G-PASCAL NEWS

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 1, NUMBER 2

NEWS

Welcome to the second edition of G-Pascal News. We hope that you are enjoying using G-Pascal and find some helpful hints in this newsletter. We hope to be able to answer some of the queries we have received in the mail recently.

G-Pascal is now being sold in Australia, New Zealand and Norway. We have also recently received an order from the Netherlands. This is a pleasing illustration that Australian software can be successful on the international market.

If there is something you like or dislike about this newsletter (or G-Pascal) please send us a note in the mail, so we know whether we are on the right track or not.

CONTENTS

CREDITS

Newsletter edited by Nick Gammon. Assistance with 'printing' programs from Steven Szczurko, Chris Brittain and Malcolm McMahon. Letters and queries from Don Colquhoun, J. Lockwood, Vladimir Vasylenko, Graham Whybrow, Mario Zolin, Tore Ostensen (Norway), Peter Macdonald, Ian Williams, Gavin Murray, Rohan Parker, P. Taylor, D. Smythe, D. Gates, Andrew Brown. Typeset on a Brother EM-100 typewriter. This page heading typeset by Hughes Phototype, 2 Spit Rd., Mosman NSW 2088. Printed by Print Mint, 18-20 Bank Place, Melbourne, Vic. Printed in January 1984.

CONTRIBUTIONS WELCOME

If you have a question about G-Pascal (or the Commodore 64), a program that you have developed that might interest other G-Pascal users, or something else interesting to say please write to us, so that we can publish it in the next issue.

GUESSING GAME

As we have been asked for more examples of how to program in G-Pascal we include a game written in G-Pascal (over the page). This is a 'no-frills' guessing game - the computer thinks of a 4-digit random number and you have to guess what the number is.

For each guess you will be told how many digits you got in the right spot (white), and how many were right but in the wrong spot (black).

For example, if the number is 1234 and you guess 1543 you will get 1 white (the '1') and 2 black (the '3' and '4').

For brevity's sake the game does not have any fancy options - you could easily modify it to stop after a pre-determined number of guesses (say 10) and say what the correct answer was.

You could also get it to actually display white or black shapes rather than just saying 'white' or 'black'.

By changing 'numdigits' (line 10) you can alter the number of digits to be guessed (from 1 to a lot). Also by changing 'largestdigit' (line 11) you can vary the range of numbers in each digit (from 1 to 9). For example, if 'largestdigit' is 5 then it will only generate numbers in the range 0 to 5.

```
 1 (* 'Guess the numbers' game
 2    Author: Nick Gammon
 3    for Commodore 64 G-Pascal *)
 4
 5 const
 6      true = 1;
 7      false = 0;
 8      cr = 13;
 9      home = 147;
10      numdigits = 4;
11      largestdigit = 9;
12
13 var ch : char ;
14     number : array [numdigits]
15              of char ;
16     number_black,
17     number_white,
18     guesses : integer ;
19
20 procedure initialize;
21 (********************)
22 const frequency = 1;
23       noise = 14;
24 begin
25   voice (3, noise, true,
26          3, frequency, 50000)
27 end ; (* initialize *)
28
29 procedure create_number;
30 (**********************)
31 var i : integer ;
32 begin
33   for i := 1 to numdigits do
34     number [i] := "0" +
35                   random mod
36                   (largestdigit + 1)
37 end ; (* create_number *)
38
39 procedure play;
40 (**************)
41 var user : array [numdigits]
42            of char ;
43     notbad,
44     i,
45     j : integer ;
46     used : array [numdigits]
47            of integer ;
48 begin
49   number_white := 0;
50   number_black := 0;
51   write ("Your guess? ");
52   read (user);
53   notbad := user [numdigits] = cr;
54   i := 0;
55   while notbad and (i < numdigits) do
56     begin
57     notbad := (user [i] >= "0") and
58               (user [i] <= "0" +
59                largestdigit);
60     i := i + 1
61     end ;
62   if not notbad then
63     begin
64     writeln ("Illegal input, try again");
65     play
66     end
67   else
68     begin
69     for i := 1 to numdigits do
70       begin
71       if number [i] = user [i - 1] then
72         begin
73         number_white := number_white + 1;
74         used [i] := true
75         end
76       else
77         used [i] := false
78       end ;
79     for i := 1 to numdigits do
80       begin
81       j := 1;
82       while j <= numdigits do
83         begin
84         if (user [i - 1] = number [j])
85         and (i <> j)
86         and not used [j] then
87           begin
88           number_black := number_black + 1;
89           used [j] := true;
90           j := numdigits
91           end ;
92         j := j + 1
93         end
94       end
95     end
96 end ; (* play *)
97
98 begin  (* main program *)
99   initialize;
100  repeat
101    write (chr (home));
102    create_number;
103    number_white := 0;
104    guesses := 0;
105    while number_white <> numdigits do
106      begin
107      play;
108      writeln (number_white, " white, ",
109               number_black, " black.");
110      guesses := guesses + 1
111      end ;
112    writeln ;
113    writeln ("Correct!");
114    writeln ("You took ",guesses,
115             " guesses.");
116    writeln ;
117    write ("Try again? ");
118    read (ch)
119  until ch <> "y"
120 end . (* main program *)
```

