G-PASCAL NEWS

Registered by Australia Post - Publication Number VBG 6589

Published by Gambit Games for Commodore 64 G-Pascal owners.

Mail: G-Pascal News, P.O. Box 124, Ivanhoe, Victoria 3079. (Australia)
Phone: (03) 497 1283

Gambit Games is a trading name of Gammon & Gobbett Computer Services Proprietary Limited, a company incorporated in the State of Victoria.

VOLUME 2, NUMBER 3 — July 1985 Annual Subscription $12

WHAT'S IN THIS ISSUE

NEXT ISSUE

Our fourth (and final) issue will contain articles about other subjects that readers indicate are of interest. Any suggestions for other articles, or questions to be answered are welcome. Please write if you have a comment — do not assume that 'someone else' will. We in fact get very little mail, favourable or otherwise, about the contents of G-Pascal News.

CREDITS

Material in this issue by Nick Gammon and Sue Gobbett. Edited by Nick Gammon. Typeset on a Brother EM-100 typewriter using 'Lori', 'OCR-B', and 'Script' daisy wheels. Typesetting software written in G-Pascal to handle proportional space typefaces. Written and printed in July 1985.

COPYRIGHT

CUSTOMER RECORDS

Our mailing list for G-Pascal News is now fully computerised on a single disk file. Please let us know if we have made an error in your name or address and we will correct it for the next issue. Please indicate the name and address we used on the address label (attach it if possible), and what the correct name and address should be.

If you have not received either the first or second issue this year please let us know also.

Also please let us know if you have ordered a disk or cassette and it hasn't arrived.

We apologise for the late arrival of this issue, but we are suffering from somewhat of a staff shortage, resulting in it taking longer to do things such as preparing address lists than it should. We hope the next issue will arrive in a more timely fashion.

PRINTOUT OF PROGRAMS

We have used a new technique to print some of the example programs in this issue — namely using the 'Centronics interface' program presented on pages 5 to 12. As a result, the listings appear identically to how they would on the screen (without the screen wrap-around of course). Because of this, the number zero is no longer printed with a slash through it — be careful not to confuse it with the letter 'O'.

We have also made use of the printer interface to print some compiler output (with P-codes along the side), and to demonstrate Debug mode in operation.

We hope this is helpful, and that you can use the printer interface yourself if you have a Centronics interfaced printer.

Because space in the next issue will be short we would like to devote it to the issues you think are most important — please write and let us know.

G-PASCAL TUTORIAL – PART 2

Author: Nick Gammon

Arrays

If you have read the previous part of our G-Pascal tutorial (published in the April 1985 issue) you have probably noticed the absence of an explanation about the word ARRAY which appears in most of our published programs.

Arrays are a very powerful way of dealing with logically related groups of information, as the following introductory example will show.

Suppose that you are writing a program to process scores for a test from a classroom of students. We will start by writing a simple program that will read in five scores, store them in the computer's memory, display them back onto the screen, add them up and take an average.

First we'll try doing this without using arrays:

```
 1 VAR score1, score2, score3,
 2     score4, score5, total,
 3     average  : INTEGER;
 4 BEGIN
 5   WRITE ("Enter score 1: ");
 6   READ  (score1);
 7   WRITE ("Enter score 2: ");
 8   READ  (score2);
 9   WRITE ("Enter score 3: ");
10   READ  (score3);
11   WRITE ("Enter score 4: ");
12   READ  (score4);
13   WRITE ("Enter score 5: ");
14   READ  (score5);
15   WRITELN;
16   WRITELN ("Scores were:");
17   WRITELN ("Score 1: ", score1);
18   WRITELN ("Score 2: ", score2);
19   WRITELN ("Score 3: ", score3);
20   WRITELN ("Score 4: ", score4);
21   WRITELN ("Score 5: ", score5);
22   total := score1 + score2 + score3 +
23            score4 + score5;
24   average := total / 5;
25   WRITELN;
26   WRITELN ("Total: ", total,
27           " Average: ", average)
28 END.
```

This program is clearly a bit tedious, and would become impossible if we wanted to deal with, say, 100 students rather than five. Using arrays, however, we can 'subscript' a variable by placing a number at the end in square brackets (for example, 'score [5]'), rather than actually defining lots of slightly different variable names.

Now here is a program that achieves the same end result, but using an array:

```
 1 CONST students = 5;
 2 VAR score
 3         : ARRAY [students] OF INTEGER;
 4     total, average, count  : INTEGER;
 5 BEGIN
 6   FOR count := 1 TO students DO
 7     BEGIN
 8     WRITE ("Enter score ",count,": ");
 9     READ  (score [count])
10     END;
11   WRITELN;
12   WRITELN ("Scores were:");
13   FOR count := 1 TO students DO
14     WRITELN ("Score ", count, ": ",
15               score [count]);
16   total := 0;
17   FOR count := 1 TO students DO
18     total := total + score [count];
19   average := total / students;
20   WRITELN;
21   WRITELN ("Total: ", total,
22            " Average: ", average)
23 END.
```

The example above is not only shorter but more general – by changing the constant 'STUDENTS' on line 1 the program will automatically process any number of students, from one to thousands – whereas the first program will only work for exactly five students without major changes.

The array declaration on lines 2 and 3 tells the compiler that you wish to declare an array (group) of variables which share the name 'SCORE' but are subscripted by a number – the maximum number is specified in brackets. In this example the maximum size of the array is 'STUDENTS' (which is equivalent to 5, as it is a constant, declared to be equal to 5 in line 1).

In fact, in this case six variables are reserved by the array declaration, not five, as subscripted variables always start at subscript zero, rather than one. (So the six subscripts would be [0], [1], [2], [3], [4] and [5].) You may choose to ignore this and just

start at subscript one (as in the example program above), but in certain circumstances it is important to realise that the array really starts at subscript zero.

