PPC.



If you are running KTERM with a command-line-specified baud rate other than 9600 baud, you will need to exit KTERM (with alt-x), wait until the EXE LED starts flashing and then re-invoke it. This is because following a reboot Port 1 defaults to 9600 baud and is unaware of KTERM's higher baud rate.

PPC EDITOR

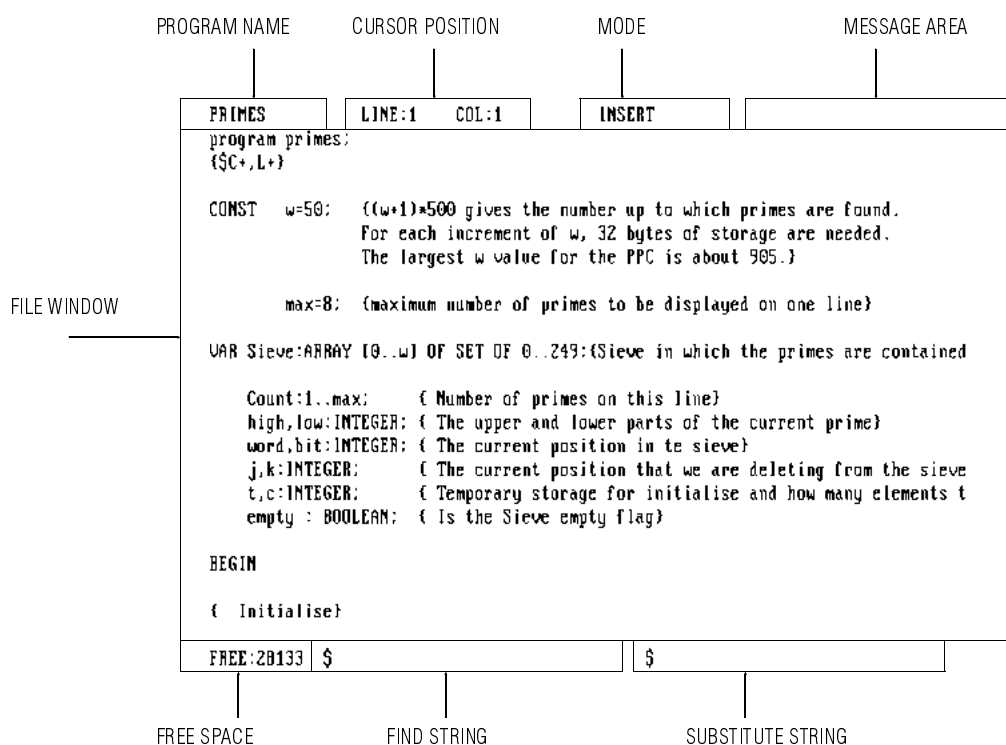
The PPC Editor is a full-screen interactive editor which allows programs of any size up to the PPC memory limit to be edited in the PPC, with an ANSI "dumb" terminal or with KTERM.

The editor uses mainly Wordstar® compatible commands and users familiar with Wordstar will feel at home right away. However, most editors and word processors available today share common features (such as block delete, copy, paste, etc) and you need only to learn a few keystrokes.

The editor is fully integrated into the PPC environment, allowing a rapid edit/compile/run program development cycle. It can edit both BASIC and PASCAL programs. It is a non-document editor, i.e. it does not perform any text justification.

Alternatively, the PPC editor can be bypassed altogether and files may be created on a PC with your favourite *non-document* editor, then transferred to the PPC.

The Look of the Editor



The screen is effectively a window into the file. This window may be moved in all four directions so that you can view any part of the file. Sideways scrolling is supported.

Initially, the editor assumes that a 25-line terminal is being used, and configures its screen layout accordingly. If KTERM is used, KTERM will tell the PPC the screen size of the current KTERM session and the editor will auto-configure to the new screen size. Within KTERM, the screen sizes can be 24 (basic mode), 42 (EGA with /VL) or 49 (VGA with /VL). These figures are one less than expected because of KTERM's permanent status line.

Typing Modes

There are two basic modes: INSERT and CHANGE. INSERT is the normal method of text entry. When you type a character in INSERT mode, all of the line to the right of the cursor is moved right one position before the character is inserted. The current line becomes longer by one character. In CHANGE mode, the new character overwrites the current character at the cursor and thus the line remains the same length. You are not allowed to go over the end of the line in CHANGE mode.

In general, INSERT mode is used to build up a file and CHANGE mode is used to alter small sections within a line.

Editor Commands

All editor commands start with a **control character**. This is optionally followed by additional characters. To enter a control character, type the character while holding-down the **CTRL** key. For example, to generate a **CONTROL-K** (also shown as **CTRL-K**): press and hold the **CTRL** key, press the **K** key (CAPS LOCK state is irrelevant), release the **K** key, and *only then* release the **CTRL** key.



Do not press **CTRL-S** while you are in any of the Executive functions (these include the editor, the compiler, BASIC, or a PASCAL program running from **Compile & Run**). **CTRL-S** is an XOFF which will stop all output to the terminal until a **CTRL-Q** (XON) is pressed.



If you are using KTERM and you exit KTERM (with **ALT-X**) while in the editor, any changes made to the file will be lost and, when you run KTERM again, you will be returned to the Executive.

Cursor-Moving Commands

The four cursor keys work as expected. In addition, the cursor can be moved with the following commands:

Character left/right

Use the ← → cursor keys

Word left/right

CTRL-A CTRL-F

Characters that constitute the boundaries between words are :-

" () [] { } = + - * / < > ^ ; :
, # \$ & \ [TAB] [CR] [SPACE]

Tab left/right

CTRL-O S CTRL-O D

Start/End of line

CTRL-Q S CTRL-Q D

With KTERM the **HOME** and **END** keys also perform the same function.

Line up/down

Use the ↑ ↓ cursor keys

After moving up or down one line, the cursor column remains fixed. Thus it may appear that the cursor is positioned beyond the end of a line. However, if any other key is pressed, the editor will behave as though the cursor was at the end of the current line. This behaviour prevents the cursor always "jumping" to the end of each line and makes editing programs much easier.

Top/Bottom of screen

CTRL-Q E CTRL-Q X

Move the cursor to the top/bottom of the screen.

Page up/down

CTRL-R CTRL-C

Start/End of file

CTRL-Q R CTRL-Q C

This moves the cursor to the start/end of the entire file.

Text Deleting Commands

Delete line

CTRL-Y

Delete the current line. Note that the line is placed into the editing buffer and can be recalled into the text by use of the CTRL-U restore line command. The deleted line will be overwritten when the user next makes a change to any line.

Delete last character

[BACKSPACE] or **CTRL-H**

Delete this character

DEL (not available on all ANSI terminals)

Delete word left/right

CTRL-O T CTRL-T

Delete from the cursor to the beginning of the last/next word. The characters that constitute the boundaries between are given under the Word left/right command above. Wraparound operates.

Delete to start of line

CTRL-Q [DEL]

Delete to end of line

CTRL-Q Y

Block Commands

The editor implements a simple block buffer scheme, where a deleted block is retained in a "block buffer". The block buffer is located in the memory area which remains free between the end of the file being edited and the end of memory. Therefore, the larger a file you are editing the smaller the block buffer will be.

No facilities are currently implemented for reading or writing blocks to the PPC filesystem. The only method of merging separate files, etc, is to transfer them to a PC (with KTERM), perform the desired operations with a non-document editor, and transfer the merged file back to the PPC.

Mark start/end of block **CTRL-K B** **CTRL-K K**

Place the block markers. A marker will be positioned at the cursor position. The markers are lost if the line containing the marker is edited.

Move block

CTRL-K V

Delete the currently marked block from the text and place in the block buffer, then insert the block at the cursor position. If there is enough space the block will be retained in the buffer, but the less space there is, the longer the command will take.

Copy block

CTRL-K C

Copy the currently marked block from the text to the cursor position.

Delete block

CTRL-K Y

Delete the currently marked block from the text and place in the buffer. The less space there is, the longer this command will take.

If the amount of free space is very small (less than 256) you are asked whether to abandon the block. If you press **Y** then the block will be deleted from the text and not placed in the block buffer. If you don't want to lose the block then **N** should be pressed.

Paste block

CTRL-K P

Insert at the cursor the block currently in the block buffer. The block remains in the buffer if there is sufficient space.

Quick Cursor Movement

Goto line

CTRL-Q I

You are prompted for a line number. [BACKSPACE] operates as expected and after ↵ the cursor is placed at the start of the line given.

Goto start/end of block

CTRL-Q B CTRL-Q K

Move the cursor to the start/end block marker.

Remember position

CTRL-K 0 (zero)

The current cursor position is stored. The marker is lost if the line in which it lies is subsequently edited.

Return to position

CTRL-Q 0 (zero)

The cursor is positioned at the previously marked position.

Go to start of file

CTRL-Q R or [HOME]

Go to end of file

CTRL-Q C or [END]

Find and Substitute

Find first

CTRL-Q F

This is really a “define Find and Substitute strings” command. The current Find string is displayed. ↵ retains the current string, otherwise you should type in the required Find string (up to a maximum of 32 characters) and then press ↵ .

[BACKSPACE] operates, CTRL-R will redisplay the previous string and CTRL-U will abort the operation.

A control code may be entered by pressing the control meta-key (CTRL-P) (see **Miscellaneous**) and then the control character (e.g. CTRL-P then ↵ enters a <CR> into the string). Pressing the meta-key and then ? will return a value which is displayed as ? and counts as a wild-char when in the Find string.

After ↵ is pressed, a prompt for the Substitute string appears. If you only require Find and do not want to Substitute anything, press ↵ . The cursor will be positioned at the start of the first occurrence of the Find string in the file.

Find next

CTRL-L

The file is searched for the next occurrence of the Find string starting from one character after the cursor. A wild-character in the Find string will match with any character at all in the file.

Substitute next

CTRL-O L

The file is searched, starting from the cursor, for the next occurrence of the Find string . A wild char in the Find string matches any character in the file. When the string is found, it is replaced by the Substitute string and the cursor is positioned after the last character of the Substitute string. Finally, the file is searched for the next occurrence of the Find string starting from the cursor.

Substitute all

CTRL-O A

Starting from the cursor, all occurrences of the Find string in the file are replaced by the Substitute string. A wild-character in the Find string will match with any character at all in the file. The cursor is then placed after the last string substituted.



You should **never** insert control characters into program files. They can cause errors which can be extremely difficult to find.



While a file is open in the editor, its lines end with a <CR>, not <CRLF>. Normally you do not need to know this, but it is relevant if you are searching for end-of-line characters.

Leaving the Editor

Exit and abandon

CTRL-K Q

You are asked whether to abandon the file. Pressing Y will cause a return to the Executive and the current text will be abandoned. Any other response will abort the command.

Exit and save

CTRL-K X

The file being edited is saved in the filing system. If there is insufficient space, an error message will appear. In such a case you must remove some text, or abandon the file.



Do not disconnect PPC power while a file is being saved. Doing so will almost certainly result in loss of data, and the loss will not be limited to the file you have been editing!

Toggles

Toggle insert mode

CTRL-V or **[INSERT]**

Switch between INSERT and CHANGE mode. A character typed in INSERT mode is inserted into the line. A character typed in CHANGE mode overwrites the current character. A <CR> cannot be overwritten in CHANGE mode.

Toggle auto indent

CTRL-O I

Auto indent only operates in INSERT mode. The message **INSERT** becomes **I/AUTO**. When indent is on and ↵ is pressed in INSERT mode, the next line will be indented so that it starts at the same column as the line above it.

Toggle free space display

CTRL-O F

A '*' following the amount of free space indicates that there is a block in the block buffer. This function is used to display the size of the block.

Miscellaneous

Deliver tab

TAB or **CTRL-I**

Tabs are fixed at 4-column stops and will be entered as a ASCII #9 code in the file and will not be changed to spaces. They are treated in the main like any other character in the file.

Restore (undelete) line

CTRL-U

If you are in the process of editing a line then this command will restore the line to what it was when you first positioned the cursor on it. If you are not in the process of editing a line then this command will insert in front of the current line, the last edited line.

This aspect of the command is useful because the DELETE LINE command places the deleted line into the line-buffer exactly as though it had just been edited. You may thus move a line from one place to another by deleting it, moving the cursor to the desired place and then issuing a CTRL-U command. The CTRL-U command can be issued multiple times to generate multiple (identical) lines which can then be edited.

Disk directory

CTRL-K F

This command displays the directory of all PPC files as they are currently saved in the filespace. It is provided for reference only.

Control meta-key

CTRL-P

Any key pressed after the meta-key is entered into the file as its literal value. This function may be used to enter control characters into the file that are normally commands or prefixes to commands (e.g. CTRL-P followed by 8 enters a CTRL-H).

The meta-key can also be used in the same way to enter control characters in the Find and Substitute strings. In this case if **?** is pressed after the meta-key a character is returned that is displayed as **?** but acts as a wild-character in the find string.



You should **never** insert control characters into program files. They can cause errors which can be extremely difficult to find.

Prompts and Messages

Prompts

There are four prompts produced by the editor which normally appear on the upper status line:

Abandon block: Sure?

This prompt requires a single character response. It appears if the user has issued the DELETE BLOCK command and there are less than 256 bytes free. If you respond **Y** then the block will be deleted and lost (note that the block is normally saved in the block buffer and thus not lost) while any other response will abort the command.

The prompt also appears if you have issued any command the execution of which would overwrite the block in the block buffer. If you respond **Y** then the block in the block buffer will be lost and the command executed, while any other response will abort the command.

Abandon file: Sure?

This prompt requires a single character response. It is produced after the Exit and Abandon command. Respond **Y** and the current text will be lost and you will be returned to the Executive. Any other response will abort the command.

Go to line:

This prompt requires a string of numbers terminated by ↵. It is produced after the GO TO LINE command. The response is interpreted as a line number and the maximum allowed length is 4 characters (only numbers are accepted). ↵ alone aborts the command.

Editor Error Messages

There are twelve messages produced by the editor and they appear mainly on the upper status-line:

Out of memory

Indicates that there is not enough space in the machine to carry out the proposed command.

Line is too long

Produced when the length of the line would exceed the maximum allowed length (255 characters) if the proposed action was taken. This might either be simply the press of a key, or the deletion of a <CR>.

Undefined command

Indicates that the initial key of a command is correct, but the second or subsequent keys do not form a valid command.

Block start unmarked/Block end unmarked

Produced after any block operation if the start/end of the block has not been marked or the mark has been lost (i.e. the line containing the mark has been edited).

Block marks reversed

Produced after any block operation if the start of the block occurs after the end.

Invalid destination

Produced after a MOVE BLOCK or COPY BLOCK and indicates that the destination (cursor) lies between the start and end of the block (inclusive).

No block in buffer

Produced after a PASTE BLOCK operation and is self-explanatory.

Marker lost

Produced after a RETURN TO POSITION command and indicates that the position marker has not been placed or has been lost.

No such line

Produced after the GO TO LINE command and indicates that the line number given is greater than the number of lines in the file.

Editor limits

The maximum line-length is 255 characters. Note that the cursor column number may exceed 255 due to tab and control characters (in which case the column number displayed on the status-line remains at 255).

The maximum file size is limited by available memory, which is approximately 29,000 bytes. It is perfectly possible to edit a file which – at any particular moment – is too large to be saved. Whenever a file's size is in this range, a '!' is displayed after the file size.

The maximum number of lines in the file is limited only by available memory but note that the line number display on the status-line is only of four digits (due to space considerations) and the GO TO LINE command can only reach line 9999 and no further. All other commands will work as normal if the number of lines exceeds 9999 (although note also that the average number of characters per line would have to be about three for the line numbers to exceed 9999).

PPC program files are stored as text lines terminated by a CRLF. There is no "end-of-file" marker. When a file is read into the editor, the LFs are stripped-out and, when the file is saved, they are inserted back in. This makes better use of the available memory. However, it is possible to edit a file which is too large to save *only when the LFs are inserted*. This is very unlikely to happen but, when it does, you will be advised the number of excess lines which must be deleted before the file can be saved.

In general, the maximum file size which can be edited is similar to the maximum file size which can be stored in the filespace, which in turn is similar to the maximum source file size which, when compiled, produces the maximum allowed amount of binary code. The various parts of the PPC have comparable capabilities.

Non-Printing Characters



You should **never** insert control characters into program files. They can cause errors which can be extremely difficult to find.

Characters of ASCII value less than 32 decimal (control characters) are treated as any other character. They may be entered into the file by pressing the control meta-key (CTRL-P) and then the control character desired.

The meta-key may also be used to specify control characters in the find and substitute strings in the same way as above. An obvious use for this feature is to find the end-of-line character (reached by [meta-key] ↵ or [meta-key]CTRL-M). The ? character when pressed after the meta-key returns FFh. This character is displayed as ? in the find and substitute strings, but is treated as a wild-character in the find string i.e. it will match with any character at all in the file.

If your terminal is capable of producing characters of value greater than 127 decimal, then these characters are entered as any other into the file and are displayed according to the capabilities of the terminal. With KTERM, this allows the entry of international characters (e.g. à). Note that such characters should be used only in quoted strings in e.g. WRITE statements; they must **not** be used in Pascal or Basic variable names.

PPC BASIC

What is PPC BASIC ?

BASIC is a simple and very popular easy-to-learn programming language developed in the 1960s which is provided in the PPC for users who need to write only relatively small programs and who perhaps have some knowledge of BASIC already.

The PPC BASIC is a subset of 'standard' BASIC known as Tiny Basic. The main difference between PPC BASIC and a full-featured BASIC language is that PPC BASIC supports **integer** variables only (no floating-point numbers or strings) and it supports only a core subset of the functions found in full-featured personal computer BASICs. However, it is adequate for simple character processing which is often the PPC's function. The reduced features also make PPC BASIC run much faster than most standard BASICs.

Performance

Like most BASICs, the PPC BASIC is an *interpreter* and it thus runs many times slower than PPC PASCAL which is a *compiled* language. One of the properties of an interpreted language is that a program containing many lines is likely to run slowly and although the PPC can accommodate BASIC programs hundreds of lines long, such programs are likely to run too slowly to be useful for most datacomms applications. For most high-performance applications a *compiled* language is essential and PPC PASCAL should be used instead. PPC PASCAL executes 20-50 times faster than PPC BASIC and also offers features such as floating-point and many others.

Example

The following simple PPC BASIC program will read characters from PPC port 3, replace every occurrence of the character "A" (65 decimal) with the character "a" (97 decimal) and output the resulting data on PPC port 4. The comments on the right are not part of the program.

10 REDIR(4)	Set destination for all subsequent output to port 4
20 X=WGET(3)	Wait for a character on port 3; load into variable X
30 IF X=65 X=97	If the character is "A", change it to "a"
40 PRINT CHR\$(X),	Output the character
50 GOTO 20	Repeat the above forever

The above should be self-explanatory, but note one important feature which most PPC programs will have in common: it **runs forever** and, unlike a program written e.g. for a personal computer, it never terminates. More programming examples are given at the end of this Section.

How to create a PPC BASIC program

A PPC BASIC program can be created either by entering PPC BASIC (the **Interactive Basic** option on the main EXECUTIVE menu) and interactively typing it in, or by using the PPC text editor. If the editor is used, an empty program of the desired name and language (Basic) must first be created with the **Create Program** option.

A BASIC program can also be "created" by creating it first on a PC with an editor, and then transferring it to the PPC. A .BAS file will, when transferred to the PPC, appear as a "Basic" file in the filing system.

How to execute (run) a PPC BASIC program

If you enter BASIC from the Executive, you must **LOAD** the required program, then type **RUN**. You may use the **DIR** command to see which BASIC programs are in the filesystem.

Alternatively, when you have finished the program and it is working as it should, you can designate it to be an 'autoexec' program (see the Executive chapter) and it will then execute automatically whenever the PPC is powered-up.

PPC BASIC language description

Below is a full list of PPC BASIC language features. Note that PPC BASIC is **case-sensitive**, and all parts of the program must be **UPPERCASE**. The only exceptions within a program are quoted strings (e.g. "ABCD") in **INPUT** or **PRINT** statements. Also, *direct* commands (such as **LIST**) may be in either case but the entire command must be in the same case (e.g. **LisT** is not allowed).

Numbers

All numbers are 16-bit integers and must always be in the range -32767 to +32767.

Variables

All variables are stored as 16-bit integers. Variable names are single letters in the range A-Z inclusive. In addition to the 26 possible A-Z variables, a single array called @ is provided which can hold up to 2000 integers and whose elements are numbered @**(0)** to @**(1999)**. All variables are set to zero whenever PPC BASIC is entered. However, variable values are retained when a program's execution is terminated, allowing the variables to be examined to assist debugging.

Functions

ABS (X)	returns the absolute value of X
RND (X)	returns a pseudo-random number between 1 and X inclusive
SEED (X)	sets a seed for the above pseudo-random sequence
GET (P)	returns a character from port P, or -1 if no character was waiting
WGET (P)	as above, but waits until a character arrives
ADC (C)	returns a reading from channel C of the analog-digital converter (ADC)
SWI (S)	returns the status (0/1) of the PPC front panel switch S

Arithmetic and Compare Operators

/	Divide. Note that with integers, 7/10=0 and 10/3=3
*	Multiply
-	Subtract
+	Add
>	Compare if greater than
<	Compare if less than
=	Compare if equal to
#	Compare if not equal to
>=	Compare if greater than or equal to
<=	Compare if less than or equal to

Be careful to ensure that all numbers always fall in the range -32767 and +32767. In particular, avoid overflow in intermediate results in calculations such as $X=A*B*C*D/E$.

The logical result of comparisons made with the compare operators is 1 if true or 0 if false.

Expressions

An expression is a grouping of numbers, variables and functions, with arithmetic or comparison operators between them. The value of an expression is evaluated from left to right. The precedence of operators is

* /	evaluated first
+ -	evaluated next
> < = >= <=	evaluated last

Parentheses can be added as required to alter the order in which an expression is evaluated. + and - signs can also be used at the beginning of any number or expression.

Statements

A 'statement' is a **line**, of BASIC program and consists of a line number between 1 and 32767 followed by one or more commands. While it is usual to have only one command per line, multiple commands separated by semicolons may be entered. A **GOTO**, **STOP** and **RETURN** command must be the last command in any line.

Program

A PPC BASIC program consists of one or more statements (lines). When the direct command RUN is issued, the line with the lowest line number is executed first, the next higher-numbered line is executed next, and so on. The **GOTO**, **GOSUB**, **STOP** and **RETURN** commands can be used to alter this sequence. Within a statement, execution of commands is from left to right. The **IF** command can be used to skip over the execution of all commands to the right of the **IF**.

In the PPC **Interactive Basic** mode, a program resides in *memory*. This applies whether it has been typed-in, or loaded with the **LOAD** command. A power loss or a PPC reset will cause that program to be lost. The **SAVE** command can be used to store the program in the PPC's filing system.

The execution of a program can be terminated with the **STOP** command, by reaching the end of the program, or with a ctrl-c, break key. However, ctrl-c, works only on programs running in the **Interactive Basic**, mode; it will not break out of a BASIC program running in autoexec mode. In other words: break checking cannot be disabled except by running the program in the "autoexec" mode.

If a BASIC program terminates due to a runtime error (e.g. an integer overflow or division by zero) then all variables are re-initialised and the program is restarted from the first line. There is no facility for ignoring runtime errors.

BASIC Functions

Functions are program elements which return a value. PPC BASIC currently contains the following functions:

ABS

This return the absolute value of an expression.

`X=ABS(-6)` returns X=6.

RND

This is a pseudo-random number function.

`X=RND(100)` returns a pseudo-random number between 1 and 100 inclusive in X

The command **SEED** can be used to set the seed for the pseudo-random number generator to a particular value, thus producing consistent results whenever the program runs.

GET

This function reads input data from a PPC port.

`X=GET(3)` returns a character in X (value 0-255) from port 3, or X=-1 if no character was waiting on port 3

The advantage of **GET** as compared to **WGET** (below) is that if no input character is waiting, the BASIC program can go away and do something else that may be useful.

WGET

This is similar to GET, but it waits until a character arrives on the specified port. The commands

```
10 X=GET(3)
20 IF X=-1 GOTO 10
```

are equivalent to

```
10 X=WGET(3)
```

The **WGET** command will cause the BASIC program to hang until data appears on the input!

ADC

This function reads the PPC 16-bit analog-digital converter (ADC). The ADC is available only if the ANALOG SUBSYSTEM option has been fitted, otherwise its execution will generate a fatal error.

`X=ADC(6)` returns a reading in X from channel 6 (AIN7) of the ADC

Returned values are in the range -32767 to +32767, corresponding to nominal input voltages -1V to +1V. ADC channel numbers are **ADC(1)** to **ADC(8)**, corresponding to analog connector pins marked AIN1-AIN8.

The 9th channel

```
X=ADC(9)
```

returns an integer which is proportional to the temperature of the analog module, for temperature compensation of ADC readings. However, due to the lack of floating-point support in PPC BASIC, little use can be made of this feature from within BASIC.

SWI

This returns an integer corresponding to the status of a PPC front panel switch. The value is 0 or 1 for switch not pressed or pressed, respectively. Switch number is 1/2/3. Example:

`X=SWI(3)` returns X=0 if switch SW3 is not pressed, otherwise X=1

BASIC Commands

PPC BASIC commands are listed below, with examples. Multiple commands separated by semicolons may be placed on the same line. A command may also be executed in a *direct* mode, by entering it *without a line number*. For example, entering the direct command

```
PRINT 25*6           (no line number)
```

will output the result of **150**. However, to build-up a multi-line program, each line must start with a line number. Program lines may be entered in any order; PPC BASIC will automatically insert each line into the correct sequence in the program according to its line number.

Each command name below (in **bold**) is followed by its description and one or more examples:

REM

This allows remarks (comments) to be inserted into a program:

```
REM This line is a comment and is ignored by BASIC
```

LET

This *optional* keyword precedes an assignment of a value to a variable:

```
LET X=25*6/5         these two examples are equivalent
X=25*6/5
```

PRINT

This is a multi-purpose command for outputting the values of BASIC variables, expressions and text strings. It can be used in many different ways, as the following examples show:

```
PRINT                outputs just a blank line (a CRLF - a carriage-return and a line-feed)
PRINT "TOTAL=",X     outputs TOTAL= followed by the decimal value of X, and a CRLF
PRINT "TOTAL=",X,    as above, but no CRLF is output at the end, allowing another PRINT statement to
                    continue output on the same line
```

Both single and double quotes may be used provided they are matched.

The numeric values which are output can be positioned on specified boundaries using the # operator:

```
PRINT #2,X,#4,Y      outputs the values of X and Y, with X right-justified in a 2-character cell and Y
                    right-justified in a 4-character cell; if the values are too long to fit into the cells
                    they are output anyway. Default cell size is 6 chars.
```

The PRINT command supports expressions:

```
PRINT "TOTAL1= ",X*45," TOTAL2= ",Y*2*X
```

Two special operators HX2 and HX4 are provided for outputting numeric values in hexadecimal:

```
PRINT HX2(X),HX4(Y)  outputs the values B3 and 43AF (if X=179 and Y=17327 decimal). Note that
                    -17327 (decimal) outputs as BC51.
```

The CHR\$ operator outputs a single character corresponding to the ASCII value of the variable:

```
PRINT CHR$(X)        outputs the character M if X=77 decimal
```

IF

This conditional tests the value of an expression and, if the value of the expression is TRUE (i.e. non-zero), executes the commands to the right of the expression. Two simple examples are:

```
IF A=22 X=30
IF A=22 X=30; GOTO 30
```

Note that the word THEN used by other BASICs is not used. Multiple logical expressions, such as

```
IF (A=22) AND (B=43) PRINT "both true"
```

are not supported. However, noting that in PPC BASIC the value FALSE is represented by 0 and TRUE is represented by 1, we can achieve the above with

```
IF ((A=22)+(B=43))=2 PRINT "both true"
```

which takes advantage of the fact that A=22 will produce a result of 1 (if TRUE) and B=43 will also produce a result of 1 (if TRUE). If they are *both* TRUE, their sum will be 2.

GOTO

This command transfers execution to the line number following the **GOTO** command. Note that the **GOTO** command must be the last command on a line; it cannot be followed by a semicolon and other commands.

GOSUB RETURN

GOSUB is like a **GOTO** but produces a subroutine call to a subroutine at the specified line. This subroutine must return with a **RETURN** command, otherwise a **STACK OVERFLOW** fatal error will eventually result.

```
GOSUB 100          standard usage
GOSUB 100+(X*10)  line number computed at runtime
```

The last example above allows one of several subroutines to be called depending on the value of X being 0, 1, 2 etc, and assumes the *corresponding* subroutines start at line numbers 100, 110, 120 etc. A fatal **MISSING SUBROUTINE** error will result if X somehow acquires a value which is out of range, so use this feature with great care. Subroutine calls can be nested to a depth limited only by stack space, currently around 50 levels.

FOR NEXT

These commands are used to execute a part of a program a specified number of times.

```
10 FOR X=0 TO 100 STEP 5
20 PRINT "X=",X," X*5=",X*5
30 Y=Y+X
40 NEXT X
```

In the above example, lines 20 and 30 will be executed with X values of 0, 5, 10, 15 etc. Execution of a **FOR** loop continues until the upper limit is met or exceeded. Negative step values may be used. If the **STEP** value is omitted

```
10 FOR X=0 TO 100
```

then the step is assumed to be +1. Variable **FOR** parameters

```
10 FOR X=A+3 TO B-7 STEP C*5
```

are supported and the expressions are evaluated only *once* before the loop starts, so there is no huge overhead in using this method. **FOR** loops can be nested to a depth limited only by stack space:

```
10 FOR X=0 TO 10
20 FOR Y=0 TO 10
30 PRINT X,Y,X*Y
40 NEXT Y
50 NEXT X
```

Be careful to **not** inadvertently modify the value of the loop variable with one of the operations within the loop. While this can be done, it is a very poor programming practice:

```
10 FOR X=0 TO 100
20 Y=Y+1
30 IF Y=250 X=300
40 NEXT X
```

In the above example, X is potentially modified in line 30 and because its then new value (300) exceeds the upper limit (100) the loop will terminate prematurely. This can produce loops which *never* terminate!

REDIR

This command specifies the PPC port to which all **output** data will be sent, until changed with another **REDIR** command. For example, the command

```
REDIR(3)
```

redirects all output to Port 3.

INPUT

This command allows direct user input of decimal numbers. The command

```
INPUT X
```

prints

```
X:
```

and waits for a number (on an expression) to be entered, followed by the ENTER key. The variable X will then be set to the value of the number or the expression. Multiple items may be read in a single **INPUT** command:

```
INPUT X,Y,Z
```

and BASIC will prompt for each one separately.

You can modify the standard **INPUT** prompt by including a string:

```
INPUT "Enter value of X" X
```

and the prompt **X:** is replaced with

```
Enter value of X:
```

Both single and double quotes may be used, provided they are matched.

The INPUT command is designed for interactive operation and is not suitable for reading e.g. numbers from a continuous input data stream.

DAC

This command outputs a value to the specified channel of the PPC 16-bit digital-analog converter (DAC). The DAC is available only if the ANALOG SUBSYSTEM option has been fitted, otherwise its execution will generate a fatal error.

```
DAC(C,X)    outputs value X on channel C of the DAC
```

The channel number is 1-2 in present 2-channel DAC versions, but could vary in the future. The output value can range from -32767 to +32767, corresponding to a nominal analogue output range of -10V to +10V. The command

```
DAC(1,16384)
```

generates a +5V output on channel AOUT1 (the first DAC channel).

SEED

This specifies a seed for the pseudo-random number generator. Any value between 1 and 32767 may be used. Example:

```
SEED(155)
```

This command can be used to initialise the random number generator, to ensure that repeated runs of a program that invokes the **RND** function produce the same results.

STOP

This command stops the execution of a program. It may appear many times in various parts of a program, but must be the *last* command on a line.

Direct Commands

The commands below can be executed only in *direct* mode, i.e. by themselves and not as part of a program. They must not be preceded by a line number. Direct commands may be entered in uppercase or lowercase, but the entire command must be in the same case.

Direct commands are applicable only in the **PPC Interactive Basic** mode.

HELP

This command lists all currently implemented BASIC commands and functions.

RUN

This command starts the execution of the current program in memory, starting at the first line.

LIST

This command prints out the current program. Partial listings are possible:

LIST 200 prints out the current program from line 200 onwards.

NEW

This command deletes the current program from memory, allowing a new program to be typed-in. It does **not**, delete anything from the PPC filing system.

SAVE

This command stores the program currently in memory in the PPC's filing system. For example, the command

SAVE PROG1 stores the program as "**PROG1**" with a language attribute of BASIC. The program can then be edited with the PPC editor, downloaded to a PC, etc. See the PPC Executive chapter for more information.

LOAD

This is the opposite of **SAVE**. The command

LOAD PROG1 loads the BASIC program PROG1 into memory, ready for a RUN command, etc.

DIR

This command lists the current PPC filing system directory, for reference.

KILL

This command **deletes a file** from the PPC filing system. For example, to delete the program saved with the above **SAVE** command:

KILL PROG1

The main purpose of this command is to make room for a new program to be **SAVED**, if you have already filled-up the PPC file directory. Exiting BASIC to delete a file would cause the BASIC program which is currently in memory to be lost.

BYE or SYSTEM

Either of the above commands returns to the PPC EXECUTIVE. Any program currently in memory which has not been **SAVED** is lost.

Programming Examples


The following program is equivalent to the translation-table-based example in the Pascal Programming Examples section. It translates characters received on Port 2 and transmits the output on Port 3:

```
10 FOR I=0 TO 255
20 @(I)=I           Initialise translation table for 1:1 translation
30 NEXT I

40 @(65)=122       Set-up any special character translations
41 @(0)=63
42 @(7)=42
43 @(35)=156
44 @(135)=102
45 @(136)=103
46 @(137)=116
47 @(138)=107
48 @(139)=121
49 @(140)=109

50 REDIR(3)       All output will go to Port 3

60 C=WGET(2)      Get char from Port 2
70 C=@(C)        Translate it through the table
80 PRINT CHR$(C), Output it
90 GOTO 60       Repeat forever
```

 The above program, like all PPC PASCAL programs, will terminate execution if a control-c char (3 decimal) is received on Port 1 – unless the program is running in **autoexec** mode.

Speed tip

For maximum execution speed, the above program can be equivalently written as:

```
5 GOTO 10
6 PRINT CHR$(@(WGET(C))),;GOTO6
10 (above lines 10...50 go here)
60 GOTO 6
```

The above version is faster partly because the main program loop is on a *single* line, and partly because it is near the start of the program. The fewer lines a program has, the less work the interpreter has to do. Also, the nearer a loop is to the start of a program, the fewer lines the interpreter has to search through to find a line number specified by GOTO or GOSUB.

However, whatever you do to speed-up *interpreted* Basic programs, *compiled* Pascal or C will generally be 10–100 times faster.

PPC PASCAL

What is PPC PASCAL ?

PASCAL is a powerful high-level programming language developed and standardised in the 1970s. The PPC PASCAL compiler provides a close implementation of ISO standard PASCAL and is suitable for writing all programs ranging from the simplest to the most sophisticated. Many extra functions and procedures not found in standard PASCAL are included to reflect the special PPC environment and applications.

PPC PASCAL is a compiled language. The compiler generates machine code directly with no intermediate stages. The code is very efficient and even complex programs can achieve sustained data rates up to 115,200 baud. Even the most complex protocols found in the RS-232 environment can usually be implemented at full speed.

Example

The following simple PPC PASCAL program will read characters from PPC port 3, convert all lowercase characters to uppercase, and output the resulting data on PPC port 4.

```
program convert;
var c:char;
begin
  repeat
    bread(port3,c);
    c:=toupper(c);
    bwrite(port4,c);
  until false;
end.
```

Note one important feature which most PPC programs will have in common: it **runs forever** and, unlike a program written e.g. for a personal computer, it never terminates. The `until false` statement produces a loop with no exit. More examples are given in the Programming Examples section and on the supplied diskette.

How am I going to learn PASCAL ?

While there are sufficient examples in this Manual to illustrate the major elements of the language and to get you started, it is beyond the scope of this Manual to teach you all of PASCAL. If you are completely new to PASCAL, please refer to the Recommended Books section.