DETECTING END-OF-FILE

When reading data from disks, you need to be able to tell when you have reached the end of the disk file. To do this, include the following function in your program. When EOF (End Of File) is true, you have reached the end of the file.

```
function eof;
begin
  eof := memc [$90] and $40 <> 0
end ;
```

AUTO-REPEAT ON ALL KEYS

To make all keys on the Commodore 64 auto-repeat (if you hold them down), run the following program:

```
begin memc [650] := 128 end .
```

TESTING THE SHIFT KEY

If you want to see if someone is pressing the SHIFT key, just AND location 653 with 1. To see if someone is pressing the COMMODORE key, just AND location 653 with 2. e.g.

```
shiftkey := memc [653] and 1;
commodorekey := memc [653] and 2 <> 0;
```

For example, if you want to make your program pause until someone presses the SHIFT key (e.g. if you are displaying instructions) you would say:

```
writeln ("Press <SHIFT> key ...");
repeat until memc [653] and 1;
```

DEMO PROGRAM ON CASSETTE

Purchasers of G-Pascal on disk will have found that there is a 'demo' program and a 'sub hunt' game included free on the disk. These programs were not included on the cassette versions because of the extra time needed to dump them, and the difficulty of locating a file halfway through the cassette.

If you would like a copy of 'demo' and 'sub hunt' on cassette however, we will be happy to supply it, if you send us a suitable blank cassette, plus $2 for postage and packaging (remember to include your name and address).

PROGRAMS AVAILABLE DIRECT FROM GAMBIT GAMES

The Gambit Games products described in this newsletter (adventure game, sprite editor and so on) are now available directly from Gambit Games if you have difficulty purchasing them from your local dealer. If your dealer does not have them in stock, please send a cheque, money order or your Bankcard number to Gambit Games (P.O. Box 124, Ivanhoe, Victoria 3079, Australia) and we will be happy to supply you promptly. Prices include postage.

Adventure game: $29.50
Sound Editor: $25
Sprite Editor: $25
Runtime System: $39.50
Modem program: $20

PROGRAM FOR TRANSMITTING FILES

We have developed a program for transmitting files by modem. It uses the Christensen protocol and is compatible with YAM and MODEM7 and similar programs that run on CP/M systems. The program is directly compatible with the format used by the Mi Computer Club in Sydney.