Another major use of arrays in G-Pascal is for the storage of 'strings' — that is, alphanumeric characters, such as a person's name. G-Pascal does not have a 'string' type as such, but by using arrays of type CHAR very useful effects can be obtained. The spelling checker program in the last issue is an example of this.

Let us try a simple program that will read in a person's name from the keyboard and echo it on the screen. To read a whole line of text at a time we must declare an array of type CHAR, and then use that array in a READ statement **without** specifying a subscript. This tells G-Pascal to fill the whole array with whatever is typed in before the user presses the RETURN key. So that you know where the end of the input text is, G-Pascal puts a carriage return (the number 13) at the end of the text.

The example below illustrates this:

```
 1 VAR name  : ARRAY [40] OF CHAR;
 2     count : INTEGER;
 3 BEGIN
 4   WRITE (CHR(147),
 5          "What is your name? ");
 6   READ  (name);
 7   WRITELN;
 8   WRITE ("Hello there, ");
 9   count := -1;
10   WHILE name [count + 1] <> 13 DO
11     BEGIN
12     count := count + 1;
13     WRITE (CHR (name [count]))
14     END;
15   WRITELN (", I hope you are well.")
16 END.
```

Use of the reserved word 'CHR' inside a WRITE or WRITELN statement outputs the contents of CHR's argument as a character rather than a number (hence its name). So on line 4 writing CHR(147) actually clears the screen, as 147 is the 'clear screen' character for the Commodore 64. On line 13 the use of CHR results in the contents of the array 'NAME' being written as characters rather than as numbers. To see what we mean, try the program without the word CHR on line 13 and you will see the decimal equivalents of whatever name you type in being displayed on the screen.

This program also illustrates the importance of realising that arrays start at subscript zero. On line 13 the first subscript displayed will be subscript zero (because 'COUNT' starts at -1, and has 1 added to it before the WRITE statement). If line 9 set COUNT to zero rather than minus 1, then you would lose the first character typed in — try it and see.

Using Debug Mode

As promised in the last issue, we will describe how to use the Debug Mode of G-Pascal.

In order to keep the amount of print-out to a manageable level we will choose a very simple example. If you want to try it yourself, key in the following small program:

```
1 (* %p *)
2 begin
3 writeln (5 + 6 * 7)
4 end .
```

If you compile this, you will get something similar or identical to the following (the %p on line 1 tells G-Pascal to 'display P-codes as they are generated'):

```
 G-Pascal compiler Version 3.1 Ser# 8374
 Written by Nick Gammon and Sue Gobbett
 Copyright 1983 Gambit Games
 P.O. Box 124 Ivanhoe 3079 Vic Australia
(401C)    1 (* %p *)
(401C)    2 begin
(401C) 3C 00 00
(401F)    3 writeln (5 + 6 * 7)
Jump at 401C changed to 401F
(401F) 3B 06 00
(4022) 85
(4023) 86
(4024) 87
(4025) 08
(4026) 04
(4027) 1E
(4028)    4 end .
(4028) 5E
(4029) 11

 P-codes ended at 402A
 Symbol table ended at C100
 <C>ompile finished: no Errors

 <E>dit, <C>ompile, <D>ebug, <F>iles,
 <R>un,  <S>yntax,  <T>race, <Q>uit ?
```

Referring to pages 79 and 80 of the G-Pascal Manual you can interpret the P-code meanings. The '3C' at 401C is an 'unconditional jump' which G-Pascal always inserts at the start of a program, procedure or function to 'jump over' any nested procedure or function declarations which might follow. In this case there aren't any, so the jump is subsequently patched to jump to the next sequential address (401F) as soon as the compiler realises this, and a message to this effect is displayed.

Then at address 401F the compiler generates an 'increment stack pointer' instruction (3B) with an operand of 0006. This reserves space for the 'stack frame linkage data' which is needed at the start of each program, procedure or function. The meaning of these six bytes is explained on page 76 of the Manual.

Then the compiler generates three 'load short literal' instructions for '5', '6' and '7' respectively, as described on the bottom of page 80 of the Manual. The addition of the '5' and the '6' is deferred because the multiplication of '6' and '7' has higher precedence.

Then at address 4025 the compiler generates a multiply (08) which will multiply the top two items on the stack.

At address 4026 the compiler generates an add (04) which will add the top two items on the stack.

At address 4027 is 'output a number' (1E) which will print the results of the computation.

At address 4028 is 'output a carriage return' (5E) as required by the 'writeln' statement, and finally at address 4029 is 'stop run' (11) which will cause the program to stop running.

Notice how most of the P-codes are only one byte long — this makes G-Pascal P-codes quite compact.

Having got this far, try running this program using Debug mode (by entering 'D' at the Main Menu), and you will see the following:

```
d
Running
(401C)  3C 03 00 3B
 Stack: CED0 = 00 00 00 00 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00
(401F)  3B 06 00 85
 Stack: CED0 = 00 00 00 00 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00
(4022)  85 86 87 08
 Stack: CECA = 00 00 00 00 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00
(4023)  86 87 08 04
 Stack: CEC7 = 05 00 00 00 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00
(4024)  87 08 04 1E
 Stack: CEC4 = 06 00 00 05 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00
(4025)  08 04 1E 5E
 Stack: CEC1 = 07 00 00 06 00 00 05 00
 Base:  CED0 = 00 00 00 00 00 00
(4026)  04 1E 5E 11
 Stack: CEC4 = 2A 00 00 05 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00
(4027)  1E 5E 11 00
 Stack: CEC7 = 2F 00 00 00 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00
47(4028)  5E 11 00 3C
 Stack: CECA = 00 00 00 00 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00

(4029)  11 00 3C 03
 Stack: CECA = 00 00 00 00 00 00 00 00
 Base:  CED0 = 00 00 00 00 00 00

run finished - press a key ...
```

In the listing above, we have underlined both the P-code currently being executed, and the current data at the top of the 'stack'. The 'stack frame linkage data' (identified as 'Base:') is always zero as no procedures or functions are currently active.