However, please remember that you are using the PPC to do a job. You are not trying to pass an examination in computer programming. Therefore, keep your program simple, and stay away from the more exotic Pascal language constructs unless you are into that type of thing. Nearly all datacomms-related PPC programs can be easily written without using records, pointers, and all the other weird language features which can be found in the Pascal books.

How to create, edit and compile a PASCAL program

Use the Create File function in the PPC EXECUTIVE, and then use the PPC text editor to enter the program. Alternatively, a program can be typed-in on a PC (using any non-document editor or word processor) and then be uploaded to the PPC. In the PPC EXECUTIVE, using the => directory file selector, select the desired PASCAL program. Choose the Compile function. If errors are found, use the editor to correct them. When a program compiles with no compilation errors, you can proceed to the Compile and Run function and run the program.

When you have finished the program and it is working as it should, you can designate it to be an autoexec program (see the PPC Executive chapter) and it will then execute automatically whenever the PPC is powered-up.

PPC PASCAL - Introduction

PPC PASCAL is a very powerful tool that enables you to build highly structured and easy to understand programs that run *very quickly*. However, as with any computer language, it is going to take you some time to get used to using PASCAL. Therefore, we strongly recommend that you adopt the following procedure when using PPC PASCAL for the first time:

- 1 Work through the entire Get Started chapter.
- 2 Read the PPC Editor chapter and work through the Editor tutorial.
- 3 Read the rest of this chapter, trying out the "quick program" below until you understand how a PASCAL program is created, compiled and executed.
- 4 Repeat the above steps until you feel comfortable using the editor and compiling/running a PASCAL program.

A Quick Program

First, we will create a trivial program called TEST1. On the main Executive menu, select the Create program function and enter the name TEST1 with language PASCAL. This creates an empty program which will appear in the directory with size=0. Return to the main menu with ESC. You will notice the => marker against the newly-created program. Now enter the editor by selecting the Edit program item and type in the program text below. Note that the indents are not important but they do make programs more readable. Use the TAB key to achieve the indents.

```
program test1;
var i:integer;
begin
  i:=0;
  repeat
    write(port1,ii);
    i:=i+1;
  until i>5;
end.
```

You may have noticed the program contains a deliberate mistake. Save the program and exit the editor (with ctrl-k x) and select the Compile only function. This will compile the program and will display the following compilation listing and an error message:

```
8900  1 program test1;
8900  2 var i:integer;
8906  3 begin
8909  4     i:=0;
890F  5     repeat
890F  6         write(port1,ii);
      *ERROR*                ^
Undeclared identifier
ESC -> EXECUTIVE, C -> continue compilation, E -> Editor:
```

The error was that you used an identifier *ii* instead of *i*. Note that the compiler has marked a point in the program *after* the exact error location; this is often the case. To correct the error, re-enter the editor by pressing **E**. Back in the editor, the cursor is automatically placed for you at the point in the program where the error was spotted. To correct it in this case, move the cursor over one of the extra *i* characters and press DEL. Save the edited program and exit the editor with ctrl-k x and re-compile with the Compile and Run function. The following correct output will appear after the successful compilation listing:

```
0 1 2 3 4 5
```

Special features of PPC PASCAL

The term "standard PASCAL" used in this Manual refers to the original PASCAL language as defined by its inventor N. Wirth and as generally taught in educational establishments (PASCAL is a popular teaching language). However, the original PASCAL was not very useful on the smaller personal computers which have appeared since the late 1970s, mainly because it had little support for the highly interactive input and output and other useful functions available on those computers. This has led to various "dialects" of PASCAL and today (1992) arguably the most popular personal computer version is Borland's Turbo Pascal which has a large range of language extensions particular to the IBM PC platform.

PPC PASCAL is best described as "standard PASCAL" with PPC-specific extensions and with a few of the "standard PASCAL" features not implemented.

PPC PASCAL extensions have little in common with the extensions in Turbo Pascal and Turbo Pascal programs which make full use of special features of Turbo Pascal are not portable to PPC PASCAL without significant changes.

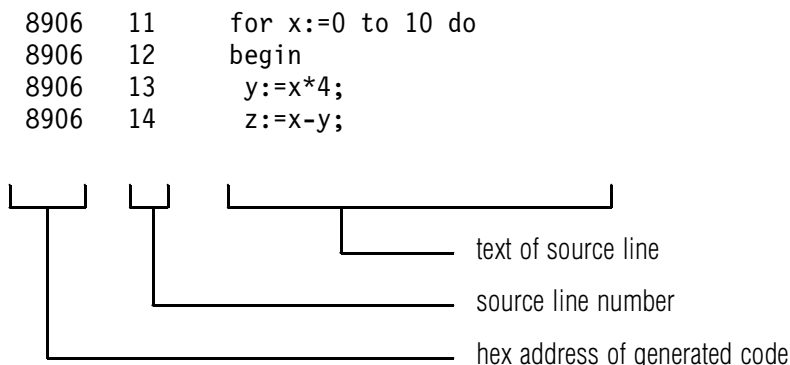
The areas where PPC PASCAL differs from "standard" PASCAL are:

- 1 PPC-specific extensions. These are special functions and procedures which support the special features of the PPC and they are all documented in the PPC PASCAL Syntax and Semantics section later in this chapter.
- 2 Compiler options. These are commands embedded in the program which control compiler operation. See the Comments and Compiler Options section.
- 3 Unimplemented "standard PASCAL" features. These are mainly in the I/O area, where the standard PASCAL file variables and functions (e.g. EOF) etc are irrelevant in the PPC's stream I/O environment. Unless otherwise stated, the implementation of PPC PASCAL is as specified in the Pascal User Manual and Report Second Edition (Jensen/Wirth).

The PPC is an I/O-intensive device and PPC PASCAL programmers must understand how the I/O functions work. In particular, users with previous PASCAL experience should study the Programming Examples section which contains examples of reading character-oriented data.

Compiler Listing

The compiler generates a listing of the form:



The source text is listed without modification, with any tabs expanded with the standard PPC tab expansion of 4-character tab stops.

The address is a 16-bit hexadecimal address and is relevant only if runtime errors occur, when the error message reports the approximate address at which the runtime error occurred.

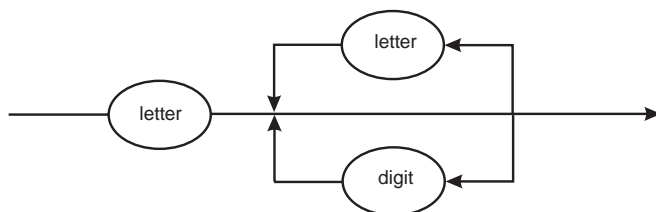
If the program terminates incorrectly (e.g. without END.) then the message **No more text** appears and control is returned to the Executive.

If no errors are detected and the program has been compiled with the Compile and Run function, the resulting binary code is immediately executed.

PPC PASCAL Syntax and Semantics

This section details the syntax and the semantics of PPC PASCAL. Unless otherwise stated the implementation is as specified in the Standard Pascal reference Pascal User Manual and Report Second Edition (Jensen/Wirth).

IDENTIFIER



Only the first 8 characters of an identifier are treated as significant. These first 8 characters should not make up a Reserved Word (see the Reserved Words section).

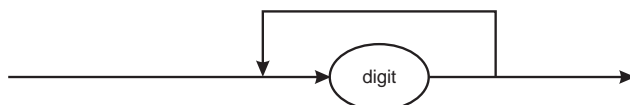
Identifiers may contain lower or upper case letters. Lower case is normally converted to upper case so that the identifiers HELLO, HELLo and heLlo are all the same. Reserved Words and predefined identifiers may be in either case.

The U+ compiler option, when used, causes the compiler to treat upper and lower-case letters as distinct so that HELLO, HELLo and heLlo are all different.

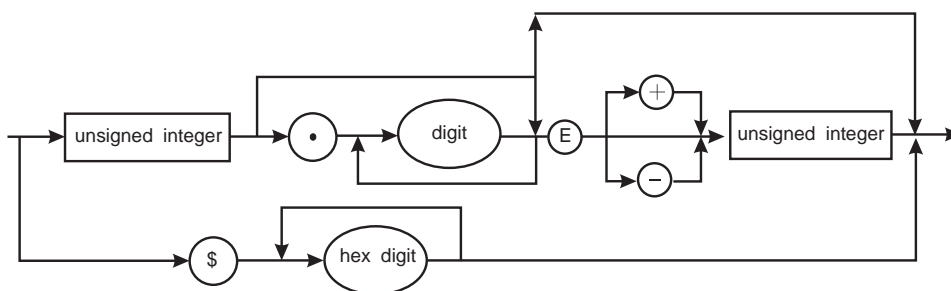


When you use the U option, you must ensure that all reserved words and predefined identifiers are in UPPERCASE. See the Compiler Options section for more information.

UNSIGNED INTEGER



UNSIGNED NUMBER



Integers have an absolute value less than or equal to 32767 (MAXINT) in PPC PASCAL. Larger whole numbers are treated as reals.

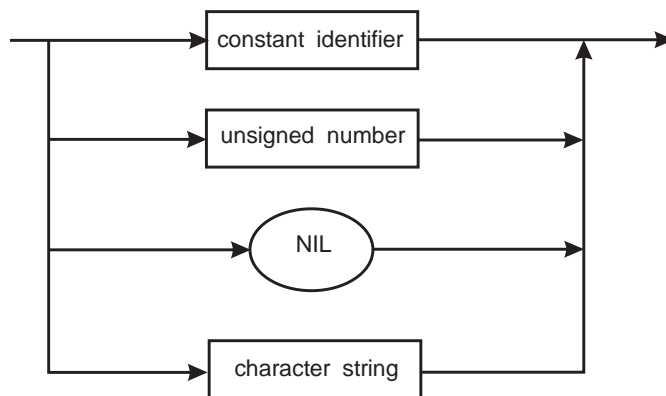
The mantissa of reals is 23 bits in length. The accuracy attained using reals is therefore about 7 significant figures. Note that accuracy is lost if the result of a calculation is much less than the absolute values of its arguments e.g. 2.00002 - 2 does not yield 0.00002. This is due to the inaccuracy involved in representing decimal fractions as binary fractions. It does not occur when integers of moderate size are represented as reals e.g. 200002 - 200000 = 2 exactly.

The largest real available is 3.4E38 while the smallest is 5.9E-39. There is no point in using more than 7 digits in the mantissa when specifying reals since extra digits are ignored except for their place value.

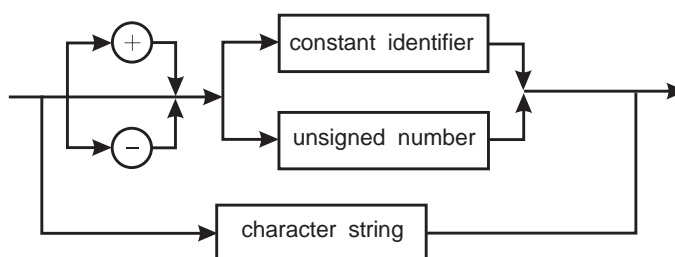
When accuracy is important avoid leading zeroes since these count as one of the digits. Thus 0.000123456 is represented less accurately than 1.23456E-4.

Hexadecimal numbers are preceded by the \$ character. Note that there must be at least one hexadecimal digit present after the \$, otherwise an error message will be generated.

UNSIGNED CONSTANT



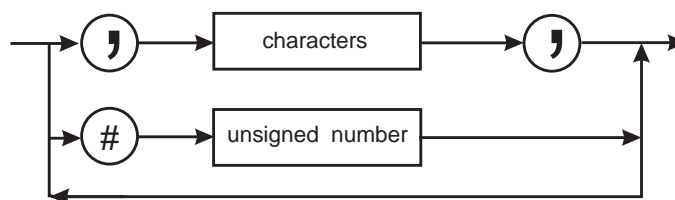
CONSTANT



Constants may be specified with the # character, which may be optionally followed by a \$ to denote a hexadecimal number. The two example lines below are equivalent.

```
CONST bs=#8; cr=#13;
CONST bs=#$8; cr=#$0d;
```

CHARACTER STRINGS



A "string" is an array of characters defined as

```
S : ARRAY [1..N] OF CHAR
```

where N is an integer between 1 and 255 inclusive. Strings defined as above can be initialised or manipulated with standard constructs such as:

```
WRITE(port1,S); S:='hello'; IF S='hello' THEN...
```

Literal strings should not contain end-of-line characters (CR, #13) - if they do then an error message is generated. Strings may not contain more than 255 characters.

The characters available are the full expanded set of ASCII values with 256 elements. To maintain compatibility with Standard PASCAL the null character is *not* represented as " (i.e. two single quotes with nothing in between); instead #0 should be used.

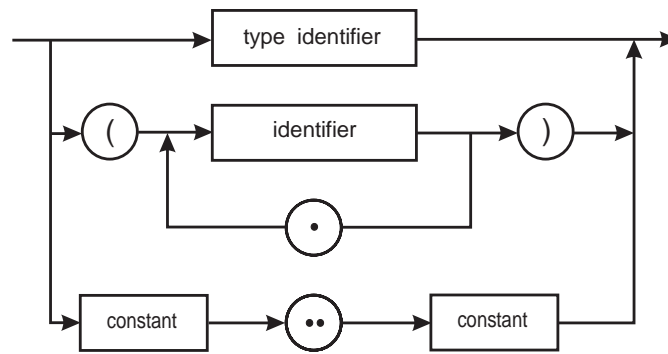
The # construct gives the corresponding ASCII character. There may not be spaces or commas between multiple components of a character string; the two lines below are not equivalent:

```
write(port1,#7 'hello')    is invalid
write(port1,#7'hello')    outputs a BELL (a beep) followed by  hello
```

In the second example line above, the character #7 forms a part of the quoted string, and an equivalent statement would be

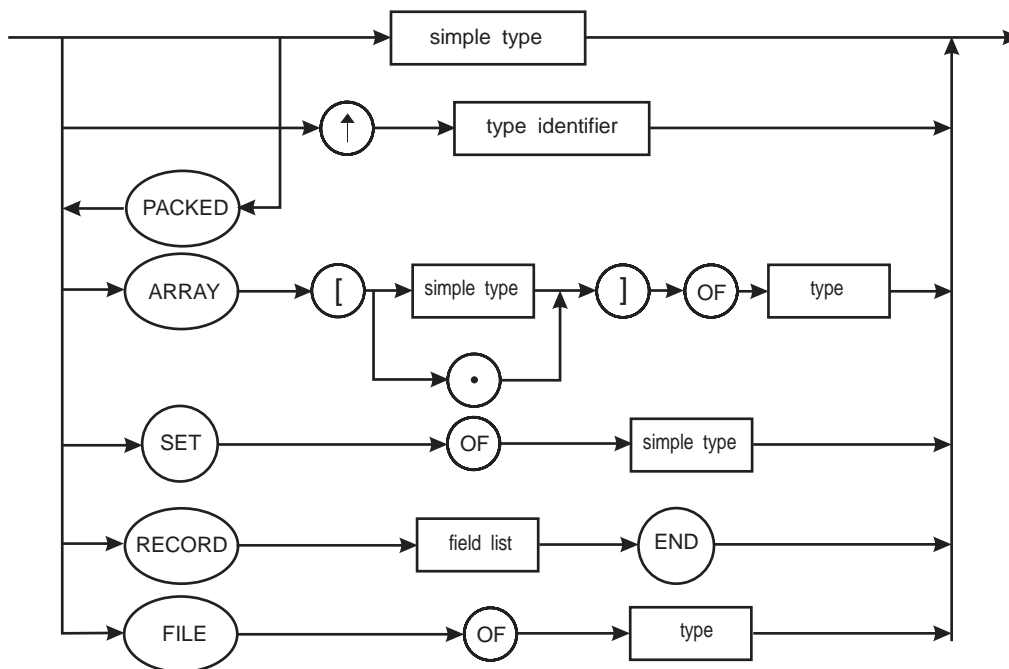
```
write(port1,#7,'hello');
```

SIMPLE TYPE



Scalar enumerated types (identifier, identifier,) may not have more than 256 elements.

TYPE



The reserved word PACKED is accepted but ignored since packing already takes place for arrays of characters etc. The only case in which the packing of arrays would be advantageous is with an array of Booleans but this is more naturally expressed as a set when packing is required.

ARRAYS and SETS

The base type of a set may have up to 256 elements. This enables SETs of CHAR to be declared together with SETs of any user-enumerated type. Note, however, that only subranges of integers can be used as base types. All subsets of integers are treated as sets of 0..255.

Full arrays of arrays, arrays of sets, records of sets etc. are supported.

Two ARRAY types are only treated as equivalent if their definition stems from the same use of the reserved word ARRAY. Thus the following types are not equivalent:

```
TYPE tablea = ARRAY[1..100] OF INTEGER;
    tableb = ARRAY[1..100] OF INTEGER;
```

So a variable of type `tablea` may not be assigned to a variable of type `tableb`. This enables mistakes to be detected such as assigning two tables representing different data. This restriction does not hold for the special case of arrays of a string type, since arrays of this type are always used to represent similar data. See the Strong TYPEing section for more information.

Pointers

PPC PASCAL allows the creation of dynamic variables through the use of the Standard PASCAL procedure NEW (see the Predefined Identifiers section). A dynamic variable, unlike a static variable which has memory space allocated for it throughout the block in which it is declared, cannot be referenced directly through an identifier since it does not have an identifier; instead a pointer variable is used. This pointer variable, which is a static variable, contains *the address of* the dynamic variable and the dynamic variable itself is accessed by including a `^` after the pointer variable. There are some restrictions on the use of pointers within PPC PASCAL:

Pointers to types that have not been declared are not allowed. This does not prevent the construction of linked list structures since type definitions may contain pointers to themselves e.g.

```
TYPE item = RECORD
    value : INTEGER;
    next : ^item
END;
link = ^item;
```

Pointers to pointers are not allowed. Pointers to the same type are regarded as equivalent, e.g. following the above declaration with

```
VAR first : link;
    current : ^item;
```

is valid. The variables `first` and `current` are equivalent (i.e. *structural* equivalence is used) and may be assigned to each other or compared.

The predefined constant NIL is supported and when this is assigned to a pointer variable then the pointer variable is deemed to contain no address.

FILES

All I/O in PPC PASCAL is done with file reads and file writes. The character-based stream I/O environment in which the PPC operates is different from the environment on a PC (i.e. reading and writing disk-based files) and it is therefore in the I/O area where PPC PASCAL differs the most from Standard Pascal. There are important concepts which you must understand about PPC file I/O:

- 1 Since all input is byte-serial (i.e. comes in one byte at a time) there is in general no way of detecting the end of an input "file", of calculating its total size, or accessing data at random in different parts of the file. This may seem obvious but needs to be stated.
- 2 The *type* of a PPC PASCAL "file" (i.e. the interpretation of its content) is *whatever you want it to be*. If you output *non-printable binary* data (e.g. with BWRITE), then that file in effect becomes a *binary* file. If you output *text-only printable data* (e.g. with WRITE or WRITELN), the file will in effect become a *"text"* file. The same applies to reading. This is contrary to Standard Pascal where files may contain printable data only and which does not allow you to input or output arbitrary binary data. It is also contrary to the extensions provided in many Pascals which support the reading and writing of various binary variables.
- 3 There is no support for "file" operations on the PPC filing system. All file I/O applies to its external data I/O ports only. Non-volatile data storage is possible, and is supported with the NVREAD, NVWRITE etc procedures.

- 4 All PPC file I/O is done only with the following routines:
- | | |
|----------------|--|
| READ, READLN | Standard PASCAL procedures for reading "text" data |
| BREAD | Special PPC PASCAL extension for reading "binary" data |
| WRITE, WRITELN | Standard PASCAL procedures for writing "text" data |
| BWRITE | Special PPC PASCAL extension for writing "binary" data |
- 5 All I/O in the PPC is buffered. If a program reads a few bytes of data from the input and then goes away to process the data, any additional input data which arrives during the processing will not be lost *provided the correct handshakes have been enabled*. These buffers are fully interrupt-driven (run in the background) and are therefore transparent to PASCAL, but they do result in smoother data flow and more efficient operation. For full details of the buffering see the PPC Ports chapter.

Some of the Standard PASCAL file accessing constructs are meaningless in the PPC environment and these have therefore been deleted:

INPUT	OUTPUT	TEXT	RESET	REWRITE
GET	PUT	EOF	f^	

Because (as described earlier) PPC PASCAL files contain whatever printable or non-printable data you want them to contain, all file type declarations like

```
file1 : FILE OF CHAR; file2 : FILE OF INTEGER; file3 : TEXT;
```

are not used. It *is* possible to read or write chars, integers, floats, etc but this is achieved by using the appropriate BREAD or BWRITE operations.

Only four filenames are allowed in PPC PASCAL and these correspond to the four PPC I/O ports:


PORT1	PORT2	PORT3	PORT4
-------	-------	-------	-------


and all four are already opened for both reading and writing when a program starts execution.

The above port names are internally predefined as constants of type TEXT and this allows the writing of port-independent I/O procedures to which the port number can be passed as a parameter. The following example shows a procedure for positioning the cursor on an ANSI terminal, and its invocation with the x,y values of 75,23:

```
procedure cursorpos(p:text;x,y:integer);
begin
  write(p,#$1b'[,y:1,';',x:1,'H'); { ':1' disables trailing spaces }
end;
cursorpos(port2,75,23);
```

Note that on the two-port PPC version (PPC-2) the files PORT3 and PORT4 are not defined.

 Contrary to Standard PASCAL, there are no predefined INPUT and OUTPUT files. All file I/O statements must specify a port; WRITE and WRITELN are the only exceptions and they assume PORT1 as the default.

 If you execute a READ, BREAD, READLN statement and there is no input data, the program will wait until the expected number of bytes arrives; in other words the program might appear to hang! In most cases this will not matter since in the absence of input data the program may not have anything to do anyway but, if desired, a "hang" can be prevented by testing the I/O buffer status with functions provided.

 The **destination** (e.g. "port1" or an address) **is mandatory** in write, writeln and bwrite statements. This is to allow the alternative construct which redirects output to an array of char and which is extremely useful.

The predefined type TEXT in PPC PASCAL has no connection with files of type TEXT etc in Standard PASCAL. As the type TEXT is an enumerated type with the predefined constants PORT1, PORT2, PORT3, PORT4 you can use items of type TEXT in record definitions etc; something which most PASCAL compilers will not allow.

Also missing from the PPC is the now-irrelevant convention in some PASCALS that there should be an *initial* blank line (EOLN=TRUE) when reading with READLN.

The following examples of file I/O briefly demonstrate how anything can be read from or written to a file simply by choosing the appropriate input or output statement:

var n:integer;	
c:char;	
read(port2,n);	expects one integer in ASCII text form (e.g. "12347")
readln(port2,n);	as above but expects an end-of-line (CR) after the number
bread(port2,n);	as above but reads the two binary bytes that constitute an integer
bread(port2,c)	reads just one byte - the most useful PPC input function

```

n:=12347;
c:='A';
write(port2,n);      outputs the integer in ASCII text form (e.g. "12347")
writeln(port2,n);   as above but follows it with a CRLF character pair
bwrite(port2,n);    outputs the integer as two binary bytes (3Bh,30h in this case)
bwrite(port2,c);    outputs just one byte (41h in this case) - very useful

```

For full details of the above file I/O functions, see the READ, WRITE etc information in the Procedures and Functions section.

RECORDS

The implementation of RECORDs, structured variables composed of a number of constituents called fields, within PPC PASCAL is as Standard Pascal except that field identifiers share the same symbol table space as record names and other identifiers, meaning that a declaration

```
TYPE ListOfTypes = (Insect, Frog, Fish, Mammal);
```

```

ANIMAL = RECORD
  LEGS : INTEGER;
  NAME : ARRAY [1 .. 20] OF CHAR;
  ANIMAL : ListOfTypes;
END;

```

will cause an error as ANIMAL is used both as the type name and as a field identifier. Likewise,

```

TYPE one = RECORD
  x, y : INTEGER
END;

two = RECORD
  z, y : INTEGER;
END;

```

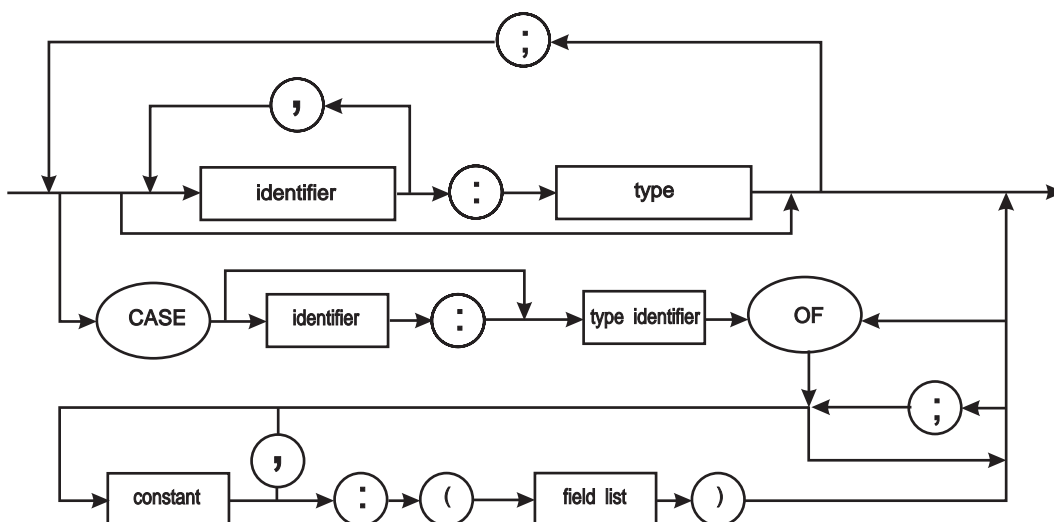
will cause an error as y exists in both records. However, any program which duplicates names in this way would be very confusing!

Records may have variant parts as per Standard PASCAL.

Two record types are only treated as equivalent if their declaration stems from the same occurrence of the reserved word RECORD. See ARRAYS and SETs section for an explanation of this.

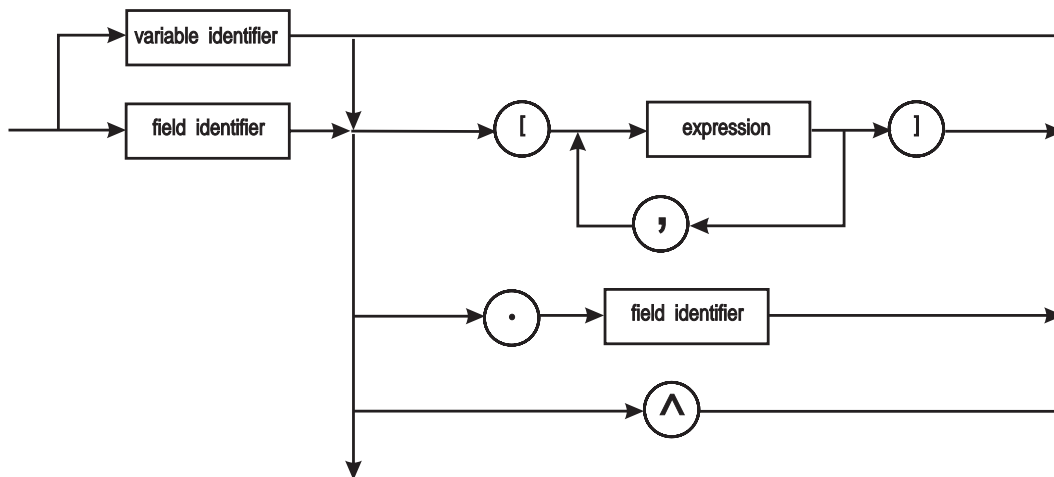
The WITH statement may be used to access the different fields within a record in a more compact and faster-executing manner than the alternative (e.g. ANIMAL.LEGS) PASCAL format.

FIELD LIST



When used in conjunction with RECORDs, see RECORDs section.

VARIABLE



Two kinds of variables are supported within PPC PASCAL; static and dynamic variables. Static variables are explicitly declared through VAR and memory is allocated for them during the entire execution of the block in which they were declared.

Dynamic variables, however, are created dynamically during program execution by the procedure NEW. They are not declared explicitly and cannot be referenced by an identifier. They are referenced indirectly by a static variable of type *pointer*, which contains the *address of* the dynamic variable.

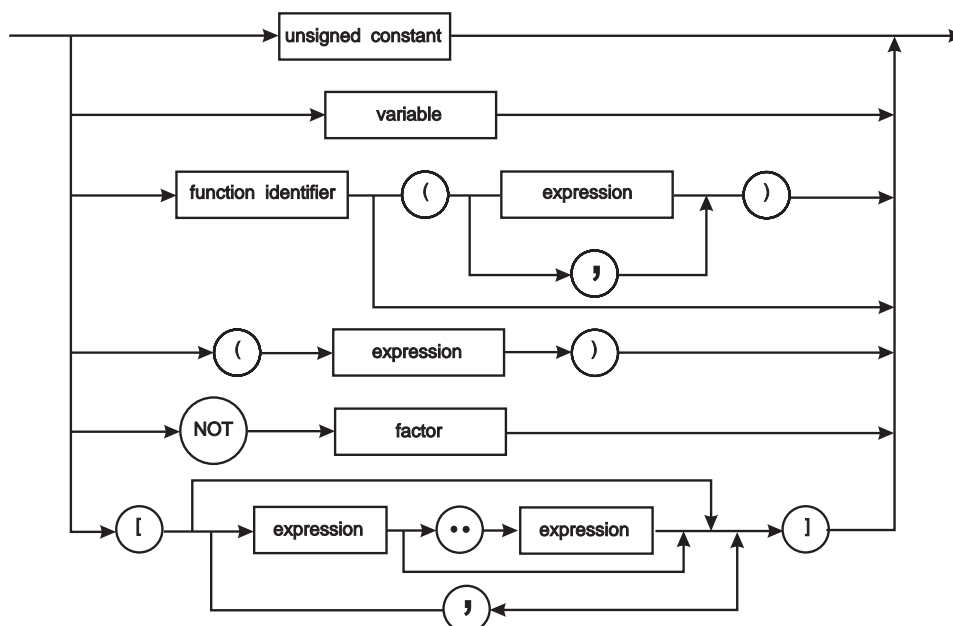
See PASCAL Dynamic Memory Allocation for more details of the use of dynamic variables.

When specifying elements of multi-dimensional arrays you are not forced to use the same form of index specification in the reference as was used in the declaration. For example, if variable **a** is declared as

```
ARRAY[1..10] OF ARRAY[1..10] OF INTEGER
```

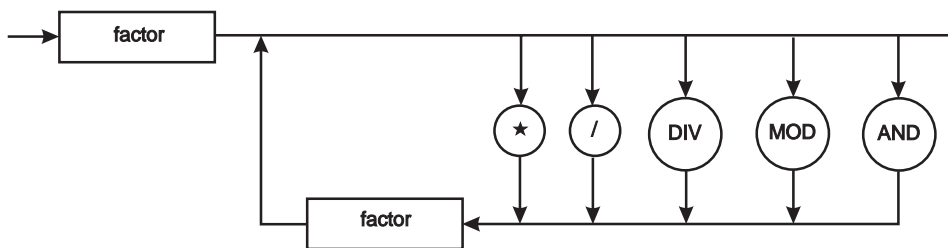
then either a[1][1] or a[1,1] may be used to access element (1,1) of the array.

FACTOR



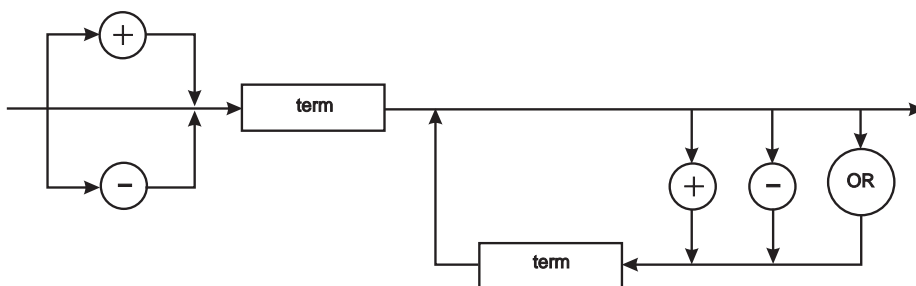
See EXPRESSION and FUNCTIONS for more details.

TERM



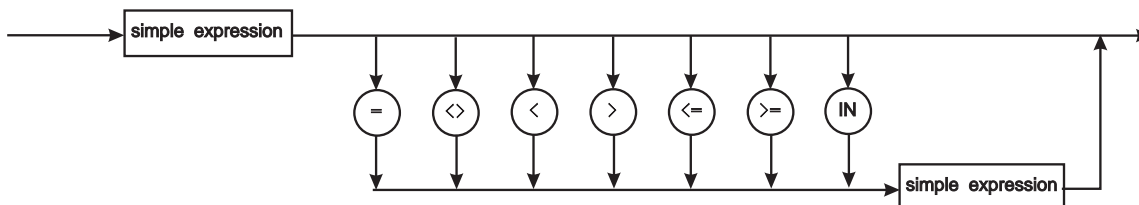
The lowerbound of a set is always zero and the set size is always the maximum of the base type of the set. Thus a SET OF CHAR always occupies 32 bytes (a possible 256 elements - one bit for each element). Similarly a SET OF 0..10 is equivalent to SET OF 0..255.

SIMPLE EXPRESSION



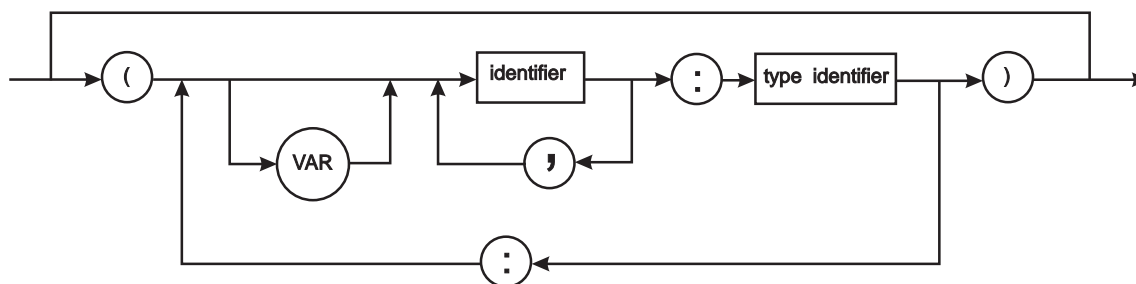
The same comments made in TERM above concerning sets apply to simple expressions.