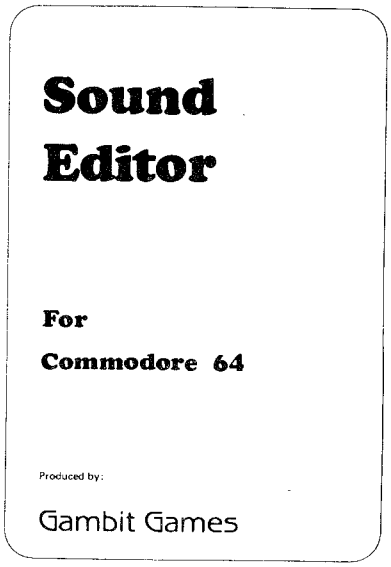
This program will be featured in an article in 'Your Computer' magazine in March 1984 (we expect).

The program allows files to be sent or received by Commodore 64 owners to each other or to any other computer that uses the Christensen protocol (such as Mi Computer Club). It features full error checking so that file transfer is very reliable, even if there are temporary line problems with the phone. It transfers 'prg' files from disk or cassette, which includes G-Pascal files and Basic files.

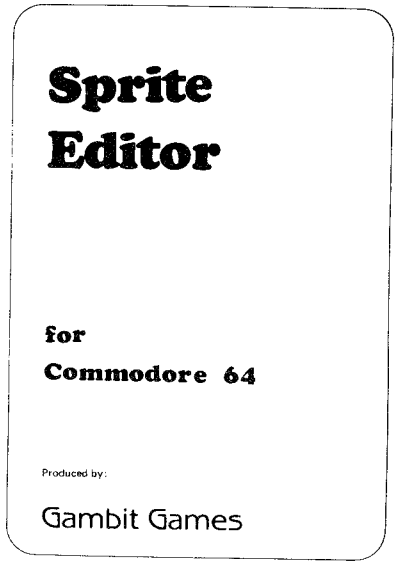
It also features full and half-duplex terminal mode for accessing Bulletin Boards, the Source and so on.

The program is 956 lines long, and demonstrates how to load and save files with 'variable' names (i.e. as typed in by the user when the program is running). It also demonstrates how to read the 'error channel' from the disk drive. For a copy of the program on disk or cassette send us $20.

To use the program you need an RS232 interface (VIC-1101A), a modem and a cable connecting them (and a phone!).

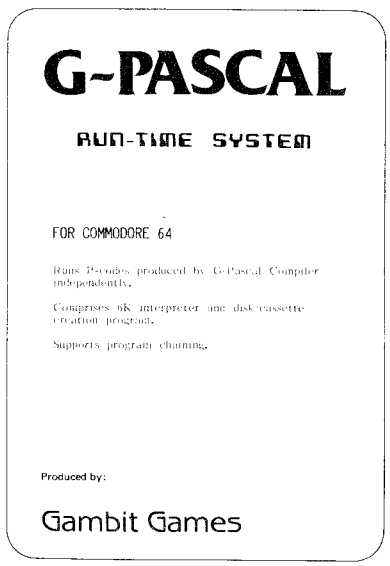


Experiment with different sound effects using the Sound Editor! Play three voices at once, change filtering, ADSR envelope, voice type, frequencies, sync and ring modulation. Full G-Pascal source code supplied. $25.00 recommended retail. Disk or cassette.



Create sprite shapes and save to disk for re-editing later! Large grid makes changing sprites easy. Uses joystick or keyboard. Outputs DEFINESPRITE statements to disk or cassette for direct inclusion in G-Pascal programs. Full G-Pascal source code supplied. $25.00 recommended retail. Disk or cassette.

Play this 784 line adventure game for fun, or modify it to your own ideas! Currently implements 21 rooms, TAKE, DROP, INVENTORY, SCORE, INSPECT, WAVE, EAT, QUIT, LOOK as well as movement verbs (north, south etc.). Includes full source listing, description of how program works and hints for modifying it. Full G-Pascal source code supplied. $29.50 recommended retail. Disk or cassette.