The display of the top of the stack is the stack contents **prior to** the P-code being executed, so you can see the effects of each P-code's execution four lines below where it appears.

For example, at P-code address 4022 is the P-code to push '5' onto the stack, and at the top of this column you can see the '5' on the stack. Numbers are always stored in reverse order, which is why the '5' appears as 05 00 00. Further on, you can see the results of the multiplication of 6 by 7 (decimal 42, hex 2A) on the top of the stack.

Then the results of the addition appear (decimal 47, hex 2F). Then the 'writeln' statement outputs the '47' which appears at the start of the line for P-code 4028. Then the '5E' P-code causes a blank line to appear.

Finally the '11' P-code causes the program to stop running.

We hope this little example clarifies the use of Debug mode, and how to interpret the P-codes. You may want to examine your own programs and see how they work.

Trace mode is similar to Debug, except that neither the contents of the stack nor the linkage data is displayed.

IMPROVED PRINTER HANDLING

We are pleased to be able to present improved methods of using/G-Pascal to print your programs, particularly if you are using a non-standard (i.e. non-Commodore) printer.

A problem in the past has been that not all G-Pascal owners have used the standard Commodore printers connected via the serial bus. (The serial bus is the same one which is used to connect to the disk drive, if you have one.)

In earlier issues of G-Pascal News we have described a stand-alone printing program which you could tailor to your needs. This is not completely satisfactory however as it does not allow you to easily print a program the moment you have typed it in, or during a compilation so as to get the P-code addresses printed alongside the source code.

MACHINE-CODE INTERFACE

In this article we describe a special machine-code interface which can be installed so that the 'Print' option in the Files Menu will cause printing to occur if you are using a Centronics type printer. The techniques described could be modified if necessary to be used with other printer types, for example an RS-232 printer connected via the VIC-1011A serial interface.

The interface routine described in this article consists of a 256-byte machine code routine (the assembly listing is provided) which will open, print to, and close a printer connected via the user port. This interface is specifically designed for a 'parallel' interface (commonly known as a Centronics interface).

HARDWARE REQUIREMENTS

The hardware is connected by using a suitable cable which connects the output of the 6526 chip (Complex Interface Adapter) number 2, which is addressed at addresses $DD00 to $DD0F and which is connected internally to the 'User Port'. We purchased a standard cable to do this from a local microcomputer dealer. One end of the cable plugs into the user port, and the other end has a Centronics connector which is plugged into your printer. In our case the printer is a Brother EM-100, but as the Centronics standard is pretty consistent this cable, and our printing routine, should work with any Centronics interfaced printer.

TYPING IN THE PROGRAM

To make the printer work with G-Pascal you will need to type in the G-Pascal program which follows. This includes the machine code routine as a series of hex constants, and also 'patches' G-Pascal using a number of MEM and MEMC statements to install suitable links from G-Pascal to the printer routine.

The G-Pascal program has a built-in sum check which should detect most keying errors you might make when entering the machine code constants, however you should still check the program carefully before running it, as the sum check will not be foolproof. Also, **save the program to disk or cassette** before running it, because an error could cause your Commodore 64 to 'lock up'.

PRINTING OPTIONS

We have endeavoured to make the printing routine as versatile as possible, so we have built in provision for automatically starting a new page after a specified number of lines, and automatically wrapping the printout around after a specified number of columns — in case your printer is narrower than what you are trying to print. There is also an 'echo' flag which you can set or disable so that what is printed is echoed on the screen as well, or not, as the case may be.

There is provision for manual intervention at the end of a page (by pressing a key) for use with 'cut sheet' printers such as daisy wheel printers, or the routine can automatically send a 'form feed' or other character if desired.

The machine code routine attempts to convert Commodore ASCII to standard ASCII as required by most non-Commodore printers. This is done in lines 104 to 119 of the machine code listing. The swapping of upper and lower case is controlled by a flag at address $C010 - make this zero if you are in 'upper case only' mode, and non-zero for normal upper and lower case mode. You may wish to adjust this part if your printer is not printing upper and lower case correctly, although it works perfectly on the printer we have here.

LOCATION OF ROUTINE IN MEMORY

Examination of how the routine works may

assist those users who do not actually have a Centronics type printer, but who can use the background information to develop something similar for their own printers.

In order to place the printing routine at an address that would be the least disruptive we have chosen to use the first 256 bytes of what is used by G-Pascal for: a) the compiler symbol table; b) doing a disk catalogue and c) the run-time stack. This is addresses $C000 to $C0FF. Lines 13 to 15 of the procedure patch G-Pascal so that it no longer uses that 256-byte block of memory. A side effect of this is that there is slightly less room available for the compiler symbol table and the run-time stack, but this would not affect most programs. If this is a problem you can place the routine at another address that you are not using, for example $2000 (if you are not using it for sprites etc.). To do this, once you have entered the program use the Replace command in the Editor to change each occurence of 'c0' to '20', by typing:

```
r.c0.20.g
```

The 'g' is very important as it tells the Editor to replace all occurences of 'c0' to '20' if there are more than one on a line (as there are in some cases). As the routine does not use hex 'c0' for other than references to addresses in the $C0xx range then this technique is simple and effective.

If you change the running address then you should delete lines 13 to 15 from the procedure as you no longer need to reserve space in the symbol table. Also the sum check will no longer work. We suggest you run the program in its original form first to confirm the sum check, and then make the modification described above.

'HOOKS' INTO G-PASCAL

Lines 48 to 51 of the G-Pascal program install the 'hooks' into the machine code routine — these consist of 'JSR' (Jump to Subroutine) instructions, referring to $C000 (open), and $C003 (close), which replace the built-in 'Commodore' open and close instructions originally in place in G-Pascal. Each 'JSR' instruction which is patched in is followed by an 'RTS' (Return from Subroutine - hex $60) which causes the remainder of the original code in G-Pascal to be bypassed.

