TABLE OF CONTENTS

Special keyboard functions 5
Getting started with G-Pascal 6
How to use G-Pascal 9
Using the Editor 10
Editor commands 11
Beginner's Guide to Pascal 15
  Programs and blocks 16
  Procedures and functions 17
  Statements 20
  Expressions 25
Notes on the Compiler 28
  MEM and MEMC 30
  LOAD and SAVE 30
  GETKEY and ABS 31
Sample program (Prime numbers) 32
Compiling your program 33
Compiler Directives 33
Compiler error messages 35
Run-time error messages 39
File Handling 40
The GRAPHICS command 43
Sprite processing overview 46
The SPRITE command 47
General sprite commands 48
  DEFINESPRITE 48
  POSITIONSPRITE and MOVESPRITE 49
  ANIMATESPRITE and SPRITESTATUS 50
  SPRITECOLLIDE 50
  GROUNDCOLLIDE 51
  STOPSPRITE and STARTSPRITE 52
  SPRITEX and SPRITEY 52
  SPRITEFREEZE and FREEZESTATUS 53
  Speeding up sprites 54
Miscellaneous graphics commands 55
  WAIT 55
  PLOT and CLEAR 56
  SCROLL, SCROLLX and SCROLLY 58
  CURSOR, SETCLOCK and CLOCK 59
  PADDLE and JOYSTICK 60
G-Pascal Sound Effects 61
The SOUND command 61
The VOICE command 63
Sound effects functions 66
  RANDOM and ENVELOPE 66
Independent modules 67
How text is stored by G-Pascal 71
Idiosyncrasies of tokenization 72
Converting from other Pascals 73
Debugging 74
  Trace and Debug mode 75
Memory Map 77
Machine language subroutines 78
Meanings of P-codes 79

COPYRIGHT

NOTICE

Gambit Games reserves the right to make improvements in the product described in this manual at any time and without notice.

G-Pascal is designed, written, manufactured and supported in Australia by Gammon & Gobbett Computer Services Proprietary Limited trading as Gambit Games.

Gammon & Gobbett Computer Services Proprietary Limited is a company incorporated in the State of Victoria.

This Manual was typeset by Hughes Phototype, Cremorne, NSW, and printed by Noosa Graphica, Noosaville, Qld.

DISCLAIMER

G-Pascal is sold or licensed "as is". The entire risk as to its quality and performance is with the buyer. Should G-Pascal prove defective following its purchase the buyer (and not Gambit Games, its distributor, or its retailer) assumes the entire cost of any necessary correction and any incidental or consequential damages. Gambit Games believe G-Pascal and this Manual are accurate and reliable and much care has been taken in their preparation, however Gambit Games make no warranties, either express or implied, with respect to this manual or with respect to the G-Pascal compiler, its quality, performance, merchantability, or fitness for any particular purpose.

REPLACEMENT POLICY

Gambit Games warrants to the original purchaser only the medium on which G-Pascal is recorded to be free from defects in materials or workmanship under normal use and service for a period of ninety (90) days from the date of purchase. If during this period a defect on the medium should occur, the medium may be returned to Gambit Games or to an authorised Gambit Games dealer, and Gambit Games will replace the medium without charge to you. Your sole and exclusive remedy in the event of a defect is expressly limited to replacement of the medium as provided above.

To provide proof that you are the original purchaser please complete and mail the enclosed Owner Warranty Card to Gambit Games.

If the failure of the medium, in the judgement of Gambit Games, resulted from accident, abuse or misapplication of the medium, then Gambit Games shall have no responsibility to replace the medium under the terms of this warranty.

POSTAL ADDRESS

Please address correspondence to:
Gambit Games,
P.O. Box 124,
Ivanhoe,
Victoria 3079.
Australia.

INTRODUCTION

Congratulations on purchasing G-Pascal! We are sure that you will find it very useful and easy to use. G-Pascal on the Commodore 64 contains very advanced features making it an excellent development tool for this computer, namely:

- High speed compiler (6,000 lines per minute) which implements a comprehensive subset of standard Pascal.
- Built-in powerful Text Editor which includes global Find and Replace capability.
- *Extensions provide extensive support for the Commodore 64's graphics, music and sound effects, time-of-day clock, interval timer, cursor control and colour control.* There are in fact 76 separate functions and actions built-in as extensions for the Commodore 64. Sprite handling is particulary well catered for, with commands to automatically move sprites around the screen with animation if desired, and to automatically stop sprites if they collide.
- Complete arcade-style games can be written without using a single PEEK or POKE.
- Error messages are in plain English with an arrow to the point of error.
- As the entire system is memory-resident, editing, compiling and testing is very fast and easy.
- Pascal programs may readily be stored on disk or cassette – programs are stored in a 'compressed' format, thereby speeding up loading and saving times, and allowing a larger program to fit into memory.
- Support for machine-code (assembler) subroutines if desired.
- Debugging aids, such as Trace and Debug modes, which can be invoked at any time from the keyboard.
- Compiler supports INTEGER and CHAR data types, and single- dimension arrays. Integers range from –8388608 to +8388607.
- Compiler supports the standard Pascal constructs: CONST, VAR, PROCEDURE, FUNCTION, WHILE, DO, REPEAT, FOR, IF and CASE.
- Arithmetic expressions may contain the relational operators as well as +, –, /, *, MOD, AND, OR, XOR, SHL, SHR and ABS.
- Compiler supports independent modules (Procedures that are compiled independently and located elsewhere in memory).
- Compiler produces relocatable P-codes – the object code produced by the compiler may be run at any memory address without change.

Run-time package

If you are planning to develop commercial programs or games in G- Pascal please enquire about our interpreter-only Run-time package. The Run-time system consists of the interpreter as a stand-alone program which is 6K long. Using the Run-time system simply consists of loading the P-codes produced by this compiler and then running the interpreter.

This Manual

This Manual has been designed with a number of purposes in mind:

- To introduce the Pascal language.
- To fully describe G-Pascal's capabilities and be a 'reference manual' for the experienced programmer.
- To describe the G-Pascal support system, such as the Editor, File system, error messages and so on, so that the G-Pascal system is easy to understand and use.

It falls beyond the scope of this Manual to provide a really comprehensive guide to the Pascal language. There are many good books on the Pascal language in bookshops and libraries, containing between them thousands of pages of information and examples about Pascal, good programming practices, games and serious applications. You are strongly recommended to find a Pascal book which covers the type of programs that you are interested in (games, mathematics, business, adventure and so on).

G-Pascal and this Manual are somewhat oriented towards arcade- style games. The reason for this is that the Commodore 64 with its sprites and 3-voice synthesizer is an excellent and low-cost method of easily writing and experimenting with arcade-style games. Consequently the discussions and examples (particularly about sprites) seem the most meaningful in this context.

However there is no reason why G-Pascal cannot be used for 'text' oriented programs, such as adventure games or more serious applications, for example: small-scale business software, mathematics or probability experiments, and educational software.

Suggestions

If you have just purchased G-Pascal and don't know where to start, these suggestions may be helpful:

- First, load G-Pascal into your Commodore 64 so that you can try out the examples for yourself, and confirm that G-Pascal does what this Manual says it will.
- Turn to 'Getting Started With G-Pascal' (over the page) and follow the step-by-step instructions to enter, compile and run your first program.
- While your first program is still in memory start experimenting! Try different Editor commands – try saving the program to disk or cassette and re-loading it – try tracing the program – try changing the program.
- G-Pascal is designed to be 'fail-safe'. Built-in checks make it very hard to do something 'wrong'. For example, it will not allow a program to be run before it has been compiled. The very worst that can happen is that a program wipes out itself or G-Pascal (possibly making it necessary to turn off the power and re-load G-Pascal) however this is extremely rare.
- If you are a beginner to Pascal, or want to know in detail about what G-Pascal will do, read the section 'Beginner's Guide to Pascal'.
- Skim through the Manual, becoming familiar with what G- Pascal can do. Try out the examples – most of them are quite small and only take a couple of minutes to type in.
- Experiment in more detail with the aspects of G-Pascal (and the Commodore 64) that you are interested in (e.g. graphics, sound effects, games, etc.)