Write your own programs for sale! The Runtime System will combine your P-codes with the G-Pascal interpreter to create a single auto-load program. $39.50 recommended retail. Disk only.

All of these products are available from your local dealer or direct from Gambit Games. To purchase direct send cheque, money order or Bankcard number to Gambit Games.

PRINTING YOUR PROGRAMS

We have had various requests from users to explain how to print their G-Pascal programs using different types of printers and connection techniques. It is difficult when developing software to cater for all possible types of printers, especially on the Commodore 64 where a printer can be connected by serial bus, RS232 serial or parallel (Centronics). The last two attach via the user port using special interfaces. Also, different printers need different control codes to do lower case, graphics and so on.

In order to solve some of these problems we present here a few different ways of getting your printer to work with G-Pascal, starting with a simple MEMC to change the way G-Pascal opens the printer file, to a complete program that will read in a G-Pascal file and print it.

Changing the secondary address

Some printers apparently need to be opened with a secondary address of 7 in order for them to print in upper and lower case. If your printer is working properly apart from the fact that it is not printing upper and lower case properly just try running the following program before using the printer:

```
begin  memc [$9E5D] := 7  end.
```

Note: this patch applies to G-Pascal versions 3.0 and 3.1 only.

Fixing strange behaviour

When G-Pascal displays its 'Main Menu' it includes two 'control' characters which may adversely affect your printer if you are in print mode. These are:

$8CD9: 5 (White letters)
$8CDA: 14 (Switch to lower case)

You may want to try changing the contents of these locations to zero if you are about to print a program. If you do this, however, you will find that the compiler itself no longer displays in lower case, or with white letters (on the screen).

Note: these patches apply to G-Pascal versions 3.0 and 3.1 only.

Writing a printer setup program

The next simplest solution to printer problems is to write a small 'setup' program that conditions the printer. In this case you would just run the setup program prior to using the printer. The following program was submitted by Steven Szczurko for initializing a 1526 printer:

```
 1 (* program Printer_init;
 2    Setup program for Commodore 1526 printer
 3 *)
 4 const printer_channel = 4;
 5       home = 147;
 6 begin
 7 (* put printer into upper/lower case mode *)
 8 open (7, printer_channel, 7, " ");
 9 put (7);
10 close (7);
11
12 (* set printer paging on *)
13 open (4, printer_channel, 0, " ");
14 put (4);
15 write (chr (home));
16 put (0);
17 close (4);
18 writeln (chr (home),"Printer initialized.")
19 end.
```

Using a stand-alone print program

The ultimate solution to catering for all printers is the following stand-alone print program. This program reads a G-Pascal program from disk or cassette and prints it. It has provision for expanding multiple space codes and reserved words. It also prints the line number at the start of each line. You could easily change it to also start a new page after every 55 (or so) lines. Another useful enhancement would be to print the name of the file and the page number at the top of each page.

In its current form the program outputs data to a 'parallel' (Centronics) type printer, however it can easily be adapted to any other sort of printer. If you have a serial bus printer then the procedure 'INIT_PRINTER' (lines 114 to 117 would just consist of an OPEN statement, e.g.:

```
OPEN (2,4,0," ");
```

If you have an RS232 interface then INIT_PRINTER would consist of a Kernal Open similar to 'OPEN_RS232_FILE' from the program on the back of the previous edition of G-Pascal News.

In both of these cases the PRINT_CHAR routine in the program would be simplified, as lines 181 to 187 would just be:

```
PUT (2); WRITE (CHR (X)); PUT (0);
```

Also, WRAP_UP would just consist of:

```
CLOSE (2);
```

If you have a parallel interface you may be able to omit lines 115 to 117, and lines 183 to 185, depending on which signal line the printer uses to tell it that data is present.

You may also be able to omit lines 175 to 180 depending on whether upper and lower case prints correctly or not.