CHANGING PRINTING OPTIONS

Lines 55 to 59 are examples of how to control the printing characteristics, you can modify or omit them as you wish. Lines 55 and 56 control the page length and width respectively, alter these to suit your printer. If either are set to zero then that disables any automatic handling for that parameter. For example, a page width of zero means 'never send an automatic carriage return no matter how wide the page is', similarly a page length of zero will disable all automatic form feeds.

Line 57 controls the 'echo to screen flag' — this should be 'true' (non-zero) to cause printed material to echo to the screen as well as print, and 'false' (zero) to cause printout to only appear at the printer.

Line 58 specifies which character will cause a new page — it is currently set to 147 — the normal 'clear screen' character. Line 59 specifies which character will be sent to the printer when the routine receives either the character specified in line 58, or when the maximum page length has been reached. In other words, as the program is currently set up, if you 'print' a 'clear screen' (CHR(147)) then the program will send a 'form feed' (CHR(12)) to the printer. If the value in line 59 set to zero then special processing occurs — a zero is not sent to the printer, rather the program waits for the [SHIFT] key to be pressed and released. This is so that a new page can be inserted into 'single sheet' printers, such as daisy wheels. No warning message occurs while the program is waiting for the [SHIFT] key to be pressed, so try pressing [SHIFT] if the program seems to have gone dead whilst printing.

You can also patch address $C010, as mentioned previously, depending on whether you want upper and lower case reversed or not.

TESTING THE PROGRAM

Once you have typed in this program and adjusted it for your requirements (and saved it to disk or cassette before testing it), you should just Run it. After a second it will just finish normally (unless there is a sum check error). If it finishes normally then you can enter the 'Files' menu and select 'Print' — you should then see the Files Menu being echoed on your printer. Then select 'Edit' and type 'L' followed by [RETURN]

and your program should list on the printer. To turn the printer off, return to the Files Menu and select 'Noprint', or alternatively press RUN/STOP and RESTORE simultaneously which will perform a 'warm start' and cancel any printing. If the program seems to go dead, try pressing the SHIFT key — you may have reached the end of the current page.

If your printer is not operational and G-Pascal 'hangs' just press RUN/STOP and RESTORE together to recover.

Once you have got the program working to your satisfaction save a copy on various disks or cassettes, ready for easy use at the start of the session. You will need to recompile and re-run the program for each session of G-Pascal as the machine code routine is lost once the power is turned off, but once it is installed it will stay operational for the entire session.

PRINTING FROM WITHIN PROGRAMS

Once you have installed this routine you can also call it from within your own programs, which provides a very powerful capability. You only need to CALL ($C000) to turn the printer on, and CALL ($C003) to turn it off again. With the printer on, all output from WRITE and WRITELN statements will go to the printer. You will probably want to turn off the 'echo' flag when using the routine from within a program. There is a small demonstration program showing how to use the printer routine from within your program in the following pages. Once the printer has been turned on once, you can just activate or deactivate it by changing location $C011 (PFLAG in the assembly listing). By doing this you will preserve the current line and column counters that the printer routine refers to.

When using the printing routines from inside your programs it is probably a good idea to check that the printing routine has been installed prior to running the program (by checking the first three bytes with a MEM statement). Our sample printing program does this at lines 12 and 13. In both programs we use the 'divide by zero' error to force a program abort.

PRINTING WITH THE RUNTIME SYSTEM

If you want to use the printing routine from a program written for use with the Runtime System then the printing routine will have to be installed as part of the program that you are running (because a program running under the Runtime System must be self-contained).

To do this you would just use the procedure INSTALL_PRINTER_INTERFACE which would be called at the start of your program, with the following changes:

1) Change line 13 to read:

```
memc [$95a5] := $C1; (* stack limit *)
```

2) Delete lines 14 and 15 (doing a catalogue, and the size of the symbol table do not apply to the Runtime System).

3) Delete lines 48 to 51 as the Runtime system does not have a 'print' option. (You will open and close print files manually from within your program).

The corrections described above apply to the line numbers **as printed** over the page — as you delete some of the lines described above then the ones further down will be renumbered — it would be best to start at the highest numbered lines and work backwards — that way the line numbers you are correcting will be the same as those printed.