EXPRESSION



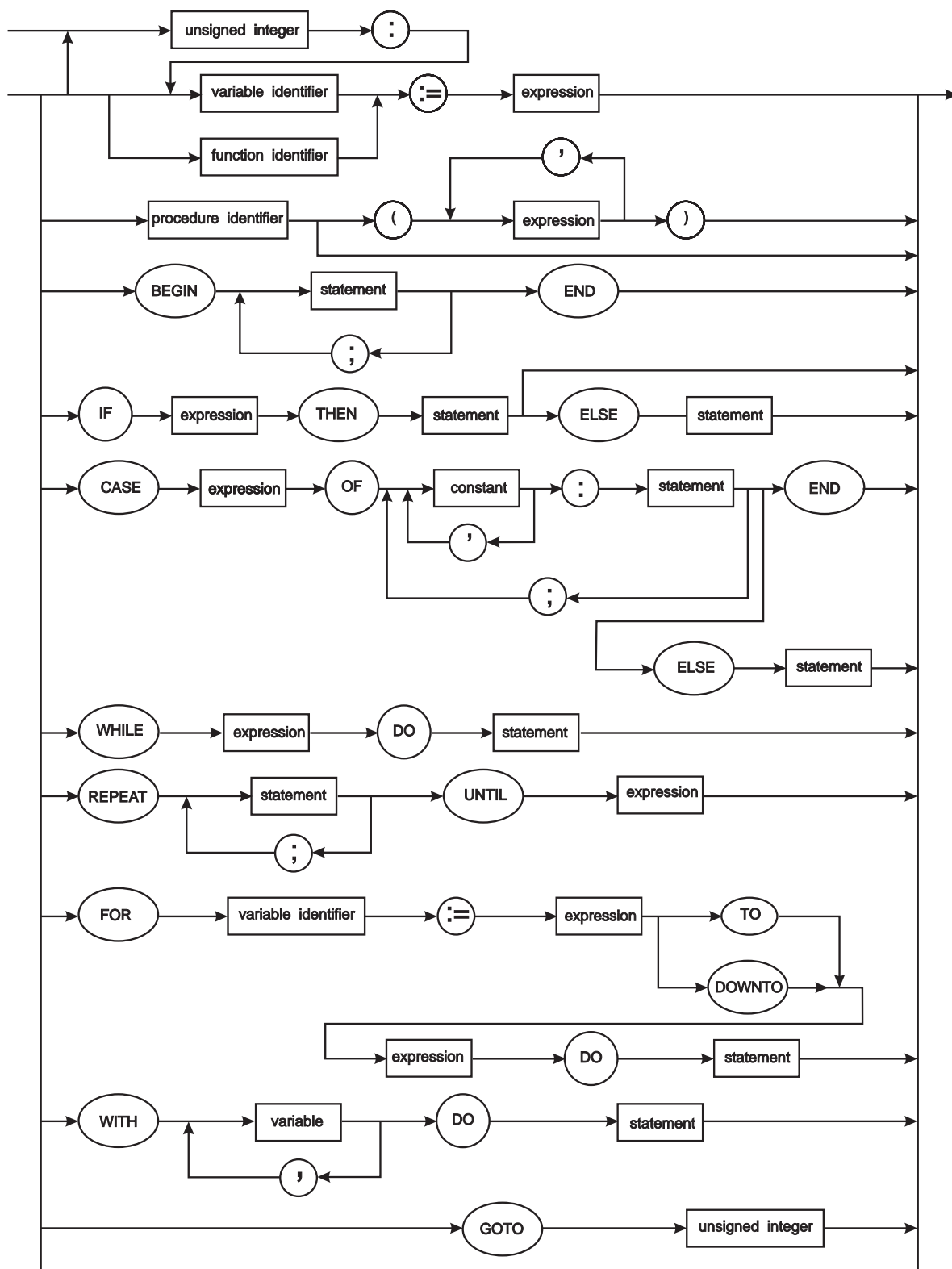
When using IN, the set attributes are the full range of the type of the simple expression with the exception of integer arguments for which the attributes are taken as if [0..255] had been encountered. The above syntax applies when comparing strings of the same length, pointers and all scalar types. Sets may be compared using >=, <=, <> or =. Pointers may only be compared using = and <>.

PARAMETER LIST



A type identifier must be used following the colon otherwise an error will result. Variable parameters as well as value parameters are fully supported. Procedures and functions are not valid as parameters. Files can only be used as variable parameters, not as value parameters.

STATEMENT



Assignment statements

See the TYPE ... RECORD sections for information on which assignment statements are valid. When assigning to subrange variables the value is not checked for being within the subrange for efficiency reasons.

CASE statements

An entirely null case list is not allowed i.e. CASE OF END; will generate an error.

The ELSE clause, which is an alternative to END, is executed if the selector (see the Expression syntax diagram) is not found in one of the case lists. This is analogous to the C `default` feature and is additional to Standard PASCAL which, to be safe, requires a CASE statement to be preceded by a check to ensure the selector is within the range of the CASE statement.

If the END terminator is used and the selector is not found then control is passed to the statement following the END. Note that you should *not* have a semi-colon before the END of a CASE statement.

FOR statements

The control variable of a FOR statement may only be an unstructured variable, not a parameter. This is half way between the Jensen/Wirth and ISO standard definitions.

Note that due to the signed integer implementation in PPC PASCAL, the absolute difference between the start and the end value in a FOR loop may not be greater than 32767. Examples:

```
FOR I := -16384 TO 16384 DO..      invalid
FOR I := -16384 TO 16383 DO..     valid
```

GOTO statements

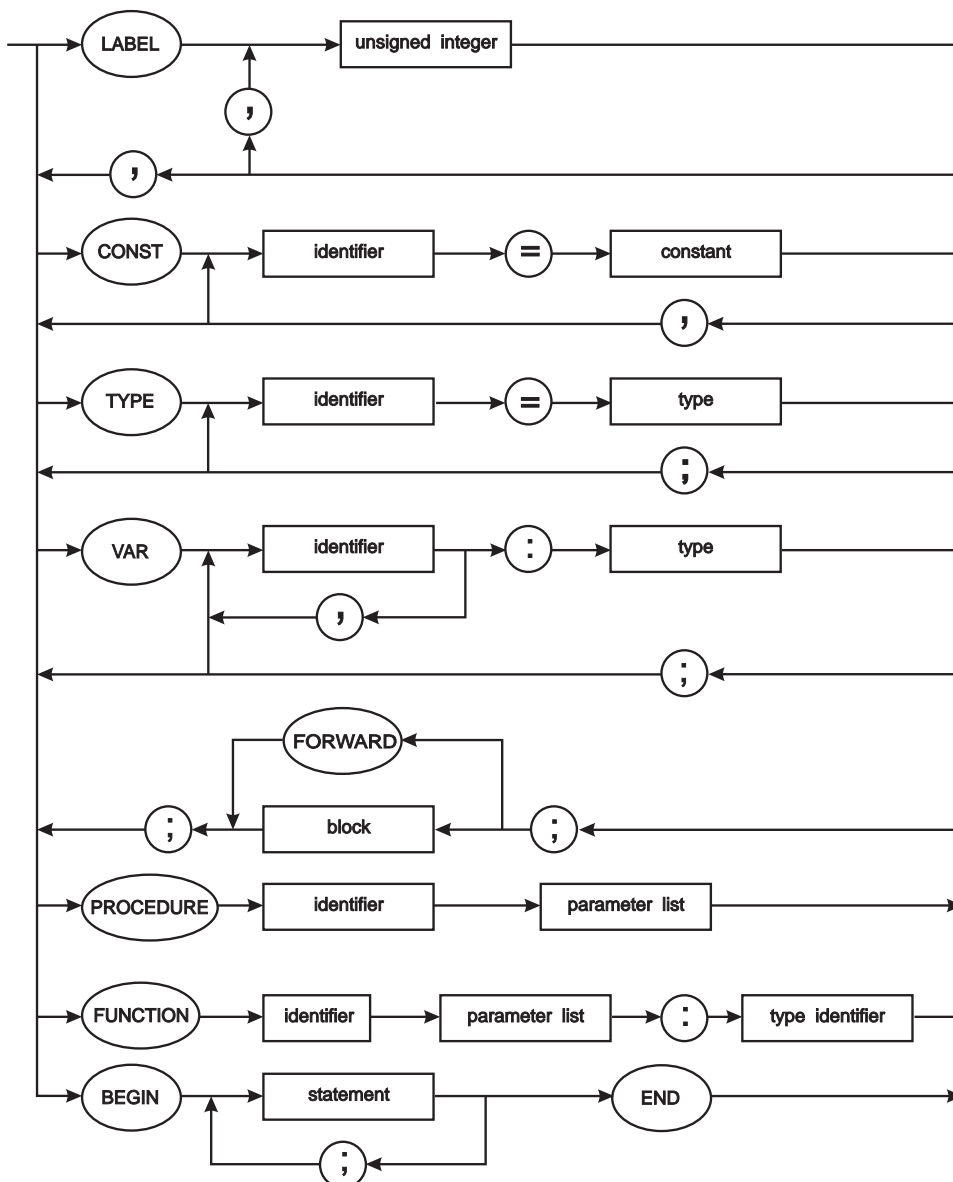
It is only possible to GOTO a label which is present in the same block as the GOTO statement and at the same level. You may *not* use GOTO to jump out of a FOR statement.

Labels must be declared (using the Reserved Word LABEL) in the block in which they are used; a label consists of at least one and up to four digits. When a label is used to mark a statement it must appear at the beginning of the statement and be followed by a colon.

WITH statements

WITH statements may not be used recursively; use the full specification of a field identifier when using it as a parameter to a procedure which is called recursively.

BLOCK



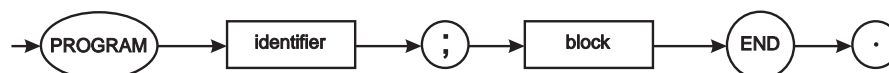
Forward References

As in the Pascal User Manual and Report (Section 11.C.1) procedures and functions may be referenced before they declared through use of the Reserved Word FORWARD e.g.

PROCEDURE a(y:t) ; FORWARD;	Procedure a is declared to be forward of this statement
PROCEDURE b(x:t);	
BEGIN	
....	
a(p);	Procedure a is now referenced
....	
END;	
PROCEDURE a;	Actual declaration of procedure a
BEGIN	
....	
b(q);	
....	
END;	

Note that the parameters and result type of the procedure a are declared along with FORWARD and are not repeated in the main declaration of the procedure. FORWARD is a Reserved Word.

PROGRAM



The list of file variables in brackets (Standard PASCAL) is optional and does nothing in PPC PASCAL.

Strong TYPEing

Different languages have different ways of ensuring that the user does not use an element of data in a manner which is inconsistent with its definition.

At one end of the scale there is machine code where no checks whatever are made on the *type* of variable being referenced. Further up the scale comes Pascal which goes as far as allowing distinct user-enumerated types. At the top of the scale (at present) is a language like ADA in which one can define different, incompatible numeric types.

There are basically two approaches used by Pascal implementations to strength of typing; structural equivalence or name equivalence. PPC PASCAL uses name equivalence for RECORDs and ARRAYs. The consequences of this are clarified in the TYPE ... RECORD sections *et al* - let it suffice to give an example here; say two variables are defined as follows:

```
VAR  A : ARRAY['A'..'C'] OF INTEGER;  
     B : ARRAY['A'..'C'] OF INTEGER;
```

then you might be tempted to think that you could write `A:=B;` but this would generate an error in PPC PASCAL since two separate TYPE records have been created by the above definitions. In other words, you have not taken the decision that A and B should represent the same type of data. You could do this by:

```
VAR A,B : ARRAY['A'..'C'] OF INTEGER;
```

and now you can freely assign A to B and vice versa since only one TYPE record has been created.

Although on the surface this name equivalence approach may seem a little complicated, in general it leads to fewer programming errors since it requires more initial thought from the programmer.

Semicolons

A semicolon should not appear before UNTIL or before END or before ELSE. Versions of PASCAL differ in their treatment of this. PPC PASCAL allows semicolons before UNTIL, END but reports an error if a semicolon appears before ELSE.

Predefined Identifiers

Constants

MAXINT	The largest integer available: 32767.
TRUE, FALSE	Constants of type Boolean.

PPC PASCAL has many other constant predefined identifiers (e.g. of type TEXT) which are used in PPC port initialisation and control procedures. See the PPC PASCAL Extensions section for details. All of these start with an underscore _ except PORT1, PORT2, PORT3, PORT4.

Types

INTEGER	See Unsigned Number section.
REAL	See Unsigned Number section.
CHAR	The full extended ASCII character set of 256 elements.
BOOLEAN	This type is used in logical operations including the results of comparisons. The result is always TRUE or FALSE.
TEXT	This type is used for the various predefined identifiers used in PPC port initialisation and control procedures. See the PPC PASCAL Extensions section for details.

Variables

There are no predefined variables in PPC PASCAL.

PROCEDURES AND FUNCTIONS

File Handling Procedures

WRITE

```
PROCEDURE WRITE(PORT:TEXT;P1,P2,P3...);
```

The procedure WRITE is intended for outputting text data. The destination must always be specified. It can be a port name or the name of a suitable array; e.g.:

```
var buffer : array [0..99] of char;
    num : integer;
....
write (buffer,'all this will be placed into the buffer',num);
write (port1, 'all this will go to Port 1',num);
```

The above facility for writing to memory instead of to a port is available on WRITE, WRITELN (but not BWRITE) procedures. It was not supported by early versions of PPC Pascal.



When writing to memory, there is no checking for overflow of the destination buffer!

The WRITE parameters P1, P2, ... Pn can have one of the following forms:

```
<e> or <e:m> or <e:m:n> or <e:m:H>
```

where e, m and n are expressions and H is a literal constant. We have 5 cases to examine:

1 e is of type INTEGER and either <e> or <e:m> is used.

The value of the integer expression **e** is converted to a character string with a trailing space. The length of the string may be increased (with leading spaces) by the use of **m** which specifies the total number of characters to be output. If **m** is not sufficient for **e** to be written or **m** is not present then **e** is written out in full, with a trailing space, and **m** is ignored. Note that if **m** is specified to be the length of **e** without the trailing space (or e=0 or e=1) then no trailing space will be output.

2 e is of type INTEGER and the form <e:m:H> is used.

In this case **e** is output in hexadecimal. If m=1 or m=2 then the value (e MOD 16^m) is output in a width of exactly **m** characters. If m=3 or m=4 then the full value of **e** is output in hexadecimal in a width of 4 characters. If m>4 then

leading spaces are inserted before the full hexadecimal value of **e** as necessary. Leading zeroes will be inserted where applicable. Examples:

```
WRITE(port1,1025:m:H);
m=1   outputs:    1
m=2   outputs:    01
m=3   outputs:    0401
m=4   outputs:    0401
m=5   outputs:    0401
```

3 **e is of type real. The forms <e>, <e:m> or <e:m:n> may be used.**

The value of **e** is converted to a character string representing a real number. The format of the representation is determined by **n**.

If **n** is not present then the number is output in scientific notation, with a mantissa and an exponent. If the number is negative then a minus sign is output prior to the mantissa, otherwise a space is output. The number is always output to at least one decimal place up to a maximum of 5 decimal places and the exponent is always signed (either with a plus or minus sign). This means that the minimum width of the scientific representation is 8 characters; if the field width **m** is less than 8 then the full width of 12 characters will always be output. If $m \geq 8$ then one or more decimal places will be output up to a maximum of 5 decimal places ($m=12$). For $m > 12$ leading spaces are inserted before the number. Examples:

```
WRITE(port1,-1.23E 10:m);
m=7   outputs:    -1.23000E+10
m=8   outputs:    -1.2E+10
m=9   outputs:    -1.23E+10
m=10  outputs:    -1.230E+10
m=11  outputs:    -1.2300E+10
m=12  outputs:    -1.23000E+10
m=13  outputs:    -1.23000E+10
```

If the form **<e:m:n>** is used then a fixed-point representation of the number **e** will be written with **n** specifying the number of decimal places to be output. No leading spaces will be output unless the field width **m** is sufficiently large. If **n** is zero then **e** is output as an integer. If **e** is too large to be output in the specified field width then it is output in scientific format with a field width of **m** (see above). Examples:

```
WRITE(port1,1E2:6:2)      outputs:    100.00
WRITE(port1,1E2:8:2)      outputs:    100.00
WRITE(port1,23.455:6:1)   outputs:    23.5
WRITE(port1,23.455:4:2)   outputs:    2.34550E+01
WRITE(port1,23.455:4:0)   outputs:    23
```

4 **e is of type character or type string.**

Either **<e>** or **<e:m>** may be used and the character or string of characters will be output in a minimum field width of 1 (for characters) or the length of the string (for string types). Leading spaces are inserted if **m** is sufficiently large.

5 **e is of type Boolean.**

Either **<e>** or **<e:m>** may be used and the ASCII strings TRUE or FALSE will be output depending on the Boolean value of **e**, using a minimum field width of 4 or 5 respectively.

WRITELN

```
PROCEDURE WRITELN(PORT:TEXT;P1,P2,P3...);
```

WRITELN is exactly identical to WRITE except it appends a CR,LF character pair to the output. It is useful only if you are outputting text data which is made up of CRLF-terminated lines.

BWRITE

```
PROCEDURE BWRITE(PORT:TEXT;V1,V2,V3...Vn);
```

This "binary write" procedure is one of the PPC extensions and allows the output of arbitrary binary data. It supports the output of **variables only** (V1...Vn) and no output formatting facilities are provided. **The port must always be specified**; if it is missing, the error message which appears may not be obvious. The following example shows how the various PPC PASCAL variables are output:

```

var a:integer;
    b:char;
    c:real;
    d:boolean;

a:=12347;
b:='A';
c:=2.3728E-5
d:=true;

```

Given the above variables contain the above values, the procedure

```

bwrite(port2,a,b,c,d);

```

will output the following values to PORT2 in the order shown:

3Bh 30h	the integer 12347 decimal (303B hex)
41h	the byte 'A' (41 hex)
BEh F0h 85h 63h	the floating-point value 2.3728E-5
01h	the boolean value TRUE (FALSE gives 00h)

In addition, Arrays, Records containing elements of any type and Sets may be output in the same way. An example of array output is below

```

var buffer : array [0..99] of char;
... (fill-up the array with data) ....
bwrite(port2,buffer);

```

where a complete 100-byte array is output in one operation. This type of output is common in PPC programming, where data is often read into a buffer, operated on while in the buffer, and the buffer is then output.

Also allowed is the form

```

bwrite(port, v:i)

```

which writes **i** bytes from the variable **v** (which would usually be an array).

The individual bytes of a multi-byte variable are output in the order in which they are stored in memory. See the Data Representation and Storage section for details of how variables are stored in PPC PASCAL.

As stated above, only **variables** may be output with BWRITE and literal values such as 'ABCD' are not supported; use WRITE or WRITELN to output those. If you need to output a mixture of formatted text data interspersed with binary data, use a sequence of WRITE, WRITELN and BWRITE invocations as required.

READ

```

PROCEDURE READ(PORT:TEXT;V1,V2,V3...Vn);

```

The procedure READ is intended for reading text (i.e. printable ASCII) data.

In general, the statement

```

READ(PORT1,V1,V2,.....Vn);

```

is equivalent to

```

BEGIN
  READ(PORT1,V1);READ(PORT1,V2);.....;READ(PORT1,Vn);
END;

```

where V1,V2 etc. may be of type character, string (an array of char), integer or real. The port number must be specified.

The statement READ(P,V); has different effects depending on the type of V. There are 4 cases to consider:

1 V is of type char.

In this case

```

READ(P,V);

```

reads a single character from the specified port and is assigned to V. If, having read this character, another character is waiting in the input buffer and this waiting character is a line marker (#13) then the function EOLN(P) will return the value TRUE.

If the character is a line-feed (LF, #10) then it is removed from the buffer and discarded and the next character is read. If you need to process LF characters, use the BREAD procedure instead.

2 V is of type string (array of char).

A string of characters may be read using READ and in this case a series of characters will be read from the specified port until the number of characters defined by the string has been read or EOLN becomes TRUE. If the string is not filled by the read (i.e. if end-of-line is reached before the whole string has been assigned) then the end of the string is filled with null (#0) characters - this enables you to scan the array and evaluate the length of the string that was read.



The READ procedure will wait until either the array has been filled, or EOLN becomes TRUE. This can cause your program to apparently hang if insufficient input data arrives. Use this method only if you are sure that the input stream will contain CRs or CRLFs at regular intervals, or if waiting for input does not matter.

3 V is of type integer.

In this case a series of characters which represent an integer is read. All preceding blanks and end-of-line markers are skipped.

If the integer read has an absolute value greater than MAXINT (32767) then the runtime error **Number too large** will be issued and execution terminated. This error can be suppressed with the O- compiler option.

If the first character read, after spaces and end-of-line characters have been skipped, is not a digit or a sign (+ or -) then the runtime error **Number expected** will be reported and the program aborted. This error can be suppressed with the R- compiler option.

4 V is of type real.

Here, a series of characters representing a real number will be read. All leading spaces and end-of-line markers are skipped and, as for integers above, the first character afterwards must be a digit or a sign. If the number read is too large or too small then an **Overflow error** will be reported (can be suppressed with the O- compiler option), if **E** is present without a following sign or digit then **Exponent expected** error will be generated and if a decimal point is present without a subsequent digit then a **Number expected** error will be given. The latter two errors can be suppressed with the R- compiler option.

READLN

This is identical to READ except that when called, it discards all input characters up to and including a CR, following which its operation is as READ. It is not particularly useful in PPC applications.

BREAD

```
PROCEDURE BREAD(PORT:TEXT;V1,V2,V3...Vn);
```

This "binary read" procedure is one of the PPC extensions and allows the input of arbitrary binary data. It supports the input of **variables only** (V1...Vn). It is the opposite of BWRITE and any variable written with BWRITE can be read with BREAD and the result is the same as the original variable. The port must always be specified; if it is missing, the error message which appears may not be obvious.

The following examples show the data expected for each type of variable:

char	1 byte
integer	2 bytes, less significant byte arrives first
real	4 bytes
array[1..76] of char	76 bytes
a[55]	2 bytes (if a is an integer array)

When BREAD is used to read *characters*, it is capable of receiving the entire 256-character set from #0 to #255 inclusive. Control characters (e.g. CR,LF) are not treated differently.

Also allowed is the form

```
bread(port, v:i)
```

which reads-in **i** bytes into the variable **v** (which would usually be an array).



When XON/XOFF is enabled as a TX handshake on this port, any XON/XOFF characters arriving at this input port will be removed from the data stream by the PPC comms processor. Your PASCAL program will not see them.



Unlike READ & READLN, BREAD performs no break (CTRL-C) checking. If your program is waiting for input, there is no means of terminating it except by resetting the PPC.

File Handling Functions

EOLN

```
FUNCTION EOLN(PORT:TEXT):BOOLEAN;
```

The function EOLN is a Boolean function which returns the value TRUE if there is an end-of-line character (#13) waiting to be removed from the input port's buffer, otherwise the function returns the value FALSE.

The default value of PORT is PORT1.

Transfer Functions

TRUNC

```
FUNCTION TRUNC(X:REAL):INTEGER;  
FUNCTION TRUNC(X:INTEGER):INTEGER;
```

The parameter **X** must be of type real or integer and the value returned by TRUNC is the greatest integer less than or equal to X if X is positive or the least integer greater than or equal to X if X is negative.

```
TRUNC(-1.5)    returns  -1  
TRUNC(1.9)    returns   1
```

ROUND

```
FUNCTION ROUND(X:REAL):INTEGER;  
FUNCTION ROUND(X:INTEGER):INTEGER;
```

X must be of type real or integer and the function returns the nearest integer to X (according to standard rounding rules). Examples:

```
ROUND(-6.5)    returns  -6  
ROUND(11.7)    returns  12  
ROUND(-6.51)   returns  -7  
ROUND(23.5)    returns  24
```

ENTIER

```
FUNCTION ENTIER(X:REAL):INTEGER;  
FUNCTION ENTIER(X:INTEGER):INTEGER;
```

X must be of type real or integer. ENTIER returns the greatest integer less than or equal to X, for all X:

```
ENTIER(-6.5)   returns  -7  
ENTIER(11.7)   returns  11
```

ENTIER is not a Standard PASCAL function but is the equivalent of BASIC's INT. It is useful when writing fast routines for many mathematical applications.

ORD

```
FUNCTION ORD(X:CHAR):INTEGER;  
FUNCTION ORD(X:INTEGER):INTEGER;  
FUNCTION ORD(X:any element of a set):INTEGER;
```

X may be of any scalar type except real. The value returned is an integer representing the ordinal number of the value of X within the set defining the type of X.

```
ORD('a')      returns  97  
ORD('@')      returns  64
```

If X is of type integer then ORD(X)=X ; this should normally be avoided.

CHR

```
FUNCTION CHR(X:INTEGER):CHAR;
```

X must be of type integer. CHR returns a character value corresponding to the ASCII value of X:

CHR(49)	returns	1
CHR(91)	returns	[

Arithmetic Functions

In all the functions within this sub-section the parameter X must be of type real or integer unless specified otherwise.

ABS

FUNCTION ABS(X:real or integer) : same type as input;

Returns the absolute value of X (e.g. ABS(-4.5) gives 4.5). The result is of the same type as X.

SQR

FUNCTION SQR(X:real or integer) : same type as input;

Returns the value X^2 i.e. the square of X. The result is of the same type as X.

SQRT

FUNCTION SQRT(X:real or integer) : REAL;

Returns the square root of X. The returned value is always of type real. A Maths Call Error is generated if the argument X is negative; this error can be disabled with the O- compiler option.

FRAC

FUNCTION FRAC(X:real or integer) : REAL;

Returns the fractional part of X: $FRAC(X) = X - ENTIER(X)$.

As with ENTIER this function is useful for writing many fast mathematical routines. Examples:

FRAC(1.5)	returns	0.5
FRAC(-12.56)	returns	0.44

SIN

FUNCTION SIN(X:real or integer) : REAL;

Returns the sine of X where X is in radians. The result is always of type real.

COS

FUNCTION COS(X:real or integer) : REAL;

Returns the cosine of X where X is in radians. The result is of type real.

TAN

FUNCTION TAN(X:real or integer) : REAL;

Returns the tangent of X where X is in radians. The result is always of type real.

ARCTAN

FUNCTION ARCTAN(X:REAL) : REAL;

Returns the angle, in radians, whose tangent is equal to the number X. The result is of type real.

EXP

FUNCTION EXP(X:real or integer) : REAL;

Returns the value e^X where $e = 2.71828$. The result is always of type real.

LN

FUNCTION LN(X:real or integer):REAL;

Returns the natural logarithm (i.e. to the base e) of X. The result is of type real. If X then a Maths Call Error will be generated. This error can be disabled with the O- compiler option.

HALT

PROCEDURE HALT;

The procedure causes program execution to stop with the message Halt at PC=XXXX where XXXX is the hexadecimal memory address of the location where the HALT was issued. Together with a compilation listing, HALT may be used to determine which of two or more paths through a program are taken. This will normally be used during debugging.

RANSEED

PROCEDURE RANSEED(X,Y,Z:INTEGER);

This procedure has three integer parameters which are used as seeds for the random numbers produced by RANDOM. These integers must be between 1 and 30000. Normally the same random numbers are generated by RANDOM for each run of a program; this procedure can be used to give different numbers.

RANDOM

FUNCTION RANDOM:REAL;

This returns a pseudo-random **real** number between 0.0 and 1.0. The algorithm used is based on that of B.A. Wichmann and I.D. Hill (NPL Report DITC 6/82). Normally the same random numbers are generated by RANDOM for each run of a program; the seeds may be changed using the procedure RANSEED described above.

SUCC(X)

X may be of any scalar type except real and SUCC(X) returns the successor of X:

SUCC('A')	returns B
SUCC('5')	returns 6.

PRED(X)

X may be of any scalar type except real; the result of the function is the predecessor of X:

PRED('j')	returns i
PRED(TRUE)	returns FALSE.

ODD

FUNCTION ODD(X:INTEGER):BOOLEAN;

ODD returns a Boolean result which is TRUE if X is odd and FALSE if X is even.

ADDR(V)

This function takes a variable identifier of any type as a parameter and returns an integer result which is the memory address of the variable identifier V. For information on how variables are held at runtime in PPC PASCAL, see the Data Representation and Storage section.

SIZE(V)

SIZE is a special function whose parameter is either a variable name or a type identifier. It returns as an integer the number of bytes of storage used by this type of variable.

Example:

```
WRITELN(PORT1,SIZE(INTEGER));    gives 2
```


RECAST(e, typ)

RECAST is a function for suspending the type checking in PPC PASCAL. It takes any expression *e* of any type and uses it as if it was of type *typ*. Both parameters must need the same number of bytes of storage. The principal use of this function is to convert one variable into another. The example below converts the contents of a character **a** (which should have the value 00h or 01h only) into a boolean **b**:

```
var a : char;           { Assuming a is a char }
    b : boolean;       { and b is boolean }
b:=recast(a,boolean);  { we can convert a -> b with this }
```

The previous operation could also have been achieved (less efficiently) with the Standard PASCAL variant record approach:

```
type fiddle = record case boolean of { define a variant record }
    true:(x:char);
    false:(y:boolean)
end;
var f:fiddle;
f.x:=a;           { this converts a -> b }
b:=f.y;
```



The compiler does not check that both parameters use the same number of bytes of storage. Use this function with care!

MEMAVAIL

```
FUNCTION MEMAVAIL:INTEGER;
```

MEMAVAIL is a parameterless function which returns the number of bytes of free space between the top of the heap and the stack. It returns the size of the largest free block of memory.

It returns an integer and can be used to detect if a program is about to run out of memory and so take corrective action. It may be possible to create a dynamic variable that is larger than MEMAVAIL would suggest since there may be sufficient free space within the heap where space has been freed using DISPOSE. For details of how memory is organised at run-time, see the Program and Data Limits section.

PASCAL Dynamic Memory Allocation

PPC PASCAL provides a set of heap management routines. Freed memory is re-used but no garbage collection is provided. These functions should be used with care and, to prevent the heap potentially becoming full as a result of fragmentation, NEW and DISPOSE should always be used in pairs; this will prevent any fragmentation.

NEW

```
PROCEDURE NEW (VAR V);
```

The procedure NEW(P) allocates space for a dynamic variable. The variable P is a pointer variable and after NEW(P) has been executed P contains the address of the newly allocated dynamic variable. The type of the dynamic variable is the same as the type of the pointer variable P and this can be of any type.

To access the dynamic variable P[^] is used. To re-allocate space used for dynamic variables use the procedure DISPOSE or alternatively MARK and RELEASE (see below).

If there is no memory left, a runtime error Out of RAM will be generated. There is no method of checking whether NEW will succeed, but if MEMAVAIL is greater than SIZE(V) then it will definitely succeed. If it is less, then NEW will probably fail but not if there is a large enough hole in the heap from a previous DISPOSE. See the note about fragmentation at the top of this page.

DISPOSE

```
PROCEDURE DISPOSE (VAR V);
```

This procedure frees the memory used by the variable pointed to be V. V must be a pointer variable which points to a variable created using the NEW procedure (see above). After the call to DISPOSE, V should be made to point somewhere else before referencing V[^]. The memory freed may then be subsequently used when creating other variables using NEW. DISPOSE must be use with care since, for example, after

```
NEW(v1);  
v2:=v1;  
DISPOSE(v1);
```

v2 will no longer point to valid data.

MARK

```
PROCEDURE MARK (VAR V);
```

This procedure saves the state of the dynamic variable heap to be saved in the pointer variable V. The state of the heap may be restored to that when the procedure MARK was executed by using the procedure RELEASE (see below).

The type of variable to which V points is irrelevant, since V should only be used with MARK and RELEASE, never NEW.

RELEASE

```
PROCEDURE RELEASE (VAR V);
```

This procedure frees space on the heap for use of dynamic variables. The state of the heap is restored to its state when MARK(V) was executed - thus effectively destroying all dynamic variables created since the execution of the MARK procedure and as such *it should be used with great care*.



If the procedure DISPOSE has been used, it is possible that not all such variables will be destroyed. To delete individual variables use DISPOSE.

Examples

The following example program allocates a 1000-byte array on the heap, reads 1000 bytes into it, operates on it (in this example it just converts any lowercase characters to uppercase), outputs it, and eventually de-allocates it:

```
type buffer=array[1..1000] of char;
   pBuffer=^buffer;
var b:pBuffer;
    i:integer;
begin
    new(b);                { allocate the buffer }
    read(port1,b^);        { fill it up }
    for i:=1 to size(buffer) do { do something with it }
    begin
        if b^[i] in ['a'..'z'] then
            b^[i]:=chr(ord(b^[i])-32);
    end;
    write(port2,b^);       { output it }
    dispose(b);            { discard it }
end;
```

The above illustrates a typical application for dynamic memory allocation: a sizeable data structure is required only temporarily, but the requirement may occur in many places within a program.

A similar effect can be achieved by declaring the buffer within a function or procedure; it will be discarded when the function or procedure exits. The difference is that using NEW and DISPOSE it is possible to discard the buffer *before* the exit.

PPC PASCAL special extensions

These functions and procedures enable PASCAL programs to access the various features of the PPC. They are all predefined and are ready to use.

SETPORT - configure a serial port

```
FUNCTION SETPORT(VAR S:SETPORTTYPE):INTEGER;
```

This function duplicates the facilities in the Executive Run-Time Serial Port Configuration menu. It configures the specified port from parameters held in a record. The address of the record is passed as a parameter to the function.

The example below sets-up Port 3 to 9600 baud, 8 bits/word, no parity, 1 stop bit, enables all receive handshakes, and disables all transmit handshakes except XON/XOFF:

```
var anyone:setporttype;      {give the record a name}
    retcode:integer;         {this accepts any 'setport' errorcode}

with anyone do begin        {now we fill-up the record as required}
  sppc_port:=port3;         {port #: 1|2|3|4}
  baud_rate:=_b9600;        {baud rate}
  bits_word:=_bw8;          {bits/word: _bw5|_bw6|_bw7|_bw8}
  parity:=_pnone;           {parity: _pnone|_peven|_podd}
  stop_bit:=_sb1;           {stop bits: _sb1|_sb2}
  rx_rts_h:=true;           {enable RX RTS handshake; true|false}
  rx_dtr_h:=true;           {enable RX DTR handshake; true|false}
  rx_x_hs:=true;            {enable RX XON/XOFF h/shake; true|false}
  tx_cts_h:=false;          {disable TX CTS handshake; true|false}
  tx_dsr_h:=false;          {disable TX DSR handshake; true|false}
  tx_x_hs:=true;            {enable TX XON/XOFF h/shake; true|false}
end;

retcode:=setport(anyone);   {this actually initialises the port}

if retcode<>0 then          {optional check on setport errorcode}
  writeln(port1,'Port 3 setport failed');
```

Note that as per Standard PASCAL the above WITH statement could equivalently be done as

```
anyone.sppc_port:=port3;
anyone.baud_rate:=_b9600;
anyone.bits_word:=_bw8;
anyone.parity:=_pnone;
anyone.stop_bit:=_sb1;
anyone.rx_rts_h:=true;
anyone.rx_dtr_h:=true;
anyone.rx_x_hs:=true;
anyone.tx_cts_h:=false;
anyone.tx_dsr_h:=false;
anyone.tx_x_hs:=true;
```

and this is more convenient if you have previously loaded-up the entire record and now wish to modify just one or two members. The former WITH style produces more compact code, however.

The following identifiers are internally predefined (enumerated types) for baud rate specifications:





<u>B</u> 30	<u>B</u> 37H	<u>B</u> 50	<u>B</u> 75	<u>B</u> 100	<u>B</u> 110
<u>B</u> 134H	<u>B</u> 150	<u>B</u> 300	<u>B</u> 600	<u>B</u> 1200	<u>B</u> 2000
<u>B</u> 2400	<u>B</u> 3600	<u>B</u> 4800	<u>B</u> 7200	<u>B</u> 9600	<u>B</u> 19200
<u>B</u> 38400	<u>B</u> 57600	<u>B</u> 115200			

The baud rate setting has been implemented with enumerated types in this way (rather than simply using integers) to support baud rate values above 32767 and also to support half-rates such as 37.5 baud (B37H). The identifiers for the other serial parameters are shown in the comments in the above program example. The record type SETPORTTYPE is also internally predefined.

The function SETPORT returns a non-zero integer return code if any of the parameters is invalid. During program debugging, this return code should be tested as shown in the above example. In any case, Standard PASCAL dictates

that a function's return code must be loaded into a variable of the appropriate type – even if that variable is never tested.

For full details of operation of the PPC serial ports please see the PPC Serial Ports chapter.

-  Due to slight differences in the internal hardware implementations of the four PPC serial ports, not all parameters are supported by all four ports. See the PPC Ports chapter for a table showing which parameters are valid on which ports.
-  You must define the value of **all** the parameters before calling SETPORT. Uninitialised parameters are undefined and will have unpredictable results.
-  If SETPORT returns a non-zero return code (i.e. an error code), the port may have been only *partly* initialised and its status is therefore undefined.
-  **When a program is running under the Executive** (i.e. with Compile and Run), **initialising Port 1 has no effect**, but with a “no error” errorcode. Only “autoexec” programs may initialise Port 1. This is done to enable you to test a program, using Compile and Run, with KTERM running at e.g. 38400 baud, even if that program initialises Port 1 to say 9600 baud.

READPORT - get serial port configuration

```
FUNCTION READPORT(VAR S:SETPORTTYPE):INTEGER;
```

This function is the opposite of SETPORT and allows a port's current configuration to be read. It uses the same record as SETPORT.