If you have a cassette player rather than a disk drive change line 14 to read:

```
medium = cassette;
```

Testing the program

Once you have typed in the program, save it to disk or cassette before testing it, just in case something drastic goes wrong.

Then run the program, and initially answer 'Y' when it asks: 'Reprint same file?' The program will just attempt to print itself. Then answer 'Y' when it asks: 'Output to screen only?' This will just display the program on the TV screen. If all is well, run the program again and answer 'N' to: 'Output to screen only?' and it should print on your printer. If not, change the procedures INIT_PRINTER and PRINT_CHAR until everything prints properly.

Then save the program to disk or cassette again, and answer 'N' when it asks: 'Reprint same file?' It will then ask for the name of the file that it is to print. Enter the file name, and you will now get a listing of your program. Once the program is printed you will find it in memory ready for editing. Alternatively, (R)un the program again and you can print another file.

```
 1 (* program to print a g-pascal file
 2    via the parallel port
 3
 4 Author: Nick Gammon of Gambit Games
 5
 6 Public Domain.
 7
 8
 9 %a $800  *)
10
11 const
12       disk = 8;
13       cassette = 1;
14       medium = disk;
15
16       strobereg = $dd00;  (* strobe register *)
17       ddra = $dd02; (* data direction registers *)
18       ddrb = $dd03;
19       start_address = $4000;
20       true = 1;
21       false = 0;
22       cr = 13;
23
24 var reprint,
25     screen_only : integer ;
26
27 procedure init;
28 (*************)
29 const home = 147;
30
31 procedure load_nominated_file;
32 (****************************)
33
34 var
35     i,
36     got_cr,
37     error,
38     length : integer ;
39     name1, name2 : array [20]
40                    of char ;
41
42 procedure get_file_name;
43 (***********************)
44 begin
45 repeat
46   writeln ;
47   write ("File name? ");
48   read (name1);
49   got_cr := false;
50   for i := 0 to 20 do
51   if not got_cr then
52     begin
53     name2 [20 - i] := name1 [i];
54     if name1 [i] = cr then
55       begin
56       length := i;
57       got_cr := true
58       end
59     end
60 until length <> 0
```

```
 61 end ;
 62
 63 procedure load_file;
 64 (******************)
 65 const
 66      areg = $2b2;
 67      xreg = $2b3;
 68      yreg = $2b4;
 69      cc = $2b1;
 70      loadit = $ffd5;
 71      setlfs = $ffba;
 72      setnam = $ffbd;
 73      readst = $ffb7;
 74
 75 begin
 76   memc [areg] := 1;
 77   memc [xreg] := medium;
 78   memc [yreg] := 0; (* relocate *)
 79   call (setlfs);
 80   memc [areg] := length;
 81   memc [xreg] := address (name2[20]);
 82   memc [yreg] := address (name2[20]) shr 8;
 83   call (setnam);
 84   memc [areg] := 0; (* load *)
 85   memc [xreg] := start_address;
 86   memc [yreg] := start_address shr 8;
 87   call (loadit);
 88   if memc [cc] and 1 then
 89     error := memc [areg]  (* got error *)
 90   else
 91     begin
 92     call (readst);
 93     error := memc [areg] and $bf
 94     end ;
 95   writeln ; writeln ;
 96   if error then
 97     writeln ("Load error, code: ",
 98            error)
 99   else
100     writeln ("Loaded ok.")
101 end ;
102
103 (***** start of : load_nominated_file ***)
104 begin
105 repeat
106   get_file_name;
107   load_file
108 until not error
109 end ;
110
111 procedure init_printer;
112 (**********************)
113 begin
114   memc [ddrb] := $ff; (* output *)
115   memc [strobereg] :=
116     memc [strobereg] or 4;  (* no data *)
117   memc [ddra] := memc [ddra] or 4
118 end ;
119
120 function yes_no;
121 (***************)
122 var reply : char ;
123 begin
124   repeat
125     read (reply)
126   until (reply = "y")
127      or (reply = "n");
128   writeln (chr (reply));
129   writeln ;
130   yes_no := reply = "y"
131 end ;
132