NEXT ISSUE

We hope this article is helpful for customising your printing requirements. We may show in the next issue how to change the routine to handle RS-232 printers if there is enough interest shown by readers.

```
 1 (* Program to install G-Pascal Centronics interface *)
 2 procedure install_printer_interface;
 3 var i, sum_check : integer ;
 4 procedure x (a, b, c);  (* install 9-byte patch *)
 5 begin
 6   mem [i    ] := a;
 7   mem [i + 3] := b;
 8   mem [i + 6] := c;
 9   sum_check := sum_check + a + b + c;
10   i := i + 9
11 end ;
12 begin (* install_printer_interface *)
13   memc [$b1c3] := $c1; (* stack limit *)
14   memc [$9d02] := $c1; (* catalogue *)
15   memc [$800b] := 15;  (* sym table size *)
16   i := $c000;          (* patch address *)
17   sum_check := 0;      (* catch keying errors *)
18 x($13b038,$44b038,$47b038);  (* $c000 *)
19 x($37c006,$930150,$000100);  (* $c009 *)
20 x($000000,$27ad00,$c0c903);  (* $c012 *)
21 x($a211f0,$26bd01,$fe9d03);  (* $c01b *)
22 x($09bdc0,$269dc0,$10ca03);  (* $c024 *)
23 x($ffa9f1,$dd038d,$dd02ad);  (* $c02d *)
24 x($8d0409,$8ddd02,$adc011);  (* $c036 *)
25 x($a9dd0d,$128d01,$138dc0);  (* $c03f *)
26 x($a960c0,$118d00,$4860c0);  (* $c048 *)
27 x($c0148e,$c00dae,$ae05d0);  (* $c051 *)
28 x($d0c011,$20480a,$68c0f9);  (* $c05a *)
29 x($c011ae,$cd4cf0,$f0c00e);  (* $c063 *)
30 x($c1c970,$c90690,$02b0db);  (* $c06c *)
31 x($ae5fe9,$f0c010,$41c912);  (* $c075 *)
32 x($c90e90,$08905b,$9061c9);  (* $c07e *)
33 x($7bc906,$4902b0,$7f2920);  (* $c087 *)
34 x($c0158d,$dd018d,$dd00ad);  (* $c090 *)
35 x($8dfb29,$48dd00,$684868);  (* $c099 *)
36 x($8d0409,$addd00,$29dd0d);  (* $c0a2 *)
37 x($f9f010,$c015ad,$c00fcd);  (* $c0ab *)
38 x($c93df0,$11f00d,$c013ee);  (* $c0b4 *)
39 x($c00cad,$cd31f0,$b0c013);  (* $c0bd *)
40 x($0da92c,$a9c3d0,$138d01);  (* $c0c6 *)
41 x($12eec0,$0badc0,$1bf0c0);  (* $c0cf *)
42 x($c012cd,$a916b0,$128d01);  (* $c0d8 *)
43 x($0fadc0,$a7d0c0,$028dae);  (* $c0e1 *)
44 x($d001e0,$8daef9,$fbd002);  (* $c0ea *)
45 x($c014ae,$601868,$c0fe6c);  (* $c0f3 *)
46   if sum_check <> -5037022 then (* detect keying errors *)
47      writeln ("Sum check error in machine code.", 1 / 0);
48   mem  [$9e55] := $c00020; (* open routine *)
49   memc [$9e58] := $60;     (* return *)
50   mem  [$9e66] := $c00320; (* close routine *)
51   memc [$9e69] := $60      (* return *)
52 end ;  (* install_printer_interface *)
53 begin  (* main program *)
54   install_printer_interface;
55   memc [$c00b] := 62;      (* page length *)
56   memc [$c00c] := 80;      (* page width *)
57   memc [$c00d] := 1;       (* echo to screen flag *)
58   memc [$c00e] := 147;     (* character to cause form feed *)
59   memc [$c00f] := 12       (* form feed to printer *)
60 end .  (* main program *)
```

```
                1    * Routine to send data to Centronics printer
                2    * connected to Commodore 64 using G-Pascal.
                3    *
                4    * Author: Nick Gammon
                5    *
                6    */  Version 9 - JULY 1985
                7    *
                8            ORG  $C000      ; WHERE SYMBOL TABLE WAS
                9    *-------------------------------------------------------------
                10   *  Below are the three entry points: open, close and print.
                11   *  Open and close are called by the user to turn the printer on
                12   *  or off.  Print is called from the 'CHROUT' output vector.
                13   *-------------------------------------------------------------
C000: 38        14   OPEN     SEC             ; HERE TO OPEN PRINTER
C001: B0 13     15            BCS  OPENIT
C003: 38        16   CLOSE    SEC             ; HERE TO CLOSE THE PRINTER
C004: B0 44     17            BCS  CLOSEIT
C006: 38        18   PRINT    SEC             ; HERE TO PRINT A CHARACTER
C007: B0 47     19            BCS  PRINTIT
C009: 06 C0     20   PRINTADR DA   PRINT      ; ADDRESS OF 'PRINT' ABOVE
                21   *-------------------------------------------------------------
                22   *  Below are various system equates.
                23   *-------------------------------------------------------------
                24   DATA:A   EQU  $DD00      ; DATA REGISTER A (USED FOR STROBE)
                25   DATA:B   EQU  $DD01      ; DATA REGISTER B (USED FOR PRINTED DATA)
                26   DDR:A    EQU  $DD02      ; DATA DIRECTION REGISTER - A
                27   DDR:B    EQU  $DD03      ; DATA DIRECTION REGISTER - B
                28   ICR      EQU  $DD0D      ; INTERRUPT CONTROL REGISTER
                29   IBSOUT   EQU  $326       ; KERNAL OUTPUT VECTOR
                30   SHFLAG   EQU  $28D       ; 1 = SHIFT KEY PRESSED
                31   CR       EQU  13         ; CARRIAGE RETURN CHARACTER
                32   *-------------------------------------------------------------
                33   * THE 5 BYTES BELOW ARE USER-ADJUSTABLE CONTROL PARAMETERS.
                34   *-------------------------------------------------------------
C00B: 37        35   PAGELEN  DFB  55         ; PAGE LENGTH
C00C: 50        36   PAGEWID  DFB  80         ; PAGE WIDTH
C00D: 01        37   ECHOFLAG DFB  1          ; NON-ZERO MEANS ECHO TO SCREEN ALSO