You can use READPORT to fill-in the record with the port's current configuration, modify a part of it, and re-initialise the port with SETPORT. The following example shows how you can use READPORT & SETPORT to change only the handshakes:

```
var anyone:setporttype;
    retcode:integer;

anyone.sppc_port:=port3;    {specify the port whose config to read}
retcode:=readport(anyone); {get the current port3 config}

with anyone do begin
    rx_rts_h:=false;        {disable RX RTS handshake}
    rx_dtr_h:=true;         {enable RX DTR handshake}
    rx_x_hs:=false;        {disable RX XON/XOFF handshake}
end;

retcode:=setport(anyone);  {re-initialise the port}
```

SETHSK - control handshake outputs

```
FUNCTION SETHSK(VAR S:SETHSKTYPE):INTEGER;
```

This sets the states of a port's output handshakes according to boolean values held in a four-member record. The following example configures the output handshake on Port 3:


```
var anyone:sethsktype;      {give the record a name}
    retcode:integer;        {this accepts any 'sethsk' errorcode}

with anyone do begin        {now we fill-up the record as required}
    hppc_port:=port3;       {port #: 1|2|3|4}
    rts_out:=true;          {set RTS3 to high level; true|false}
    dtr_out:=true;          {set DTR3 to high level; true|false}
    cdo_out:=false;         {set CDO3 to low level; true|false}
end;

retcode:=sethsk(anyone);    {this actually sets-up the handshakes}
```

SETHSK will return a non-zero return code if any of the four parameters have invalid values.

The term “output handshakes” used here refers to the three output signals which every PPC port has, regardless of whether they are actually being used as handshakes, i.e. to control data flow. For example, the CDO (carrier detect output) is definitely not a handshake signal.

-  If either of RTS or DTR has been *enabled* (with SETPORT, or in the Executive Run-Time Serial Port Configuration menu) it will be controlled *automatically* by the PPC and any attempt to control it with SETHSK will be ignored. If you want to control RTS or DTR with SETHSK, that output handshake must be *disabled* in SETPORT or in the Executive.

READHSK - read handshake inputs

```
FUNCTION READHSK(VAR R:READHSKTYPE):INTEGER;
```

This function fills-in a record with the current states of a port's input handshakes. It allows the states of the port's CTS, DSR or CDI input signals to be read. The states are read into a record containing the following four members:

```
rppc_port : port1|port2|port3|port4
cts_in    : boolean
dsr_in    : boolean
cdi_in    : boolean
```

The following Port 3 example shows how to read the states of the CTS, DSR, CDI signals for Port 3:

```
var anyname:readhsktype;      {give the record a name}
    retcode:integer;          {this accepts any 'readhsk' errorcode}
anyname.rppc_port:=port3;     {initialise the Port # to read; 1|2|3|4}
retcode:=readhsk(anyname);    {this fills-in the signals' states}
with anyname do               {print out the states}
    writeln(port1, 'CTS=',cts_in,' DSR=',dsr_in,' CDI=',cdi_in);
```

IPQCOUNT - get #bytes in input queue

```
FUNCTION IPQCOUNT(PORT:TEXT):INTEGER;
```

Each PPC port has a 255-byte input queue which receives data under interrupts. If there is any unread data, this function will return an integer in the range 1..255. In practice, if the handshakes are operating correctly, the queue size should never exceed 128 by more than a few chars. For full details of the serial port queues, see the PPC Ports chapter.

This function allows a program to check if any data has arrived on a specified port's input queue. It allows a program to read data only when data is present, rather than "hang" until data arrives.

When used in conjunction with the PPC timers, this function permits the construction of a program which waits for input data for a certain maximum period and then times-out and does something else. For an example of this, see the LOADTIMER and READTIMER functions.

When used in conjunction with the OPQSPACE function, this function also permits the construction of a program which services multiple ports in an apparently "multitasking" manner. For an example see OPQSPACE below.

After this function has been invoked, the number of bytes in the input queue may *increase* as a result of new input data arriving, but it can never decrease except by being read by your program.

IPQCLEAR - clear input queue

```
PROCEDURE IPQCLEAR(PORT:TEXT);
```

This procedure clears-out any unread data in the specified port's input queue.

OPQSPACE - test output queue space

```
FUNCTION OPQSPACE(PORT:TEXT):INTEGER;
```

Each PPC port has a 64-byte output queue which is emptied under interrupts to the output device (handshakes permitting). Four bytes are always kept unused. If there is any room in this queue, this function will return an integer in the range 1..60. If the queue is initially empty, a program can freely generate up to 60 bytes of output data – even if the output device has been "busy" all the time.

This function allows a program to check if there is any room in the specified port's output queue. It allows a program to write output data only when there is room for it, rather than "hang" until room becomes available.

When used in conjunction with the PPC timers, this function permits the construction of a program which times-out if the output device has been "busy" for too long.

When used in conjunction with the IPQCOUNT function, this function also permits the construction of a program which services multiple ports in an apparently "multitasking" manner. The following example shows a trivial "multitasking" program which copies data PORT1→PORT2 and simultaneously PORT3→PORT4 and which can never hang:

```

procedure copydata(p_in,p_out:text);
var c:char;
begin
  bread(p_in,c); bwrite(p_out,c);  {copy a char p_in -> p_out}
end;
repeat
  if (ipqcount(port1)>0) and (opqspace(port2)>0) then
    copydata(port1,port2);
  if (ipqcount(port3)>0) and (opqspace(port4)>0) then
    copydata(port3,port4);
until false;

```

Note how `copydata` is invoked only if there is some input data *and* if there is room for output data. This important principle can be extended to produce sophisticated “multitasking”-like programs.

After this function has been invoked, the amount of space in the output queue may *increase* as a result of data being output, but it can never decrease except by being filled by your program.

OPQCOUNT - get #bytes in output queue

```
FUNCTION OPQCOUNT(PORT:TEXT): INTEGER;
```

This function is the complement to `OPQSPACE`. It returns the number of bytes of data currently in the output queue.

To assist special programs which must at times ensure that the output queue is completely empty, this function has been carefully implemented to return the *total* of the number of bytes in the 64-byte output queue *plus* the number of bytes held in the UART transmit circuitry. This ensures that when the value returned is zero, all data generated by your program has been transmitted out of the PPC. **However, this works perfectly only on ports 3 and 4.** On ports 1 and 2, a return value below 2 is ambiguous and means the following:

value=1	never returned
value=0	there are 0 or 1 bytes still in the UART

Therefore, on ports 1,2 detection of “everything sent out” must be done with a timer, by waiting until the `OPQCOUNT` value falls below 2 and then waiting for at least 2 character periods.

After this function has been invoked, the number of bytes in the output queue may *decrease* as a result of data being output, but it can never increase except by being filled by your program.

SETRTC - set real-time clock

```
FUNCTION SETRTC(VAR T:RTCTYPE): INTEGER;
```

This function sets the PPC real-time clock from date and time values held in a record which contains the following members:

tm_sec	: integer	{seconds after the minute	0..59}
tm_min	: integer	{minutes after the hour	0..59}
tm_hour	: integer	{hours since midnight	0..23}
tm_mday	: integer	{day of the month	0..30}
tm_mon	: integer	{months since January	0..11}
tm_year	: integer	{years since 1900	50..149}
tm_wday	: integer	{days since Sunday	0..6}

The following example loads the clock with 18:48:55, Sunday 10th May 2002:

```

var anyone:rtctype;           {give the record a name}
    retcode:integer;         {this accepts any 'setrtc' errorcode}


with anyone do begin        {now we fill-up the record as required}
  tm_sec:=55;
  tm_min:=48;
  tm_hour:=18;
  tm_mday:=9;                {10th day = 9}
  tm_mon:=4;                 {May = 5th month = 4}
  tm_year:=102;
  tm_wday:=0;                {Sunday = 0}
end;


```

```

retcode:=setrtc(anyname);    {this actually loads the clock}
if retcode<>0 then          {optional check on setrtc errorcode}
  writeln(port1,'SETRTC failed');

```

 You must define the values of **all** the seven parameters before calling SETRTC. Uninitialised parameters are undefined and will have unpredictable results.

 Years over 1999: be careful to specify the “year since 1900” correctly! Year 2005 is specified as 105. It is also important that your program handles this correctly; it is no good to store only 2 digits for the year and expect the date to correctly roll from 1999 to 2000. The best way to store the year value is as a full integer equal to the actual year, and any year-related text in the user interface should show the full 4 digits too.

The RTC, in common with most RTC chips on the market, stores the year as only two BCD digits 00..99. PPC firmware supports years past 1999 by adding an offset of 50, thus supporting years 1950..2049. This is done transparently and you need to know nothing about it.

If SETRTC returns a non-zero return code (i.e. an error code), this indicates that some of the parameters are invalid, and the clock will not have been accessed. A non-zero return code can also indicate that the RTC option is not installed.

GETRTC - read real-time clock

```

FUNCTION GETRTC(VAR T:RTCTYPE):INTEGER;

```

This function reads the PPC real-time clock into the same record which is used for SETRTC. A non-zero return code indicates that the RTC option is not installed.

LOADTIMER - load a timer

```


PROCEDURE LOADTIMER(TIMERNO,VALUE:INTEGER);

```

The PPC provides eight timers, numbered 0..7, which are decremented to zero and which provide 1ms resolution. Because the timers are decremented in the background (i.e. with interrupts) the PASCAL program, having started a timer by loading a non-zero value into it, is free to do anything while the timer is being decremented, and it needs to test the state of the timer only occasionally.

Delays up to 65535ms are possible; delays over 32767ms will involve *negative* timer values. The timer is decremented to zero only; the expiration of a timer is detected with the function READTIMER returning a zero value. If an invalid timer number is specified, this procedure has no effect.

The 1ms increment is not affected by the -H2 or -H3 PPC options which merely increase the CPU speed.

 The timer is decremented every 1ms, i.e. *with 1ms resolution*. Therefore, if you load a time constant of e.g. 3 then the actual delay before the timer reaches zero could be *between 2ms and 4ms*. Very short time delays are not exact and the minimum time constant which should ever be loaded is 2.

See the READTIMER function below for a program example.

READTIMER - read a timer

```

FUNCTION READTIMER(TIMERNO:INTEGER):INTEGER;

```

The following program code reads characters from Port 3 until 20 characters have been received, or until 300ms have elapsed, whichever occurs first:

```

var timeleft,count:integer; c:char;

loadtimer(7,300);           {load timer 7 with 300ms}
chrcount:=20;              {# of chars to wait for}

repeat
  if ipqcount(port3)>0 then begin {if there is any input data}
    read(port3,c);             { then read a char}
    count:=count-1;           { and decrement chr count}
    loadtimer(7,300);         { and reload the timer}
  end;
until (count=0) or (readtimer(7)=0); {and drop out if done|timeout}

if timeleft=0 then writeln(port1,'Timeout on Port 3');

```


To check if a timer has expired, use an *equality* test, e.g. `timeleft=0`, not e.g. `timeleft>0`. The latter case might not work if the initial timer load value was `>32767`.

A channel number which is invalid (other than 0..7) will cause a zero timer value to be returned.

ULED - control user LED

```
PROCEDURE ULED(ONOFF:BOOLEAN);
```

This procedure allows the user-accessible front panel LED labelled USR to be switched on/off according to the state of a Boolean variable. This feature can be used to indicate the progress of a program, or act as a prompt for some user action, or simply for debugging.

C programs can access all eight LEDs, or all 16 on PPCs with two LED displays.

READSWI - read front panel switches

```
PROCEDURE READSWI(VAR R:SWITCHTYPE);
```

This procedure allows the three front panel switches to be tested. Their states are returned in a record of type SWITCHTYPE containing three Boolean variables which are TRUE if the switch is being pressed, FALSE otherwise. The record contains the following members:

```
SW1STAT:BOOLEAN;  
SW2STAT:BOOLEAN;  
SW3STAT:BOOLEAN;
```

The following example program prints out the current states of the switches:

```
var anyname:switchtype;           {give the record a name}  
readswi(anyname);                 {get the switch states into the record}  
with anyname do                   {output them}  
    writeln(port1, 'SW1=',sw1stat,' SW2=',sw2stat,' SW3='sw3stat);
```

Note that if SW2 and SW3 are pressed together for more than approximately 1 second, the PPC will be reset! This cannot be disabled by any program.

VAL - convert a number

```
PROCEDURE VAL(VAR STRING; VAR I:INTEGER; VAR NEXTPOS:INTEGER; VAR SUCCESS:BOOLEAN);  
PROCEDURE VAL(VAR STRING; VAR R:REAL; VAR NEXTPOS:INTEGER; VAR SUCCESS:BOOLEAN);
```

This procedure converts an ASCII character string representing an **integer** or **real** number into the number's binary equivalent. It is useful if e.g. your program has read-in a line of text (containing numbers) into an array without conversion, and the numbers then have to be "read". It performs the same operations as the READ procedure when reading integers or reals; except that READ converts the characters *as they are being received* and cannot process characters already stored somewhere.

Characters from the variable STRING are converted into a numeric value which is stored in the variable I or R. The integer NEXTPOS returns the offset of the first character after the number *relative to the start* of the variable STRING; this allows successive invocations of VAL to read a series of numbers without requiring a preceding forward scan to find the end of each number.

SUCCESS is returned as TRUE if the value converted successfully. If FALSE, NEXTPOS then holds the location of the error, as an offset from the start of the string variable *which was given to VAL*.

Normally STRING will be a string variable (an array of char) but other types can be used. The rules which determine what constitutes a valid number, and what terminates a number, are as for the READ procedure. The following examples show some of the various uses of VAL.

Assume that the array src contains the following characters (where Δ represents white space)

```
ΔΔΔ234Δ23.45ΔΔΔ-2.56E8ΔΔ-2.56e8ΔΔ+28ΔΔ-289aaaa23k.....
```

then the following program code will act as commented on the right:

```
var src:array [1..100] of char; suc:boolean; ivalue,npos:integer; rvalue:real;  
val(src,ivalue,npos,suc);                    {skips initial 3 spaces and returns ivalue=234}  
val(src[1],ivalue,npos,suc);                 {same as above}  
val(src[4],ivalue,npos,suc);                 {same as above}
```

```

val (src [4], rvalue, npos, suc);      {reads 1st # as a real and returns rvalue=234}
val (src [7], rvalue, npos, suc);      {skips 1 space and returns rvalue=23.45}
val (src [7], ivalue, npos, suc);      {returns ivalue=23 ('.' is a valid integer terminator)}
val (src [16], rvalue, npos, suc);     {returns rvalue=-2.56E8}
val (src [25], rvalue, npos, suc);     {as above, lowercase E accepted}
val (src [34], ivalue, npos, suc);     {returns ivalue=28, leading + accepted}
val (src [39], ivalue, npos, suc);     {returns ivalue=-289, number terminated by trailing
                                        non-numeric characters ('a')}

val (src [46], ivalue, npos, suc);     {returns suc=FALSE because a number cannot
                                        start with a non-numeric character}

val (src [47], ivalue, npos, suc);     {returns ivalue=23}

```

The above examples do not make use of the `npos` return value. `npos` greatly assists with the conversion of multiple numbers, allowing VAL to pick-up from where the previous VAL finished. The following example reads the above array `src` and places the numbers into variables `x1,x2..`:

```

var x1,x5,x6,x7,last:integer; x2,x3,x4:real;

val (src [1],x1,npos,suc); last:=npos+1;    {last:=offset of last+1 char of the first #}
val (src [last],x2,npos,suc); last:=last+npos; {get the next number}
val (src [last],x3,npos,suc); last:=last+npos; {etc}
val (src [last],x4,npos,suc); last:=last+npos;
val (src [last],x5,npos,suc); last:=last+npos;
val (src [last],x6,npos,suc); last:=last+npos;
val (src [last+4],x7,npos,suc);             {this one needs to skip leading 'a' chars!}

```

Remember that `npos` is the *offset* of the last+1 character of the number relative to the start of the number *which was supplied to VAL*. The above program is therefore slightly subtle.

In practice, especially when reading mixed alpha/numeric data, you may need to use VAL in conjunction with a purpose-written procedure which scans the input string for the items which you actually want and then invokes VAL to convert the numbers, and which handles errors, etc.


INIT - initialise data


```
PROCEDURE INIT (VAR S; CONSTANT; CONSTANT ...);
```

This procedure is provided to allow easy initialisation of PASCAL variables (e.g. arrays), with a string of integer, real or string **constants**.

The constants are simply placed in memory starting at the address of **S**. The syntax diagram for a constant applies. The interpretation of the constant is according to the way it is typed-in, **not** according to the *type* of the variable being loaded:

'a'	loads a single byte 97 (dec), 61 (hex)
'abcd'	loads four bytes: 61,62,63,64 (hex)
#97	loads a single byte 97 (dec), 61 (hex) – '#' denotes a character
#\$61	as above
4	loads an integer (2 bytes): 04 00 (hex), 04 is loaded in the lower address
1234	loads an integer (2 bytes)
\$4d2	as above, hex value – '\$' by itself denotes an integer
\$04d2	as above
0	loads an integer (2 bytes)
4.0	loads a real (4 bytes) – a real must contain a decimal point or an exponent!
1234.0	loads a real (4 bytes)
1234E0	as above
-27.578E-23	loads a real (4 bytes)
-27.578e-23	as above
100000	loads a real (number is above 32767)

 INIT does not perform type checking. You can load anything – anywhere! Also, INIT does not prevent the loading of data into addresses *past the end* of the variable!

 Because of the lack of error checking, use INIT with great care and, if in any doubt, print-out the contents of the variable(s) to check that the result is what you intended.

Examples:

```

var c:array[1..11] of char;
init(c,'hello there');           {fills the array with the 11 chars}
init(c[1],'hello there');       {as above}
init(c,'hello',' there');       {as above}
init(c,#104,#101,#108,#108,#111,#$20,#$74,#$68,#$65,#$72,#$65); {as above}
init(c,#104#101#108#108#111#$20#$74#$68#$65#$72#$65);           {as above}
init(c,'hello');                {fills-in 1st 5 chars only: [1]..[5]}
init(c[7],'hello');             {fills-in the last 5 chars only: [7]..[11]}

var i:array[1..4] of integer;
init(i,2,23,234,0);             {fills the array with the 4 integers}
init(i[3],10000,11000);        {fills-in elements [3] and [4] only}

var r:array[1..4] of real;
init(r,1.0,2.5,-3.4,245.56E6);  {fills the array with the 4 reals}

```

Any PPC PASCAL variable can be initialised with INIT, including records and sets. Also, for longer statements, the constants can be spread over multiple lines:

```

var rec : record
  m : array [1..8] of char;
  n : array [1..4] of integer;
  p : integer;
  q : array [1..4] of real
end;

init(rec,'somedata',           {initialises member m}
      23,25,27,2567,           {initialises member n}
      123,                     {initialises member p}
      2.5,3.6,45E7,0.05e-8);  {initialises member q}

```

SUMBUF - compute a checksum over a buffer

```
FUNCTION SUMBUF(START,SIZE:INTEGER):CHAR;
```

This function computes a MOD 256 checksum over a buffer containing **char** data. The parameter START is the *address* of the buffer. The following example computes the checksum over 399 bytes of data and places the result into the last byte:

```

var buffer : array [1..400] of char;
buffer[400]:=sumbuf(addr(buffer),399);

```

XORBUF - compute a XOR over a buffer

```
FUNCTION XORBUF(START,SIZE:INTEGER):CHAR;
```

This function computes a XOR over a buffer containing **char** data. The parameter START is the *address* of the buffer. The buffer size must be 2 or greater. The first character is XORed with the second, the result is XORed with the third, the result of that is XORed with the fourth, etc. Example:

```
buffer[400]:=xorbuf(addr(buffer),399);
```

CRCBUF - compute a CRC over a buffer

```
FUNCTION CRCBUF(START,SIZE,INITIAL:INTEGER):INTEGER;
```

This function computes a 16-bit CRC over a buffer containing **char** data. The parameter START is the *address* of the buffer. The value INITIAL is the initialisation value for the CRC generator. The algorithm used is based on the standard CRC polynomial $x^{16}+x^{15}+x^2+1$.

This CRC is **not** the one used in MODBUS messages.

The following example loads the CRC of a 400-byte buffer (computed over its first 398 bytes) into its last two bytes:

```
buffer[399]:=crcbuf(addr(buffer),398,0);
```

The INITIAL parameter facilitates the computation of a single CRC over *multiple* buffers, as the following example shows:

```

var buffer1,buffer2 : array [1..10] of char;
  crc : integer;

buffer1:='aaaaabbbb';
buffer2:='ccccdddd';
crc:=crcbuf(addr(buffer1),10,0);      { get CRC of 1st buffer, init=0 }
crc:=crcbuf(addr(buffer2),10,crc);   { get CRC of 2nd buffer, init=last }

```

The final crc result above is identical to that produced by:

```

var buffer3 : array [1..20] of char;

buffer3:='aaaaabbbbccccdddd';
crc:=crcbuf(addr(buffer3),20,0);

```

_BIT - integer bit test

```

FUNCTION _BIT(VALUE,BITNUM:INTEGER):BOOLEAN;

```

This returns the state of a bit in an integer, as TRUE or FALSE if the bit is 1 or 0 respectively. Only the four least significant bits of BITNUM are used, limiting it to 0..15.

_AND - integer bit-wise AND

```

FUNCTION _AND(VALUE1,VALUE2:INTEGER):INTEGER;

```

This performs a bit-wise AND of the 16 bits constituting each of the two integers and returns the integer result. For example:

```

var a:integer;

a:=_and(a,$0FFF);      { zero top four bits }

```

_OR - integer bit-wise OR

```

FUNCTION _OR(VALUE1,VALUE2:INTEGER):INTEGER;

```

This performs a bit-wise OR of the 16 bits constituting each of the two integers and returns the integer result. For example:

```

var a:integer;

a:=_or(a,$000F);      { set lowest four bits }

```

_XOR - integer bit-wise XOR

```

FUNCTION _XOR(VALUE1,VALUE2:INTEGER):INTEGER;

```

This performs a bit-wise XOR of the 16 bits constituting each of the two integers and returns the integer result.

_CPL - integer bit-wise complement

```

FUNCTION _CPL(VALUE:INTEGER):INTEGER;

```

This inverts the bits in the integer VALUE and returns the integer result.

_SHL - integer shift left

```

FUNCTION _SHL(VALUE,PLACES:INTEGER):INTEGER;

```

This shifts VALUE left by PLACES bit positions, with zero fill, and returns the integer result. Any carry out from the top bit (bit 15) is discarded. This function is faster than the alternative: shifting left by multiplying by a power of two. Only the four least significant bits of PLACES are used, limiting it to 0..15.

_SHR - integer shift right

```

FUNCTION _SHR(VALUE,PLACES:INTEGER):INTEGER;

```

This shifts VALUE right by PLACES bit positions, with zero fill, and returns the integer result. Any carry out from the lowest bit (bit 0) is discarded. This function is faster than the alternative: shifting right by dividing by a power of two. Only the four least significant bits of PLACES are used, limiting it to 0..15.

ROL - integer rotate left

```
FUNCTION _ROL(VALUE, PLACES: INTEGER): INTEGER;
```

This rotates VALUE left by PLACES bit positions and returns the integer result. Any carry from bit 15 is fed back into bit 0. Only the four least significant bits of PLACES are used, limiting it to 0..15.

ROR - integer rotate right

```
FUNCTION _ROR(VALUE, PLACES: INTEGER): INTEGER;
```

This rotates VALUE right by PLACES bit positions and returns the integer result. Any carry from bit 0 is fed back into bit 15. Only the four least significant bits of PLACES are used, limiting it to 0..15.

BITC - char bit test

```
FUNCTION _BITC(VALUE, BITNUM: CHAR): BOOLEAN;
```

This returns the state of a bit in a character, as TRUE or FALSE if the bit is 1 or 0 respectively. Only the three least significant bits of BITNUM are used, limiting it to 0..7. Note that the bit number is specified as a `char` so, for example, if you want to test bit 'n' where 'n' is held as an integer, the `chr` function must be used to convert the integer into a `char`:

```
var n:integer;
    c:char;
    state:boolean;
state:=_bitc(c,chr(n));
```

The bit number can of course be specified as an immediate value, using the `#` operator:

```
state:=_bitc(c,#6);      { test bit 6 }
```

ANDC - char bit-wise AND

```
FUNCTION _ANDC(VALUE1, VALUE2: CHAR): CHAR;
```

This performs a bit-wise AND of the 8 bits constituting the two chars and returns the char result. For example:

```
var a:char;
a:=_andc(a,#$0F);      { zero top four bits }
```

ORC - char bit-wise OR

```
FUNCTION _ORC(VALUE1, VALUE2: CHAR): CHAR;
```

This performs a bit-wise OR of the 8 bits constituting the two chars and returns the char result. For example:

```
var a:char;
a:=_orc(a,#$0F);      { set lowest four bits }
```

XORC - char bit-wise XOR

```
FUNCTION _XORC(VALUE1, VALUE2: CHAR): CHAR;
```

This performs a bit-wise XOR of the 8 bits constituting the two chars and returns the char result.

CPLC - char bit-wise complement

```
FUNCTION _CPLC(VALUE: CHAR): CHAR;
```

This inverts the bits in the char VALUE and returns the char result.

_SHLC - char shift left

```
FUNCTION _SHLC(VALUE, PLACES:CHAR):CHAR;
```

This shifts VALUE left by PLACES bit positions, with zero fill, and returns the char result. Any carry out from the top bit (bit 7) is discarded. This function is faster than the alternative: shifting left by multiplying by a power of two. Only the three least significant bits of PLACES are used, limiting it to 0..7. See the _BITC function for information on specifying the PLACES variable.

_SHRC - char shift right

```
FUNCTION _SHRC(VALUE, PLACES:CHAR):CHAR;
```

This shifts VALUE right by PLACES bit positions, with zero fill, and returns the char result. Any carry out from the lowest bit (bit 0) is discarded. This function is faster than the alternative: shifting right by dividing by a power of two. Only the three least significant bits of PLACES are used, limiting it to 0..7. See the _BITC function for information on specifying the PLACES variable.

_ROLC - char rotate left

```
FUNCTION _ROLC(VALUE, PLACES:CHAR):CHAR;
```

This rotates VALUE left by PLACES bit positions and returns the char result. Any carry from bit 7 is fed back into bit 0. Only the three least significant bits of PLACES are used, limiting it to 0..7. See the _BITC function for information on specifying the PLACES variable.

_RORC - char rotate right

```
FUNCTION _RORC(VALUE, PLACES:CHAR):CHAR;
```

This rotates VALUE right by PLACES bit positions and returns the char result. Any carry from bit 0 is fed back into bit 7. Only the three least significant bits of PLACES are used, limiting it to 0..7. See the _BITC function for information on specifying the PLACES variable.

STRCMP - compare strings

```
FUNCTION STRCMP(S1, S2:INTEGER):INTEGER;
```

This compares string S1 with string S2. Both strings are assumed to be terminated with #0. The return value is negative, zero or positive if S1<S2, S1=S2, S1>S2 respectively. Note that S1 and S2 are *addresses* of the strings.

```
var s1,s2 : array [1..20] of char;  
init(s1, 'Hello', #0);  
init(s2, 'Helloq', #0);  
writeln(port1, strcmp(addr(s1), addr(s2)));
```

produces a result of -2 whose sign indicates that s1<s2. Note that the magnitude of the result is simply the difference between the mismatched characters.

STRNCMP - compare strings, n chars

```
FUNCTION STRNCMP(S1, S2, N:INTEGER):INTEGER;
```

This is as STRCMP but only the first N characters are compared. Referring to STRCMP example above:

```
writeln(port1, strncmp(addr(s1), addr(s2), 4)); { result = 0 }  
writeln(port1, strncmp(addr(s1), addr(s2), 5)); { result = -2 }
```

MEMCPY - memory copy

```
PROCEDURE MEMCPY(SOURCE, DEST, BLOCKSIZE:INTEGER);
```

A block of bytes, size BLOCKSIZE, is copied from SOURCE to DEST. The source and destination blocks may overlap. No checking takes place on any of the input values! Example:

```
var block1, block2 : array [1..20] of char;  
init(block1[3], 'hello');  
memcpy(addr(block1[3]), addr(block2[11]), 5);
```

After the above is executed, block2 contains

```
?????????Hello?????
```

where ? represents an undefined character.

TOUPPER - convert to uppercase

```
FUNCTION TOUPPER(INCHR:CHAR):CHAR;
```

If the character INCHR is in the range "a" to "z", it is converted to uppercase.

CBRK - check for a break character

OBRK - output a break character

```
FUNCTION CBRK (PORT:TEXT) : BOOLEAN;  
PROCEDURE OBRK (PORT:TEXT);
```

The above functions work on ports **3** and **4** only. They are not supported on a PPC-2.

The above functions support detection and generation of line break conditions. A "break" is a sequence of "1" bits on the line whose duration exceeds the duration of a complete character at that baud rate. A "break" therefore represents a framing error. Breaks are used with some equipment to indicate a special condition.



The PPC checks for a break condition in the UART (on P3 & P4 only) every 1ms and latches any break condition in a flag. CBRK returns the state of this flag, and clears the flag. CBRK may therefore fail to detect a break if the break duration is less than 1ms.

OBRK outputs a break of a fixed duration, 200ms, regardless of the current baud rate.

PPC break condition checking

As well as using CBRK, you can detect received break conditions through the reception of a null (00h) character. This character always appears in the receive queue as a result of a break condition on the input port. However, the behaviour of ports 1,2 differs significantly from ports 3,4:

On ports 3,4 the reception of a break character returns a *single* null (00h) character in the receive queue. This is regardless of how long the break condition persists.

On ports 1,2 the reception of a break character returns *multiple* nulls; the number of nulls is equivalent to the number of ordinary characters that would be received at the current baud rate during the duration of the break condition. For example, if the port is set to 9600 baud, and the break persists for 270ms, then approximately 270 nulls are received. This would cause the input queue to overflow, and cause the PPC to return an 'input busy' handshake condition.

In applications where break characters are expected on port 1/2, this situation can be easily handled in software, as the following Port 2 example code shows:

```
var c:char;  
    brkrecd:boolean;  
  
brkrecd:=false;           { assume break not received }  
bread (port2,c);         { wait for a char on port 2 }  
if c=#0 then begin      { if null received }  
    loadtimer(0,300);    { then wait for 300ms }  
    while readtimer(0)<>0 do;  
        ipqclear(port2); { and clear the input queue }  
        brkrecd:=true;   { and mark 'break received ' }  
end;
```

The above code assumes that the input break condition will last for less than 300ms.

Analog Subsystem PASCAL functions

These functions are available only on a PPC fitted with the optional 16-bit Analog Subsystem option. For full details of this option, see the Analog Subsystem Option chapter.

ADCINI - initialise ADC

```
FUNCTION ADCINI:INTEGER;
```

This function should be called at the start of any program which accesses the PPC Analog Subsystem. It invokes the ADC self-calibration cycle which takes about 1.8 seconds. A non-zero return indicates that the Analog Subsystem option is not installed, or is faulty. This function also sets the output of the two D-A converters to zero. It is also the recommended way of checking if the Analog Subsystem is fitted.

ADC - read analog-digital converter ADCC - read analog-digital converter (calibrated)

```
FUNCTION ADC (CHAN:INTEGER) : INTEGER;  
FUNCTION ADCC (CHAN:INTEGER) : INTEGER;
```

This function initiates an A-D conversion on the channel specified by **CHAN** and returns a signed integer which represents the analog input voltage reading. The channel number **CHAN** is 1..18, used as follows:

- 1..8 channels AIN1..AIN8, use with inputs configured for voltage mode ($\pm 100\text{mV}/\pm 1\text{V}$)
(either ADC or ADCC function may be used)
- 9 on-board temperature sensor (use ADC function only, not ADCC)
- 10 invalid
- 11..18 channels AIN1..AIN8, use with inputs configured for current loop mode (4-20mA)
(use ADCC function only, not ADC)

On the standard product version, a $\pm 100\text{mV}/\pm 1\text{V}$ nominal input range produces a digital reading in the range -32767 to +32767. A positive overvoltage is returned as +32767, and a negative overvoltage is returned as -32767.

The temperature (channel 9) would be used for temperature compensation only and the precise temperature does not need to be known. However, it can be converted to °K by dividing by 81.92.

The ADC function returns a raw uncalibrated analog-digital converter reading which can have a scale error of up to $\pm 5\%$ of full scale. The zero error is within $\pm 3\text{LSBs}$, however. (The input voltage range of $\pm 100\text{mV}/\pm 1\text{V}$ is guaranteed to be accommodated). It is slightly faster than ADCC.

The ADCC function returns a highly accurate calibrated value. The calibration is transparent but you can see the functions ANEERD and ANEEWR for details of how it is done.

When using ADCC to read in current loop mode, it returns a calibrated value in the range of 6553..32767 corresponding to a transducer current in the range 4..20mA. If you require the transducer's "zero" (current=4mA) to return a zero reading, you must remove the "4mA" offset in your software.

ADCGAIN - set $\pm 100\text{mV}$ / $\pm 1\text{V}$ gain

```
PROCEDURE ADCGAIN(GAIN:INTEGER);
```

This sets the gain for all subsequent readings for channels 1..8 in voltage mode. The permitted values of **GAIN** are 1 (for $\pm 1\text{V}$ range) or 10 (for $\pm 100\text{mV}$ range). Other values have no effect. The current gain setting is ignored when reading in 4-20mA current mode.

DAC - drive digital-analog converter DACC - drive digital-analog converter (calibrated)

```
PROCEDURE DAC (CHAN,VALUE:INTEGER);  
PROCEDURE DACC (CHAN,VALUE:INTEGER);
```

This outputs a signed integer **VALUE** to the D-A channel **CHAN**. Input values in the range -32767 to +32767 produce an output in the range -10V to +10V. Valid channel numbers are 1 or 2.

The DAC procedure outputs **VALUE** direct to the converter, with no corrections. It is slightly faster than DACC but the output can exhibit a scale error up to $\pm 5\%$, plus a zero error up to $\pm 3\%$.

The DACC procedure outputs a calibrated value. See ANEERD and ANEEWR for details.

Execution times

The following table shows the approximate execution times of functions listed above:

	Standard PPC	-H2 option
ADC	350µs	200µs
ADCC (voltage mode)	490µs	250µs
ADCC (4-20mA mode)	650µs	330µs
DAC	280µs	130µs
DACC	430µs	190µs

The above timings do not apply to the on-board temperature sensor (channel 9), for which the figures are approximately 800µs and 450µs for the standard and -H2 PPC respectively.

ANEERD - read analog subsystem calibration EEPROM

ANEWR - write analog subsystem calibration EEPROM

FUNCTION ANEERD (ADDRESS:INTEGER) : INTEGER;

FUNCTION ANEWR (ADDRESS,VALUE:INTEGER) : INTEGER;



You do not need to know about these functions unless you wish to alter the PPC's analog calibration table. This table is factory-loaded and if you modify it incorrectly you may have to return the PPC for re-calibration!

When the ADCC or DACC functions are invoked, the PPC performs an automatic zero and scale calibration on the 16-bit integer value involved. The calibration values are stored in a dedicated EEPROM located on the analog card. The EEPROM can hold up to 64 16-bit integer values of which only the first 21 are currently used, as shown below:

Address	Nominal value	Function
0,1	0,16384	zero & scale cal for all 8 ADC inputs, ±1V
2,3	0,16384	zero & scale cal for all 8 ADC inputs, ±100mV
4..11	16384	scale cal for current loop mode on inputs 1..8
12..19	0,16384..	zero & scale cal pairs for DAC outputs 1..4
20	16384	scale cal for +10V auxiliary output (uninitialised)
21..63	—	not used at present

The functions ANEERD and ANEWR can be used to read and write, respectively, individual integer locations in the EEPROM. Function ANEWR returns an errorcode which, if non-zero, indicates either an invalid address or a EEPROM write error.

If you modify any of the calibration values, the new value will not become active until the ADCINI function has been called, or until power-down.



If you must use ANEWR, test your program first on one of the unused locations 21..63.



In common with all EEPROMs, the EEPROM has a limit of 10,000 writes per location.

DIN1 - read TTL input #1

DIN2 - read TTL input #2

FUNCTION DIN1 : BOOLEAN;

FUNCTION DIN2 : BOOLEAN;

The Boolean value returned by this function is the current unlatched state of the TTL level input AIN1 or AIN2. A TTL HIGH level is returned as TRUE.

DOUT1 - drive TTL output #1

DOUT2 - drive TTL output #2

PROCEDURE DOUT1 (ONOFF:BOOLEAN);

PROCEDURE DOUT2 (ONOFF:BOOLEAN);

The Boolean value supplied to this procedure controls the state of the DOUT1 or DOUT2 output. A TRUE value sets it high (+5V), and a FALSE value sets it low (0V).