HOW TO LOAD AND RUN G-PASCAL

1. Load and run G-Pascal as directed by the instructions that came with your disk, cassette or cartridge.
2. Your G-Pascal is now running! You will see:

```
G-Pascal compiler Version 3.0 Ser 4321
Written by Nick Gammon and Sue Gobbett
Copyright 1982 Gambit Games
P.O. Box 124 Ivanhoe 3079 Vic Australia

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

Now see the section 'Getting started with G-Pascal' which is a step-by-step guide to entering your first program.

SPECIAL KEYBOARD FUNCTIONS

The following keys have special meaning to G-Pascal.

Space bar

Pressing 'space' will freeze a display on the screen. Press any key (except RUN/STOP) and the display will continue. RUN/STOP will abort the display.

RUN/STOP

This will abort a display, or if a program is running, will abort the program with a message showing which P-code was currently being executed.

RUN/STOP/RESTORE

Holding down the RUN/STOP key and then pressing RESTORE will do a 'warm boot' aborting whatever G-Pascal is doing, clearing the screen, and returning to the Main Menu. Any program in memory is not lost, however. Normally just pressing RUN/STOP (without RESTORE) will be sufficient to stop a program running, however press RESTORE as well if nothing seems to be happening.

COMMODORE/T

If a program is running will cause a Trace to start. (See 'Debugging your program' for more details). *To enter COMMODORE/T hold down the key with the Commodore logo (bottom left hand corner of the keyboard) and then press the 'T' key.*

COMMODORE/D

If a program is running will cause Debug mode to start. (See 'Debugging your program' for more details). *To enter COMMODORE/D hold down the key with the Commodore logo (bottom left hand corner of the keyboard) and then press the 'D' key.*

COMMODORE/N

If a program is running will stop a Trace or Debug. (N = Normal). If the program is not Tracing or Debugging pressing COMMODORE/N will have no effect. *To enter COMMODORE/N hold down the key with the Commodore logo (bottom left hand corner of the keyboard) and then press the 'N' key.*

Arrow keys

The arrow keys (and INST/DEL key) function the same as in Basic – useful when in the Editor or when typing in a file name to Load or Save.

GETTING STARTED WITH G-PASCAL

This section provides a step-by-step introduction to using the G- Pascal system. We will key in, compile and run a simple program. Please work through this example using your Commodore 64 – it will give you familiarity with the various phases of the system (editing, compiling and running).

To distinguish between what you enter and what G-Pascal responds, all input to be entered by you is in ***bold italics.*** All Editor commands are terminated by pressing the RETURN key.

After loading G-Pascal you will see the Main Menu – enter 'E' to invoke the Editor ...

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

The Editor prompts with a colon (:). Type L (for List) to confirm that there is no program currently in memory ...

```
:L
```

Now enter your program by placing the Editor in input mode. To do this, type I (for Input) ...

```
:I
```

The Editor will now prompt you with a '1' – indicating that you are about to enter line number 1 of your program. Type in the program as shown, pressing RETURN at the end of each line. If you make a typing mistake while entering a line, you can use the INST/DEL or cursor movement keys (arrow keys) to correct it ...

```
1 (* program to display the squares
2 of numbers from 1 to 10 *)
3 const limit = 10;
4 var k : integer;
5 begin
6 for k := 1 to limit do
7 writeln ("square of ",k," is ",k * k)
8 end.
9
```

Just press RETURN on its own here and the Editor will leave Input mode, and display a colon again, indicating it is ready for another command.

```
:
```

Now list your program by typing L again. You should now see a listing of your program which looks identical to the program above ...

```
:L
```

If you notice a mistake at this stage the easiest way to correct it is to use the Editor's Modify command. For example, if you misspelt 'begin' on line 5 as 'began', then you would type in 'M5' to modify line 5. The Editor will echo the current contents of line 5, and then enter Input mode to allow you to type one or more lines to replace it. The whole operation would look like this ...

```
:M 5
5 began
5 begin
6
```

Rather than retyping the line you may wish to just move the cursor over the line in error (using the arrow keys), correct the line and then press RETURN.

Just press RETURN here to leave Input mode again. You can now attempt to compile it by entering C (for Compile). Compiling the program converts it to object code (P-codes) ready for Running.

If you have made an error in typing in your program you will get an error message, and be returned to the Editor, with the colon (:) displayed again. If this happens, Modify the offending line and try again.

If you have not made any errors the whole process will look like this ...

```
:C

G-Pascal compiler Version 3.0 Ser# 4321

Written by Nick Gammon and Sue Gobbett
Copyright 1983 Gambit Games
P.O. Box 124 Ivanhoe 3079 Vic Australia

P-codes ended at 412E
Symbol table ended at C026

(C)ompile finished: no Errors

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

At this stage the Main Menu will be displayed again – now enter R (for Run) and you will see the results being displayed on the screen ...

```
(E)dit, (C)ompile, (D)ebug, (F)iles,
(R)un, (S)yntax, (T)race, (Q)uit ? R
Running
square of  1 is   1
square of  2 is   4
square of  3 is   9
square of  4 is  16
square of  5 is  25
square of  6 is  36
square of  7 is  49
square of  8 is  64
square of  9 is  81
square of 10 is 100

run finished – press a key ...
```

G-Pascal now displays its 'end of run' message, as above. This is to give you a chance to read the results. When you press a key (such as RETURN), the screen will clear and you will see the Main Menu again ...

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

You can now enter the Editor again and try something else! You may want to experiment with minor changes to this demonstration program to get used to the Editor and the Compiler. This would be a good time to read the part of this Manual pertaining to the Editor and experiment with the other Editor commands, such as Find and Replace. When you are confident about the Editor try some of the other examples in the book, such as those demonstrating the various graphics and sound effects capabilities of G-Pascal. If you want to delete the program currently in memory so that you can 'start from scratch', just enter: 'D 1-9999' as in the following illustration ...

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

:D 1-9999

Do you want to delete 8 lines ? Y/N Y

8 deleted

:L

:
```

For a summary of Editor commands, just enter 'H' (Help).

HOW TO USE G-PASCAL

After loading and running G-Pascal or after a compile or run, you will see the 'Main Menu' :

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

It is called a menu because you have various choices to make, depending on what you want.

To choose one, press the letter corresponding to your choice (the letter in brackets). For example press C for Compile. Do not press the RETURN key as well. If you make an incorrect choice the Commodore 64 will re-display the Main Menu.

The choices are :

Edit

Enters the Editor to allow you to type in or change a Pascal program. When the Editor is active it will display a colon (:) to let you know it is awaiting an Editor command. See 'Using the Editor' for more details.

Compile

Compiles a program (that is, converts it to P-codes) that you have loaded or edited. If the compile has no errors you may then type R (for Run) if you want to run your program. A program must be compiled before it can be run. See 'Compiling your program' for more details.

Syntax

This does a 'syntax check' of your program, letting you know if it has any errors. It does not generate P-codes however so you must do a compile as well before running the program. See 'Compiling your program' for more details.

Run

This will run your program. It must have been successfully compiled first or the message 'No valid compile done before' will be displayed. G-Pascal will display:

Running

and then commence running your program.

Debug and Trace

These are a variation of the 'Run' command which, in addition to running your program, display debugging information as your program runs. You will not normally choose these options as the information they display clutters up the screen somewhat and slows down execution of the program considerably. See 'Debugging your program' for more details.

Files

Enters the 'Files Menu' to allow you to load or save programs, turn on or off your printer, and use the Commodore 64 DOS to delete files, etc. See 'File handling' for more details.

Quit

Leaves G-Pascal, but first checks that you wanted to quit by asking:

Quit ? Y/N

If you do not type 'Y' the quit has no effect. Do not Quit unless you have finished with G-Pascal for now.

USING THE EDITOR

The Editor displays a colon (:) when it is ready for a command, so if you see a : you know you are in the Editor. To leave the Editor enter: Q (RETURN) and you will return to the Main Menu, or QF (RETURN) and you will go straight to the Files Menu.

Line numbers

The Editor is line-number oriented. This means that all Editor commands refer to the line-numbers of each line in your program. The line numbers are allocated automatically by the Editor, starting at 1 and going up by 1 each line. The line number allocation is 'dynamic', that is, if you insert a line between what is currently line 6 and line 7 then the new line becomes line 7 and the line that used to be line 7 becomes line 8, etc.

Commands

All Editor commands consist of *one* letter followed by none, one or possibly two line numbers. For example:

```
L 10,20
```

would list lines 10 to 20. In some commands (Find and Replace), the line number range is also followed by a 'string' enclosed within delimiters. If you are specifying a line-number range, there should be either no spaces, or one space between the command and the line numbers.

If you are specifying two line numbers (as in the above example) you can use any non-numeric 'delimiter' between the line numbers (except '.' which is used by Find and Replace to delimit strings). Therefore use whatever you feel comfortable with. For example, all the following are equivalent:

```
L 15-30
L 15 30
L 15,30
```

The commands are :

```
<C>ompile
<D>elete Line Number Range
<F>ind Line Number Range . string .
<I>nsert Line Number
<L>ist Line Number Range
<M>odify Line Number Range
<Q>uit
<R>eplace Line Number Range . old . new .
<S>yntax
```

(The above command summary appears if you type 'HELP' when in the Editor).
The commands are explained in detail on the following pages.

EDITOR COMMANDS

DELETE

Deletes one or more lines. Any lines following those deleted are renumbered.

To delete one line:

D *line-number*

e.g.

D 5

would delete line 5.

To delete a range of lines:

D *first-line, last-line*

e.g.

D 10,20

would delete lines 10 to 20.

If you attempt to delete 5 or more lines the Editor will ask you (for example):

Do you want to delete 200 lines ? Y/N

This is in case you meant to delete 20 lines but accidentally tried to delete 200 lines. If you want the delete to go ahead press 'Y' otherwise press any other key. You do not need to press (RETURN) as well.

INSERT

Inserts new lines into your program. To insert, enter:

I *line-number*

where 'line-number' is the line that you want to insert *after*. To stop inserting just type (RETURN) when the cursor is over a blank line (or a line containing only the line number). Usually this means just press RETURN before typing in anything. To insert a blank line enter SHIFT/SPACE and then press RETURN.

For example:

I 10

inserts after line 10 (starting at line 11).

The Editor reminds you what line you are currently inserting by displaying its line number at the start of the line. To put a line at the very start of the program (line 1) enter:

I

To put a line at the end of the program enter:

I 9999

LIST

Lists your program.

To list the whole program: L

To list one line: L *line-number*

To list a range of lines:

L *first-line,last-line* e.g.

L 10

will list line number 10.

L 40, 60

will list lines 40 to 60.

You can temporarily 'freeze' your list by pressing the space bar. You can stop the listing altogether by pressing the RUN/STOP key.

To list without line numbers:

There is a variation of the List command called the 'No-line- number List' (N for short). It works the same as List except that line numbers are not displayed on the left. Its intended purpose is for listing the file when the line numbers would be a nuisance – for example if you use the editor for non-Pascal uses such as producing a letter or a name and address list. The 'N' command does not appear on the command summary.

e.g.
N 100-500
would list lines 100 to 500 without line numbers.

MODIFY

Allows you to change one or more lines in your program.

To change one line:

M *line-number*

To change a range of lines:

M *first-line, last-line*

Modify works by:

a) listing the line or lines requested.
b) deleting the lines requested.
c) entering Insert mode so that you can replace them.

Like the Delete command, Modify will warn you if you are about to change more than 4 lines.

The intention is that for making minor changes to small numbers of lines you use the 'cursor control' keys to move the cursor over the erroneous lines, making corrections where necessary. Once a line is corrected just press RETURN to copy the corrected line back into the program. The cursor does not have to be at the end of the line – anywhere on the line will do.

COPYING AND MOVING LINES

The Insert command can be used to copy or move text from one part of the program to another (in groups of up to 20 lines). The method of doing this is:

a) List the lines to be copied/moved.
b) Enter 'Insert' mode at the point where the lines are to be copied or moved to.
c) Move the cursor to the start of the lines listed (in (a) above) using the upwards cursor control key.
d) Press RETURN to copy that line into the new spot. If more than one line is to be copied keep pressing RETURN until all the desired lines are copied. Then leave Insert mode by positioning the cursor over a blank line and pressing RETURN (an easy way to do this is to press SHIFT/CLEAR/HOME and then RETURN).
e) Delete them from the old place in the program if desired. (Warning - they may have different line numbers now if you inserted before them.)

To move or copy larger blocks of code (hundreds of lines) you can use the 'append' command in the file menu – just save to disk or cassette a temporary file containing the code to be moved and then append it back into the right spot.

FIND

The Find command is a powerful method of quickly locating a string of text within your program. By 'string' we just mean a sequence of characters.

For example – to find the start of every procedure in your program:

```
F.procedure.
```

As the editor finds each line containing the word 'procedure' (in this example) it displays that line with its line number. When it has finished it displays the total number found.

The Find command can also be used on a range of lines (or a single line) by specifying a line range in the usual way –

F *first-line, last-line . string . options*

e.g.

```
F 100-200.begin.
```

The string delimiters *must* be the '.' symbol, they must *both* be present, and directly follow the line number range (if present) or the letter 'F' (if not). Pressing RUN/STOP will abort the Find.

Options on the Find

The second delimiter may optionally be followed by up to three parameters – each represented by a single letter. They may appear in any order or not at all. The options are:

T – Translate – this will temporarily translate any lower-case letters in the program to upper case before comparing them to the specified string. If the 'T' option is *not* used then the Find will differentiate between 'FRED' and 'fred', for example. Since the compiler considers upper and lower case identical (except inside string literals) then it may be wise to use the 'T' option when checking for how often words occur. If 'T' is used then the string specified in the Find *must* itself be entered in lower- case.

e.g.

```
F.fred.T
```

Q – Quiet – this will not display lines which contain the specified string – only the count at the end of the Find will be displayed. This is useful for just counting the number of times (for example) the word 'BEGIN' occurs without actually listing all the lines which contain it.

e.g.

```
F.begin.q
```

G – Global – this will search each line in the program for *every* occurrence of the specified string, not just the first. The only difference this will make to the Find is that if a line contains more than one occurrence of the string then the count at the end of the Find will be different.

e.g.

```
F.begin.gtq
```

REPLACE

The Replace command is a powerful method of quickly replacing a string of text within your program with another one. By 'string' we just mean a sequence of characters.

For example – to replace the first occurrence of the word 'enemies' on each line with 'klingons' just say:

```
R.enemies.klingons.
```

As the editor finds each line containing the word 'enemies' (in this example) it replaces the *first* occurrence on that line with the word 'klingons' and then displays the line with its line number. When it has finished it displays the total number replaced.

The Replace command can also be used on a range of lines (or a single line) by specifying a line range in the usual way –

R *first-line, last-line. old string . new string . options*

e.g.