C00E: 93        38   FFIN     DFB  147        ; 'CLEAR SCREEN' WILL CAUSE FORM FEED
C00F: 00        39   FFOUT    DFB  0          ; MAKE 12 FOR FORM FEED, 0 FOR PAUSE
C010: 01        40   CASESWAP DFB  1          ; 1 = SWAP UPPER/LOWER CASE
C011: 00        41   PFLAG    DFB  0          ; PRINTING IN PROGRESS FLAG
                42   *-------------------------------------------------------------
                43   * INTERNAL COUNTERS AND SAVE AREAS.
                44   *-------------------------------------------------------------
C012: 00        45   LINES    DFB  0          ; LINE COUNT
C013: 00        46   COLS     DFB  0          ; COLUMN COUNT
C014: 00        47   XSAVE    DFB  0          ; SAVE AREA FOR X-REGISTER
C015: 00        48   CHAR     DFB  0          ; THE CHARACTER WE ARE PRINTING
                49   *-------------------------------------------------
                50   * OPEN THE PRINTER FILE.
                51   *  Once the printer has been opened from within a program it
                52   *  can be re-opened by simply moving non-zero to PFLAG.
                53   *  Doing this would preserve current line and column counters.
                54   *-------------------------------------------------
                55   OPENIT   EQU  *
C016: AD 27 03  56            LDA  IBSOUT+1   ; ALREADY INSTALLED?
C019: C9 C0     57            CMP  #>PRINT
C01B: F0 11     58            BEQ  OPEN2      ; YES
C01D: A2 01     59            LDX  #1         ; NO - COPY VECTOR/INSTALL NEW ONE
```

```
C01F: BD 26 03  60   OPEN1    LDA  IBSOUT,X   ; OLD VECTOR
C022: 9D FE C0  61            STA  OUTVEC,X   ; MAKE A COPY OF IT
C025: BD 09 C0  62            LDA  PRINTADR,X ; NEW OUTPUT ROUTINE ADDRESS
C028: 9D 26 03  63            STA  IBSOUT,X
C02B: CA        64            DEX
C02C: 10 F1     65            BPL  OPEN1
                66   OPEN2    EQU  *          ; ON WITH OPEN
C02E: A9 FF     67            LDA  #$FF       ; SET PORT B DATA DIRECTION TO OUTPUT
C030: 8D 03 DD  68            STA  DDR:B
C033: AD 02 DD  69            LDA  DDR:A      ; AND BIT 4 OF PORT A
C036: 09 04     70            ORA  #4
C038: 8D 02 DD  71            STA  DDR:A
C03B: 8D 11 C0  72            STA  PFLAG      ; SET PRINTING FLAG
C03E: AD 0D DD  73            LDA  ICR        ; CLEAR ANY OUTSTANDING INTERRUPT FLAG
C041: A9 01     74            LDA  #1
C043: 8D 12 C0  75            STA  LINES      ; SET LINE COUNT TO 1
C046: 8D 13 C0  76            STA  COLS       ; DITTO FOR COLUMN COUNT
C049: 60        77            RTS             ; AND RETURN
                78   *-----------------------------------------------------------
                79   * CLOSE THE PRINTER FILE
                80   *-----------------------------------------------------------
                81   CLOSEIT  EQU  *
C04A: A9 00     82            LDA  #0
C04C: 8D 11 C0  83            STA  PFLAG      ; CLEAR PRINTING FLAG
C04F: 60        84            RTS             ; ALL DONE
                85   *-------------------------------------------------
                86   * Print a character (in A-Register). All registers preserved.
                87   *-------------------------------------------------
                88   PRINTIT  EQU  *
C050: 48        89            PHA             ; SAVE A-REGISTER
C051: 8E 14 C0  90            STX  XSAVE      ; SAVE X-REGISTER
C054: AE 0D C0  91            LDX  ECHOFLAG   ; ECHO ON SCREEN?
C057: D0 05     92            BNE  PSCREEN    ; YES
C059: AE 11 C0  93            LDX  PFLAG      ; PRINTING?
C05C: D0 0A     94            BNE  PRINTNOW   ; YES - SO PRINT ONLY
                95   PSCREEN  EQU  *          ; HERE TO OUTPUT TO SCREEN
C05E: 48        96            PHA             ; SAVE THE CHARACTER FOR LATER
C05F: 20 F9 C0  97            JSR  OUTPUT     ; USE NORMAL KERNAL OUTPUT ROUTINE
C062: 68        98            PLA             ; BACK TO OUR ORIGINAL CHARACTER
C063: AE 11 C0  99            LDX  PFLAG      ; PRINT ALSO?
C066: F0 4C     100           BEQ  PFINISHJ   ; NOPE - FINISH VIA PFINISHJ
                101  PRINTNOW EQU  *          ; HERE TO PRINT THE CHARACTER
C068: CD 0E C0  102           CMP  FFIN       ; FORM FEED?
C06B: F0 70     103           BEQ  GOTFF      ; YES - SPECIAL HANDLING
C06D: C9 C1     104           CMP  #"A"       ; UPPER CASE (8-BIT ON)?
C06F: 90 06     105           BLT  PRINT1
C071: C9 DB     106           CMP  #"Z"+1
C073: B0 02     107           BGE  PRINT1
C075: E9 5F     108           SBC  #$5F
C077: AE 10 C0  109  PRINT1   LDX  CASESWAP   ; CASE SWAP REQUIRED?
C07A: F0 12     110           BEQ  PRINT3     ; NO
C07C: C9 41     111           CMP  #'A        ; REVERSE CASE OF LETTERS
C07E: 90 0E     112           BLT  PRINT3
C080: C9 5B     113           CMP  #'Z'+1
C082: 90 08     114           BLT  PRINT2
C084: C9 61     115           CMP  #'a'
C086: 90 06     116           BLT  PRINT3
C088: C9 7B     117           CMP  #'z'+1
C08A: B0 02     118           BGE  PRINT3
C08C: 49 20     119  PRINT2   EOR  #$20        ; CONVERT TO NORMAL ASCII
```

```
                    120  PRINT3   EQU  *
C08E: 29 7F         121           AND  #$7F        ; NOW STRIP HIGH-ORDER BIT
C090: 8D 15 C0      122           STA  CHAR        ; SAVE OUR CHARACTER