DG1 Parallel Digital I/O Card Pascal functions

These functions are available only on a PPC fitted with the -DG1 option, the Parallel Digital I/O Card. This card offers 16 TTL-level schmitt inputs, and 16 high-current open-collector outputs.

PDIN - read parallel digital inputs

```
FUNCTION PDIN : INTEGER;
```

This function returns the states of the 16 inputs in the corresponding 16 bits of the 16-bit integer. For example, if the current states of the 16 input port bits are

```
0001 0010 0011 1111
```

then this function will return the value 123F hex (4671 decimal) in the integer variable. If the most significant (leftmost) input is high, the returned integer will be negative.

PDOUT - write parallel digital outputs

```
PROCEDURE PDOUT(VALUE:INTEGER);
```

This procedure outputs the 16 bits of the integer VALUE to the corresponding output port bits. The example

```
pdout($123f);
```

will output the following bit pattern on the 16 outputs:

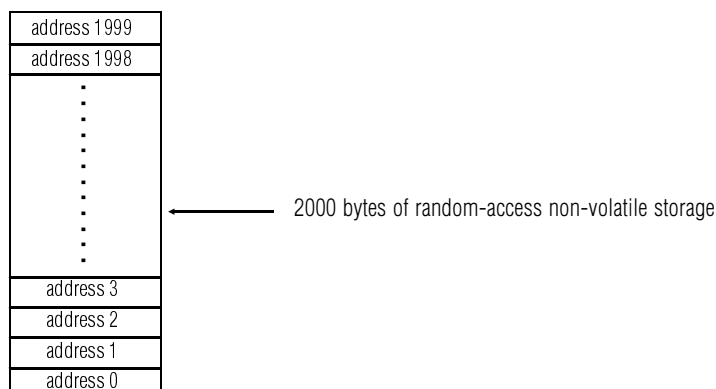
```
0001 0010 0011 1111
```

With both above functions, the most significant bit of the integer corresponds to bit 15 of the input or output port. The least significant bit of the integer corresponds to bit 0 of the port.

Further information on the DG1 card is in the I/O Expansion Options chapter.

EEPROM non-volatile access functions

The following functions support the storage and retrieval of data in a specially-reserved 2000-byte area of the EEPROM non-volatile memory:



This is a very useful feature which permits the construction of a device which can self-modify its properties and retain the changes through power-down. Examples include the storage of passwords in a security device, or the storage of calibration coefficients for the analog subsystem.

Before using any of these functions, you must understand that **an EEPROM is not the same as a RAM** – even though the functions provided do allow the EEPROM to be treated as a random-access device. The following considerations are most important:

LIMITED WRITE ENDURANCE: An EEPROM has a limit on the number of writes which can be done on any particular address. If this limit is exceeded, the EEPROM may be permanently damaged. The write limit in the device used in the PPC (currently XICOR 28C256) is guaranteed at **10,000 writes**, although manufacturers' data shows that failures do not typically occur until the number of writes *to the same location* is well past 100,000. This limit applies on a *per-location* basis, i.e. if you write a particular address until that location is totally destroyed (e.g. 1,000,000 times) you will **not** have affected any other addresses in the EEPROM.

Because of the above, whenever you write a program which writes to the EEPROM, you should comment-out the actual write function and insert debug (e.g. WRITE) statements which output the EEPROM addresses being written. This should prove that your program is writing the EEPROM only when it should be (i.e. not very often!). If your application requires the storage function in order to run at all, you can debug your program with a 2000-byte array simulating the EEPROM.

EXECUTION TIME: The write operation takes approximately **5ms** (0.005 of a second) *per byte written*. The NVWRITEC function therefore takes approximately 5ms and the NVWRITEI takes about 10ms. If speed is essential, these functions are suitable only for writing a small number of bytes. The NVWRBLK function uses a different hardware approach known as "fast page mode" and takes approximately 5ms per 64 bytes written; a 700-byte block will therefore write in about 60ms. Note that these timings are *typical* and your program must not rely on them.

SECURITY: The Delete All function in the PPC Executive does not affect the 2000-byte storage area. This means that the only way of deleting confidential data (e.g. passwords) stored in this area is to write a program which overwrites it.

There are no limits on *reading* the EEPROM. Reads are very fast, although the software overhead involved in the function call means they are not as fast as e.g. reading data from an array in RAM.

The functions NVWRITEC and NVWRITEI (only) perform a pre-read of the location and an actual write of the EEPROM takes place only if the existing contents of the location *differs* from the value being written. Depending on the application, this can prolong the typical life of the EEPROM but the main reason for it is to prevent damage if a runaway program loops on a *constant* EEPROM address while writing a *constant* value to it. This pre-read scheme means that the actual execution of an EEPROM write operation can vary greatly – down to a few % of the 5ms typical figure. The block function NVWRBLK does **not** do a pre-read and must be used with even greater care.

Each of the EEPROM functions returns an integer error code. A non-zero value indicates an error:

- 1:** Invalid EEPROM address (return code = 1). Note that an attempt to read or write e.g. a 500-byte block starting at address 1700 will also return an error, because the block must be *entirely* contained within the 2000-byte area. Similarly, the highest address for integer reads/writes (NVWRITEI or NVREADI) is 1998, not 1999.
- 2:** An EEPROM write error (return code = 2). This should never occur, unless the EEPROM device is faulty or your program has destroyed one or more locations in it.

All the write functions perform a verify after the write, so provided that the return code is zero, there is normally no need to read back the EEPROM to ensure the data was correctly written.

An EEPROM has great advantages over the "battery-backed CMOS RAM" non-volatile storage schemes used by other equipment manufacturers: extremely high reliability and long life. It is almost impossible to corrupt the data stored in an EEPROM, other than by modifying it with the functions provided. The storage life is guaranteed at 10 years and the predicted typical figure (at 25°C) is >100 years.



A power loss during an EEPROM write operation will cause the write operation to be terminated, and could even result in loss of data at addresses other than those being written at the time of the power loss.

NVWRITEC - write a char

```
FUNCTION NVWRITEC(EEPROMADDR: INTEGER; C: CHAR): INTEGER;
```

This function stores a character at a specified address in the 2000-byte EEPROM non-volatile area. Valid addresses are 0..1999 (decimal). Execution time is 10ms max (5ms typ). The following example reads chars from Port 3, discards everything up to a '*' and then loads the next 20 chars into the non-volatile area starting at address 1500:

```
var inputchar:char;
    chrcount,nvaddress,retcode:integer;

repeat
    bread(port3,inputchar);           {discard chars until '*'}
until inputchar='*';

nvaddress:=1500;                      {initial EEPROM address}
chrcount:=20;                          {process 20 more chars}

repeat
    bread(port3,inputchar);           {wait for a char}
    retcode:=nvwritec(nvaddress,inputchar); {write it into EEPROM}
    nvaddress:=nvaddress+1;           {EEPROM address +1}
    chrcount:=chrcount-1;             {char counter -1}
until (chrcount=0) or (retcode<>0);   {and repeat}

if retcode<>0 then                     {optional retcode test}
    writeln(port1,'NVWRITEC error:',retcode); {if error, report}
```

Note how the above program tests the return code. This should always be done, at least during program debugging.

NVREADC - read a char

```
FUNCTION NVREADC(EEPROMADDR: INTEGER; VAR C: CHAR): INTEGER;
```

This function is the opposite of the NVWRITEC function above. It reads a character from the EEPROM. The following code will print-out the data stored by the above NVWRITEC program:

```
repeat
    retcode:=nvreadc(nvaddress,inputchar); {read the EEPROM}
    write(port1,inputchar);                 {output the char to P1}
    nvaddress:=nvaddress+1;                 {EEPROM address +1}
    chrcount:=chrcount-1;                   {char counter -1}
until (chrcount=0) or (retcode<>0);        {and repeat}

if retcode<>0 then                           {optional retcode test}
    writeln(port1,'NVREADC error:',retcode); {if error, report}
```

NVWRITEI - write an integer

```
FUNCTION NVWRITEI(EEPROMADDR,VALUE: INTEGER): INTEGER;
```

This is identical to NVWRITEC except that two bytes representing an integer are written. Execution time is 20ms max (10ms typ). Note that valid addresses are 0..1998 (decimal), **not** 0..1999. The *lower* (less significant) byte of the integer is stored at the *lower* address, although you would not normally need to know this.

NVREADI - read an integer

```
FUNCTION NVREADI(EEPROMADDR: INTEGER; VAR I: INTEGER): INTEGER;
```

This function is the opposite of the NVWRITEI function above. It reads two bytes representing an integer from the EEPROM. Valid addresses are 0..1998 (decimal).

NVWRBLK - write a block

FUNCTION NVWRBLK(SOURCE, EEPROMADDR, BYTES: INTEGER): INTEGER;

This function writes a block up to 2000 bytes in length from any PASCAL data structure (e.g. from an array) to the EEPROM, in a single operation, given a memory *source* address and an EEPROM *destination* address. The same functionality could of course be achieved with a sequence of NVWRITEC calls but NVWRBLK is much faster for large blocks of data, because it uses a different underlying hardware mechanism to access the EEPROM. The execution time varies according to several factors but is around 12ms max (6ms typ) for every 64 bytes in the block being written; a 640-byte block will therefore write in 120ms max (60ms typ).

The following conditions will return an errorcode=1:

- 1 An "overlapping" block. Valid EEPROM addresses are 0..1999 but the entire block being written must be written *inside* the 2000-byte area. Thus, when writing e.g. a 500-byte block, the valid starting EEPROM address range is 0..1499; the block will be written to addresses 1499..1999 inclusive.
- 2 Starting EEPROM address out of the 0..1999 range.
- 3 A *source* address which starts below the PASCAL program/data memory area (currently 8900 hex, 35072 decimal). This helps to prevent some errors because, in any functioning program, only data stored in PASCAL data structures should be written to the EEPROM.

Note that since the *source* address will normally be above 32767, it will be represented by a *negative* integer.

The following example program will write the *second half* of a 400-byte array into the EEPROM at addresses 1000..1199:

```
var buffer:array[1..400] of char;
    memaddr, eeaddr, blksize, retcode: integer;

memaddr:=addr(buffer[201]);
eeaddr:=1000;
blksize:=200;

retcode:=nvwrblk(memaddr, eeaddr, blksize);
if retcode<>0 then
    writeln(port1, 'NVWRBLK failed: ', memaddr, eeaddr, blksize, retcode);
```

NVRDBLK - read a block

FUNCTION NVRDBLK(EEPROMADDR, DEST, BYTES: INTEGER): INTEGER;

This function is the opposite of NVWRBLK. It reads a block up to 2000 bytes in length from the EEPROM to any PASCAL data structure. The returned error codes are as per NVWRBLK. To trap some errors, this function will refuse to write the data into memory whose address is below PASCAL code/data areas.

The following example program will fill the *second half* of a 400-byte array from the EEPROM addresses 1000..1199 which were written in the NVWRBLK example program:

```
var buffer:array[1..400] of char;
    memaddr, eeaddr, blksize, retcode: integer;

memaddr:=addr(buffer[201]); eeaddr:=1000; blksize:=200;

retcode:=nvrdblkc(eeaddr, memaddr, blksize);
if retcode<>0 then
    writeln(port1, 'NVRDBLK failed: ', eeaddr, memaddr, blksize, retcode);
```

COMMENTS

A comment within a PASCAL program may occur between any two reserved words, numbers, identifiers or special symbols. A comment starts with a { character or the (* character pair. Unless the next character is a \$ all characters are ignored until the next } character or *) character pair. If a \$ was found then the compiler looks for a series of compiler options (see below) after which characters are skipped until a } or *) is found. For example:

```
i:=i+1; {bump the loop count}
i:=i+1; (* bump the loop count *)

{$L+ turn the listing on from now}
(*$L+ turn the listing on from now*)
```

COMPILER OPTIONS

These one-letter options that can be used to control the compilation process; some of them can be used throughout your program while others can only appear at the start of the program. For example:

```
{$C-,A- this text can be anything}
```

turns off break checks and array index checks.

General Options

These may appear anywhere in the program.

Option L

Controls the listing of the program text and object code address by the compiler.

If L+ then a full listing is given. If L- then lines are only listed when an error is detected. Default is L+.

A compilation listing of an "autoexec" or an encrypted program is suppressed, regardless of the setting of this option.

Option S

Controls whether or not stack checks are made.

If S+ is selected then at the beginning of each procedure and function call a check is made to see if the stack will *probably* overflow in this block. If the runtime stack overflows the dynamic variable heap or the program then the message **Out of RAM at PC=XXXX** is displayed and execution aborted. Naturally this is not foolproof; if a procedure has a large amount of stack usage within itself then the program may 'crash'. Alternatively, if a function contains very little stack usage while utilising recursion then it is possible for the function to be halted unnecessarily.

If S- is selected then no stack checks are performed; this improves execution speed. Default is S+.

Option A

Controls whether checks are made to ensure that array indices are within the bounds specified in the array's declaration.

If A+ is selected and an array index is too high or too low then the message **Index too high** or **Index too low** will be displayed and the program execution halted. If A- is selected then no such checks are made; this improves execution speed. Default is A+.

Option I

When using 16 bit 2's complement integer arithmetic, overflow occurs when performing a >, <, >=, or <= operation if the arguments differ by more than MAXINT (32767). If this occurs then the result of the comparison will be incorrect. This will not normally present any difficulties; however, should you wish to compare such numbers, the use of I+ ensures that the results of the comparison will be correct, at the cost of a reduction in execution speed. The equivalent situation may arise with real arithmetic in which case an overflow error will be issued if the arguments differ by more than approximately 3.4E38; this cannot be avoided.

If I- is selected then no check for the result of the above comparisons is made. Default is I-.

Option P

This option is used to send the compilation listing to a specified PPC port, which is assumed to have an ASCII printer connected to it. The Postscript output mode is not available.

The letter P is followed by a single digit in the range 1-4 and this specifies the port. For example

```
{ $P2 }
```

will send the compilation listing to a printer on port 2. If no P option is specified then the compilation listing is sent to the terminal on port 1. Note that the compilation listing is generated only if the L+ option has been selected.

The printer initialisation string which is configurable in the Executive is sent to the printer just before the listing starts.

Top-of-File Options

The following options must appear before any PROGRAM statement since they all have an effect on the object code produced. Normally, therefore, you would place these options in the first line of your program.

Option R

This controls whether errors generated *when reading numbers* are fatal. R- suppresses the **Number expected** or **Exponent expected** errors in READ or READLN data input. Default is R+.

If your program is correct and if the input data is correct then errors should not occur, but spurious characters can appear after power-up and unnecessarily lead to fatal errors. It is prudent programming practice to read data one byte at a time into a buffer and, at a suitable point, process the buffer (e.g. with the VAL procedure); this avoids the above errors in a proper manner.

Option O

This controls whether general *runtime* errors are fatal. If O+ (the default) is selected then checks are made for errors in the various integer and real arithmetic operations including log and trig functions, and for overflow when reading numeric values with READ or READLN. O- disables these checks and results in higher execution speed, and suppresses the following error messages: **Number too large**, **Overflow**, **DIV by zero**.

The O- option is intended for *finished and debugged* programs which execute in the "autoexec" mode and therefore may have no means of reporting runtime errors.



Ignored runtime errors will result in undefined variables, or worse! A program in which an error occurs is almost certainly not going to do what it was written to do. Disabling runtime errors is not a substitute for writing and debugging your program properly.

Option C

Controls whether or not keyboard checks are made during object code program execution.

If C+ (the default) is selected then a CTRL-C character (#3) received on Port 1 breaks program execution, reports the address where the break was detected, and passes control to the Executive.

When enabled, this check is made at the beginning of all loops, procedures and functions. Thus you may use this facility to detect e.g. which loop is causing the program to hang. However, break checking is not implemented in some I/O loops: e.g. it will not work if your program is "hanging" because an output port has been prevented from transmitting by a BUSY handshake from the receiving device. The only exit then is to reset the PPC, or to write the program to check the handshake before outputting and, if BUSY, check if a CTRL-C character has arrived on Port 1.

If C- is selected then the check is not made, resulting in higher execution speed.



If, after your program has started executing, you have pressed any key *other than ctrl-c*, then the break checking will cease to function. This is because the ctrl-c character (#3) is seen as a "break" character only if it appears *at the front* of the Port 1 input queue. The only way to stop a program in such a case is with the front panel reset (SW2+SW3). Obviously, one way to keep break working is to regularly read from Port 1.

Break checks are automatically disabled on "autoexec" programs, regardless of the setting of this option. However, for the highest execution speed, C- should still be selected.

Option U

When this option is selected, lowercase and uppercase are not treated as equivalent. U+ (or just U by itself) makes the compiler accept upper and lower case letters as being different so that the identifiers `Loop`, `LOOP`, `LoOp` are 3 separate identifiers; without the U+ option these 3 would be converted to one identifier `LOOP` by the compiler.

The default is no U option, i.e. upper/lowercase equivalence. This allows all of the PASCAL program to be entered in lowercase (except literal constants such as 'ABCD' and 'A' which are *always* case-sensitive) and makes it more readable.



When using U+ you must make sure that all reserved words and predefined identifiers (of the latter there are many!) are in UPPERCASE.

Program and Data limits

After each successful compilation (in the Compile Only mode) the compiler displays figures which show how much of the available memory has been used by the program. This section describes the meaning of those figures.

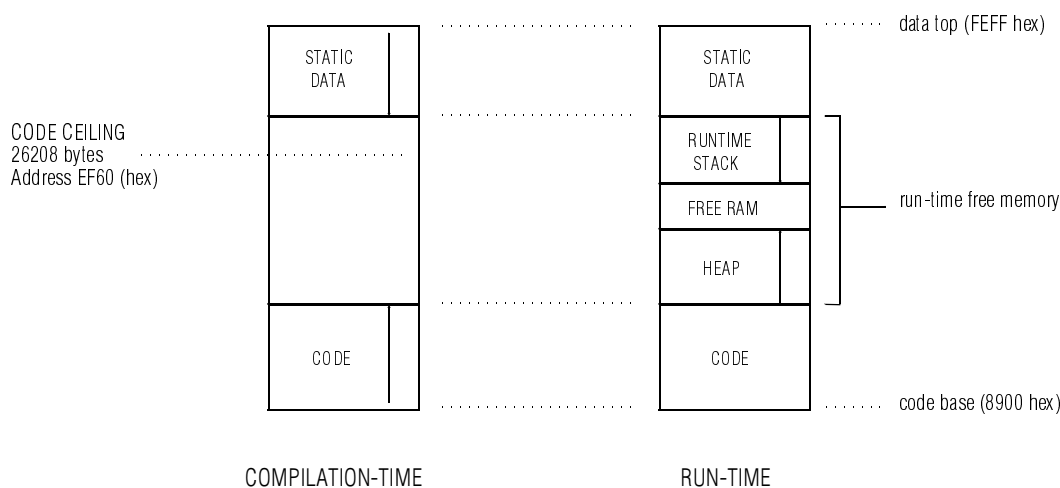
The information is in the form:

RAM usage (decimal): code=AAAA/BBBB (XX%) statics=CCCC/DDDD (YY%)

where

AAAA is the number of bytes of code generated
 BBBB is the total amount of space available for code (approx 26,000 bytes)
 CCCC is the number of bytes of static data allocated during compilation
 DDDD is the total amount of space available for static data (approx 30,000 bytes)
 XX is simply $AAAA*100/BBBB$
 YY is simply $CCCC*100/DDDD$

The slight complication is that code and data share the *same* 30,000-byte (approx) block of memory. Therefore, the space for code is reduced by the amount of data generated, and the space for data is reduced by the amount of code generated. During compilation, code is placed into memory from bottom upwards, and static data is allocated from top downwards. When the two meet (somewhere in the middle), you have run out of memory! The following diagram illustrates the memory allocation:



Static data is allocated during compilation from top down, and can extend *almost* all the way down to top of code. What is left in between will be the *run-time free memory*, and the compiler will always leave at least 512 bytes spare. As an example, in a program which contains

```
var a : array [1..10000] of integer;    {this uses 20,000 bytes}
```

there will be less than 10,000 bytes left for all the code. In addition, there is a code ceiling of 26208 bytes which corresponds to a hex address (the value displayed during compilation) of EF60. This ceiling is imposed by implementation factors beyond the scope of this manual.

Static data represents all variables declared with **VAR** statements in the *outermost* block of a program, and such variables are always accessible from everywhere in the program. The **heap** is memory used for dynamically-allocated variables, i.e. those allocated with **NEW**. The **run-time stack** is used to implement function and procedure call and returns, and holds all local variables (variables declared *within* functions and procedures).

In summary, the sum of **code plus static data** must not exceed approximately 30,000 bytes. Programs which violate this will not be executed. However, if you periodically compile your program as it is being developed, you will have a good idea how much space is left at any point.

Error messages

Compilation error messages

An arrow ^ indicates the approximate position of the error within the program. The actual error location is usually *just before* the arrow and, if a Pascal statement is not entirely resident on a single line, the error could be on a line preceding the arrow position.

Most of the error messages are self-explanatory.

Number is too large

Semi-colon or END expected before here

Undeclared identifier

Identifier expected

'=' not ':=' in constant declarations

'=' expected

This identifier can't begin a statement

':=' expected

)' expected

Wrong type combination

.' expected

Factor expected

Constant expected

Identifier is not a constant

'THEN' expected

'DO' expected

'TO' or 'DOWNTO' expected

(' expected

Can't write this type of expression

'OF' expected

comma expected

colon expected

'PROGRAM' expected

Variable expected as parameter

'BEGIN' expected

Variable expected in READ

Cannot compare expressions of this type

Type INTEGER or REAL expected

Can't read this type of variable

Identifier is not a type

Exponent expected in real number

Scalar expression expected

Null strings not allowed: use CHR(0)

'[' expected

']' expected

Array index type must be scalar

..' expected

']' or ',' expected in ARRAY decl

Lowerbound upperbound

Set too large (256 elements)

Function result must be type identifier

',' or ']' expected in set

..' or ',' or ']' expected in set

Parameter type must be type identifier

Null set not allowed here

Scalar (including real) expected

Scalar (not real) expected

Sets incompatible

'>' and '<' can't be used with sets

Hex digit expected

Array too large (64K!)

'END' or ',' expected in RECORD defn

Field identifier expected

Variable expected after 'WITH'
Variable in WITH must be RECORD type
No associated WITH statement
Unsigned integer expected after LABEL
Unsigned integer expected after GOTO
Label at wrong level
Undeclared label
Can only use equality tests on pointers
Bad string constant
Too many ':'s. Only e:m:h valid
Strings can't have EOLNs
Parameter must be a POINTER variable
Variable parameter required
BREAD/BWRITE supports variables only
SIZE takes variable or type identifier
Symbol Table Full
TOO MUCH CODE or TOO MUCH DATA

Runtime error messages

When a runtime error is detected then one of the following messages will be displayed, followed by at PC=XXXX where XXXX is the approximate program address at which the error occurred. Note that in certain cases XXXX may be inaccurate: when you have broken out of a program (with ctrl-c) then the address given may be that of the low-level I/O routine rather than the address of the READ etc statement which was executing, or it could be entirely inaccurate!

All runtime error messages are sent to Port 1.

Halt
Overflow
Out of RAM
DIV by Zero
Index too Low
Index too High
Maths Call Error
Number expected
Number too large
Exponent expected

Reserved Words and Predefined Identifiers

Reserved Words

AND	ARRAY	BEGIN	CASE
CONST	DIV	DO	DOWNTO
ELSE	END	FOR	
FORWARD	FUNCTION	GOTO	IF
IN	LABEL	MOD	NIL
NOT	OF	OR	PACKED
PROCEDURE	PROGRAM	RECORD	REPEAT
SET	THEN	TO	TYPE
UNTIL	VAR	WHILE	WITH

Special Symbols

The following symbols are used by PPC PASCAL and have a reserved meaning:

```
+ - * /
= <> < <= >= >
( ) [ ]
{ } (* *)
^ := . , ; :
' ..
```

Predefined Identifiers

The following entities may be thought of as declared in a block surrounding the whole program and they are therefore available throughout the program unless re-defined by the programmer within an inner block.

```
CONST MAXINT = 32767;
TYPE BOOLEAN = (FALSE, TRUE);
CHAR { The expanded ASCII character set };
INTEGER = -MAXINT..MAXINT;
REAL { A subset of the real numbers }
```

The following special TYPEs are also predefined as part of the PPC PASCAL extensions:

```
TEXT {type of PORT1,PORT2,PORT3,PORT4}
      i.e.: TYPE TEXT = (PORT1,PORT2,PORT3,PORT4);
BRTYPE {type of the baud rate values (see list below)}
BWTYP E {type of the bits/word values _BW5, _BW6, _BW7, _BW8}
PRTYPE {type of the parity values _PNONE, _PEVEN, _PODD}
SBTYPE {type of the stop bit values _SB1, _SB2}
SWITCHTYPE {type of SW1STAT,SW2STAT,SW3STAT}
SETPORTTYPE {type of the SETPORT record}
SETHSKTYPE {type of the SETHSK record}
READHSKTYPE {type of the READHSK record}
RTCTYPE {type of the SETRTC and GETRTC records}
```

The following additional predefined identifiers exist in connection with the above:

PORT1	PORT2	PORT3	PORT4		
_B30	_B37H	_B50	_B75	_B100	_B110
_B134H	_B150	_B300	_B600	_B1200	_B2000
_B2400	_B3600	_B4800	_B7200	_B9600	_B19200
_B38400	_B57600	_B115200			
_BW5	_BW6	_BW7	_BW8		
_PNONE	_PEVEN	_PODD			
_SB1	_SB2				

The following predefined identifiers are names of record members:

RTS_OUT	DTR_OUT	CDO_OUT		{Boolean}
CTS_IN	DSR_IN	CDI_IN		{Boolean}
TM_SEC	TM_MIN	TM_HOUR	TM_MDAY	{integer}
TM_MON	TM_YEAR	TM_WDAY		{integer}
SW1STAT	SW2STAT	SW3STAT		{Boolean}
SPPC_PORT	RPPC_PORT	HPPC_PORT		
BAUD_RATE	BITS_WORD	PARITY	STOP_BIT	
RX_RTS_H	RX_DTR_H	RX_X_HS		
TX_CTS_H	TX_DSR_H	TX_X_HS		

The following is a complete list of procedures and functions.

PROCEDURE	WRITE;	WRITELN;	BWRITE;	
	READ;	READLN;	BREAD;	
	HALT;	NEW;	MARK;	
	RELEASE;	RANSEED;	DISPOSE;	
	INIT;	VAL;	READSWI;	
	LOADTIMER;	ULED;	DOUT1;	
	DOUT2;	OBRK;	ADCGAIN;	
	DAC;	DACC;	MEMCPY;	
	PDOUT;			
FUNCTION	ABS;	SQR;	ODD;	
	RANDOM;	ORD;	SUCC;	
	PRED;	EOLN;	CHR;	
	SQRT;	ENTIER;	ROUND;	
	TRUNC;	FRAC;	SIN;	
	COS;	TAN;	ARCTAN;	
	EXP;	LN;	ADDR;	
	SIZE;	RECAST;	MEMAVAIL;	
	SETPORT;	SETHSK;	READHSK;	
	SETRTC;	GETRTC;	ANEERD;	ANEWR;
	NVRBLK;	NVWRBLK;	NVREADC;	NVWRITEC;
	NVREADI;	NVWRITEI;		
	READTIMER;	ADCINI;	ADC;	ADCC;
	CBRK;	PDIN;		
	IPQCOUNT;	IPQCLEAR;	OPQCOUNT;	OPQSPACE;
	SUMBUF;	XORBUF;	CRCBUF;	
	_BIT;	_AND;	_OR;	_XOR;
	_SHL;	_SHR;	_ROL;	_ROR;
	_BITC;	_ANDC;	_ORC;	_XORC;
	_SHLC;	_SHRC;	_ROLC;	_RORC;
	STRCMP;	STRNCPM;	TOUPPER;	_CPL;
				_CPLC;



Where a predefined identifier is longer than eight characters (e.g. SETPORTTYPE), you should use the **full** name, even though the PPC Pascal compiler will ignore any characters beyond the first eight. This rule is recommended because the supplied compression utility program KPACK.EXE does not implement the 8-character limit and may erroneously compress abbreviated identifiers.

Programming Examples

Please look through the various programs supplied with the PPC, both on the diskette and in the filespace. They contain examples of how to invoke most of the PPC PASCAL functions and procedures.

The most important requirement of a datacomms product is a total absence of bugs. If you are writing bug-ridden PC-based applications (as many people are), you can charge your customers for an “upgrade” every 6 months (as many people do) and the upgrade can be loaded with yet more bugs. You cannot do this with a datacomms product, however, and careful programming and testing is the only answer. Fortunately, most datacomms programs – especially those on the PPC – will be quite small and easy to thoroughly test.

Robust reading of input data

This is one of the most important requirements in a datacomms product. Garbage can appear at the input due to a variety of reasons which can include the powering-up/down of connected equipment and the plugging/unplugging of cables. The actual method used depends on the type of data being read:

Character data

Read input data 1 char at a time, discarding everything until a valid character appears. In packet-oriented protocols, look for a “start of packet” character. Next, load everything into an **array of char** buffer until one of the following happens: an “end of packet” char is received, or the buffer is about to overflow.

```
const startpkt = #16;
      endpkt = #02;
      inport = port2;
var c : char;
    i : integer;
    buffer : array [1..100] of char;
repeat
  bread(inport,c);      { read input and discard until 'start' char appears }
until c=startpkt;
i:=1;
repeat
  bread(inport,c);      { read a char }
  buffer[i]:=c;         { put it into the buffer }
  i:=i+1;
until (c=endpacket) or (i>100);      { and repeat }
```

ASCII numbers

The safest way of reading these is by reading input data 1 char at a time, discarding everything until a valid numeric character (e.g. a digit 0..9) appears, loading remaining chars into a buffer and, when the end of the number is received, using the VAL procedure to evaluate the number. See the description of the VAL procedure for examples of this.

Because VAL returns a “pointer” to the end of each number converted, multiple numbers can be converted using a simple loop. To prevent VAL seeing a valid number in “garbage” that may exist beyond the end of the received data in the buffer, the conversions should not continue beyond the data which has been actually read-in.

The following example uses VAL to read integers from a buffer, the buffer having been previously filled by reading characters until an end-of-line character is received. It assumes that the 100-byte buffer is large enough for the longest possible line. Input is on Port 2, output is on Port 3.

```
var c : char;
    i,pos,lastpos,value : integer;
    suc : boolean;
    buffer : array [1..100] of char;
{ Fill the buffer }
i:=1;
repeat
  bread(port2,c);      { read a char }
  if (c in ['0'..'9']) or (c=' ') or (c=#0d) then begin
```

```

        buffer[i]:=c; { if numeric or space, put it into the buffer }
        i:=i+1;
    end;
until (c=#$0d) or (i>100);      { and repeat until end of line }
{ Convert the buffer into integers }
pos:=1;                          { start converting at start of buffer }
repeat
    val(buffer[pos],value,lastpos,suc);      { convert number @ pos }
    pos:=pos+lastpos; { set 'pos' for next number }
    if (pos<i) and suc then write(port3,value); { output value }
until (pos>=i) or (not suc); { repeat until all recd data converted }

```

If the buffer contains a *mixture* of integers and reals, a *single* VAL loop cannot be used. The easy solution is to read everything as reals. VAL can read reals; the only change to the above code would be to define `value` as `real`.

An alternative way is to read numbers with the READ procedure, with O- and R- compiler options to prevent any runtime errors. However, this method does not allow detection of obvious garbage; READ will always return a number even if it is meaningless. The following example shows how READ can be used to do the equivalent of the example above:

```

while not eoln(inport) do begin
    read(inport,value);
    write(port1,value);
end;

```

Multi-byte binary numbers

Because all character values can appear with equal probability, no character-based error checking is possible. However, if you know that the values must fall within a certain range, you can discard any which are outside.

An additional problem with binary data is that if a byte is lost or misinterpreted, all subsequent data becomes garbage. Any applications involving the transfer of *non*-character (i.e. integer, real, etc) binary data must prefix a block of data with a "sync" character which can then be searched for at the receiving end.

Character translation

If just one or two character codes need translating, a simple IF is sufficient:

```

var c:char;
repeat
    bread(inport,c);
    if c='a' then c:='b'; { translate 'a' into 'b' }
    bwrite(outport,c);
until false;

```

If several, or many, characters need translating, multiple IF statements are cumbersome. For the translation of several characters, a CASE statement is better:

```

repeat
    bread(inport,c);
    case c of
        'a': c:='b';      { convert 'a' to 'b' }
        'x': c:='y';      { convert 'x' to 'y' }
        'z': c:='?';      { convert 'z' to '?' }
    end;
    bwrite(outport,c);
until false;

```

However, for the translation of *many* characters, a translation table must be used. This is a 256-byte array which, at initialisation, is preloaded with the output characters. The input character forms the index into the array. The resulting program (the `repeat..until` loop) is very fast:

```

xltab : array [0..255] of char;
c : char;
i : integer;

```

```

{ Preload the table with 1:1 translation, i.e. no translation }
for i:=0 to 255 do
  xltab[i]:=chr(i);
{ Define the chars to be translated }
xltab[65]:='z';           { '65' is 'A', translate it to 'z' }
xltab[0] :='?';          { translate a null char to '?' }
xltab[7] :='*';          { translate a BELL to '*' }
xltab[35]:='£';          { etc }
init(xltab[135], 'fgtkym'); { this defines input chars 135..140 }
{ The main program }
repeat
  bread(inport,c);       { read a char }
  c:=xltab[ord(c)];      { translate it }
  bwrite(outport,c);    { output it }
until false;

```

String translation

This can be done in many different ways, depending on whether the input data can be read one "line" at a time into a buffer, whether more than one string must be searched for, and on the way in which the strings are stored.

The following complete program (supplied with the PPC) reads the input data stream and replaces every occurrence of string `searchstr` with string `substistr`. It does this "on the fly", i.e. it does not buffer the input data.

Both strings are stored in char arrays and both must be terminated by a null (#0). This means that neither string may contain the character #0.

The program works by comparing each input char with the first char of the search string; if there is a match then it compares subsequent chars. If there is a mismatch before the end of the search string has been reached then the input chars matched so far are passed-on without change; recursion is used at this point to ensure that if the search string matches partly, and the first non-matching character is the start of a new search string, the program identifies the new search string correctly.

The program allows the substitute string to be of zero length, i.e. its #0 terminator character being in position `substistr[1]`. This mode will remove the search string and replace it with nothing. The search string must be at least one character long, plus the #0 terminator.

Doubtless you will see more "elegant" versions of this program in Pascal books, but this one works and should be fairly easy to understand.

```

program stringxl;
var ch:char;
    inport,outport : text;
    searchstr : array [1..20] of char;
    substistr : array [1..20] of char;
procedure search(c:char);
{ Procedure which accepts a char and, if it matches the first char of
  the search string, it gets more chars & sees if they match, etc }
var i,j : integer;
    match : boolean;
begin
  if c=searchstr[1] then begin          { check 1st char match }
    i:=1;
    match:=false;
    repeat
      { loop here while input matches the search string }
      bread(inport,c);
      i:=i+1;

```



```

        if (c=searchstr[i]) and (searchstr[i+1]=#0) then match:=true;
until ( c <> searchstr[i] ) or match;
if match then begin
    { output the substitute string, up to a #0 }
    j:=1;
    while substistr[j] <> #0 do begin
        bwrite(outport,substistr[j]);
        j:=j+1;
    end;
end
else begin
    { output the matched part of the search string }
    for j:=1 to i-1 do
        bwrite(outport,searchstr[j]);
    search(c);    { and recursively output the non-matching char }
end;
end
else begin
    bwrite(outport,c);
end;
end; { search }
{ The main program }
begin
    init(searchstr,'Hello',#0);           { initialise search string }
    init(substistr,'How are you',#0);     { initialise substitute string }
    inport:=port1;
    outport:=port1;
    repeat
        bread(inport,ch);
        search(ch);
    until false;
end.

```



Every string search/replace algorithm suffers from one problem: if the input data stream ends, and its last few characters just happened to be the start of a string which is to be replaced, the system will hang until the rest of the string has arrived. The only solution is a timeout.