```
R 100-200.fred.nurk.
```

The string delimiters *must* be the '.' symbol, all three *must* be present, and directly follow the line number range (if present) or the letter 'R' (if not). Pressing RUN/STOP will abort the Replace.

Options on the Replace

The third delimiter may optionally be followed by up to three parameters – each represented by a single letter. They may appear in any order or not at all. The options are:

T – Translate – this will temporarily translate any lower-case letters in the program to upper case before comparing them to the target string. If the 'T' option is *not* used then the Replace will differentiate between 'FRED' and 'fred', for example. See the discussion under the 'Find' command for more details.

Q – Quiet – this will not display lines which have been replaced - only the count at the end of the Replace will be displayed. This is useful if you know there will be a lot replaced – and you are sure that you are replacing the right things!

G – Global – this will replace *every* occurrence of the string on each nominated line – not just the first – use with discretion. *Warning:* if you use the 'Global' option indiscriminately and make a program line longer than 88 characters (the maximum the system can handle) then the Editor will not function correctly. For space reasons there is no check built in to prevent this happening, so make sure that in the process of replacing that program lines are kept to a reasonable length.

COMPILE

This starts a compile of your program. It is the same as entering 'C' from the 'Main Menu'. See 'Compiling your program' for more details.

SYNTAX

This does a 'syntax check' of your program (to let you know if it has errors). It is the same as entering 'S' from the 'Main Menu'. See 'Compiling your program' for more details.

QUIT

Typing 'Q' leaves the Editor (quits) returning you to the 'Main Menu'. Typing 'QF' takes you directly to the 'Files' menu.

BEGINNER'S GUIDE TO PASCAL

This guide is not intended to be a comprehensive Pascal tutorial, however it will introduce you to G-Pascal's basic concepts. Try out the examples yourself to become familiar with the language structure and what the statements do.

General format

Spaces or new lines may freely be used within your program (except in the middle of words or symbols) to make the program more readable. Upper or lower case characters may be used interchangeably as the compiler converts everything to upper case internally (except string constants, e.g. "fred").

Comments

Comments may be freely interspersed within your program – they must be enclosed within the (* and *) symbols. Comments may appear anywhere (except within the middle of a word or symbol).

Reserved words

Reserved words are those that have a specific meaning to the compiler and must be used in the correct context. They are shown in this section in upper case in order to distinguish them from other words. However they may be entered in lower case in your programs if you wish. In fact, since all reserved words are 'tokenised' by the compiler they will always be displayed in lower case, regardless of how they are entered.

User-defined words

All words other than reserved words are chosen by the programmer (yourself). They may be any length (all characters are significant), must start with a letter, and after that consist of letters, numbers or the underscore character. (The underscore is shown in this Manual as _ however on the Commodore 64 it looks like a 'left-arrow', and is entered by the key on the top left-hand side of the keyboard.) All user-defined words must be declared before they are used. 'Declaring' a word means telling the compiler to what use you are putting the word. You do this by using the word in a CONST, VAR, PROCEDURE or FUNCTION declaration.

GENERIC WORDS

In this section of the Manual 'generic' words are entered in *italics*. For example, the definition of an IF statement is:

IF *expression* THEN *statement* ELSE *statement*

In this case the words *expression* and *statement* are neither reserved words or user-defined words. Instead they indicate that they are to be replaced with a valid expression or statement as appropriate, referring to other parts of this Manual for the correct construction of expressions and statements.

If an ellipsis (...) is used it indicates that the preceding syntactical item may be repeated (usually indefinitely). Exactly what part may be repeated is usually obvious from the context or the examples following. For example, the definition of a compound statement is:

BEGIN *statement; statement* ... END

In this case zero or more statements may appear between the BEGIN and the END. If more than one statement appears they are separated by semicolons.

PROGRAM

A G-Pascal program consists of a 'block' followed by a period. A block is explained below. For example, a simple program is:

```
BEGIN
  WRITE ("Hello")
END.
```

BLOCK

A block consists of the following:

CONST *constant-declarations*

VAR *variable-declarations*

PROCEDURE }
FUNCTION } *procedure and function declarations*

BEGIN *statements* END

The 'declarations' are all optional, however if used they must appear in the above order. Since a valid statement is a 'null' statement (i.e. nothing) then the minimum block is: BEGIN END

CONST

The constant declarations are used to assign a name to a constant, that is, a value that will not change during the execution of the program. The form of the constant declaration is:

CONST *identifier* = *constant*; ...

For example:

```
CONST cr = 13;
  clearscreen = 147;
  numberofplayers = 2;
  true = 1;
  false = 0;
  on = true;
  off = false;
```

As in the above example, a constant declaration can use an identifier which has already been declared as a constant (e.g. on = true;).

VAR

Variable declarations are used to name the variable data. Variables are either INTEGER or CHAR type, and may optionally be an array. Multiple variables of the same type may be separated by commas. This is the general format of variable declarations:

VAR *list-of-identifiers* : INTEGER;
list-of-identifiers : CHAR;
list-of-identifiers : ARRAY [*array-size*] OF INTEGER;
list-of-identifiers : ARRAY [*array-size*] OF CHAR;

The list-of-identifiers consists of one or more variable names, separated by commas.

The array-size is expressed as the maximum occurrence number of that array item. Arrays start at subscript zero, so that when you say ARRAY [10] you are referring to an array of 11 items, numbered 0 to 10.

Some examples:

```
VAR a, b, c, d, e, f : INTEGER;
  enemystrength,
  enemysize,
  enemydexterity : ARRAY [ 10 ] OF INTEGER;
  inputline : ARRAY [80] OF CHAR;
  ch : CHAR;
  amount1, amount2,
  amount3, amount4
  : INTEGER;
```

INTEGER variables are three bytes long each (in other words, three bytes of memory are reserved by the compiler for the contents of each variable or array element in the case of INTEGER arrays). This means that INTEGER variables can contain a number in the range -8388608 to +8388607. (In hexadecimal the range is: $0 to $FFFFFF). Integer variables can also hold up to a three-byte 'string' of characters, for example: "hi" or "ZZZ".

CHAR variables are one byte long each. This means that CHAR variables can contain a number in the range 0 to 255 (hexadecimal $0 to $FF). They can also contain a single character string, for example: "b" or "5".

The Compiler does not carry out 'type-checking' so that INTEGER and CHAR variables can generally be intermixed in expressions and elsewhere without problem. However, normally INTEGER variables are used to hold numbers, and CHAR variables are used to hold characters, particularly arrays of characters. If the result of an expression is stored in a CHAR variable and the result exceeds 255 (the maximum that it can hold) then the result is truncated – only the low-order byte is stored. Mathematically this means that the result stored is the number modulus 256.

G-Pascal does not support other data types, nor the TYPE declaration. However see the section 'Converting from other Pascals' for hints on how to simulate other data types in G-Pascal.

PROCEDURE and FUNCTION

Procedures and Functions are 'subroutines' which may be called later on in the program to achieve a specific purpose. The format of Procedure and Function declarations is identical. The difference between them is that functions return a value, and thus form part of expressions (explained later), whereas procedures do not return a value, and do not form part of expressions. The format of procedure and function declarations is:

PROCEDURE *procedure-name (argument-list); block ;*

FUNCTION *function-name (argument-list); block ;*

The '(argument-list)' is optional, however if used it consists of one or more arguments, separated by commas, inside parentheses. These are known as 'formal arguments'. In the body of the procedure or function the formal arguments are used as if they had occurred in a VAR declaration (of type INTEGER), although in fact no declaration is needed for formal arguments. During the execution of the program, when the procedure or function is invoked arguments are supplied (the actual arguments) – copies of these are then used by the procedure or function. For example, a function to double a number would have one argument (the number which is to be doubled) – the value returned by the function would be the double of the formal argument. (See example below).

You will note that the definition of a procedure or function declaration includes a 'block', whereas procedure and function declarations are themselves an (optional) part of a block. This is known as a recursive definition. In other words, a block may contain a procedure declaration which itself contains a block, which in turn may contain another procedure declaration and so on. This means that procedures and functions may themselves contain CONST, VAR, PROCEDURE and FUNCTION declarations.

If the procedure or function contains its own VAR declarations then these are known as 'local' variables as they can only be referred to inside the procedure or function in which they are declared. Outside the procedure or function they have no meaning and may not be referred to (this is referred to as the 'scope' of that declaration). It is good programming practice to use local variables for data that is not needed outside the procedure or function, such as loop counters, flags and so on. This avoids possible unintentional conflict with other variables used elsewhere.

The values of local variables are lost when the procedure or function in which they are declared is exited. That is, when a procedure or function is invoked the contents of all of its local variables are not defined – they are not retained from any previous invocation. They may only be used while the procedure or function is active, and are discarded when the end of the procedure or function is reached.

Examples of Procedures and Functions

Here are some examples of procedure and function declarations:

```
PROCEDURE clearscreen;
CONST home = 147;
BEGIN
WRITE ( CHR(home) )
END; (* of clearscreen *)

PROCEDURE printdoubles ( firstnumber, lastnumber );
VAR number : INTEGER;

FUNCTION double ( x );
BEGIN
double := x * 2
END; (* of double *)
```

```
BEGIN
FOR number := firstnumber TO lastnumber DO
   WRITELN ( number, " times 2 is ", double ( number ))
END; (* printdoubles *)

BEGIN (* main program *)

clearscreen;
printdoubles (20,30)