133 (****** start: init ***********)
134
135 begin
136   writeln (chr (home),
137     "Centronics G-Pascal File Print");
138   writeln ;
139   write ("Reprint same file? <Y>es/<N>o ... ");
140   reprint := yes_no;
141   if not reprint then
142     load_nominated_file;
143   writeln ;
144   write ("Output to screen only? <Y>es/<N>o ... ");
145   screen_only := yes_no;
146   if not screen_only then
147     init_printer
148 end ;
149
150 procedure print_file;
151 (********************)
152 var
153    line,
154    pointer,
155    limit,
156    count : integer ;
157    ch : char ;
158
159 procedure next_char;
160 (******************)
161 begin
162   ch := memc [pointer];
163   pointer := pointer + 1
164 end ;
165
166 procedure print_char (x);
167 (***********************)
168 const
169      flag = $dd0d;
170      datareg = $dd01;
171 begin
172   write (chr (x));  (* echo on screen *)
173   if not screen_only then
174     begin
175     (* convert x to ASCII *)
176     if x >= 192 then
177       x := x and $7f
178     else
179       if (x > 64) and ((x and $5f) < 91) then
180         x := x xor $20;  (* swap upper/lower case *)
```

```
181     (* send data to printer *)
182     memc [datareg] := x;
183     (* tell printer data there *)
184     memc [strobereg] := memc [strobereg] and $fb;
185     memc [strobereg] := memc [strobereg] or 4;
186     (* wait until acknowledge *)
187     repeat until memc [flag] and $10
188     end
189 end ;
190
191 procedure print_reserved_word (x);
192 (*******************************)
193 const table = $81b5;
194 var position,
195     length
196     : integer ;
197
198 begin
199 position := table;
200 while (memc [position + 1] <> x)
201   and (memc [position] <> 0) do
202   position := position +
203               memc [position] + 2;
204 if memc [position + 1] <> x then
205   print_char (x)
206 else
207   begin
208   length := memc [position];
209   repeat
210     print_char (memc [position + 2]);
211     position := position + 1;
212     length := length - 1
213   until length <= 0;
214   print_char (" ")
215   end
216 end ;
217
218 procedure print_line;
219 (********************)
220 var i,
221     leading_zero : integer ;
222
223 procedure print_power (which);
224 (****************************)
225 begin
226   if (i / which > 0)
227   or not leading_zero then
228     begin
229     leading_zero := false;
230     print_char (i / which + "0")
231     end
232   else
233     print_char (" ");
234   i := i - line / which * which
235 end ;
236
237 (****** start: print_line ******)
238 begin
239   line := line + 1;
240   i := line;
241   leading_zero := true;
242   print_power (1000);
243   print_power (100);
244   print_power (10);
245   print_char (line mod 10 + "0");
246   print_char (":");
247   print_char (" ")
248 end ;
249
250 (******* start: print_file *******)
251 begin
252   pointer := start_address;
253   line := 0;
254   print_line;
255   next_char;  (* get first character *)
256   if ch <> 0 then  (* not blank file *)
257   repeat
258     if ch = $10 then (* space count *)
259       begin
260       next_char;
261       limit := ch and $7f;
262       for count := 1 to limit do
263         print_char (" ")
264       end
265     else
266       if ch > $80 then
267         print_reserved_word (ch)
268       else
269         print_char (ch);
270     if ch = cr then
271       print_line;
272     next_char  (* next character if any *)
273 until ch = 0
274 end ;
275
276 procedure wrap_up;
277 (****************)
278 begin
279   memc [ddrb] := 0
280 end ;
281
282 (****** program starts here ******)
283 begin
284   init;
285   print_file;
286   wrap_up
287 end .
```

ENTERING UNDERSCORES

The 'underscore' character in the above listing (shown as '_') is entered by typing the 'left-arrow' key on the Commodore 64. This is on the top left hand side of the keyboard (above the CTRL key).