Performance hints

The following general suggestions apply to PPC PASCAL programs, and are not in any particular order of importance:

Use global variables instead of local variables. Global variables are faster to access and produce a more compact program.

Use inline code rather than function or procedure calls.

Avoid unnecessary copying of data from one buffer to another.

Pass all large variables (e.g. arrays) by address (i.e. make them a VAR) rather than by value.

For fast I/O, use BREAD and BWRITE. They are faster than READ and WRITE. When outputting an array, use the form

```

    bwrite(port,arrayname);    { to output the whole array }
    bwrite(port,arrayname:n); { to output the first n bytes only }

```

The above constructs are much faster than reading or writing single characters from within a loop.

Program too large

In the unlikely case of symbol table overflow, the solution is to use shorter symbol names, or use fewer symbols.

In the equally unlikely case that your program is too large to be stored in the PPC's filing system, a MS-DOS program called KPACK.EXE is available which processes the Pascal source file and considerably reduces its size. It does this by removing comments, blank lines, replacing long variable names with single- and double-letter names, and other techniques. The result is still a Pascal source file (albeit not a very readable one) which can be compiled and run on the PPC just like any other. It cannot, however, be edited with the PPC editor. KPACK.EXE is documented in the file KPACK.DOC and is supplied on the diskette.

Data Representation

The following discussion details how data is represented internally by PPC PASCAL.



Read this chapter only if you are reading or writing PPC PASCAL variables (other than characters) with the BREAD or BWRITE procedures, or performing type conversions with the procedure RECAST. The information here is not required to program in PASCAL and could even be confusing.

Integers

Integers occupy 2 bytes of storage each, in 2's complement form. Examples:

1	≡	\$0001
256	≡	\$0100
-256	≡	\$FF00

The less significant byte is stored at the lower memory address. When reading integers with BREAD, the less significant byte is expected first. When writing integers with BWRITE, the less significant byte is output first.

Characters, Booleans and other Scalars

These occupy 1 byte of storage each, in pure, unsigned binary.

Characters:

8 bit, extended ASCII is used.

'E'	≡	#\$45
'['	≡	#\$5B

Note that, on Port 1, characters above EFh (239 dec) cannot be entered from a terminal when the PPC is in Executive Mode, because those codes are used internally.

Booleans:

TRUE	≡	\$01
FALSE	≡	\$00

Reals

The real number is stored in a mantissa+exponent form in four bytes; one byte for the exponent and three bytes for the mantissa.

The way in which floating-point numbers are represented in binary is complex and beyond the scope of this Manual. The floating point format used is not the IEEE floating-point format sometimes used in microcomputers, and it is therefore extremely unlikely that you will be able to communicate using *binary* reals (with BREAD or BWRITE) with another piece of equipment.

The only situation in which you might be reading or writing *binary* reals is when two or more interconnected PPCs need to communicate floating point values to each other *in the most efficient manner possible*. If floating point data must be exchanged with external devices, the *text* format (i.e. using READ or WRITE for the I/O) should normally be used; this requires more CPU work but at least the *textual* representation of floats is universal.

Records and Arrays

Records use the same amount of storage as the total of their components. Variant records use the amount of storage required by their largest variant; different variant parts use the same storage.

Arrays: if n=number of elements in the array; s=size of each element then the # of bytes occupied by the array is n*s.

Examples:

an ARRAY [1..10] OF INTEGER requires $10*2 = 20$ bytes

an ARRAY [2..12,1..10] OF CHAR has $11*10=110$ elements and so requires 110 bytes.

Sets

Sets are stored as bit strings and so if the base type has n elements then the number of bytes used is: $(n-1) \text{ DIV } 8 + 1$. Examples:

a SET OF CHAR requires $(256-1) \text{ DIV } 8 + 1 = 32$ bytes

a SET OF (blue,green,yellow) requires $(3-1) \text{ DIV } 8 + 1 = 1$ byte.

Pointers

Pointers occupy 2 bytes which contain the address (in Intel format, low byte first) of the variable to which they point.

Recommended Books

The Pascal books below are useful for learning the language. Most Standard Pascal books devote large sections to teaching the more “elegant” aspects of Pascal programming which are likely to be rarely used in datacomms work, so almost any book which describes the main elements of the **Standard Pascal** language will be useful.

The major differences between PPC PASCAL and Standard Pascal are in the area of I/O, and for details of these you must consult the relevant sections in this chapter.

You should avoid books which describe Borland Turbo Pascal. This version of Pascal has many IBM PC-specific features and is quite different from PPC PASCAL.

Tutorial Style Books

Jim Welsh, John Elder

Introduction to Pascal

3rd Edition 1988. Prentice Hall. ISBN 0-13-491549-6

A thorough overview of Pascal, with examples and exercises. Many examples of more complex programs. Not recommended for a complete novice to programming.

S. Eisenbach C. Sadler

Pascal for Programmers

1st Edition 1981. Springer-Verlag. ISBN 0-387-10473-9 and 3-540-10473-9.

A good readable introduction to Pascal with examples but not very useful for I/O.

Richard Meadows

Pascal for Electronics and Communications

1st Edition 1985. Pitman. ISBN 0-273-02155-9

A book which illustrates Pascal through programs which solve common electrical and electronic problems. A good readable introduction to Pascal but not recommended for general use unless you are into electronics.

W. Findlay

Pascal: An Introduction to Methodical Programming

D.A. Watt 3rd Edition 1985. Pitman. ISBN 0-273-02188-5

Boris Allan

Introducing Pascal.

1st Edition 1984. Granada. ISBN 0-246-12323-0

Reference Books

N. Wirth

Pascal User Manual and Report.

K. Jensen 2nd Edition 1975. Springer-Verlag. ISBN 0-387-90144-2

The Standard Pascal reference. Written before interactive systems were commonplace and, in the area of real-time I/O, very dated.

J. Tiberghien

The Pascal Handbook

1st Edition 1981. Sybex. ISBN 0-89588-053-9

KTERM Terminal Emulator

What is KTERM ?

KTERM.EXE is a ANSI-compatible terminal emulation program which allows users of IBM and IBM-compatible PCs to access the PPC Executive.

Why use KTERM ?

All PPC features are accessible with just a dumb ANSI-compatible terminal. In addition, if you wish to use your IBM or IBM-compatible PC as the terminal, most of the various commercially available terminal emulation programs will work. However, the supplied KTERM.EXE terminal emulator offers special features not available on any other terminal emulation program:

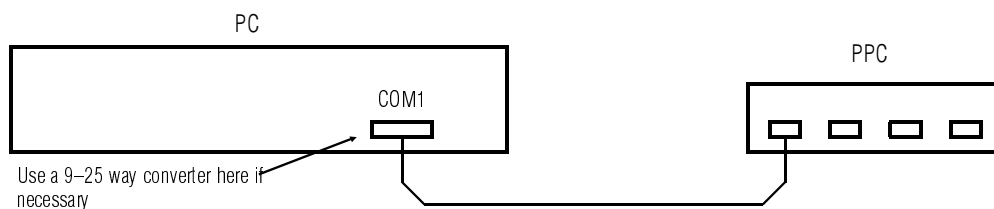
- KTERM supports **file transfer** between the PPC and your PC. This is extremely useful in two ways: you can maintain an unlimited number of PPC programs on the PC, either on a floppy or on a hard disk, and download these files to the PPC as required. You can also use your favourite *non-document* editor to create and edit PPC programs.
- KTERM supports the **43- and 50-line EGA/VGA display modes** in the PPC editor. These modes are preferred for editing because they make visible a larger part of the program being edited. Note that although some commercial programs (e.g. PROCOMM PLUS) do support extended video modes, these modes are not easily usable with the PPC because KTERM uses special ESCape sequences to tell the PPC the "terminal" screen dimensions.

KTERM installation

No "installation" is required. Just copy KTERM.EXE from the diskette to a convenient directory on your hard disk (make sure it is in the PATH) or, on a floppy-only PC, copy it to your working diskette.

KTERM operation

By default, KTERM uses your PC's COM1 serial port at 9600 baud, no parity, 8 bits/word and 1 stop bit. Therefore, connect the PC-PPC cable (see Serial Ports and Cables chapter) as shown in the following diagram:



Ensure that the PPC is powered-up and the EXE indicator is flashing. If EXE is not flashing then the PPC has probably been programmed to *run a program at power-up*; please see the PPC Executive chapter on how to enter the PPC Executive mode.

KTERM sets its own serial port parameters and there is no need for e.g. a preceding MODE or SETSER command to configure the serial port. Therefore, you can simply type

KTERM ↵

and the PPC Executive main menu should appear on your PC. If it does not, please see the KTERM Troubleshooting text later in this chapter. All Executive functions are now accessible to you.

To exit from KTERM, type alt-x; that means typing x while holding-down the ALT key.

KTERM File Transfer

This function allows transfer of user programs between the PPC internal filing system and your PC's hard or floppy disk.

1. To transfer a file from the PPC to your PC

Select the file in the Executive directory with the => file selector, highlight the

```
Transfer PPC --> PC
```

menu item and press ENTER. The selected file will be transferred to your PC and will be stored in your current directory.

2. To transfer a file from your PC to the PPC

Highlight the

```
Transfer PC --> PPC
```

menu item and press ENTER. KTERM will now prompt you to enter the name of the file resident on your PC; alternatively you can press the TAB key for a menu of candidate files. Either way, select the desired file and press ENTER. The selected file will be transferred to the PPC and will be stored in the PPC's filing system.

PPC - PC filename relationships

You will have noticed that files stored in the PPC filing system (and displayed in the Executive file directory) *have no filetypes*. (A *filetype* is the rightmost 3-character part of an MS-DOS filename). However, the PPC files do have a *language attribute* which is displayed in the Executive directory. This attribute determines the filetype which the MS-DOS version of the file will acquire upon transfer.

PPC language attribute	MS-DOS filetype
BASIC	.BAS
PASCAL	.PAS
BINARY	.BIN

and the above relationship applies in both directions.

PC filename directory control

As default, all files transferred from the PPC to the PC are stored in the *current* directory. However, this can be changed with the KTERMFILES environment variable. For example, an environment variable

```
KTERMFILES=C:\KTERM\TRANSFER
```

or

```
KTERMFILES=C:\KTERM\TRANSFER\
```

will, on a PPC→PC transfer, write the files into the directory specified. On a PC→PPC transfer, KTERM will *expect* the files in the specified directory.

Environment variables are a feature of MS-DOS and are a method of passing information to executing programs. To set the above environment variable, place the command

```
SET KTERMFILES=C:\KTERM\TRANSFER
```

anywhere in your AUTOEXEC.BAT file and reboot your PC. To cancel the KTERMFILES environment variable, remove the command from AUTOEXEC.BAT and reboot the PC. Alternatively, a *temporary* cancellation can be achieved by typing

```
SET KTERMFILES=
```

at the MS-DOS command line – this also deletes the KTERMFILES environment variable. Note from the above examples that the “\” at the end of the KTERMFILES path specification is optional – this is a common point of confusion with many application programs, some of which require the “\” and others must not have it. KTERM is more intelligent and accepts both formats.



Do not try to change the KTERMFILES environment variable when you have *shelled-out* of KTERM (with control-z). MS-DOS will discard your changes when you un-shell back into KTERM. This is normal MS-DOS behaviour.

File transfer security


All file transfers are CRC-checked. A corrupted file will be refused by its destination (KTERM or the PPC) and you will have to re-initiate the transfer. If errors persist, choose a lower baud rate.

DOS shell

KTERM has a very useful feature known as “shelling-out”. Type alt-z (type z while holding-down the ALT key). KTERM now opens-up another copy of COMMAND.COM (the MS-DOS command line processor) and you can run normal MS-DOS commands and programs. While shelled-out, KTERM remains resident (which reduces your PC’s conventional memory by approximately 80k bytes) and it can still receive data sent by the PPC. To “un-shell” back into KTERM, type

EXIT

at the MS-DOS command prompt. KTERM will restore its original screen content which may also have been updated with any data received from the PPC while shelled-out.

 While shelled-out, do not load any *resident* programs (TSRs). When you un-shell back into KTERM, the PC will probably crash. This is normal MS-DOS behaviour.

 While shelled-out, do not run any programs which *might* access the serial port on which KTERM is running. Examples include communications programs of all types.

KTERM command line options

The following command line options (displayed with KTERM /H) are supported by KTERM:

COMx Specifies the PC serial port to which the PPC is connected. Default is COM1. Only COM1 and COM2 are supported.

/B Specifies the PC serial port baud rate. The default is 9600 baud. Only 9600, 19200 and 38400 baud are supported. 38400 baud is highly desirable when editing larger programs, but may not work reliably on slower PCs.

- Specifying a baud rate value in KTERM causes KTERM to send to the PPC a special ESCape sequence which instructs the PPC to re-configure *its* serial port to that baud rate. When you exit KTERM (with alt-x), it will change the PPC back to 9600 baud. Therefore, the **PPC should always be set** (in the **Executive Mode Port 1 Config** menu) to **9600 baud** regardless of which baud rate KTERM is running at.
- KTERM always fully configures the serial port to its requirements, and restores the original configuration upon exit. There is no need for e.g. a preceding MODE or SETSER command.

/VL Selects large screen video mode: 43 lines on EGA and 50 lines on VGA displays. The default is 25 lines which gives the standard KTERM display of 24 lines plus the bottom status line.

/M Forces a monochrome display. Can produce a more readable display on portables with LCD or plasma displays. The default is a display which matches your PC’s display: monochrome or colour on a monochrome or colour PC respectively. KTERM obtains display size information from the BIOS and this is not 100% reliable; one can confuse KTERM by using a *monochrome* monitor with a VGA card configured for a *colour* monitor.

/MI Forces an inverse monochrome display. Above comments for **/m** option apply.

/VB Forces KTERM to access your display *via the PC’s video BIOS*. This mode is automatically selected with monochrome and CGA displays. The main use of this option is to prevent “snowing” on the display, or to achieve correct operation with “not-quite-IBM-compatible” display hardware.

/VF Forces KTERM to access your display directly *via the hardware*. This mode is automatically selected with EGA, MCGA and VGA displays. This option can produce faster display updating when running on a slow PC (especially at 38400 baud), but can cause “snowing” on some displays. The “snowing” can be suppressed with the **/vb** option.

/NI You should never need to use this option. It disables the mechanism which KTERM uses to advise the PPC of the new baud rate. The only time this option is useful is if you inadvertently configured the PPC to power-up with an *Executive Mode baud rate other than 9600 baud*. (Rates of 1200-38400 can be configured in the PPC for special scenarios such as modems or fast dumb terminals). If this happens, you must use SETSER (or an equivalent program) to configure the PC baud rate to the baud rate which you think the PPC might be set to, and use KTERM with the **/ni** option:

```
SETSER COM1:38400,N,8,1  
KTERM COM1 /NI
```

If the **/ni** option is used, any baud rate specified on the command line is ignored.

/OV This “override” option forces KTERM to connect to the specified serial port even if that port appears to be in use by another program (e.g. a mouse driver). Some PCs initialise their COM1/COM2 interrupt vectors to point

to memory addresses below F000:0000 (i.e. possibly to a memory-resident program) and KTERM then issues a "port already in use" message and aborts. This option overrides the checking.

An example of where the **/ov** option is commonly required is when your CONFIG.SYS file contains a STACKS=A,B command (where A or B are non-zero). This is likely to be the case if you have installed Windows 3.1, or are using DOS version 5 or higher.

/L This option logs (copies) everything sent to the screen to a file called KTERM.LOG located either in the current directory, or in the KTERMFILERS path. It is useful for technical support assistance. The file KTERM.LOG is reset to zero size whenever KTERM is invoked.

KTERM examples

KTERM	use COM1 at 9600 baud
KTERM COM2	use COM2 at 9600 baud
KTERM COM1:38400	use COM1 at 38400 baud
KTERM COM1:38400 /vl	use COM1 at 38400 baud with a 43- or 50-line display (this is the recommended mode for high performance on VGA)
KTERM /b19200 /m	use COM1 at 19200 baud, force a monochrome display
KTERM /b19200 /mi	use COM1 at 19200 baud, force an inverse mono display
KTERM /ov	use COM1, 9600 baud, override interrupt vector checking

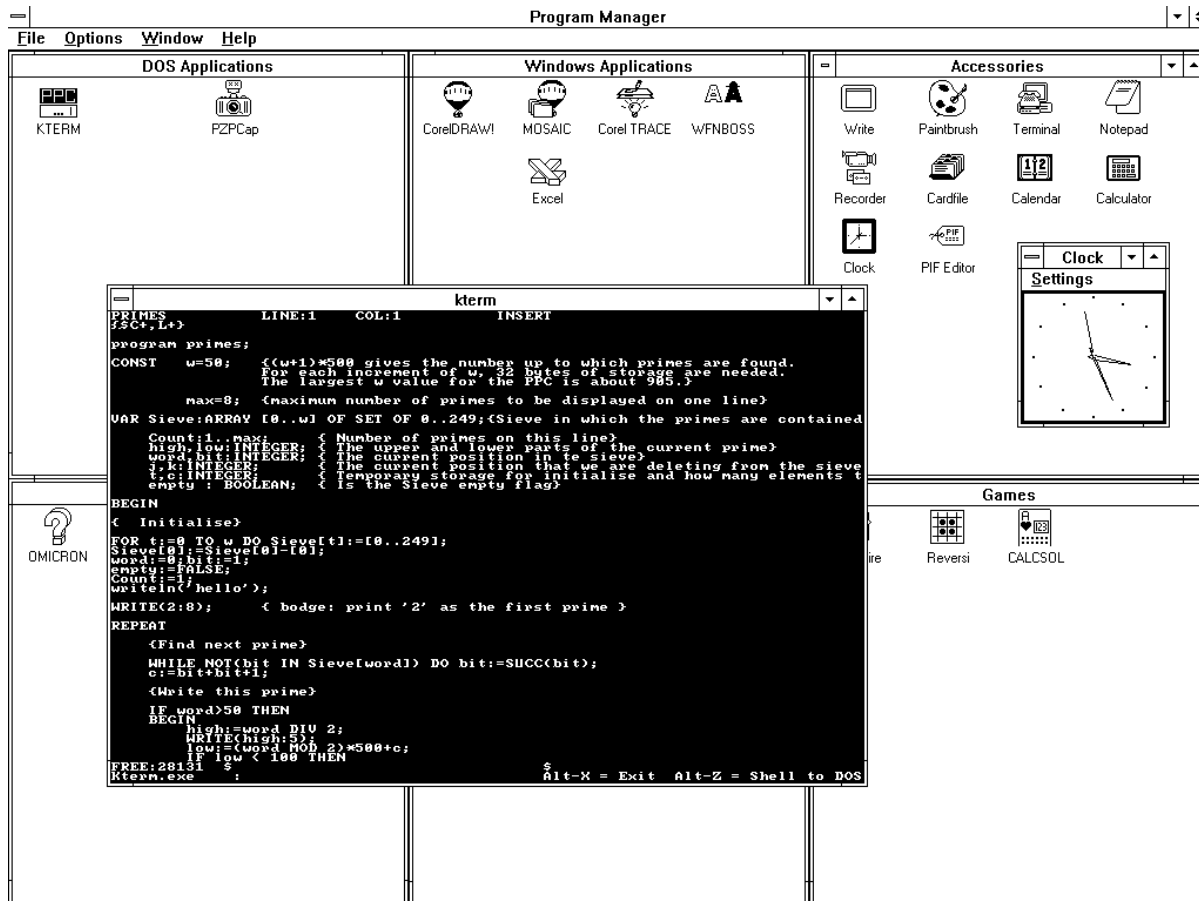
In general, you should use the highest baud rate possible (38400 baud) because this will give the fastest screen updates. However, slow PCs will lose serial receive data at such high baud rates and this will manifest itself as screen corruption, or persistent error messages during file transfers.



If you intend to use KTERM as a general-purpose terminal emulator in non-PPC applications, note that it does *not* support *all* the ANSI commands. It supports only a limited set of commands which are actually used by the PPC. It also transmits special commands when it starts up.

KTERM under Windows 3.x, Windows 95, Windows NT 4

KTERM runs under the above, either full-screen or in a DOS box. All functions including file transfer are supported, except (sometimes) the ctrl-z DOS shell command. The supplied KTERM.PIF and KTERM.ICO files are required and must be copied to your hard disk; a suggested location is the WINDOWS directory. Then select the **File** option, select **New** and create a new program item called **KTERM**, with the appropriate path pointing to where KTERM.EXE is stored. This will create the KTERM icon which appears in the top left corner of the screen below. Double-clicking on the KTERM icon invokes KTERM as shown below:



Note that KTERM will initially be invoked in the basic mode with no command line options, at 9600 baud. You may wish to edit the command line (in the KTERM.PIF file or in the Program Manager Properties menu; see your Windows Manual) to select e.g. a 50-line VGA mode (/VL option) and a higher baud rate. The above screen shows KTERM running with the /VL option under Windows 3.0 with a super-VGA (1024x768) display in 386 enhanced mode.

Note also that any serial port baud rate settings set-up in the Windows Control Panel are irrelevant because KTERM accesses the serial port hardware directly, configures the port as required and, upon exit with alt-x, restores the original configuration.

Using the standard Windows 3.x cut and paste functions, it is possible to cut and paste text from KTERM running in a window to a Windows 3.x application but the amount of text which can be pasted may be limited by the application's capabilities.

When running a DOS application such as KTERM *in a window*, Windows waits for the application's screen contents to stop changing before it updates the window. This has the side-effect that, during file transfer when KTERM updates the displayed byte count very rapidly, the byte count shown in the Windows window appears "stuck" until the file transfer is finished.

Instead of KTERM, it is possible to use the TERMINAL.EXE program which is supplied free with Windows. However, it has poor performance, is likely to lose data at speeds above 9600 baud (even on a 486/33) and does not support file transfer with the PPC.

Running KTERM over a modem link

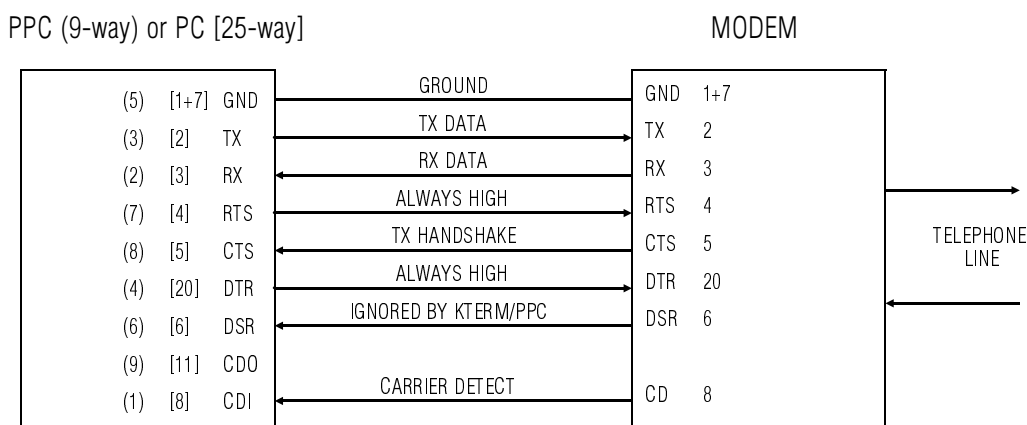
This is desirable for applications where the configuration of a remote PPC, or a program resident on that PPC, need to be modified remotely.

This is possible because if an autoexec program terminates, the PPC returns to the Executive. You must also write your PPC Pascal program so that it terminates when it receives a specific key (or key sequence) from Port 1. Alternatively, you must instruct someone at the remote location to set the PPC into the Executive mode prior to your dial-up.

Accessing the PPC via a modem is not difficult, but there are the standard pitfalls when using modern modems which do automatic baud rate detection. This is explained below.

Cables

A cable suitable for connection between an IBM PC 25-way serial port and a standard Hayes-compatible asynch modem is below. The same cable can also be used for the connection between the remote modem and the PPC.



PPC configuration

It is assumed that you want to access the PPC Executive over the modem link and edit files with the PPC editor, perform PC-PPC file transfer, etc (and not just communicate with a program which is running in autoexec mode). You must therefore configure the **Executive Mode baud rate** to match the modem's configuration. In addition, you may want to see data which is emitted by your program to the Executive port (Port 1), and you must therefore configure the **Runtime baud rate** of Port 1 to match the modem's configuration also.

Where configuration is possible, use 8 bits/word, 1 stop bit, no parity, RTS/CTS handshake ON, XON/XOFF handshake OFF. Use the highest baud rate which your modem supports.

The PPC port (Port 1) should be configured with the following handshakes:

RX handshakes:	RTS	DTR	XON/XOFF	TX handshakes:	CTS	DSR	XON/XOFF
	ON	OFF	OFF		ON	OFF	OFF

The **Executive Mode** baud rate must match the **Port 1 Runtime** baud rate.

Modem configuration

No special configuration is required in most cases. Most modern modems are auto baud rate sensing at their DTE port and the local (PC-attached) modem will usually auto-configure when it receives the first data from KTERM.

Most modems do their auto baud rate detection from the "A" of the "AT" command. Make sure that the first thing you send to the modem is something starting with "AT".

When KTERM is started, it always transmits a special initialisation sequence. This is intended for the PPC, but when KTERM is used to communicate with a modem, the modem will treat this as garbage, not least because the sequence does not start with the AT command. Therefore, you need to transmit the AT command to the modem, at the proper baud rate to which you wish to set the modem, before starting KTERM. The simplest way to do this is with a batch file containing a MODE or SETSER command, followed by "COPY FILE COM1" (where "file" contains an AT command), followed by KTERM.

The remote (PPC-attached) modem must be configured to auto-answer. With Hayes-compatible modems this is usually done with the command

```
ATSO=1 <CR>
```

where the "1" specifies the number of rings to wait before answering. Your PPC-based program must transmit the above string to the modem, with the suggested Pascal statement

```
writeln(port1, 'ATSO=1');
```

Alternatively, the modem can be pre-configured (by the installer) to power-up in auto-answer mode. However, this can cause a problem because when the modem answers the incoming call, it has not yet received anything from the PPC and therefore does not know the PPC's Port 1 baud rate. In this situation many modems will set their DTE port to a baud rate which matches the rate of the incoming call, i.e. an incoming call at V22bis (=2400 baud) will set the DTE port to 2400 baud. This is fine if the PPC is also set to 2400 baud *and* the connection is *always* established at 2400 baud! Many modems can be pre-configured to a specified non-varying DTE baud rate which is independent of the link rate, but the commands to do this vary widely and are outside the scope of this Manual. Please consult your modem manual for details. Transmitting the ATSO command to the modem after each power-up usually avoids the problem.

KTERM usage

You must use KTERM version 1.1E or higher. There are four cases to consider:

- 1 Your modem supports 9600 baud DTE rate** (or higher) but does not support a line connection at 9600 baud (or higher), i.e. it is a V22, V22bis or V32 modem only. This is the most common case. The commands

```
setser com1:9600,n,8,1  
kterm /r /ni
```

will establish a PC-modem connection at 9600 baud.

- 2 Your modem supports more than 9600 baud DTE rate** and supports a line connection at more than 9600 baud, i.e. it is V32bis (or higher) modem. The commands

```
setser com1:38400,n,8,1  
kterm /r /ni
```

will establish a PC-modem connection at e.g. 38400 baud.

- 3 Your modem does not support 9600 baud at all** and a lower baud rate (2400 or 4800) must be used. The commands

```
setser com1:2400,n,8,1  
kterm /r /ni
```

will establish a PC-modem connection at e.g. 2400 baud.

- 4 Your modem supports 1200 baud or less**, i.e. it is a V21 or V2123 modem.

Since the lowest settable Executive Mode baud rate is 2400 baud, you cannot perform any Executive functions over a modem connection. Sorry.

The `/r` option instructs KTERM to use the standard RTS/CTS modem handshake. It also slightly modifies the PC-PPC file transfer timings to handle short gaps in data flow caused by the handshaking with buffered (MNP) modems.

Operation

When KTERM is invoked as shown above, you will initially see a blank screen. Typing

```
AT <CR>
```

must return the standard "OK" modem response. This also sets the modem's port to the PC's baud rate. Enter the standard Hayes dial command, e.g.

```
ATDT 0712345678 <CR>
```

and the modem dials the remote PPC's modem. When a connection is established between the modems, the standard response "CONNECT 2400" (or similar) appears. At this point, press one or more of the following keys until the Executive menu appears:

```
a key (or key sequence) which terminates your autoexec program  
ESC  
control-q
```

Note that control-c will not usually terminate an autoexec program. Even if you have not disabled break checks (with C-), the PPC automatically disables them when a program is running in autoexec mode. If you do want control-c to


terminate your program, you must specifically test for it (#3 character) and, when detected, either exit the main block, or execute the HALT statement.

When the Executive menu appears, press the F10 function key. This tells the PPC the current KTERM screen size.

You can now perform any Executive function (except changing the Port 1 or Executive Mode baud rates !!)

When modifications etc are finished, the PPC program (which should still be marked "autoexec") can be restarted by selecting the Reboot function on the main menu.

 If you have uploaded new programs to the PPC, the "autoexec" marker could be pointing to a different program from its previous position! Check this.

 Dont exit KTERM while the modem is online. It might drop the PC's DTR which would then hang-up the modem. Shelling-out to DOS from KTERM (alt-z) is OK, however.