END. (* of main program *)
```

The above example, as well as illustrating procedures and functions, is a complete program which you may want to key in and try for yourself. Note that the function 'double' is declared within the procedure 'printdoubles'. This is an example of a 'nested' function. Procedures and functions may be nested indefinitely. Notice the different way that procedure and functions are invoked (called). Procedures are invoked by naming them as a statement. The main program above in fact merely consists of two procedure invocations. Functions however are invoked by naming them as part of an 'expression' – in this case the function 'double' forms a part of the WRITELN statement.

The function 'double' also illustrates how a function can use its parameters (arguments) in a calculation. When 'double' is called it is passed the parameter 'number' (i.e. double (number)). A copy of the value contained in 'number' is made and passed to 'double' which then considers it to be its argument 'x'. 'x' is then multiplied by two and assigned to the name of the function, which is the way that the function returns its result to the part of the program that invoked it. (i.e. double := x * 2).

Recursive Procedures and Functions

Procedures and functions can also invoke themselves recursively which can sometimes be useful. For example, here is a way of calculating factorials by recursion:

```
VAR number : INTEGER;

FUNCTION factorial ( x );
BEGIN
  IF x = 1 THEN
    factorial := x
  ELSE
    factorial := x * factorial (x – 1)
END; (* factorial *)

BEGIN (* main program *)
  FOR number := 1 TO 10 DO
  WRITELN (number, "! = ", factorial (number))
END.
```

The example above is also a complete program which you may wish to try for yourself. The function 'factorial' keeps calling itself (recursing) until it can go no further (1 factorial is 1). There are other ways of calculating factorials but this example illustrates recursion quite well.

(A factorial of an integer is the product of all the integers from 1 to itself. e.g. 4 factorial, which is represented mathematically as 4!, is 4 * 3 * 2 * 1, which is 24. 5 factorial is 120 and so on.)

STATEMENTS

The parts of the program (or block) that 'do something' are the statements. The beginning of the statements part of a block is signalled by the word BEGIN and the end by the word END. Like the definition of a block, the definition of a statement is recursive – that is, some statements may contain other statements. The simplest example of this is the 'compound statement'. A compound statement looks like this:

BEGIN
statement; statement; statement ...
END

In other words, wherever a statement is allowed, the word BEGIN may be placed, followed by any number of statements separated by semi- colons, followed by the word END. For example, the definition of an IF statement is as follows:

IF *expression* THEN *statement* *ELSE statement*

The following examples are valid IF statements:

```
IF x = 3 THEN
  b := 5
ELSE
  b := 10;

IF word = "abc" THEN
  BEGIN
  a := 1;
  b := 2
  END
ELSE
  BEGIN
  a := 5;
  b := 10
  END;
```

Null statements

A valid statement is the 'null' statement (in other words, nothing). So the following is also a valid (although rather meaningless) IF statement:

```
IF a = 21 THEN ELSE;
```

The absence of any symbols between the THEN and the ELSE and also after the ELSE indicates the null statement. In other words, nothing will happen. Occasionally this is useful – for example, during debugging you may temporarily convert a statement to comments (by enclosing it in (* and *)) – this will have no ill effect.

We now summarise the various G-Pascal statement formats, with a brief description of their purpose.

Assignment statement

variable := *expression*

The assignment statement evaluates the expression (expressions are explained later) and places the result in the named variable. e.g.

```
A := B * 25;
```

Procedure invocation

procedure-name (expression, expression ...)

By naming a previously defined procedure you invoke it – that is, transfer control to the procedure to carry out the statements defined within that procedure's declaration. If the procedure declaration contained a list of formal arguments then the actual values to be passed to that procedure are now supplied in the form of a list of expressions, separated by commas, all in parentheses. e.g. PLAYANOTE (PITCH, DURATION); If no arguments are required then the parentheses are not required. e.g. NEXTGAME;

IF

IF *expression* THEN *statement* ELSE *statement*

The expression is evaluated. If it is true (not zero) then the first statement is executed, if it is false (zero) then the second statement is executed. The 'ELSE statement' clause is optional. (Example previously).

WHILE

WHILE *expression* DO *statement*

The expression is evaluated. If it is true (non-zero) the statement is executed. This process is repeated until the expression becomes false (zero).

e.g.

```
x := 0;
   WHILE x < 10 DO
     BEGIN
     x := x + 1;
     WRITELN (x," squared is ",x * x)
     END
```

REPEAT

REPEAT *statement ; statement ...* UNTIL *expression*

The statements between REPEAT and UNTIL are executed. Then the expression is evaluated. If it is false (zero) this process is repeated. e.g.

```
REPEAT x := x + 1; y = y + 5 UNTIL x = 50
```

CASE

CASE *expression* OF
label-list1 : *statement1; ...*
label-list999 : *statement999*
ELSE *statement*
END

The expression is evaluated. This is called the 'case selector'. It is then compared in turn to the lists of expressions (labels) preceding each statement. If it matches one of them then the statement following the colon is executed and the CASE statement terminates. If no label matches the selector then the statement following the ELSE is executed. As in the IF statement the ELSE clause is optional. The label list consists of one or more expressions, separated by commas. This is an extension of standard Pascal which only allows a list of constants. e.g.

```
CASE actionnumber OF
  1    : processaction1;
  2, 3, 4 : processactions234;
  5, 6 : processactions56
ELSE   processerroraction
END
```

The END is part of the CASE statement and should not be confused with the normal BEGIN END pairs normally found elsewhere in Pascal.

Here is an example of using expressions in a CASE statement:

```
          CASE true OF
  (x > 0) AND (x < 10) : smallx;
  (x >= 10) AND (x < 100) : mediumx;
  x >= 100          : largex
ELSE            toosmallx
END
```

WRITE and WRITELN

WRITE (*output-list*)
WRITELN (*output-list*)
WRITELN

WRITE and WRITELN function identically except that WRITELN automatically writes a 'carriage return' at the end of the output- list. In other words, at the end of a WRITELN the cursor (or printer if printing) commences a new line. WRITELN without any arguments just writes a carriage return.

The output list is evaluated and written to the screen. The output list consists of one or more of: expressions, string constants, or the special functions HEX and CHR, separated by commas.

String constants are written 'as is', e.g. WRITE ("hello") would display the word 'hello' on the screen. If you want the quote symbol to appear in the string itself then two consecutive quotes should be used. For example, to display: MY NAME IS "SUE" you would enter: WRITELN ("MY NAME IS ""SUE""").

If an expression is written on its own it is evaluated and the result displayed as a decimal number. e.g. WRITE (5 * 6) would display '30' on the screen.

To write 'screen control' codes (such as clear screen, shift to upper-case and graphics and so on) the number should be written as: CHR (x) where x is an expression. e.g. WRITELN (CHR(147)) would clear the screen. (This is the same as: PRINT CHR$(147) in Basic).

To write a number in hexadecimal the number should be written as: HEX (x) where x is an expression. e.g. WRITE (HEX(10)) would display '00000A'.

The functions HEX and CHR are only meaningful within a WRITE or WRITELN statement.

```
WRITELN (65," in hex is ",hex(65)," and in ascii is ",chr(65));
```

READ (*input-list*);

The input list consists of a list of one or more variables, separated by commas. Each item in the list is treated separately, as if they had appeared in separate READ statements. There is no built-in method of accepting a list of numbers from one line of input, although you could write a procedure to do this by accepting a string and decoding it as desired. The effects of the various types of input-list variables are as follows:

a) Variable of type CHAR – a single character is accepted from the keyboard and placed in the named variable. The system does not wait for the RETURN key to be pressed. The character is not echoed on the screen.

b) Array of type CHAR with no subscript supplied – a line of input is accepted from the keyboard and placed in the named array, starting at subscript zero. The line is echoed on the screen as it is typed in, and the normal cursor control keys may be used to edit the line prior to the RETURN key being pressed. If the line entered is smaller than the array size then a RETURN symbol will appear in the array (i.e. the value 13) after the last character entered. *Warning – the Commodore 64's full-screen editor will cause text on the screen to the right of the cursor to be considered part of the input, even if it wasn't actually typed in at that time.* This means that a 'default' answer can be displayed and the cursor positioned to the start of it. If the user just presses RETURN then the text to the right of the cursor will be returned automatically.

c) Variable of type INTEGER – a line of input is accepted from the keyboard. The line is echoed on the screen as it is typed in, and the normal cursor control keys may be used to edit the line prior to the RETURN key being pressed. After RETURN is pressed an attempt is made to interpret it as a decimal number and place the result in the named variable. The first character must be a number or minus sign, the following characters must be numbers, until the RETURN is pressed. If these conditions are not met the built-in function INVALID is set true, otherwise (if the number is OK) INVALID is false. *Warning - the Commodore 64's full-screen editor will cause text on the screen to the right of the cursor to be considered part of the input, even if it wasn't actually typed in at that time.* This means that a 'default' answer can be displayed and the cursor positioned to the start of it. If the user just presses RETURN then the text to the right of the cursor will be returned automatically.

d) Variable of type INTEGER followed by a $ symbol. A line of input is accepted from the keyboard. The line is echoed on the screen as it is typed in, and the normal cursor control keys may be used to edit the line prior to the RETURN key being pressed. After RETURN is pressed an attempt is made to interpret it as a hexadecimal number and place the result in the named variable. The numeric string input must be from 1 to 6 hexadecimal digits for the attempt to succeed. If the number is not recognized as a hexadecimal number then the built-in function INVALID is set true, otherwise (if the number is OK) then INVALID is false.

Some examples:

```
VAR number, hexnumber : INTEGER;
  character : CHAR;
  line    : ARRAY [80] OF CHAR;
BEGIN
READ (number);    (* read a decimal number *)
IF INVALID THEN WRITELN ("bad number");
READ (hexnumber $ ); (* read a hex number *)
IF INVALID THEN WRITELN ("bad hex number");
READ (character);    (* read a single character *)
READ (line);        (* read a line into an array *)
END.
```

FOR

FOR *variable* := *expression1* TO *expression2* DO *statement*

Expression1 is evaluated and assigned to the variable. It is then compared to expression2. If it is less than or equal to expression2 then the statement is executed. Then 1 is added to the variable, and it is compared to expression2 again and so on, until the variable is greater than expression2.

FOR *variable* := *expression1* DOWNTO *expression2* DO *statement*

Expression1 is evaluated and assigned to the variable. It is then compared to expression2. If it is greater than or equal to expression2 then the statement is executed. Then one is subtracted from the variable, and it is compared to expression2 again and so on, until the variable is less than expression2.

For example:

```
FOR j := 1 TO 20 DO
  WRITELN (j);

FOR k := 100 DOWNTO 20 DO
  WRITELN (k)
```

CALL

CALL (*expression*)

The CALL statement transfers control to the machine-code subroutine located at the address which the expression evaluates to. For example, to call the 'close all files' routine in the Monitor Kernal (at address $FFE7) you would say:

```
CALL ( $FFE7 );
```

It is possible to set up the A, X, Y registers and condition codes before the CALL and examine them after the CALL See the section on 'Machine Language Subroutines' for more details.

Other statements

There are other statements with specialized uses which are G- Pascal extensions for the Commodore 64, which control the cursor, graphics, sound effects, clock etc. These are described in more detail in other sections of this Manual.

EXPRESSIONS

An expression (or 'arithmetic expression') forms part of the definition of many statements, as seen previously. For example, an assignment statement assigns the result of the evaluation of an expression to a variable. Like the definitions for a block and a statement, the definition of an expression is recursive, that is, an expression can contain another expression. Expressions consist of a combination of operands and operators. e.g.

```
5
A < 2
(a * b / c) + 14
```

Operands

array-name [index-expression]
constant
function-name
function-name (argument-list)
variable
(expression)

Operators

(In decreasing order of precedence):

NOT
* / DIV MOD AND SHL SHR
+ – OR XOR
= <> < > <= >=

Precedence controls in which order the expression is evaluated. Since * is on a higher precedence than + then the expression:

```
5 + 8 * 10
```

evaluates to 85, not 130, since the multiplication is done before the addition. Where operations of equal precedence are presented then they are evaluated from left to right. Since the operators AND and OR are on a higher precedence than the relational operators (=, <, › and so on) then conditional expressions generally require parentheses around expressions combined by AND and OR. For example:

```
IF ( a > 5 ) OR ( b < 10 ) THEN<
```

If the parentheses were omitted G-Pascal would evaluate '5 OR b' before doing the relational tests, which would probably not have the desired result.

MEANING OF OPERATORS

NOT

NOT reverses the value of a boolean operand. Unlike all other operators which are inserted *between* two operands (e.g. in the expression: 4 + 6 the operator '+' comes between the '4' and the '6'), NOT *precedes* a single operand. If its operand is true (non-zero) then NOT makes it false (zero). If its operand is false (zero) then NOT makes it true (1). e.g.

```
REPEAT game UNTIL NOT alive;
```

*

* is the symbol for multiplication. It multiplies two numbers together. e.g.

```
salary := wage * 52;
```

/ and DIV

/ and DIV are synonymous and are the symbols for division. They result in integer division. (If the division cannot be performed exactly then the remainder is discarded). e.g.

```
inches := feet / 12;
```

The remainder of a division can however be established with the MOD operator.

MOD

MOD returns the remainder of a division (often known as the *modulus* – hence its name). e.g.

```
diceroll := RANDOM MOD 6 + 1;
```

AND

AND performs a boolean 'and' of each of the 24 bits in the two operands. That is, a result bit is on only if both corresponding bits in the operands are on. The normal use for AND is in the conventional sense of checking that two conditions are true simultaneously, however it can also be used to 'mask' out unwanted bits in an integer. e.g.

```
IF (a > 4) AND (b < 5) THEN     (* check both conditions are true *)
IF PADDLE (2) AND $FF THEN      (* isolate low order byte *)
```

SHL

SHL performs a 'shift left' operation. It takes the first operand and shifts it left the number of bits specified in the second operand (maximum of 24 shifts). Each shift left is equivalent to multiplying the first operand by 2, so SHL is a simple and quick way of raising a number to a power of 2. For example, to multiply a number by 2 to the power 6:

```
result := number SHL 6;
```

SHR

SHR performs a 'shift right' operation. It takes the first operand and shifts it right the number of bits specified in the second operand (maximum of 24 shifts). Each shift right is equivalent to dividing the first operand by 2, so SHR is a simple and quick way of dividing a number by a power of 2. For example, to divide a number by 2 to the power 6:

```
result := number SHR 6;
```

+

+ is the symbol for addition. It adds its two operands together. e.g.

```
result := a + b;
```

–

– is the symbol for subtraction. It subtracts the second operand from the first. e.g.

```
klingons := klingons – 1;
```

OR

OR performs a boolean inclusive 'or' of each of the 24 bits in the two operands. That is, a result bit is on if either or both of the corresponding bits in the operands are on. The normal use for OR is in the conventional sense of checking that either of two conditions are true, however it can also be used to 'turn on' certain bits in an integer. e.g.

```
IF (a > 4) OR (b < 5) THEN    (* check either condition is true *)
mask := mask OR 1;        (* turn on low order bit *)
```

XOR

XOR performs a boolean exclusive 'or' of each of the 24 bits in the two operands. That is, a result bit is on if either of the corresponding bits in the operands are on *but not both*. XOR is a G-Pascal extension and is less likely to be used than OR, however one important use for XOR is for toggling a bit – that is, turning it off if it is on, or turning it on if it is off. This can be useful when defining your own character sets – the 'inverse' of any character can be obtained by XORing the character with $FF. This will switch any 1's to 0's, and any 0's to 1's. e.g.

```
b := b XOR 1; (* toggle contents of b *)
byte := byte XOR $FF; (* invert contents of byte *)
```

= <> < > <= >=

The above symbols are all 'relational operators'. That is, they return a boolean value (true or false) depending on whether or not the relation is true. The result of a relational operation will always be zero (false) or 1 (true). Relational operators are usually used in IF, WHILE or REPEAT statements, however they can be used anywhere that an expression is permitted. e.g.