REAL NUMBERS

We have had a number of queries about the absence of REAL (floating-point) numbers in G-Pascal. The reasons are as follows ...

A design criteria of G-Pascal was that the compiler should fit into 16K of memory, so that is could eventually be placed into a plug-in cartridge. It presently uses all but a few bytes of that 16K, leaving very little room for any additional features.

In fact, a lot of effort has gone into packing as much into G-Pascal as is presently there - for example the full English error messages, helpful menu-driven operation, and HELP facility in the Editor, are all accomplished by tokenizing all the words used in messages.

The Commodore 64 has a lot of powerful features - sprites, SID chip, hardware clock and timer, bitmapped graphics and so on. We wanted to support all of these features so that G-Pascal truly was a useful programming tool. Each feature that is supported takes room, leaving less room for 'standard' Pascal features, such as REAL numbers and TYPE declarations.

You will find that most other Pascal compilers on microcomputers provide support for REAL numbers and so on at the expense of being very slow to use. Most compilers are disk based, meaning that during the compilation process parts of the compiler are read into memory from the disk. Also these compilers usually read the program to be compiled from disk as they go. The overall effect of all of this is that such compilers compile at (say) 200 lines per minute, not 6,000 lines per minute as G-Pascal does.

It quickly becomes very frustrating attempting to get rid of syntax errors and debug programs if you have to wait 15 minutes for your program to compile each time. We firmly believe that G-Pascal users would be happy to forgo some of Pascal's more esoteric features, in return for a fast, easy to use system. Also a memory-resident compiler is feasible to supply on cassette, making it available to a wider range of users.

By providing 3-byte integers in G-Pascal it is possible to obtain a reasonable degree of precision in arithmetic operations (almost 7 significant digits). This is ample for many applications.

WRITING TEXT IN BITMAP MODE

If you are writing a program that displays in bitmap mode (high-resolution graphics) you may find the need to put text on the screen as well. For example, you may be drawing a graph and want to label the axes, or you may be doing a game that uses bitmapped graphics and need to display the game score or other textual information.

The easiest way to accomplish this is to use the technique presented in the following program. The program that follows is a demonstration of mixing text and bitmapped graphics - in your program you merely need to include the procedure INIT_BITMAP_WRITING and call it once at the start of the program.

INIT_BITMAP_WRITING places a machine-code routine at address $1F00 and links it into the Kernal output routines (this is commonly called a 'wedge'). From then on any WRITE or WRITELN statements will function normally if the computer is in 'normal' display mode, but in 'bitmap' mode the machine-code routine will map the appropriate character shapes onto the bitmap screen at the current cursor position.

The only control character supported is 'carriage return' - others will be ignored. Key in the program, taking care with the machine-code constants. Then save it to disk or cassette before testing it! Once the program tests OK, you can use the INIT_BITMAP_WRITING procedure in your other programs.