C093: 8D 01 DD      123           STA  DATA:B      ; Send data to printer
C096: AD 00 DD      124           LDA  DATA:A
C099: 29 FB         125           AND  #$FB        ; STROBE PRINTER
C09B: 8D 00 DD      126           STA  DATA:A
C09E: 48            127           PHA              ; GIVE PRINTER TIME TO NOTICE IT
C09F: 68            128           PLA
C0A0: 48            129           PHA
C0A1: 68            130           PLA
C0A2: 09 04         131           ORA  #4
C0A4: 8D 00 DD      132           STA  DATA:A
C0A7: AD 0D DD      133  PWAIT     LDA  ICR         ; WAIT FOR ACKNOWLEDGE
C0AA: 29 10         134           AND  #$10        ; CHECK 'FLAG' BIT
C0AC: F0 F9         135           BEQ  PWAIT       ; NOT SET YET
C0AE: AD 15 C0      136           LDA  CHAR        ; SEE IF IT WAS <RETURN> OR FORM FEED
C0B1: CD 0F C0      137           CMP  FFOUT       ; FORM FEED?
C0B4: F0 3D         138  PFINISHJ BEQ  PFINISH     ; YES - DON'T COUNT AS A COLUMN
C0B6: C9 0D         139           CMP  #CR         ; WAS IT A CARRIAGE RETURN?
C0B8: F0 11         140           BEQ  GOTCR       ; YES - SPECIAL PROCESSING
C0BA: EE 13 C0      141           INC  COLS        ; COUNT COLUMNS
C0BD: AD 0C C0      142           LDA  PAGEWID     ; GET PAGE WIDTH
C0C0: F0 31         143           BEQ  PFINISH     ; ZERO MEANS IGNORE WIDTH
C0C2: CD 13 C0      144           CMP  COLS        ; PAST WIDTH?
C0C5: B0 2C         145           BGE  PFINISH     ; NOT YET
C0C7: A9 0D         146           LDA  #CR         ; SEND A CARRIAGE RETURN
C0C9: D0 C3         147           BNE  PRINT3
                    148  GOTCR    EQU  *           ; HERE WHEN WE GOT A CARRIAGE RETURN
C0CB: A9 01         149           LDA  #1
C0CD: 8D 13 C0      150           STA  COLS        ; COLUMN COUNT IS BACK TO 1
C0D0: EE 12 C0      151           INC  LINES       ; AND COUNT LINES
C0D3: AD 0B C0      152           LDA  PAGELEN     ; UP TO MAXIMUM?
C0D6: F0 1B         153           BEQ  PFINISH     ; ZERO MEANS ANY LENGTH OK
C0D8: CD 12 C0      154           CMP  LINES
C0DB: B0 16         155           BGE  PFINISH     ; NOT YET
                    156  GOTFF    EQU  *           ; HERE WHEN NEW PAGE WANTED
C0DD: A9 01         157           LDA  #1
C0DF: 8D 12 C0      158           STA  LINES       ; LINE COUNTER BACK TO 1
C0E2: AD 0F C0      159           LDA  FFOUT       ; SEND FORM FEED?
C0E5: D0 A7         160           BNE  PRINT3      ; YES - DO IT
C0E7: AE 8D 02      161  SHWAIT   LDX  SHFLAG      ; WAIT FOR SHIFT KEY
C0EA: E0 01         162           CPX  #1
C0EC: D0 F9         163           BNE  SHWAIT      ; NOT YET
C0EE: AE 8D 02      164  SHWAIT2  LDX  SHFLAG      ; NOW WAIT FOR THEM TO LET GO
C0F1: D0 FB         165           BNE  SHWAIT2
                    166  PFINISH  EQU  *           ; FINISHED PRINTING
C0F3: AE 14 C0      167           LDX  XSAVE       ; RESTORE X-REGISTER
C0F6: 68            168           PLA              ; RESTORE A-REGISTER
C0F7: 18            169           CLC              ; KEEP KERNAL HAPPY (CLC = NO ERROR)
C0F8: 60            170           RTS              ; RETURN TO CALLER
                    171  *
C0F9: 6C FE C0      172  OUTPUT   JMP  (OUTVEC)    ; INDIRECT JUMP TO KERNAL OUTPUT
                    173  *
                    174  OUTVEC   EQU  OPEN+254   ; USE LAST 2 BYTES IN THIS PAGE


--End assembly, 252 bytes, Errors: 0
```

```
 G-Pascal compiler Version 3.1 Ser# 8374
 Written by Nick Gammon and Sue Gobbett
 Copyright 1983 Gambit Games
 P.O. Box 124 Ivanhoe 3079 Vic Australia
(42F1)    1 (* %LIST - give us a compile listing *)
(42F1)    2 (* Printer demonstration program *)
(42F1)    3 const openprint  = $c000;
(42F4)    4       closeprint = $c003;
(42F4)    5       echoflag   = $c00d;
(42F4)    6       printflag  = $c011;
(42F4)    7       false      = 0;
(42F4)    8       true       = 1;
(42F4)    9 var   i,
(42F4)   10       number     : integer ;
(42F4)   11 begin
(42F4)   12   if mem [openprint] <> $13b038 then (* check machine code *)
(4301)   13     writeln ("Printer routine not installed.", 1 / 0 );
(4329)   14
(4329)   15   call (openprint);           (* open print file *)
(432E)   16   memc [printflag] := false;  (* don't print yet *)
(4334)   17   memc [echoflag]  := false;  (* no echo to screen *)
(433A)   18
(433A)   19   write ("Which multiplication table? ");
(4358)   20   read  (number);
(435D)   21   memc [printflag] := true;
(4363)   22   writeln ;
(4364)   23   writeln ("Multiplication table for ", number);
(4385)   24   writeln ;
(4386)   25   for i := 1 to 12 do
(4395)   26      writeln (i, " times ", number, " = ", number * i);
(43C7)   27   writeln ;
(43C8)   28   memc [printflag] := false;
(43CE)   29
(43CE)   30   writeln ("All finished!");
(43DE)   31   memc [echoflag]  := true;   (* echo to screen *)
(43E4)   32   call (closeprint)
(43E8)   33 end .

 P-codes ended at 43EA
 Symbol table ended at C184
 <C>ompile finished: no Errors

 <E>dit, <C>ompile, <D>ebug, <F>iles,
 <R>un,  <S>yntax,  <T>race, <Q>uit ? r
Running

Multiplication table for 7

1 times 7 = 7
2 times 7 = 14
3 times 7 = 21
4 times 7 = 28
5 times 7 = 35
6 times 7 = 42
7 times 7 = 49
8 times 7 = 56
9 times 7 = 63
10 times 7 = 70
11 times 7 = 77
12 times 7 = 84
```