Finally, disconnect the modem link by hanging-up your modem. This can be done from your modem's front panel, or by powering it down, or by typing the command

```
~~~+++~~~ATH0 <CR>
```

while still in KTERM. Each of the above "~" characters represents a 1 second (at least) delay; it is not actually typed. The three "+" characters must be typed in rapid succession.

KTERM Troubleshooting

Executive menu does not appear at all

- 1 Press the ESC key several times.
- 2 Exit KTERM (with alt-x) and re-invoke it.
- 3 Exit KTERM, reset the PPC (by holding SW2+SW3 together until the RST LED illuminates), wait for the EXE LED to start flashing, then re-invoke KTERM. If the EXE LED does not start flashing, the PPC is probably configured to automatically run a program, and you must use the Executive Mode force entry procedure: press all three switches until RST illuminates and then, while still holding SW1, release SW2+SW3. The PPC will now enter Executive Mode.
- 4 Check you are using the correct PC port: COM1 or COM2.
- 5 Check you are connected to the correct PPC connector marked PORT1/PORT3.
- 6 Check you have the correct cable (see Serial Ports and Cables chapter) between the PC and the PPC. If you have made your own cable, it must have at least those connections shown on the PPC bottom label.

Garbled data appears

- 1 Exit KTERM (with alt-x) and re-invoke it. If the problem is due to mismatched baud rates, this will equalize them.
- 2 If only a few characters are corrupted, your baud rate is probably too high. Select 19200 or, if the problem persists, 9600 baud.

Editor display is incorrect, or other display faults

Exit KTERM (with alt-x) and re-invoke it. For an explanation, see the "invalid cursor position" error message further below.

Control-c does not stop program execution

- 1 There is already some data in the Port 1 receive queue. This will prevent the ctrl-c being seen.
- 2 In a PASCAL program, you have disabled break checking (with the C- compiler option).
- 3 KTERM has a 1024-byte receive buffer. This means that data can be appearing on the screen for several seconds after the PPC has stopped transmitting.

KTERM Error Messages

Video subsystem does not support large screen mode

You have used the **/vl** option but your display is not EGA or VGA.

Specified COM channel is already in use

Before configuring its serial port, KTERM has determined that an existing program has possibly set-up its own interrupt vectors for that port (COM1 if no port has been specified).

- 1 Check the contents of your AUTOEXEC.BAT and CONFIG.SYS files and remove any programs likely to be accessing the serial port used by KTERM. Likely candidates are mouse or digitiser drivers, some of which "grab" every serial port present in an attempt to detect if a mouse/digitiser is attached, and do not release them afterwards!
- 2 If you are sure that no program is meant to be using the port, use the **/ov** option.
- 3 If your CONFIG.SYS file contains a STACKS=A,B command where A or B is non-zero, or you are running DOS version 5 or higher, then use the **/ov** option.

Kterm is already active, type exit to return

You have shelled-out of KTERM (with alt-z) and then tried to invoke another copy of KTERM. This is not supported! Type EXIT to return to the original copy of KTERM.

Invalid cursor position detected. Exit KTERM and reenter.

This error should occur only while in the PPC Editor and is more subtle: when KTERM is invoked, it transmits a special ESCape sequence to the PPC to tell it "you are now running with a XX-line terminal" where XX is normally 24 (KTERM uses the 25th line for its status display) or it can be 42/49 (EGA/VGA with the /vl option). The PPC Editor reconfigures itself to the new screen size specified by KTERM. However, *if you have powered-off or rebooted the PPC without exiting KTERM*, the PPC will power-up assuming a basic "dumb" 25-line terminal and the Editor will use all 25 lines. KTERM, being a 24-line terminal, will report an attempt to place the cursor on the 25th line.

The solution is to exit KTERM (with alt-x) and re-invoke it *while the PPC is connected*. Alternatively, with KTERM version 1.01E or higher, pressing F10 when in the Executive menu transmits the screen size to the PPC and prevents subsequent occurrences of this error message. The F10 method is recommended for remote (modem) KTERM operation because exiting KTERM might drop DTR and hang up the modem.

Communications line error, transfer aborted. Press any key.**Incorrect data length, transfer aborted. Press any key.****Communications CRC error, transfer aborted. Press any key.**

Any of the above errors indicates loss or corruption of data during a PPC-PC file transfer. This should never happen unless your PC is too slow to handle the specified baud rate. Try a lower baud rate: 19200 or 9600 baud.

Invalid filename, transfer aborted. Press any key.**Invalid file extension, transfer aborted. Press any key.**

These messages indicate that the version of your copy of KTERM.EXE does not support a new feature of the PPC. Obtain the latest copy of KTERM.EXE from your PPC supplier.

Overwrite file (Y/N)

A file of the same name as the file being transferred already exists on your PC.

Error writing**Error overwriting**

The file you are transferring to the PC cannot be written; the possible reasons include:

- 1 A file of the same name already exists and is marked read-only.
- 2 A *directory* of the same name already exists.
- 3 Your disk is full
- 4 Your disk is write-protected

Directory does not contain any KTERM files. Transfer aborted.**Unable to open file. Transmission aborted.**

- 1 You have requested a PC→PPC file transfer, but KTERM cannot find any .BAS or .PAS files.
- 2 The files to be transferred to the PPC must reside either in the current directory, or in the path given by the environment variable KTERMFILERS=path.

File is empty. Transmission aborted.

The filesize is zero. Zero length files cannot be transferred.

File is too large. Transmission aborted.

Maximum file size supported by KTERM is 32k bytes. In addition, a lower limit may apply when transferring PC→PPC and there are already some files in the PPC's filespace.

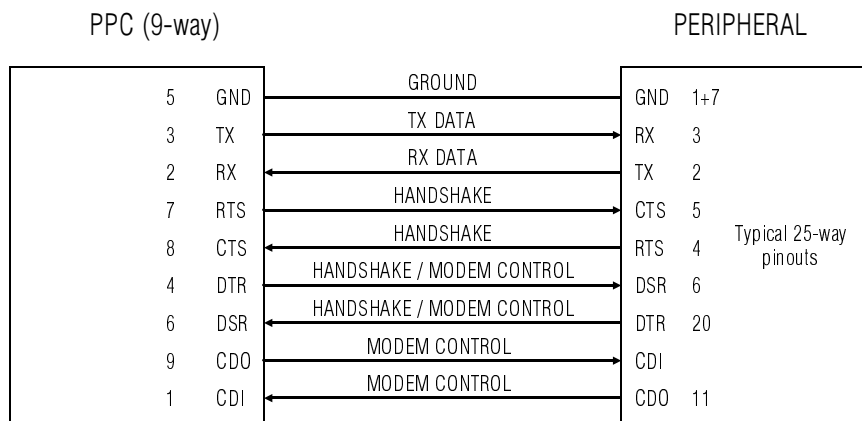
PPC Serial Ports & Cables

This chapter covers the details of the four serial ports. Each of the four ports is normally RS232 but can be configured for RS422 or RS485, 2-wire or 4-wire.

 PPC D-connector pin numbers shown here refer to the PPC-4 male **9-way** connectors only.

Hardware Interface - RS232

All four ports have an identical interface which supports the full set of RS-232 data and handshake signals. The following diagram shows the PPC connected to an RS-232 peripheral device, which could be a computer, a printer, plotter, a modem, or almost anything else:



All supported signals are shown connected. However, with most devices, only a subset of the connections shown will be required depending on the handshake requirements. The only common device which might use all of the supported signals is a modem; see **Modem Handshaking**.

Pin Functions

GROUND

This signal must always be connected. This combines both Protective Ground and Signal Ground functions.

TX

“Transmit Data” output. A “write” statement transmits data from this pin, subject to all enabled TX handshakes being READY. Also, when XON/XOFF is enabled as an RX handshake, XON/XOFF characters are generated by the PPC and are transmitted from this pin to control the data flow into the RX input.

RX

“Receive Data” input. A “read” statement reads data arriving at this pin. Also, when XON/XOFF is enabled as a TX handshake, any XON/XOFF characters arriving at this pin will be stripped by the PPC and will control the data flow from the TX output.

RTS

“Request To Send” output. When enabled as an RX handshake, this signal is automatically controlled by the PPC and controls the data flow into the RX input. When not enabled as an RX handshake, this signal can be freely controlled from PASCAL or C. The *name* of this signal dates back many years and no longer means very much except when used with modems.

CTS

“Clear To Send” input. When enabled as a TX handshake, this signal is automatically monitored by the PPC and controls the data flow from the TX output. This signal can also be freely monitored from PASCAL or C. If no connection is made to this pin, it will be internally pulled-up to a READY (RS-232 high level) state. The *name* of this signal dates back many years and no longer means very much except when used with modems.

DTR

“Data Terminal Ready” output. When enabled as an RX handshake, this signal is automatically controlled by the PPC and controls the data flow into the RX input. When not enabled as an RX handshake, this signal can be freely controlled from PASCAL or C. The *name* of this signal dates back many years and no longer means much except when used with modems.

- DSR** **“Data Set Ready” input.** When enabled as a TX handshake, this signal is automatically monitored by the PPC and controls the data flow from the TX output. This signal can also be freely monitored from PASCAL or C. If no connection is made to this pin, it will be internally pulled-up to a READY (RS-232 high level) state. The *name* of this signal dates back many years and no longer means very much except when used with modems.
- CDO** **“Carrier Detect” output.** This general-purpose signal can be freely controlled from PASCAL or C. It is intended to support the “pass-through” of the carrier detect signal, to allow a PPC to emulate a modem, e.g. when acting as a dial-up security device.
- CDI** **“Carrier Detect” input.** This general-purpose signal can be freely monitored from PASCAL or C. It supports the connection of a modem to the PPC: the state of this signal then indicates whether the modem is on-line or off-line. With Hayes-compatible modems the CD connection is not essential (because they can return their status in the form of text strings), but monitoring the modem’s CD signal is the only *unambiguous* and *instantaneous* way of knowing the modem’s status. If no connection is made to this pin, it will be internally pulled-down to a FALSE (RS-232 low level) state.

Hardware Interface - RS422/RS485

First, let’s straighten out a few common misconceptions:

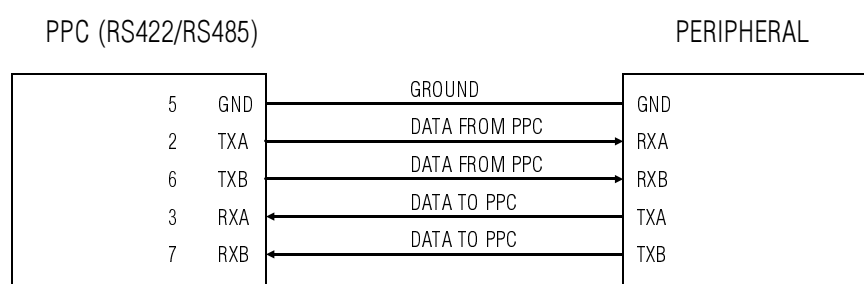
RS422 and RS485 use differential signalling. For each RS232 1-wire signal there are now 2 wires, marked A and B. Also, the hardware handshakes and modem controls are not used. RS422 and RS485 are identical in their voltage levels and distance capabilities.

RS422 uses a driver which is always enabled and so RS422 is used only for **point to point** communications. Like RS232, the data flow can be **full-duplex**, i.e. simultaneous in both directions.

RS485 has a “tri-state” driver capable of being disabled when not transmitting, and is used in **multidrop** systems. The data flow is typically **half-duplex**, i.e. only in one direction at any one time. Moreover, RS485 systems are normally Master/Slave systems where the Master periodically polls the Slaves, and only the Master can initiate communication. A Slave transmits data only in response to a poll from a Master. RS485 can also be wired as “4-wire” or as “2-wire”.

A PPC RS422/485 port can operate in any of the above modes. Your program can leave the driver state in its power-up condition, in which case it will be always enabled and you have an RS422 port. If you control the driver (using the software-controllable RTS signal) then you have effectively created a 4-wire RS485 port. If you also externally interconnect TXA-RXA and TXB-RXB then you have a 2-wire RS485 port.

The following diagram shows the connection between a PPC-4 port configured for RS422/RS485, and another device:



In a multidrop system, with the PPC acting as the Master, there can be multiple peripherals (Slaves). All the Slaves have their connections paralleled.

To convert the above 4-wire port to a 2-wire port, interconnect TXA-RXA (pins 2-3) and TXB-RXB (pins 6-7).

RS422/RS485 grounding

All RS422/485 receivers have a finite common mode voltage range, typically -7V to +12V. If you can guarantee that in your system this will never be exceeded, then the GND connection is not required. In practice this may be hard to guarantee, and the GND connection should always be present except where there is a serious risk of a ground loop with resulting heavy ground currents.

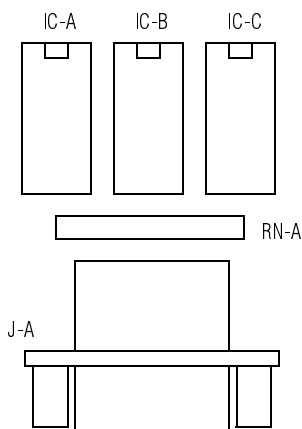
RS422/RS485 Termination

If your cable length exceeds several hundred metres then a termination resistor (equal to the characteristic impedance of the cable) should be connected between the two wires representing each signal. A terminator is required at each end of the cable; four terminators are therefore required on a RS422 or 4-wire RS485 system, and two on 2-wire RS485.

In practice, terminators are much over-rated and are usually not required. They also waste a lot of DC power; this can be avoided by connecting a series capacitor, e.g. 1000pF, with the terminator.

RS232 to RS422/RS485 port conversion

With power OFF, remove the four rear panel screws and withdraw the rear panel complete with the attached PCB. You will notice that next to each of the four 9-way D-connectors there are three IC sockets and an 8-pin resistor network. The following diagram shows the arrangement for each PPC-4 port. The PPC-E uses identical components; refer to the table further below for which components apply to which port.



Component positions shown for PPC-4 board. PPC-E uses identical components - see tables below for identification.

The following table shows which components are used for each port function:

	IC-A	IC-B	IC-C	RN-A
RS232	14C88 or 75C188	14C89 or 75C189	-	4x4k7
RS422/485	-	-	LTC491	-

The following tables show which components apply to which of the four ports:

PPC-4	J-A	IC-A	IC-B	IC-C	RN-A
Port 1	J1	IC1	IC2	IC3	RN1
Port 2	J2	IC4	IC5	IC6	RN2
Port 3	J3	IC7	IC8	IC9	RN3
Port 4	J4	IC10	IC11	IC12	RN4

PPC-E	J-A	IC-A	IC-B	IC-C	RN-A
Port 1	J1	IC9	IC10	IC27	RN1
Port 2	J1	IC11	IC12	IC28	RN2
Port 3	J1	IC13	IC14	IC29	RN3
Port 4	J1	IC15	IC16	IC30	RN4

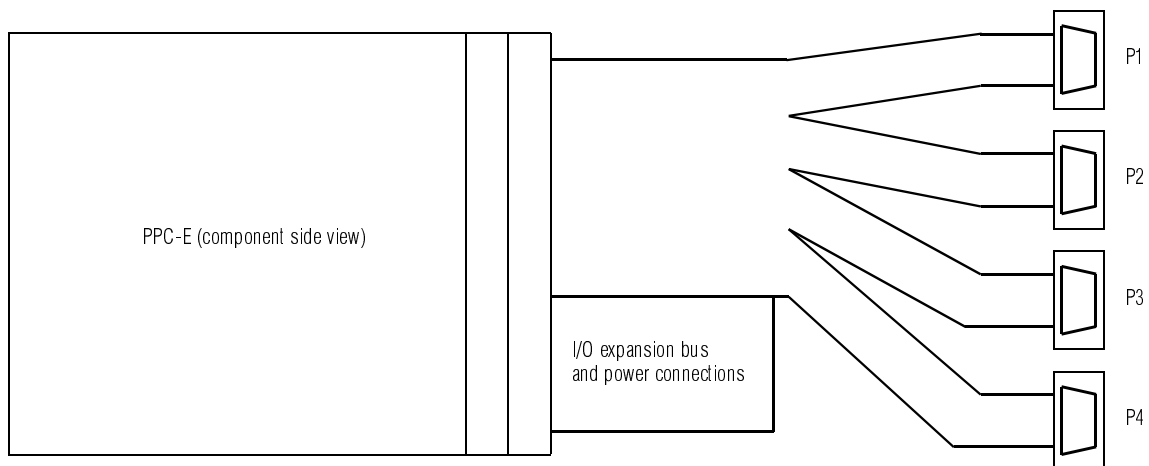
The LTC491 has various equivalents, e.g. MAX491, or MAX489 where lower slew rate (better for EMC) is required. The RN network is an 8-pin inline resistor network containing four separate 4k7 5% resistors. The polarity of the network in its socket is not important.

Fit a 14-way pin header into the now-vacant IC-B (14C89 or 75C189) socket. This header must be wired to interconnect pins 6,7,8,11. This has the effect of wiring the CTS, DSR inputs to READY (HIGH) levels, and it also ties the CDI input to a HIGH level. These signals are unused on an RS422/485 port, but incorrect levels on CTS or DSR can prevent data transmission from that port.

- ☞ Observe standard anti-static precautions when working with any unprotected circuitry. This means working on an anti-static (conductive) mat and wearing a wrist-strap which is connected to the mat.
- ☞ Avoid short-circuiting the RTC battery. If this has been shorted, even momentarily, you must run the Reset ("R") command in the RTC Configuration in the PPC Executive, and of course re-enter the correct date and time.
- ☞ For above reasons, do not store the PPC-E (Eurocard version) in a metallised conductive bag. Some such bags will run the battery down very quickly and can even short-circuit it, resulting in a theoretical fire hazard.

PPC-E DIN41612 connections

The PPC-E uses a standard Eurocard 64-way 2-row DIN41612 connector. In most applications this product will plug into a custom backplane. However, the DIN41612 pin connections on this connector are arranged so that it is possible to make up an IDC (insulation-displacement) cable terminating in four DB9 **male** IDC connectors, as shown below:



With the above cable arrangement, the pin connections of the DB9 connectors are identical to those of the standard PPC-4 product.

RS485 driver control

For an RS422 port, leave the driver permanently enabled, by not changing RTS.

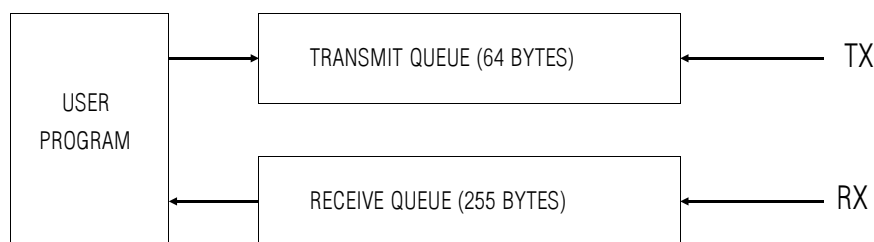
For an RS485 port, to enable the RS485 driver, set RTS to a 1, and vice versa. To control RTS, use the `SETHSK` or `setporthsk` in Pascal or C respectively. The DTR and CDO signals are irrelevant. The PPC Utilities disk contains example RS422/485 programs.

When turning off the driver, make sure the entire message has been transmitted first. The function `OPQCOUNT` is ideal for this, and is most accurate on ports 3,4 where its returned value includes the exact contents of the UART transmit circuitry. Therefore, **ports best suited to high speed RS485 use are ports 3 and 4**. If Ports 1 or 2 are used for RS485, a different procedure is used to determine the correct driver turn-off time - see the `OPQCOUNT` function description.

Remember also that on a *2-wire* RS485 system you will also be receiving your own transmission, and may need to discard it by issuing an `IPQCLEAR` after the message has been transmitted. This can however be useful as the reception of the last character of your transmitted message can be used to immediately turn off the driver!

I/O Queues

Each PPC port contains two queues: a **transmit (TX) queue** and a **receive (RX) queue**:



A user program never accesses a port *directly*; any **READ, WRITE** etc statements communicate only with the queues. The transfer of data between a queue and its port is performed as a background process (i.e. under interrupts) by the comms processor and is transparent to the user program.

Transmit Queues

A TX queue greatly enhances performance in both computational and datacomms areas, by allowing a user program to perform computations while previously-generated data is being transmitted under interrupts.

With the 64-byte TX queue, a user program could theoretically generate up to 64 bytes of data even if the output device was BUSY throughout that time. However, to allow for insertion of XON/XOFF characters into the data stream and other reasons, the PPC limits the filling-up of its TX queues to **60** bytes.

Receive Queues

An RX queue is necessary to avoid data loss when handshaking the incoming data. When the space in this queue falls below 128 bytes, the PPC will return a BUSY handshake, if enabled. See the **RX Handshakes** description on the following page for full details.

Handshaking

The PPC supports two types of handshakes: **transmit (TX) handshakes** and **receive (RX) handshakes**. A **TX** handshake signal is *received by* the PPC and controls the data flow *from the PPC*. An **RX** handshake signal is *transmitted by* the PPC and controls the data flow *into the PPC*.

Each PPC serial port is equipped with two hardware handshakes and an XON/XOFF handshake - for both transmit and receive modes. Therefore, each port supports up to a total of six distinct handshakes, each of which can be individually enabled or disabled. The only exceptions are the Port 1 & 2 CTS TX handshakes; these cannot be disabled.

Since there is no “standard” on handshaking, you will generally have to examine what handshakes the other devices require or support, and *enable those handshakes only* on the PPC. In many cases only a small subset of the PPC’s handshake modes will be enabled, although it usually does no harm – subject to the cables being correct – to simply enable them all.

The following is a detailed description of the handshakes’ operation:

TX Handshakes


The PPC transmits data from the TX pin only if **all** enabled TX handshakes are **READY** (RS-232 high level or XON). TX handshakes which may be enabled are CTS, DSR and XON/XOFF. More than one TX handshake may be enabled.


In any handshake system there is a delay between the handshake’s change of state, and how long it takes for the transmit data flow to start/stop. The following approximate timings apply to the PPC:

Port 1 & 2 CTS TX handshake:	data flow starts/stops within 1 character period
All other CTS & DSR TX handshakes:	data flow stops within 4 character periods data flow restarts within 1 ms
XON/XOFF handshakes:	data flow stops within 5 character periods data flow restarts within 1 ms

RX Handshakes

When the PPC port is receiving data, if the free space in its 255-byte RX queue falls below 128 bytes, it sets all enabled RX handshakes to the **BUSY** state. A **BUSY** state is an RS-232 low level and, if XON/XOFF is enabled, the transmission (by the PPC) of an XOFF character. When the space in the RX queue increases above 200 bytes (as a result of a user program removing the data from it), all enabled handshakes enter the **READY** state. RX handshakes which may be enabled are RTS, DTR and XON/XOFF. More than one RX handshake may be enabled.

 From the above it is clear that when a device which is sending data to the PPC is told by the PPC to stop transmitting, it must not transmit more than **128** additional bytes after that, otherwise data may be lost.

 The above-mentioned XON/XOFF characters can be transmitted by the PPC only if all enabled **TX** hardware handshakes are **READY**.

Binary Data Handshaking

When transferring binary data, XON/XOFF characters could appear accidentally within the data. In such cases, an XON/XOFF handshake in which the XON/XOFFs flow in the same direction as the data must be **disabled**.

For example, if a PPC port is *receiving* binary data, that port’s TX XON/XOFF handshake must be disabled. If a PPC port is *transmitting* binary data, the receiving device’s TX XON/XOFF handshake (if available) must be disabled. If two devices are exchanging *full-duplex* binary data, XON/XOFF handshake cannot be used at all.

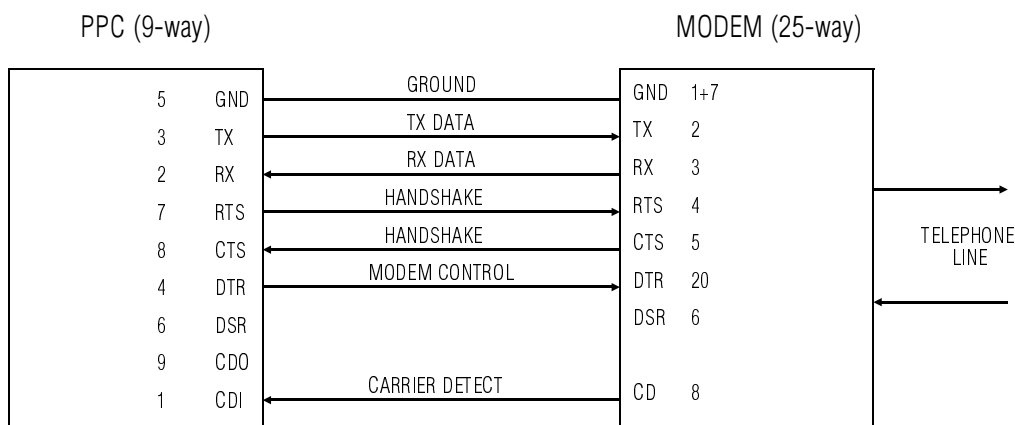
 Failure to observe the above will result in the XON/XOFF codes (11 and 13 hex) to disappear from your data. This may not be immediately apparent, and will certainly not be apparent if you are testing your system with printable data only!

Modem Control

The following information applies to modern Hayes-compatible asynchronous modems with internal data buffering. It may not apply to old modems, or special-purpose modems.

The data flow *from the PPC to the modem* is handshaked by the modem's CTS signal. The data flow *from the modem to the PPC* is handshaked by the PPC's RTS signal. This is equivalent to a standard PC-modem connection with hardware handshakes. XON/XOFF handshaking should be avoided because it prevents the transfer of 8-bit binary data.

The following diagram illustrates the usual connections. Note that the PPC pin-outs are for its 9-way D-connectors whereas the modem pin-outs are for the standard 25-way modem D-connector:



The PPC port should be configured with the following handshakes:

RX handshakes:	RTS	DTR	XON/XOFF	TX handshakes:	CTS	DSR	XON/XOFF
	ON	OFF	OFF		ON	OFF	OFF

The **RTS** signal is controlled automatically by the PPC. It is normally HIGH, unless the modem is sending data which is not being retrieved by your program fast enough.

The **DSR** signal (not connected) is not generally useful although its function can be programmed on some modems.

The **DTR** signal is programmable on most modems. It can be used to force the modem to hang-up without using the time-consuming “+++” sequence which would be necessary otherwise.

The modem's **CD** (carrier detect) signal can be monitored by the PPC and provides an accurate indication of the modem's online/offline status.

With internally-buffered modems, the PPC–modem baud rate can be much higher than the data rate on the telephone line. The modem will handshake the data as required, using CTS.

Since no handshake system exists over a telephone line, any data which the modem receives from the telephone line is immediately passed to the PPC and enters the PPC's 255-byte RX buffer. To avoid buffer overflow, the PPC's program must promptly read this data and should not rely on any handshakes to hold-up the data flow.

Modems vary a great deal in their functions and most have a large number of software-programmable features. These features must be correctly configured. Please read your modem programming manual.

Ensure that your PPC-modem cable has only those connections actually required for the job. Many modems have signals additional to those shown above, and some special-purpose modems have test voltages present on certain pins. A connection of such voltages to the PPC will damage the PPC.

Note that modems use a different pin naming convention from most serial devices. Modems are “DCE” devices, which means that TX, RTS, DTR are all inputs, and RX, CTS, DSR are all outputs - the opposite of almost everything else.

Port Parameters

A range of serial parameters is supported by each port. These can be configured in two ways:

- 1 From the Executive **Runtime Port Config** menu. The settings done here are non-volatile and are stored until again changed in this menu.
- 2 From a PASCAL or C program, using the `SETPORT` or `setportcfg` functions respectively. The settings done here are volatile and are lost at power-down.

Due to slight differences in the internal hardware implementations of the ports, not all possible parameters can be configured on every port. The table below shows the valid combinations:

	PORT 1	PORT 2	PORT 3	PORT 4
baud rates	600‡ 1200 2400 4800 9600 19200 38400	300 600‡ 1200 2400 4800 9600 19200 38400	30 37.5 50 75 100 110 134.5 150 300 600 1200 2000 2400 3600 4800 7200 9600 19200 38400 57600 115200	30 37.5 50 75 100 110 134.5 150 300 600 1200 2000 2400 3600 4800 7200 9600 19200 38400 57600 115200
bits/word	7 8	7 8	5 6 7 8	5 6 7 8
parity	none even odd	none even odd	none even odd	none even odd
stop bits	1 2	1 2	1 2	1 2
RX RTS h/shake	on off	on off	on off	on off
RX DTR h/shake	on off	on off	on off	on off
RX XON/XOFF h/shake	on off	on off	on off	on off
TX CTS h/shake	always on †	always on †	on off	on off
TX DSR h/shake	on off	on off	on off	on off
TX XON/XOFF h/shake	on off	on off	on off	on off

When configuring from the Executive Port Config function, the Executive ensures that invalid combinations cannot be set. When configuring from a PASCAL program, the PASCAL function `SETPORT` returns a non-zero errorcode if an invalid combination is detected; the only exceptions are marked in the above table with † and these do not generate errors in `SETPORT`.

Baud rates marked with ‡ are not supported in the -H2 (2x higher speed) version of the PPC.

Executive Mode configuration




When the PPC is in its Executive mode, Port 1 is automatically configured to work correctly with a terminal or with `KTERM`. This “Executive configuration” is most likely different from the “runtime configuration” which you may have configured in the Runtime Port Config menu, or with the `SETPORT/setportcfg` functions.

If a program is running under the Executive, the Port 1 configuration operative during the program’s execution will be the “Executive configuration”. However, if the program is running in “autoexec” mode then the Port 1 configuration will be the Runtime Port Config configuration.

The reason for the above implementation is to support testing of programs when using `KTERM` with a command-line baud rate specification. For example, if you are running `KTERM` with a `/b38400` specification, then the Port 1 baud rate *during the KTERM session only* will be 38400 baud. If your program is designed to do something with Port 1 at 9600 baud and configures Port 1 accordingly, your debug statements would not be receivable with `KTERM`.

Cables

With the exception of the CDO signal which is not provided on a PC, all the PPC signals have the same input/output direction as the corresponding pins on the 9-way serial port of an IBM-compatible "PC-AT". Therefore, when driving most "output-type" devices (printers, plotters, etc) the cable which previously worked between a PC and the device can often be successfully used between a PPC and the same device.

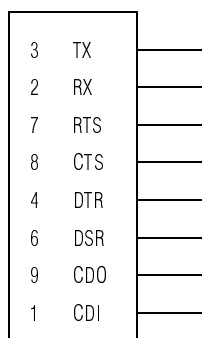
-  The cable information in this section is for guidance only. Always refer to the manuals for the equipment being interconnected before proceeding. A "standard" serial cable does not exist.
-  Ensure that your cables contain only those connections required for the job. Some devices have special voltages on certain pins, and these can damage the PPC. Avoid using "standard" cables containing all the 25 possible connections.
-  Connect cables only when the equipment being interconnected is switched OFF.

Modems

Please refer to the **Modem Control** section in this chapter for details.

PPC Loopback Adaptors

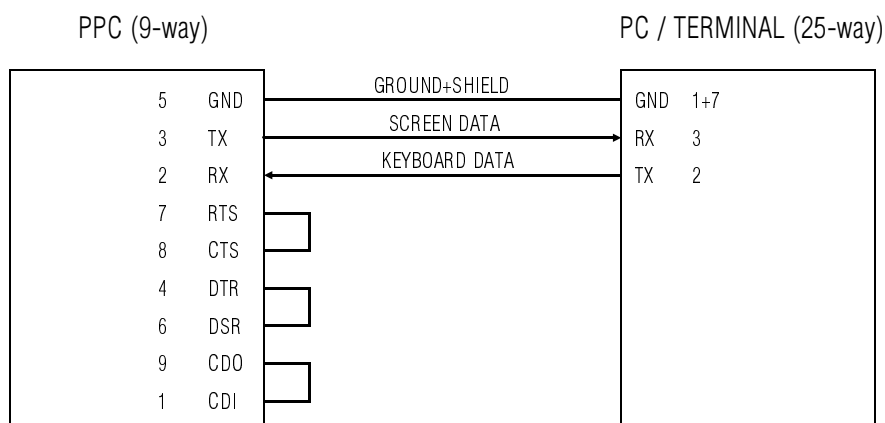
The **Serial Port Tests** function in the Executive requires special loopback adaptors to be connected to ports 2,3,4. The connections for each adaptor are below.



For Port 1, the recommended PC-PPC terminal cable (below) already contains the loopback connections required.

PC/Terminal ↔ PPC Cable

The following connections are recommended:



This cable is designed for use with the KTERM.EXE terminal emulation program. Some terminals and terminal programs (e.g. PROCOMM) require a high level on their CTS and/or DSR inputs (pins 5 & 6) before transmitting data; this can be most easily achieved by interconnecting RTS+CTS+DSR (pins 4+5+6) on the terminal.

The terminal must be configured for the PPC Executive Mode default: **9600 baud, 8 bits/word, no parity, 1 stop bit, XON/XOFF handshake only**. If your terminal supports 19200 or 38400 baud, you can set the PPC to default to those baud rates with the **Executive Mode Port 1 Config** menu.

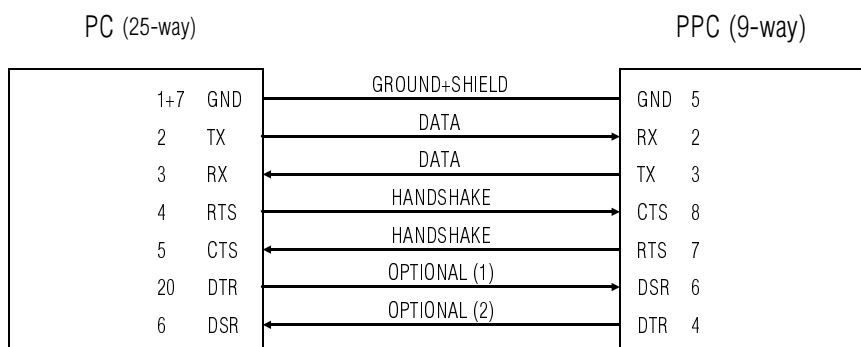
If your terminal does not support XON/XOFF but does support hardware handshake: connect the terminal's handshake signal (usually RTS, pin 4) to the CTS input on the PPC (pin 5).



The above cable is not suitable for sending data from a PC with the MS-DOS COPY or similar commands, because the BIOS will not transmit unless the CTS+DSR (pins 5+6 on a 25-way connector) inputs on the PC are both HIGH.

PC ↔ PPC cable

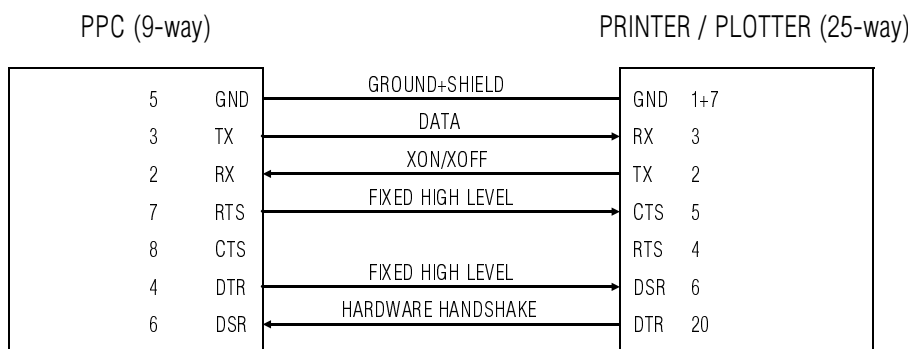
The following cable is suitable for use between an IBM-compatible PC and the PPC. The connections actually required will depend on the type of communication taking place, but this example shows all the possible connections for an application involving full-duplex binary data with full-duplex hardware handshakes. The cable will also be suitable if XON/XOFF handshaking is used, but note that binary data cannot be sent in the same direction as any XON/XOFF characters.



The connections named "optional" above perform no useful function, except that OPTIONAL(2) supplies an RS-232 HIGH level to the PC's DSR input as required by most PCs if transmitting data with BIOS functions.

PPC → printer/plotter cable

The following cable is suitable for driving most types of "DTE" output devices and supports both hardware and XON/XOFF handshakes:



The un-named connections do not usually serve any purpose except to supply an RS-232 HIGH level to the device's CTS and DSR inputs. For example, some plotters will ignore incoming data, or (e.g. in case of HP pen plotters) will not respond to interrogation commands, unless those inputs are HIGH.

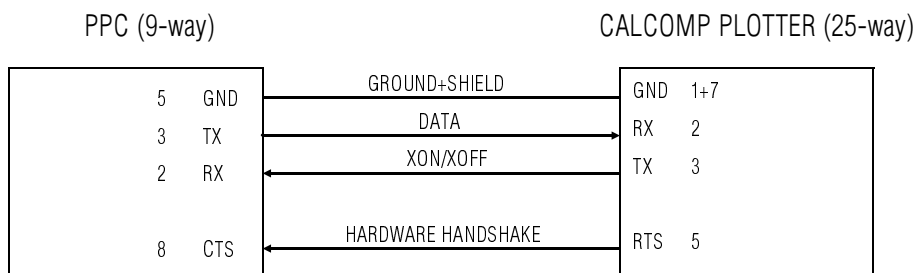
The PPC port should be configured with the following handshakes:

RX handshakes:	RTS	DTR	XON/XOFF	TX handshakes:	CTS	DSR	XON/XOFF
	OFF	OFF	OFF		ON	ON	ON

or as required by the device being driven.

PPC → CalComp plotter cable (“DTE Host”)

CalComp pen plotters (except the A3/A4 models ‡, Colorview ‡ and Plotmaster ‡) are unusual in that they use “DCE” straight-through connections. The following cable is suitable for driving most of the old CalComp pen plotters and supports both hardware and XON/XOFF handshakes:



The PPC port should be configured with the following handshakes:

RX handshakes:	RTS	DTR	XON/XOFF	TX handshakes:	CTS	DSR	XON/XOFF
	OFF	OFF	OFF		ON	OFF	ON

or as required by the device being driven.

Where more than one connector is provided on the plotter, use the one marked “DTE HOST”. If using the connector marked “DCE HOST”, or for the devices marked ‡ (whose single connector is effectively “DCE HOST”), use the more common PPC–printer/plotter cable on the previous page.

I/O Expansion Options

The PPC has an I/O expansion bus which supports the connection of custom I/O subsystems.

The connector is a 34-way 0.1"-pitch 2-row header. The signals available are an 8-bit data bus, several address lines and several control lines.

A clock is available whose frequency is 6.144MHz, 12.288MHz or 18.432MHz with the standard, -H2 or -H3 CPU speed options respectively.

Interrupts are possible and must be serviced using a user-installed ISR.

The address lines A14,A15 must be decoded to 1,1 to map the card into the 0xC000..0xFFFF Z180 I/O space.

If any signal on the expansion connector will be driving more than 2 loads, that signal should be buffered. This includes the data bus.

It is very easy to design I/O expansion cards featuring ADCs, DACs, FLASH memory, digital I/O - in fact just about anything. The I/O decoding is done with simple logic, or with a 16V8 GAL. The circuitry can be accessed from Pascal programs (using the "undocumented" `_INBC`, `_OUTBC` instructions) or from C programs using all normal methods.

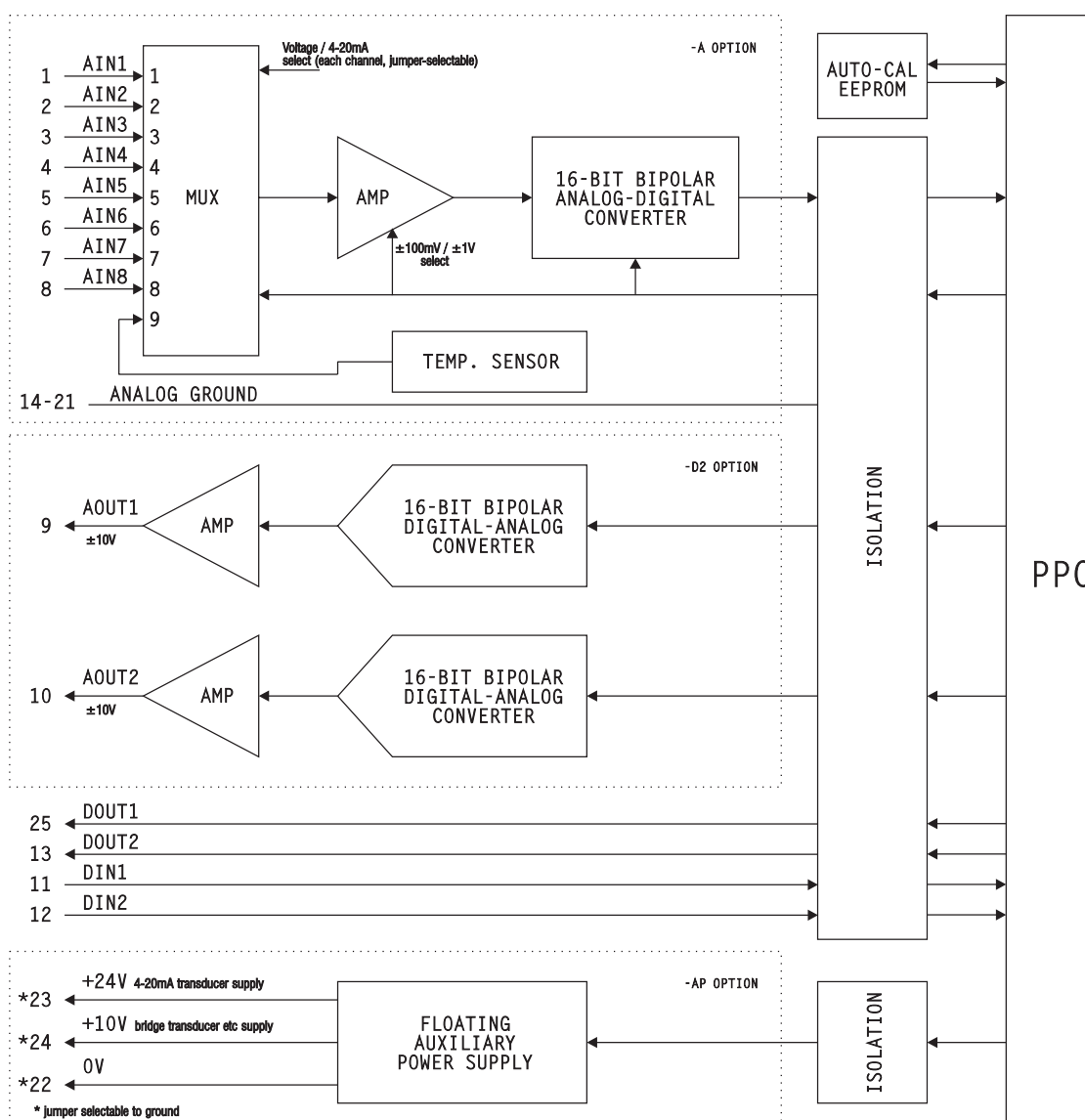
For further information on how to design your own I/O expansion cards, please contact factory.

16-Bit Analog Subsystem



The specification of this product is subject to change. Contact factory for information.

The following diagram shows the current version of the PPC Analog Subsystem option, with the pin numbers and pin names of the 25-way D-connector which the present card uses:



Facilities and Operating Modes

The PPC Analog Subsystem can be accessed from Basic, Pascal and C programs.

For measurements, the PPC Analog Subsystem supports both voltage and 4-20mA current loop inputs. Voltage inputs are software switchable to $\pm 100\text{mV}$ or $\pm 1\text{V}$. In the current loop mode the PPC can read and power up to eight 4-20mA transducers. The selection between voltage and current loop mode is jumper-selectable on a channel-by-channel basis.

In voltage mode, all readings are returned as 16-bit signed integers (all 16 bits are significant) and can be further operated upon using PPC's fast 32-bit floating point maths facilities.

For current loop mode the -AP option is required. This provides a +24V (nominal) power supply for powering standard 4-20mA self-powered transducers. Each of the eight possible transducers connects between this supply and its analog input terminal, each of which has a jumper-selectable 47R resistor to ground. The voltage across the 47R is measured (on the 1V range). After removing the "4mA" zero offset, the resulting unipolar reading is valid to just under 15 bits.

If a *floating* regulated +10V supply is required (e.g. for bridge transducer excitation) the -AP option provides a suitable floating output. However, if the +10V output is utilised, the +24V output is not generally usable because its ground

reference is now floating. For similar reasons, the floating +10V output can be used to power only *one* bridge transducer at any one time.

For control applications, two DAC outputs (–D2 option) are available. These share a common ground with the ADC circuitry.

Two TTL-level outputs and two TTL-level inputs are provided. These share a common ground with the ADC/DAC circuitry.

Hardware Requirements

To support the Analog Subsystem, the minimum requirement is a PPC-4.

Accuracy and Performance

The Analog Subsystem utilises calibration coefficients (corrections) which are generated during factory test and stored in an on-board EEPROM non-volatile memory. Separate corrections are stored and automatically applied to zero and scale errors in all voltage measurements, 4-20mA current loop measurements, and DAC outputs. All component tolerances are thus adjusted out in a manner which is transparent to the PPC programmer. The initial factory calibration is $\pm 0.02\%$ of full scale reading. Temperature effects are not calibrated-out but are typically within $\pm 100\text{ppm}/^\circ\text{C}$. Special build options are possible with lower TCs.

The calibration EEPROM is accessible from the PPC with PPC PASCAL commands (**ANEERD**, **ANEWR**) and its contents can be modified if necessary. Approximately 40 16-bit integer locations are free for other user storage; this is

For voltage inputs, two functions are available: calibrated (**ADCC**) and uncalibrated (**ADC**). The latter returns the raw uncorrected ADC reading and is therefore slightly faster. Similar functions are provided for the output.

Installation

The Analog Subsystem is normally factory-installed in the PPC. A field upgrade is possible and involves a PPC ROM change and a larger power supply unit.

Protection

The analog inputs are protected against overvoltage to $\pm 30\text{V}$.

Isolation

The analog side is isolated from the PPC side with components that are rated at 500V DC or higher and all PCB clearances support this also. Note, however, that the 25-way D connector which is used for all I/O cannot be properly specified at more than 50V DC although it can withstand much more. The isolation cannot be claimed to withstand 240V AC; this would require an isolation specification and test voltage of at least 4500V AC RMS.

DG1 Digital I/O Card

The -DG1 option card offers 16 CMOS-level schmitt trigger (74HC14) inputs and 16 high current (ULN2803A) open collector outputs. It can be accessed from Pascal and C.

The external connector is a 44-pin female "sub-miniature" D-connector. The metal shell of this connector is the same size as that of a standard 25-way D-connector.

Hardware Requirements

The minimum requirement is a PPC-4. The present version (schematic shown on next page) does not support the -H2 or -H3 options, due to the use of the Z80 PIO.

Installation

The DG1 card is normally factory-installed in the PPC. A field upgrade is possible and involves a PPC ROM change.

Protection

The inputs are protected against overvoltage to $\pm 30V$ by means of series 10k resistors. The outputs are not protected.

Isolation

There is no isolation.

Troubleshooting

Cannot enter the Executive

- 1 If the EXE LED is **not** flashing then the PPC is configured to execute ("autoexec") a program at power-up. To force entry into the Executive mode, press SW1+SW2+SW3 until the RST LED comes on, then release SW2+SW3 (only), holding-down SW1 until the EXE LED starts flashing.
- 2 If the EXE LED is flashing then the PPC is already in the Executive, and it most likely is a communications problem in the connection to the terminal. Try the following:
 - Reset the PPC, by holding SW2+SW3 together until the RST LED comes on, wait for EXE to start flashing again, and try again.
 - If using a non-KTERM terminal, press ESC on the terminal several times.
 - If using a non-KTERM terminal, check it is configured for **XON/XOFF only** (not hardware handshake) and is set to 9600,N,8,1.
 - If using KTERM, press ESC several times.
 - If using KTERM, exit it (with ALT-X) and re-enter.
 - If using KTERM, try a different serial port on the PC (e.g. COM2).
 - Someone may have changed the Executive mode baud rate from its 9600 baud default. It can be 1200-38400. If using a non-KTERM terminal, try setting it to other likely baud rates. If using KTERM, try setting the PC to different baud rates (with e.g. **SETSER COM1:38400,N,8,1**) and then running KTERM with the /NI override (**KTERM /NI**).

The PPC "hangs" on entry to Runtime Port Configuration menu

You are not using the correct cable. See the **Cables** section in the **PPC Serial Ports** chapter for details.

A program terminates for no apparent reason

- 1 An execution-time ("run-time") error has occurred. This should be indicated by the EER LED. Run your program from the Executive and, if PASCAL, with runtime errors *not* disabled. If this does not reveal it, check your program for places where any of the following might occur:
 - Division by zero
 - Overflow during arithmetic operations. Note that just simply adding 1 to an integer will cause an overflow after at most 65535 additions.
 - Excessive memory allocation (with the NEW function, etc)
 - Excessive recursion
- 2 The program has crashed. This is hard to achieve in BASIC or PASCAL, other than by a stack overflow (e.g. excessive subroutine nesting) or by allocating too much memory with NEW. Such "out of memory" errors cannot be disabled and will always be fatal.

In PASCAL, runtime errors can be suppressed with the O- and R- compiler options; however, this is not a substitute for a carefully-written program. In BASIC, runtime errors are suppressed when running in the autoexec mode, but such an error causes the program to terminate and restart from the first line.

A program terminates while reading data

- 1 A 03h character (CTRL-C) has appeared in Port 1 input data. In BASIC, the only way to disable break checking is to run the program in autoexec mode. In PASCAL, break checking is disabled with the C- compiler option.
- 2 An invalid number has appeared in the input data when reading numbers. In BASIC this error cannot be suppressed, other than by reading characters one at a time and processing them "intelligently". In PASCAL this error can be suppressed with the O- and R- compiler options; however, if your input data is likely to contain occasional garbage then you should first read the input as "chars" into a buffer, and process the buffer afterwards.

A program hangs during communications

This can be a handshaking problem, caused by the XON/XOFF handshake being enabled when it should not be. For the explanation, see below.

Loopback tests hang-up the PPC

You have the XON/XOFF handshake enabled. If your loopback test uses the full 256-code character set (most do) then an XOFF character will appear within the data.

However often this is a standard data input reading problem: you need to write any input code to allow for errors, with an overall timeout if necessary. If your program is written to always expect e.g. 20-byte packets to arrive, and for some reason only 19 bytes arrive in one packet, it will hang forever, and then probably mis-read the next packet.

RS422/485 Problems

Due to a general lack of standardisation in this area, one often needs to experiment when connecting to other equipment.

Check that the A/B connections are connected to the corresponding terminals at the other end. You may need to experiment by swapping A/B.

Ensure that the -7V to +12V RS422/485 common mode range is not being exceeded. This will usually manifest itself as corrupted data, or blown-up interface chips. If in doubt, make sure the GND connection is present, or use a port isolator.

In multidrop systems, one sometimes needs pullup/pulldown resistors to ensure that the bus is in the proper "space" state when no device is driving it. The PPC does not have these resistors built-in. Some devices have them but connected the wrong way round, causing all receivers to receive long break levels whenever the bus is not driven. This manifests itself as garbage received before each message.

C Introduction

This chapter is a quick “get started” guide to the PPC C programming option.

With the C option installed, the PPC retains all its functionality as described elsewhere in this Manual. Both Pascal and Basic remain available.

The C option has been implemented in a way which makes programming very straightforward and – apart from the obvious necessity for you to know *some* C – much easier than C programming on other hardware devices. The compiler is a full ANSI C compiler which generates state-of-the-art highly optimised code. It is not a “tiny” C or other C with reduced functionality. Many PPC-specific comms and other extensions have been added.

How was the C option implemented on the PPC? A standard PPC can handle files of type Pascal and Basic. With the C option, support for type Binary has been added. You simply compile your .c file(s) into a single .bin file which is transferred to the PPC using the file transfer functions of the supplied KTERM.EXE terminal emulator. Once this .bin file is designated “autoexec” it runs when the PPC is rebooted. Also, with PPC v1.07+, you can run it from the Compile and Run menu item. That is all there is to it!

The C option consists of the following:

- 1 An enlarged PPC ROM which contains the C runtime library routines. Because almost the entire runtime library is in ROM (as is also the case with PPC Pascal), the entire 30,464-byte RAM area is available for compiled code. This enables the generation of large programs. Because C programs are cross-compiled on a PC and also due to greater code efficiency of C versus Pascal, C programs can be typically 5 times larger (in # of lines) than can be achieved with PPC Pascal. This makes C recommended for highly complex PPC applications.
- 2 A Hi-Tech Z180/64180 C cross compiler, version 7.31 or higher. This compiler is available in MS-DOS and other versions. It is priced separately from the PPC, and must be ordered separately. This compiler is a standard off-the-shelf unmodified Hi-Tech product.
- 3 A diskette containing a PPC-specific runtime library symbol table, in .OBJ format. Your C program is linked with this symbol table and this ensures that it calls the various (ROM-based) library functions at the correct addresses. Also supplied is a C example program, a makefile and other utilities required to produce the final .bin file.

You will also need a “make” utility to run the supplied makefile. Virtually any such utility, including those supplied with Borland or Microsoft languages can be used. The standard protected-mode make is preferred because it leaves more memory for the compiler.

Current versions of the Hi-Tech compiler are 386-only protected mode versions. These work in the same way as the old “real-mode” version.

Not all features of the Hi-Tech compiler are supported with the PPC:

The compiler includes an IDE (integrated development environment). However, because of the ease with which a makefile-based system can be set-up (and because many programmers prefer to use their favourite editor anyway), the use of the IDE to achieve the various required compiler and linker operations is not documented here. Also, the simpler makefile approach may be preferred for projects which may need to be maintained far into the future.

The Hi-Tech compiler kit includes a source-level debugger called Lucifer. At present this is not supported. It is much less necessary than would be the case with other targets because the PPC’s interrupt-driven background transmit makes it very easy to insert debug statements into a program without slowing its execution in a way which would make the results meaningless.

The compiler also supports “large-model” programs. These use code banking to enable generation of programs with >64k of code. This mode is not presently supported in the PPC, although it could be used if the PPC was used as the basis for a custom product.

Step 1: Get to know the PPC

You are strongly advised to acquire a general understanding of how the PPC works. At least, please run through the Get Started section in the PPC Manual, and quickly scan the rest.

Step 2: Compiler Installation

The Hi-Tech C compiler is a DOS cross compiler which is a general purpose Z80/Z180 compiler and is not specific to the PPC in any way. To install it, follow the steps in the compiler documentation.

Keep the original disk safely, and always install from a *copy* of it. Besides this being good practice, the *installed* copy of the compiler on the hard disk is copy-protected with a scheme which ties it to some property of the hard disk. If the hard disk is reformatted you will need to re-install from the diskette. You are right - we don't like this either!

Unless this Hi-Tech compiler is the only Hi-Tech compiler which you are ever likely to use, choose an installation directory name like \ht180 rather than just \ht. This is because a version of the compiler for another CPU (e.g. H8/300) may by default install in \ht also.

The Hi-Tech compiler disk includes C and assembler sources for the standard Hi-Tech library. There is no need to install these although you might as well do so, unless you are short of hard disk space. With very few exceptions, the standard library is what is provided in the PPC ROM. The PPC-specific extensions are additional. The Hi-Tech-supplied libraries are not used for the PPC.

Step 3: PPC C Software Utilities Installation

Create a directory to work in, e.g. c:\ppc. Copy all files from the \c directory on the PPC Software Utilities diskette into it. **Read the read.me file.**



The C option requires **KTERM.EXE** version 1.02 or later (.EXE dated 1/FEB/94 or later). If you are an existing PPC user, make sure you have an up-to-date version of KTERM.

Step 4: C Option installation

The C Option is factory-installed in the PPC ROM. Its existence in your PPC can be confirmed by invoking the **Config Dump** menu item and selecting the output (in ASCII) to port 1. Existing PPCs without the C Option can be field-upgraded. Contact your PPC supplier for details.

Step 5: Reboot your PC

This is necessary to make effective any changes to your autoexec.bat file made by the compiler installation program. These changes are usually limited to a) adding the compiler path to your path and b) adding several environment variables to your environment. As is usual with programs which modify your autoexec.bat file, you should examine the changes in autoexec.bat to make sure they have not affected something else!

Step 6: Create a program

Using your favourite *non-document* editor, create the following program called hello.c