The routine displays characters in upper and lower case - it is not really designed to display 'graphics' characters but may work with some of them. If there is enough interest shown by readers we will publish the assembler listing of the machine code portion next time.

```
 1 (* bitmap mode writing program
 2    demonstration *)
 3
 4 (* Author: Nick Gammon.
 5    Uses $1f00 to $1fed
 6    for machine code subroutine *)
 7
 8 const bitmap = 1;
 9       chargenbase = 8;
10       black = 0;
11       yellow = 7;
12       on = 1;
13       home = 147;
14 var x : integer ;
15
16 procedure init_bitmap_writing;
17 (***************************)
```

```
18 var m : integer ;
19
20 procedure x(i,j,k);
21 (*---------------*)
22 begin
23 mem [m] := i;
24 mem [m + 3] := j;
25 mem [m + 6] := k;
26 m := m + 9
27 end ;
28
29 begin
30 m := $1F02;
31 x($0327AD,$F01FC9,$018D13);
32 x($26AD1F,$008D03,$1DA91F);
33 x($03268D,$8D1FA9,$600327);
34 x($11AD48,$2029D0,$6804D0);
35 x($1F006C,$208568,$488A48);
36 x($384898,$FFF020,$9028C0);
37 x($389805,$A828E9,$841E86);
38 x($20A51F,$380310,$C960E9);
39 x($74F00D,$B060C9,$40C90B);
40 x($380A90,$D040E9,$03F005);
41 x($20E938,$782085,$DC0EAD);
42 x($8DFE29,$A5DC0E,$FB2901);
43 x($A90185,$5F8500,$0A20A5);
44 x($0A5F26,$0A5F26,$855F26);
45 x($5FA55E,$85D809,$00A95F);
46 x($A54B85,$664A1E,$664A4B);
47 x($65184B,$4C851E,$0A1FA5);
48 x($080A0A,$4B6518,$A54B85);
49 x($00694C,$284C85,$6520A9);
50 x($4C854C,$B107A0,$4B915E);
51 x($F91088,$A51FE6,$28C91F);
52 x($A90690,$1F8500,$A51EE6);
53 x($19C91E,$A90490,$1E8500);
54 x($0901A5,$018504,$DC0EAD);
55 x($8D0109,$58DC0E,$1FA418);
56 x($201EA6,$68FFF0,$AA68A8);
57 x($006068,0,0);
58 call ($1F02)   (* set up vector *)
59 end ;
60
61 (******** main program *********)
62
63 begin
64 init_bitmap_writing;
65 writeln (chr (home),
66 "Testing ... this is written in");
67 writeln  ("normal (not bitmap) mode");
68 writeln ;
69 writeln ("Press a key for next part ...");
70 repeat until getkey ;
71 graphics (bitmap, on,
72           chargenbase, 4);
73 clear (yellow, black);
74 for x := 1 to 190 do
75   plot (on, x, x);
76 cursor (10, 10);
77 writeln ("Hi there - this is written");
78 writeln ("in bitmap mode.")
79 end .
```

ENTERING SEMICOLONS

Some readers are mystified about when it is necessary to use a semicolon in Pascal programs. We hope to be able to clear this up now ...

Semicolons are used as statement **separators** - in other words, when two statements appear in sequence they are separated by a semicolon. Semicolons are not needed at the **end** of a statement (unless another statement follows).

Semicolons are also used:

1) Following the final END in a procedure or function.

2) Following a CONST declaration (e.g. cr = 13;).

3) Following the data type in a VAR declaration.

4) As part of a procedure or function declaration (e.g. procedure fred;).

Here is a simple example which illustrates the use of semicolons:

```
 1 var a, b, c : integer;
 2 begin
 3   a := 1;
 4   b := 2;
 5   if a = 1 then
 6     begin
 7     c := b * 2;
 8     b := 4
 9     end;
10   a := 5
11 end.
```

There is no semicolon on line 2 because 'begin' is not a statement. Lines 3 and 4 are followed by statements and so end with semicolons. Line 5 does not end with a semicolon because the word 'then' is not a statement itself, but is to be followed by a statement, in this case the compound statement at lines 6 to 9.

Line 8 has no semicolon because it is followed by the word 'end' which is not a statement - the same applies to line 10.

In some cases an extra semicolon does no harm - for example, if line 8 had a semicolon at the end it would not change the execution of the program. In this case it would create a 'null' statement at the end of line 8 which would not have any effect.

However if the 'then' in line 5 was followed by a semicolon then that would terminate the 'if' statement (so that the only thing executed conditionally would be the null statement between the 'then' and the semicolon). In this case the extra semicolon would change the execution of the program from what would be intended.

Perusal of the programs in the issue will help appreciation of where semicolons should be used.