USING SPRITES FROM G-PASCAL

The example program on the right demonstrates the use of various aspects of sprite handling within G-Pascal. It is a bit more elaborate than the example in the G-Pascal Manual on page 46, but is still reasonably simple.

Lines 3 to 15 are CONST declarations of various constants needed by the SPRITE, SOUND and GRAPHICS commands, as suggested in the G-Pascal Manual to make the program more readable.

Line 16 declares three variables for use within the program.

Lines 18 to 21 define the shape of the sprites. This definition was set up using the SUPERSPRITE EDITOR written by Craig Brookes (sold separately by Gambit Games). You could also edit the sprite using the SPRITE EDITOR supplied on later G-Pascal disks, and the G-Pascal Update Disk.

Line 22 clears the screen by writing the Commodore 'clear screen' character (decimal 147).

Lines 23 to 28 set up all eight sprites to have the same characteristics. Notice that they all share the same sprite shape (placed at sprite location 128) so only one DEFINESPRITE statement is necessary.

The 'sprite' statement can accept an indefinite number of arguments, provided they are given in groups of three, namely: 1] Sprite number; 2] What 'command number' to do to that sprite (e.g. EXPANDX) and 3] the 'argument' to that command (e.g. ON or OFF).

Lines 29 to 32 use the GRAPHICS statement to change screen colours, and set up the multi-coloured sprites second and third colours.

Lines 33 to 45 use the MOVESPRITE statement to automatically move the sprites across the screen in three 'waves' (run the program and you will see what we mean).

The arguments to 'movesprite' are: 1] sprite number; 2] initial X position; 3] initial Y position; 4] increment in X direction (-512 means two pixels per frame backwards); 5] increment in Y direction (0 means 'fly horizontally') and 6] number of frames to move.

Line 44 waits for 0.35 seconds before bringing in the next wave.

Lines 46 and 47 just wait for the sixth sprite to have stopped moving before allowing the program to end.

```
 1 (* sprite movement demo program.
 2    Written by Sue Gobbett.    *)
 3 const
 4      blue = 6; red = 2;
 5      black = 0;
 6      multicolour = 3;
 7      spritecolour0 = 16;
 8      spritecolour1 = 17;
 9      on = 1; off = 0;
10      expandx = 4;
11      point = 2;
12      active = 7;
13      background = 12;
14      delay = 3;
15      clearscreen = 147;
16 var i, j, spnum : integer ;
17 begin
18 definesprite (128,
19   $00050, $001f4, $007fc,
20   $01ffc, $1fff5, $01ffc,
21   $007fc, $001f4, $00050);
22 write (chr (clearscreen));
23 for i := 1 to 8 do
24   sprite (
25     i, point, 128,
26     i, multicolour, on,
27     i, expandx, on,
28     i, active, on);
29 graphics (
30   background, black,
31   spritecolour0, red,
32   spritecolour1, blue);
33 spnum := 0;
34 for i := 1 to 3 do
35   begin
36   for j := 1 to i do
37     begin
38     spnum := spnum + 1;
39     movesprite (spnum,
40       (350 - (spnum * 4)),
41       (150 - (i * 20) + (j * 40)),
42       -512, 0, 200)
43     end ;
44   sound (delay, 35)
45   end ;
46 repeat
47    until spritestatus (6)  = 0
48 end .
```

PROGRAMMING WITHOUT BUGS

Programming in Pascal can be a rewarding experience because it is generally possible to write programs that do not suffer from major, unexplainable bugs — unlike programming in unstructured languages like Basic.

There are a couple of reasons for this, one is the ability to use 'local' variables in procedures, thus isolating unexpected side-effects. Another is the absence of the 'GO TO' instruction which, when foolishly or carelessly used, can cause major debugging headaches.

For example, suppose you are writing an Adventure game and you want to write a procedure to describe the contents of a given room. If you declare the procedure as:

```
PROCEDURE describe_room (room_number);
```

and use local variables for any internal processing that is required within the procedure, then DESCRIBE_ROOM has got very clearly defined inputs and output — its input is the 'room_number' argument, and its output is a room description on the screen. Any other processing necessary is confined to local variables and cannot have side-effects in the rest of the program.

If you program like this then you can debug individual procedures simply (for example, by writing a test program that just calls that procedure with nominated arguments), and once they are debugged they can be forgotten.

Unlike Basic, you cannot 'accidentally' jump into the procedure, or jump out of it. The procedure itself cannot do anything except start at the start, and procede in an orderly fashion to the last statement in it (it can, of course, loop by using DO, REPEAT and WHILE statements, but these have a straightforward operation).

You are also free of the restrictions imposed by line numbers in Basic — as the G-Pascal editor allows you to insert an indefinite number of lines between two consecutive lines, you can code a 'skeleton' program initially, and easily 'flesh it out' later on.

For example, a skeleton adventure game might look like this initially:

```
CONST true = 1;
      false = 0;

VAR   room, game_over : integer;
      input_line : array [80] of char;

PROCEDURE describe_object (which_one);
BEGIN
END;

PROCEDURE describe_room (which_room);
BEGIN
END;

PROCEDURE get_line_from_player;
BEGIN
READ (input_line)
END;

PROCEDURE take_object (which_object);
BEGIN
END;

PROCEDURE drop_object (which_object);
BEGIN
END;

BEGIN   (* main program *)
room := 1;
game_over := false;
REPEAT
  get_line_from_player;
  describe_room (room)
UNTIL game_over
END.
```

Whilst this simple example won't play a very exciting game, it gives you the basis for building up the program in manageable steps, and can actually be compiled and tested even in its skeleton form.

A few useful debugging tips can make programming easier, too. One idea is to test a key on the keyboard (such as the 'Commodore' key) and if it is down, display debugging information which would otherwise not appear. To do this you could say:

```
IF MEMC [653] = 2 THEN
  WRITELN ("You are currently in room ",
           room);
```

Another way of handling debug information is to have a 'start debug' command. For example, if the adventure game player typed in 'START DEBUG' the program could start outputting debugging information. (I have noticed that entering 'T' to Scott Adams' adventures seems to have this effect).

In conclusion, Pascal allows you to write and debug programs with a minimum of frustration, thus making programming an enjoyable and rewarding activity.