```
IF ships < 3 THEN .
IF answer = 6 THEN
result := b >= 4;
```

NOTES ON THE COMPILER

ARRAYS

Single dimension arrays only are permitted. If multi-dimensional arrays are needed it is easy to write functions to simulate them. For example, if you want an 8 by 15 array, declare your array as having 120 elements (8 times 15) and write a function that multiplies the first subscript by 8 and adds the second subscript. That way each combination of subscripts 'maps' onto a unique element.

Subscripts must be enclosed within square brackets ([and]). These are located above the colon and semicolon (: and ;) on the keyboard.

Range checking of arrays

No range checking is carried out on any arrays. This means that if you declare an array of 10 items and write to the 11th upwards, or to the –1st downwards then you will 'clobber' something unexpected. This may cause your program to 'crash' or behave strangely, or quite possibly the entire G-Pascal system may crash.

Part of the reason for this is that 'return' addresses for procedure or function calls, and also 'stack frame addresses' are stored on the stack, right next to your variables. Therefore if an array is defined as the first variable within a procedure or function then exceeding the bounds of a declared array by just *one* element could make the procedure or function lose track of where to return to. *If you think an array variable may exceed its declared bounds build a check into your program to make sure it doesn't.*

Array allocation

Arrays are allocated *downwards* in memory. This is important if you are storing a machine language program in an array, for example. In a given array, the highest array element has the lowest memory address.

'PROGRAM' STATEMENTS

Programs do not start with the word PROGRAM. A program must start with one of: CONST, VAR, PROCEDURE, FUNCTION or BEGIN. Failing this, the error message 'BEGIN expected' will appear.

CONSTANTS

Constants (a constant is a factor in an expression that does not change throughout the execution of a program) can be expressed in one of four ways interchangeably:

1. Decimal constants: for example: 1096, –33, +99

Decimal constants are simply signed numbers to the base 10. They may range from –8388608 to +8388607.

Warning – since the compiler interprets a plus or minus sign directly in front of a number as being the 'sign' of that number (e.g. –5) then when using plus or minus signs for arithmetic (addition and subtraction) then at least one space must follow the sign if the next symbol is a number. In other words:

```
a := b+1; would
```

give an error because the '+' would be interpreted as the sign of '1', not an addition operation. The correct way to represent the above would be:

```
a := b + 1;
```

2. Hexadecimal constants, for example: $A1, $FFC.

Hexadecimal constants are unsigned numbers to the base 16. They may range from $0 to $FFFFFF. The '$' is required.

3. String constants, for example: "Z", "NO", "YES".

String constants are a string of between one and three characters, enclosed in quotation marks. If only one or two characters appear in the string then the high-order byte/s are set to zero. Therefore when comparing to, or moving to or from, a CHAR type variable only one character should appear between the quotes as CHAR variables are only one byte long. The use of two or three character string constants would primarily be restricted to applications which involve a lot of 'word' work, such as Adventure-type games or other applications which involve three letter commands.

A string constant "abc" will be stored as:

```
"a" + "b" * 256 + "c" * 65536
```

or in other words:

```
"a" OR "b" SHL 8 OR "c" SHL 16
```

4. Identifiers which appear in a CONST declaration.

For example, if the declaration: CONST cr = 13;
appears in the program then 'cr' is considered to be a constant. The use of CONST declarations for constants is *highly recommended* as they make the program much more readable and easy to follow. For example: WRITE (chr(147)) and WRITE (chr(clearscreen)) will both clear the screen (if 'clearscreen' is declared to be equivalent to 147 in a CONST declaration) but the latter makes the intention much clearer when reading the program.

DATA TYPES

Integer

Integers are stored as 3-byte signed numbers and therefore range from –8388608 to 8388607.

Char

CHAR type variables are stored as one byte each. No type checking is carried out by the compiler so in fact INTEGER and CHAR variables may be used completely interchangeably in G-Pascal, except:

a) CHAR variables have a different meaning to integer variables in the READ procedure.

b) The result of an expression stored in a CHAR variable will be MOD 256 (that is only the low order byte will be stored – the high order two bytes will be discarded). Therefore a CHAR variable will always be in the range of 0 to 255 (or $0 to $FF).

c) CHAR variables only use one third the amount of memory to store as integers – particularly important in big arrays.

Other data types

Other data types are not supported, nor is the TYPE statement. (But see 'Converting from other Pascals' for hints on how to simulate other data types in G-Pascal).

ELSE CLAUSE

The ELSE clause may be used with the IF statement, in which case it is always associated with the most recent un-ELSE-ed IF.

The ELSE clause may also be used with the CASE statement to cause a statement to be executed if none of the labels agree with the case selector.

MEM ARRAY

(read this if you like peeking/pokeing)

The MEM array is a pre-defined integer array starting at address zero.
For example:

```
FRED := mem [ $5000 ];
```

would place in FRED the 3 bytes in memory, starting at address $5000;

or:

```
mem [ $5000 ] := $ABCDEF;
```

would store $ABCDEF in memory addresses $5000 to $5002.

Caution: Careless use of MEM as a receiving field (as in the second example) could 'crash' your G-Pascal system or destroy your program. Use with care!

Another caution: Use of absolute addresses via MEM and MEMC makes your programs machine-dependent and not portable.

A third caution: The subscript of the MEM array is always interpreted as an absolute address so that MEM [0] and MEM [1] in fact overlap by two bytes – they both share addresses 1 and 2. This is unlike ordinary integer arrays where each 'occurrence' represents an address 3 bytes away from its neighbour.

MEMC ARRAY

The MEMC array is a pre-defined CHAR array starting at address zero. Use MEMC for peeking/pokeing single bytes.

ADDRESS

The run-time address of a variable can be established by the

```
ADDRESS (identifier)
```

function. For example, to find the address in memory of FRED:

```
J := ADDRESS ( FRED );
```

The ADDRESS function would only be of interest to more experienced programmers for special purposes such as calling machine code routines contained within an array, or passing the address of a variable to a procedure. It is unique to G-Pascal and therefore not portable.

LOAD

The LOAD statement will load a file from disk or cassette under program control. The LOAD statement takes the following form:

```
LOAD (device, address, flag, filename );
```

The 'device' represents the device number which is 1 for cassette and 8 for disk. The 'address' is the address to which the data is to be loaded. The 'flag' is 0 for a Load and 1 for a Verify. The 'filename' is a string containing the filename.

After the load, the built-in function INVALID should be checked. If it is zero then the load was OK. If it is non-zero then INVALID contains the error code.
e.g.

```
LOAD (8, $1000, 0, "FILEA");
IF INVALID THEN WRITELN ("ERROR", INVALID , "ON LOAD");
```

SAVE

The SAVE statement will save a file to disk or cassette under program control. The SAVE statement takes the following form:

```
SAVE (device, start-address, end-address, filename );
```

The 'device' represents the device number which is 1 for cassette and 8 for disk. The 'start-address' is the start address from which the data is to be saved. The 'end-address' is the end address of the block of data to be saved. The 'filename' is a string containing the filename.

If the start address is greater than or equal to the end address a run-time error will occur.

After the save, the built-in function INVALID should be checked. If it is zero then the save was OK. If it is non-zero then INVALID contains the error code. e.g.

```
SAVE (8, $1000, $1800, "FILEA");
IF INVALID THEN WRITELN ("ERROR ",INVALID, " ON SAVE");
```

SAVE may be used to save a block of variables from a program so that they can be loaded back again later on (for example, saving the current position in an adventure game). In this case remember that variables are allocated *downwards* on the stack, so that the lowest memory address is in fact the last variable declared. The safest way of doing this is to declare two 'dummy' variables to pinpoint each end of the variable data, like this:

```
var firstvar : char; (* first program variable – highest address *)
a,b,c,d,e,f : integer; (* all program variables go here *)
i,j,k,l,m : char;
(* and so on *)
lastvar : char; (* last program variable – lowest address *)
begin
save (8, address(lastvar), address(firstvar), "varfile");
load (8, address(lastvar), 0, "varfile");
end.
```

Of course, this technique will only work if the number of variables is the same between the SAVE and the LOAD, in other words, if the data declarations in the program are the same.

GETKEY

GETKEY is a function that indicates whether or not a key is being pressed on the keyboard (or is in the keyboard queue). If no key is currently being pressed it returns zero – if a key is being pressed it returns a value corresponding to that key.

To make a program wait until any key is pressed for example just say:

```
REPEAT UNTIL GETKEY;
```

However if you need to know what the value of the key is then it must be saved in a temporary variable. e.g.