```
#include "ppclib.h"
#pragma psect text=r_text
#pragma psect const=r_const
#pragma psect strings=r_strings
#pragma psect data=r_data
#pragma psect bss=r_bss

int main()
{ int i;
  for (i=0; i<10; i++)
  {
    fprintf(port1,"Hello World \r\n");
    loadtimer(0,500); while (readtimer(0));
  }
}
```

The above program will output "Hello World" 10 times (at 500ms intervals) and then exit the main function. This causes a return to the PPC Executive which, provided that the Executive Mode baud rate matches the Port 1 Runtime baud rate, will appear and enable you to reboot the PPC and run it all over again.

A copy of hello.c is on the diskette.

The #pragma directives in the last example ensure that the linker correctly locates the various data, code etc sections. They appear unchanged at the start of every PPC C program.

A ready-to-use makefile is provided which compiles, links and locates a program hello.c into a file hello.bin which is suitable for immediate transfer to the PPC. Type

```
make hello.bin
```

and, when the file has been produced *with no errors*, use KTERM to transfer it to the PPC's internal filing system. (To avoid confusion with baud rates, run KTERM (version 1.02 or later, .EXE dated 1/FEB/94 or later) at 9600 baud for this exercise, i.e. without any /b command line switches).

PPC firmware v1.06 or lower: Using the Define Power-Up Action menu item, mark hello to execute at power-up. Then reboot the PPC, either using the Reboot menu item, or with the front panel reset. Following completion of the brief power-up test, the program will run and the text output will appear on your KTERM terminal connected to Port 1.

PPC firmware v1.07 or higher: use **Compile and Run** to run the program, or as above.

The supplied makefile contains two targets: hello.bin and test.bin. The latter is a larger C program which tests all the PPC-specific C extensions, and its source test.c is a very useful reference for writing comms programs for the PPC.

Creating larger programs

The procedure is the same. You need to edit the supplied makefile and add a new target to it. A new linkfile must also be created.

Larger programs should be split-up into several modules. These are normally compiled separately and then linked together. This is good programming practice.

From this point on, C programming on the PPC is just like C programming on a PC. There are differences, of course, but these are mainly in the areas of input/output where the PPC differs greatly from the very rudimentary comms facilities which a PC provides.

The next section is a reference covering all PPC-specific C extensions and differences.

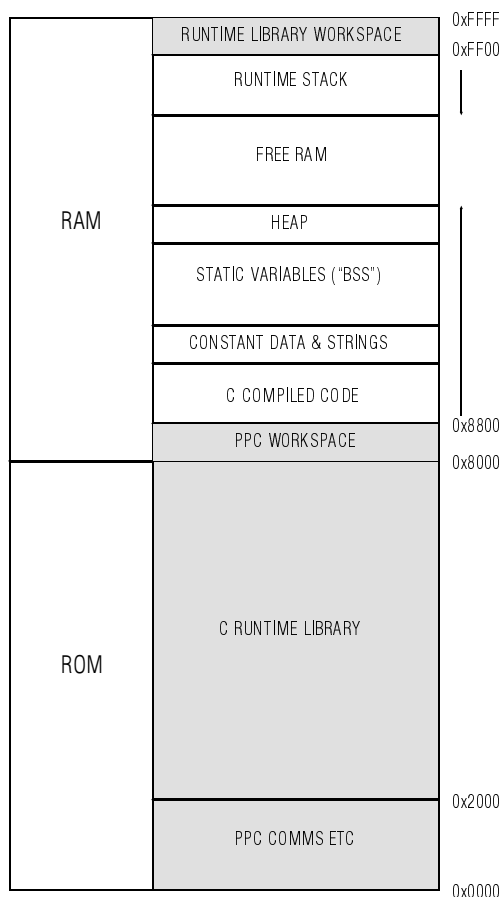
C Reference

This section covers the C Option for the PPC. It discusses PPC-specific C extensions and other features of the PPC C programming environment which a C programmer needs to know about.

The memory map

This is very similar to that used in PPC Pascal, in that the runtime library is ROM-resident and the binary program executes in RAM.

Code and data share a single 30,464-byte RAM area from 0x8800 to 0xFF00. The following diagram illustrates the memory allocation:



The shaded areas are shown for information only. You do not need to know anything about them.

The .bin disk file which you create contains code, any **const** strings and any **const** data. Upon PPC reboot, it is loaded by the PPC firmware at 0x8800. As you may note from the linker invocation in the supplied makefile, at the code entry point (at 0x8800) is a small piece of code (jp2main). This performs two important functions:

- 1 It checks that the version of your compiler is compatible with the version which was used to produce the ROM-based runtime library; a mis-match (which is a fatal error) is indicated by a permanent moving 00000001 pattern on the PPC LED display.
- 2 It jumps to the `main` function in your program, which may be anywhere in the program.

Because the entire program is in RAM, any **const** data or strings are modifiable.

To ensure that your compiled program calls the ROM-based library functions at the correct ROM addresses, you link your program to a supplied .OBJ module which is basically just a compiled symbol table which contains the addresses of all ROM-based functions. Unlike other environments, you do *not* link your program to any "library" as such. The only exception to this is the `ppcextra.lib` library which contains some rarely-used C functions which were not included in the PPC ROM due to space considerations.

C Extensions

All functions which are implemented in the PPC C system are defined in the `ppclib.h` file. Nearly all of these are contained in the ROM-based runtime library.

This runtime library also contains `long` and `float` versions of all relevant functions, including `fprintf` and `scanf`, so the usual huge program size penalty involved in using floats is avoided.

A few rarely-used functions (e.g. `malloc`) are in a separate library, **`ppcextra.lib`**, which you must link to your program if you need them.



`ppclib.h` is the only file which you need to `#include` in your program. Do *not* include `stdio.h`, `conio.h`, or any other of the `.h` files supplied with the Hi-Tech compiler.

This section documents all PPC-specific functions, i.e. those which are *not* part of ANSI C. It also documents all functions which the PPC implements *differently* from the ANSI C standard. Please refer to the Hi-Tech manual (or a C reference book) for details of all remaining PPC C functions (their prototypes are listed in the `ppclib.h` file).

Because many of the PPC-specific extensions are similar or identical to those provided in PPC Pascal, you are advised to turn to the PPC Pascal section and check the description of the corresponding PPC Pascal function (if present) for a more detailed description.

Note that Hi-Tech C supports comments with the “C++” `//` sequence, as well as the more commonly used `/* ... */` sequence.

For usage examples of all PPC-specific functions, see the supplied **`test.c`** program.

I/O Read/Write Functions

The I/O has been implemented similarly to PPC Pascal: the four PPC ports correspond to four streams called `port1`, `port2`, `port3`, `port4`. These streams are supplied as the first parameter to each of the various C I/O functions provided. For convenience, the supplied `ppclib.h` file `#defines` the symbols `port1`, `port2`, `port3`, `port4` as `1,2,3,4`.

These streams are **not like disk files**. They are in fact much simpler to understand and use:

- Since data flowing through the PPC is “continuous”, **you will never get EOF** returned by any I/O function. (This is why most of the PPC-specific character-based functions use type `char`, (not `int`) to represent character data – this also produces more efficient code.) The standard C “file” behaviour (reading what there is to be read and then getting EOF) is generally not useful in a datacomms product because you would be getting EOF very frequently – whenever you have emptied the input queue.
- **All I/O is binary**; there is no built-in conversion of LFs to CRLFs, no trapping of CRs, no hidden buffering of input until a CR arrives, no trapping of control-C (0x03) – anywhere. Pure and simple!
- **All input functions wait** (i.e. hang!) until the appropriate number of chars has been read from the PPC input queue. If this is not desired, use an input queue test like `ipqcount` to check the queue before reading. The PPC has a 255-byte input (RX) queue for each port.
- **All output functions wait** (i.e. hang!) until the appropriate number of chars has been written to the PPC output queue. If this is not desired, use an output queue test like `opqspace` to check the queue before writing. The PPC has a 60-byte output (TX) queue for each port.
- There are no `stdin/stdout` functions like `getch`, `getchar`, `putch`, `putchar`, `getc`, `putc`, etc etc – whose precise operation (e.g. trapping of 0x03) differs from compiler to compiler in any case.
- In porting C to the PPC, the emphasis has been on simplicity, while providing everything that is needed to build the most complex datacomms applications.

The following are the *only* single-character I/O functions provided:

```
char fgetc ( int port );           // returns a char from specified port.
int fgetcnd ( int port );         // as above, non-destructive (i.e. does an ungetc)
int fputc ( int port, char char ); // outputs a char to specified port
```

The following are simplified character-oriented versions of `fwrite` and `fread`:

```
int fwritec ( int port, char * bufname, int charstowrite );
int freadc ( int port, char * bufname, int charstoread );
```

The number of chars to write/read is specified by the third parameter. These functions are pure binary – they do **not** stop at a null (0x00). For situations where they can be used, they are more efficient than making repeat calls to `fputc/fgetc` within a loop.

The following are the only `printf`-style functions provided:

```
int fprintf ( int port, char * fmt, ... );
int vfprintf ( int port, char * fmt, va_list ap );
```

The above differ from the standard C versions only in that the first parameter is an `int` which specifies the PPC port in the range 1..4. All ANSI features are supported including longs and floats. There is no `printf` although you could define one using a macro in terms of `fprintf(stdout,...)` where `stdout` is defined in `ppclib.h` as `port1`.

The following are identical to ANSI C and are therefore listed here for completeness only:

```
int sprintf ( char * bufname, char * fmt, ... );
int vsprintf ( char * bufname, char * fmt, va_list ap );
```

All “printf”-type functions return -1 if an invalid port is specified. Other returns are as per ANSI C.

The following are the only `scanf`-style functions provided, and are identical to ANSI C:

```
int sscanf ( char * bufname, char * fmt, ... );
int vsscanf ( char * bufname, char * fmt, va_list ap );
```

There is no `fscanf` (i.e. no conversion of incoming data “as it arrives”) because such a function is practically useless for building a robust system. Robust operation involves reading some data into a buffer and then stepping through the buffer and “intelligently” identifying its content. In any case, typical `scanf` implementations read chars into a buffer (whose existence is hidden somewhere in the runtime library and whose size is fixed at e.g. 256 bytes!) and, upon receipt of a LF (0x0A, \n) process the buffer. Such mode of operation would be useless for data which does not have an LF, or which overflows the buffer.



Avoid using the name `port` for anything. This is a reserved word in Hi-Tech C. It is a typedef for a pointer to CPU I/O addresses. The Hi-Tech manual contains full details. When programming the PPC, you never need to access I/O ports directly unless you are doing special hardware-specific software or are accessing a custom I/O card.

I/O Queue Management Functions

In a device like the PPC, data flow generally never ends (although it may of course have gaps in it) and one common “PC” disk-file-oriented programming style, where a program terminates when e.g. an EOF is returned when reading a file, is not suitable for the PPC. Instead, your program will typically contain a loop where it tests for data in the input queue, tests space in the output queue, if both are OK it reads the input data, processes it and outputs it, then returns to wait for more input. The following functions support this programming style and also allow the creation of programs which process communications on multiple ports concurrently:

```
int ipqcount ( int port ); // returns # of bytes in input queue, 0..255
int ipqclear ( int port ); // clears (discards) input queue
int opqspace ( int port ); // returns space in output queue, 0..60
int opqcount ( int port ); // returns # of bytes in output queue, 0..60
```

For a more detailed description, see the similarly-named functions in the PPC Pascal section.

I/O Port Configuration Functions

You may never need to use these, because the entire configuration of each PPC port can be configured from the **Run-Time Serial Port Configuration** menu, and this is how it is usually done. These functions are needed only if your program needs to read and/or modify the port settings during its operation, or if you need to override the above settings.

The first of the following two functions enables your C program to configure a PPC port, and the second allows the present configuration to be read-back:

```
int setportcfg ( struct sp_t * anyone ); // configure port from sp_t
int getportcfg ( struct sp_t * anyone ); // fills in sp_t
```


Both make use of the same structure which is defined in `ppclib.h` as `sp_t`. Note that with `getportcfg` you must fill-in the port number; `getportcfg` then fills-in the rest.

I/O Port Direct Handshake Control

You may never need to use these, because the handshake operation of each PPC port can be configured from the Executive **Run-Time Serial Port Configuration** menu, and this is how it is usually done. These functions are needed typically only if your program needs to operate the various signals as *modem* control lines (in which case you may need to directly access DTR, CDI and possibly CDO) or if you are doing RS485 communications (in which case you will be controlling the RTS signal).

The first of the following two functions enables your C program to directly control the three “handshake” output signals, and the second allows the present states of the three “handshake” input signals to be read-back. Similarly to `setportcfg/getportcfg` above, these operations are done via structures `sph_t` and `gph_t` which are defined in `ppclib.h`.

```
int setporthsk ( struct sph_t * anyone ); // configure handshakes from sph_t
int getporthsk ( struct gph_t * anyone ); // fills-in gph_t
```

 If either of RTS or DTR has been *enabled* (with `setportcfg`, or in the Executive **Run-Time Serial Port Configuration** menu) it will be controlled *automatically* by the PPC and any attempt to control it with `setporthsk` will be ignored. If you want to control RTS or DTR with `setporthsk`, that output handshake must be *disabled* with `setportcfg` or in the Executive.

“Break” Sequences

Ports 3,4 are capable of detecting and generating an RS232 “break” level. This is in effect an over-length start bit, and can be used to indicate (to the receiving end) a condition which could not be represented simply by sending a normal message.

Unless you must use break generation for a very good reason, avoid it. It is rarely necessary in a properly designed system and is largely a hang-over from the origins of RS232 back in the 1960s.

```
int brkchk ( int port ); // return 1 if break detected, and clear the break detect flag
int brkout ( int port, unsigned int duration ); // send a break, length in ms
```

 `brkout` uses PPC timer #7 internally!

 The PPC tests the port 3,4 UARTs for a break condition from within a 1kHz interrupt, and `brkchk` may therefore fail to report an incoming break level if it lasts < 1ms.

Timing Functions

The PPC contains eight timers, numbered 0..7, each of which is decremented to zero (under interrupt) at 1kHz. Each timer can be loaded with a value up to 65535ms.

```
int loadtimer ( int timernum, unsigned int timervalue );
unsigned int readtimer ( int timernum );
```

`loadtimer` returns -1 if an invalid timer # is specified, 0 otherwise.

The above functions do not require the RTC option.

Real-Time Clock Functions

The PPC optionally contains a lithium-battery-backed real-time-clock. This can be set and read with the two following functions:

```
int getrtc ( struct tm * anyone ); // read RTC into tm
int setrtc ( struct tm * anyone ); // set RTC from tm
```

Both functions use the standard ANSI C structure `tm` which is defined in `ppclib.h`. Some members of `tm` (e.g. the daylight saving flag) are not implemented. Legal *year* values are 1950 .. 2049, represented (years since 1900 - see `tm`) as 50 .. 149.

Note that there is no PPC function which returns date/time in the form of a long int representing seconds elapsed since year 1900, or similar. Some additional ANSI C functions which provide additional support are defined in `ppclib.h` and are provided in a separate library `ppcextra.lib` which must be linked to your program.

Checksums and CRC

Many data communications systems use checksums or CRCs to ensure data integrity. A CRC function, in particular, is inefficient to implement in C and the following functions (written in assembler) are therefore provided:


```

unsigned char sumbuf ( char * bufname, int buflen );
unsigned char xorbuf ( char * bufname, int buflen );
unsigned int  crcbuf ( char * bufname, int buflen, int initvalue );

```

For a more detailed description, see the similarly-named functions in the PPC Pascal section.

Bit Operations and Rotation

These operations are not provided in standard C and, although they are easy to implement in C, the following self-explanatory functions have been provided for convenience and for performance- critical applications:

```

int testbit16 ( int bitnum, unsigned int value ); // returns 0|1
int testbit8  ( int bitnum, unsigned char value ); // returns 0|1
unsigned int  rotleft16 ( int rotations, unsigned int value );
unsigned int  rotright16 ( int rotations, unsigned int value );
unsigned char rotleft8  ( int rotations, unsigned char value );
unsigned char rotright8 ( int rotations, unsigned char value );

```

Extended Memory Functions

The PPC -96 option provides extra 96k bytes of RAM. This additional RAM is not used by anything in the PPC. The following functions access this extra memory as a linear 96k-byte array of chars (a random access file):

```

unsigned char readmem ( unsigned long address );
void writemem ( unsigned long address, unsigned char chartowrite );

```

where address can be any value 0 .. 98303.

Miscellaneous Functions

The following function returns the states of the three front panel switches into a structure (defined in ppplib.h) `sw_t` :

```

void getsw ( struct sw_t * anyname );

```

The following function copies `status` (0 or 1) into the front panel LED labelled USR:

```

void uled ( int status );

```

Other functions enable direct access to all eight front panel LEDs, or all 16 if it is a dual-display PPC.

The following function fills-in a buffer with information on the PPC firmware version, number of ports, firmware date, CPU speed, PPC serial number, and other data:

```

int sysdata ( char * bufname );

```

At present, a buffer of approximately 23 bytes is required. `sysdata(0)` returns the exact size. See the example program `test.c` for more details.

The following two functions support the PPC Digital I/O Card (Type 1):

```

unsigned int pdin ( void ); // read 16 inputs
void pdout ( unsigned int bitpattern ); // set 16 outputs

```

For a more detailed description, see the similarly-named functions in the PPC Pascal section.

The following functions support the PPC 16-bit Analog Subsystem:

```

int readadc ( void ); // read ADC (calibrated)
int readadcu ( void ); // read ADC (uncalibrated)
int setdac ( int channel, int value ); // output to DAC (calibrated)
int setdacu ( int channel, int value ); // output to DAC (uncalibrated)
void setadcgain ( int gain ); // set ADC range
void setdout ( unsigned int state ); // set two TTL output lines
unsigned int getdin ( void ); // read two TTL input lines

```

For a more detailed description, see the similarly-named functions in the PPC Pascal section.

The following functions provide access to the 2000-byte user-accessible non-volatile (EEPROM) area:

```

int nvwrc ( unsigned int addr, unsigned char value ); // write a char
int nvrdc ( unsigned int addr ); // read a char

```

```
int nvwrblk ( char * src, unsigned int dest, unsigned int blksize ); // write a block
int nvrdbl ( unsigned int src, char * dest, unsigned int blksize ); // read a block
```

For a more detailed description, see the similarly-named functions in the PPC Pascal section.

DES Encryption Functions

The runtime code for these is present only if the PPC is fitted with the **DES Option**.

```
int des_defkey ( char *key, char *subkeybuf ); // specify a key
void des_encrypt ( char *src, char *dest, char *subkeybuf ); // encrypt 8 bytes
void des_decrypt ( char *src, char *dest, char *subkeybuf ); // decrypt 8 bytes
```

Further information on these functions is provided with the DES Option.

Miscellaneous

PPCEXTRA.LIB Extra Library

The following ANSI C functions are not provided in the ROM-based runtime library. If you need to use any of these, you need to add **ppcextra.lib** to the end of the linker command line.

qsort	asctime	sinh	strdup
malloc	ctime	cosh	bsearch
calloc	gmtime	tanh	
free	localtime		
brk	time_zone		
sbrk			

When `ppcextra.lib` is added to the linker command line, only those modules containing the functions which are actually used by your program are added to your program. The increase in program size is usually therefore small.

Checking of “printf”-type Function Parameters

The following C compiler `#pragma` statements (in `ppclib.h`) enable very useful compilation-time checking of the corresponding functions:

```
#pragma printf_check (sprintf)
#pragma printf_check (vsprintf)
#pragma printf_check (fprintf)
#pragma printf_check (vfprintf)
```

For example, the following (very common!) type of error is detected:

```
fprintf( port1, "a=%d, b=%d, c=%d", a, b, c, d ); // one extra parameter
```

For this to work, the compiler must be invoked with a warning level of `-1` (`-w-1` on the compiler command line) or lower.

Performance Hints

The following general suggestions apply to C programs generally, and are not in any particular order of importance. They are relevant only if you find you need the highest possible performance, otherwise you can safely ignore them all!

Use statically allocated variables instead of local (i.e. stack-based) variables. Global variables are much faster to access and produce a program which can be 30-50% smaller.

Pass structures, arrays and other “large” objects by reference rather than by value.

Dont use `fprintf` or `sscanf` unless you really need the flexibility which these very comprehensive functions offer. For example, when reading integers from input data, using `atoi` or `atol` can be much faster than using `sscanf`. However, doing so will not reduce program *size* because all these functions are already provided in the ROM-based C runtime library.

In I/O involving blocks of data, use `freadc` and `fwritec` in preference to calling `fgetc` or `fputc` many times in a loop.

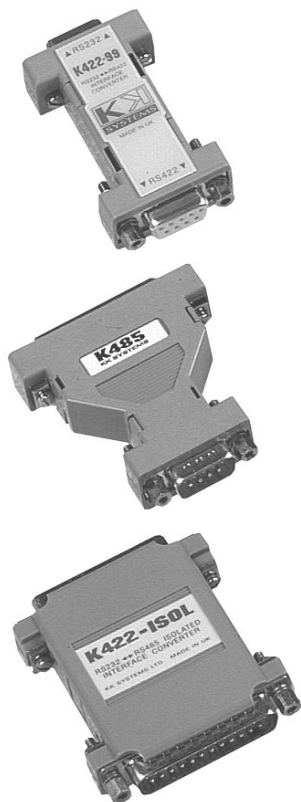
Avoid using the heap functions (i.e. `malloc`, `free` etc) unless you always free memory blocks in precisely the reverse order in which they were allocated, to eliminate fragmentation. Unlike a PC, the PPC is unable to advise the user that the program has run out of memory!

Technical Support

Questions on how to use the PPC, any of the PPC-specific C extensions, and any suggestions are all welcome.

Other Products

Inline RS232/RS4xx Converters and Isolators



A range of high quality very low cost RS232 to RS422/485 converters is available. All units plug directly into an IBM PC serial port (25-way or 9-way) and are line-powered from the RS232 interface's RTS or DTR signals. Isolated models (1kV RMS test) are also available.

Please see www.kksystems.com for full details and pricing of the whole range.

KD485 DIN Rail Converter



The KD485 is a multi-purpose isolated RS232-RS422/485/20mA interface converter with intelligent data processing options.

Three standard product versions cover most industrial interface and protocol conversion applications from a simple interface converter to a fully user-programmable unit.

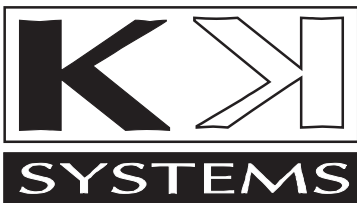
A MODBUS RTU comms library is available for rapid development of MODBUS-to-custom protocol converters using ANSI C.

Features include Auto Driver Enable, an Addressable Adapter Mode for multidropping non-addressable devices, baud rates 30-115200 baud.

The KD485 is 7-35V DC powered.

C programs are generally very portable between the PPC and the KD485.

We welcome enquiries for custom products. We have delivered many custom solutions, generally based on the PPC or the KD485.



KK Systems Ltd
Tates, London Road
Pyecombe, Brighton
BN45 7ED
Great Britain

☎ +44 1273 857185
fax +44 1273 857186
e-mail: info@kksystems.com
www: www.kksystems.com

Any questions or suggestions are welcome.