```
REPEAT
  KEYVALUE := GETKEY;
UNTIL KEYVALUE;
CASE KEYVALUE OF ..... (* and so on *)
```

Normally you would use the READ statement if you want to wait until a key is pressed. GETKEY is intended for applications where the program wants to occasionally check for keys being pressed on the keyboard but do other things in the meanwhile if they are not.

ABS (value);

The built-in function ABS returns the absolute value of its argument. In other words, ABS will always return a positive result, whether or not the argument is positive. This can be handy for establishing the distance between two points, regardless of which one is greater than the other. e.g.

```
distance := ABS ( x1 – x2 );
```

SAMPLE PROGRAM

```
(* ERATOSTHENES SIEVE PRIME NUMBER GENERATOR   %L *)

CONST SIZE = 1000;
   TRUE = 1;
   FALSE = 0;
   HOME = 147;
   PERLINE = 5;

VAR FLAGS : ARRAY [SIZE] OF CHAR;
I,PRIME,K,COUNT,ONLINE : INTEGER;

BEGIN
COUNT := 0;
WRITELN (CHR(HOME));
ONLINE := 0;
FOR I := 0 TO SIZE DO
   FLAGS [I] := TRUE;
FOR I := 0 TO SIZE DO
   IF FLAGS [I] THEN
   BEGIN
      PRIME := I + I + 3;
      K := I + PRIME;
      WHILE K <= SIZE DO
      BEGIN
      FLAGS [K] := FALSE;
      K := K + PRIME
      END;
      IF ONLINE > PERLINE THEN
      BEGIN
      WRITELN; (* NEW LINE *)
      ONLINE := 0
      END;
      ONLINE := ONLINE + 1;

      COUNT := COUNT + 1;
      WRITE (PRIME," ")
   END;
   WRITELN;    WRITELN (COUNT, " PRIMES")
END.
```

COMPILING YOUR PROGRAM

Compile

If you select 'Compile' from the Main Menu or Editor your program will be compiled and converted to P-codes, ready for Running. You may select a listing of the program to appear during compilation, or change the address at which the P-codes are located by using the Compiler Directives described below. If the compile is successful (no errors) you may immediately press 'R' (for Run) to test your program.

Syntax

The 'Syntax' option also compiles your program, however it does not generate P-codes so you must do a Compile afterwards if you have no errors and wish to run your program. As P-codes are not generated the %P compiler directive (display P-codes) does not function (acts the same as %L). In all other respects Syntax and Compile are identical. The main use for 'Syntax' is to check for errors if the P- codes are going to clobber the source code during the compilation process (this can only happen if the %A compiler directive is used). In this case, use the 'Syntax' option to check that the program has no errors, then save it to disk or cassette, then Compile it.

Asterisks

If you do not request a listing an '*' is displayed on the screen as every 32 source program lines are compiled. This is to reassure you that 'something is happening', and give a visual indication of how fast (and how far) the compilation is proceeding.

Errors

If there is an error in your program the compilation will stop with an arrow pointing to the symbol being processed when the error was detected (this not necessarily being the actual cause of the error as such), the word '*** Error' and an English error message. See the section 'Compiler Error Messages' for details about the meaning of errors.

COMPILER DIRECTIVES

Compiler directives are special symbols inserted within comments to cause G-Pascal to take special action during the compilation, such as producing a listing or placing the P-codes at a different address.

Compiler directives must appear within comments (i.e. after a (* symbol, and before its corresponding *) symbol) as they are not part of the Pascal language as such. They consist of the percentage symbol (%) directly followed by a code letter indicating the type of directive. The code letter may be in upper or lower case. These are described below.

%L (Listing)

The %L directive causes a listing of the program (with the P-code addresses displayed on the left in parentheses) commencing with the line on which the %L directive appears. The listing is the same as an Editor List, except for the P-codes on the left. At times knowing the P-code addresses is useful, particularly when a run-time error occurs (such as Divide by zero). In this case the address displayed when the run-time error occurs can be used to locate the line in the program which caused the error.

The %L directive would normally appear in the first line of the program, however it may be placed further down if only a partial listing is desired. Listing may be turned off with the %N directive.

```
(* %L *)
```

%N (No listing)

The %N directive stops the compiler from listing the program as it compiles. Instead, an asterisk is displayed as each 32 program lines are compiled. This is the default.

```
(* %N *)
```

%P (P-code listing)

The %P directive causes the compiler to list the program, and also to list each P-code that is generated for each statement. This directive would not normally be used, unless you are interested in what P-codes are generated for each statement. See 'Meanings of P- codes' for a description of what each P-code means. Like the %L directive %P is cancelled with an %N directive.

```
(* %P *)
```

%A (Address of P-codes)

The %A directive defines where the P-codes are to be placed during the compilation process. It should be followed by a decimal or hexadecimal address or the error 'Constant expected' will appear.

It is essential that the %A directive appear at the very start of the program, before the first CONST, VAR, PROCEDURE, FUNCTION or BEGIN. If this is not done the P-codes will not be compiled contiguously and a run-time error (or worse) will happen when the program is run.

If the %A directive is omitted then the P-codes will be placed directly following the end of the source program. This is the usual and recommended method of compiling. The %A directive should only be used if the 'Memory Full' error appears during compilation (which means that there is insufficient room at the end of the source program for the P-codes), or when compiling independent modules.

The address for the P-codes must be chosen with care. The permitted range is $800 (just after screen memory) to $4000 (overlapping the source program). Outside this range the error message 'Number out of range' will appear. A P-code address of $800 would be sensible for many applications, however care must be taken that DEFINESPRITE statements and bit-mapped graphics do not clash with the P-codes. See the 'Memory Map' section in this Manual for more details about what memory addresses may be used for which purpose.

In the event that the P-codes are placed at $4000 (the start of the source program), or just below, then the compilation process will cause the source program to be replaced by the P-codes, thus *effectively destroying the source program. Under these circumstances it is essential that the source program be saved to disk or cassette before a Compile, otherwise the source program will be lost.*

As the compiler is a single-pass compiler the technique of 'clobbering' the source program with the P-codes is an effective method of making maximum use of available memory, however there is the inconvenience of having to re-load the program from disk or cassette after each Compile. Remember that the 'Syntax' option is a method of compiling the program without generating P-codes, and should be used to ensure that there are no compile errors before the final compilation which does generate P-codes is done.

Examples of the %A option:

```
(* %A $800 – place P-codes after screen memory *)

(* %A $4000 – overwrite source code with P-codes *)
```

COMPILER ERROR MESSAGES

If the compiler detects an error in the G-Pascal program it will return one of 36 error messages. The messages appear in English, as given below. An upwards arrow will point to the symbol currently being processed when the error was detected. If the compiler has detected the end of the program unexpectedly the arrow will point to the last symbol in the program (for example if the final period (.) is missing).

In some cases the error will be in the symbol that the arrow is actually pointing to (for example, 'Undeclared Identifier' or 'Number out of range'). However in a lot of cases the error will be an error of omission (For example, the message '; expected' usually means that a semicolon has been omitted from the end of the previous line). In these cases the error message usually refers to some problem in the previous statement or clause. Therefore, if the meaning of an error is not obvious, *list and examine carefully the last ten or so lines prior to and including the line containing the error.*

All errors are fatal – in other words as soon as an error occurs the compilation is halted and the compiler automatically returns to the Editor so that the error can be corrected. As G-Pascal compiles very rapidly detection and correction of errors is a quick and easy process.

The list of error messages below is intended as a guide to the usual circumstances surrounding a given error – they should help in understanding a particular error.

As the error messages are in upper and lower case the compiler automatically switches the character set to upper and lower case before displaying errors.

= expected

The compiler is processing a CONST declaration and is expecting an '=' sign to come between the constant name and its value. (e.g. CONST TRUE = 1;).

:= expected

The compiler is processing an assignment statement or a FOR statement and is expecting a ':=' to follow the variable name. (e.g. K : 1;).

; expected

The compiler is expecting a semicolon. It is probably missing from the end of the previous statement.

; or END expected

The compiler is processing a compound statement – that is, one beginning with a BEGIN and ending with an END. It has come to the end of a statement and now expects either a semicolon followed by another statement, or the word END.

, expected

The compiler is processing a list of arguments and now expects a comma, followed by another argument. (e.g. CURSOR (6, 7);).

, or : expected

The compiler is processing a CASE statement and is expecting a comma or colon to follow the case selector, or the compiler is processing a VAR declaration and is expecting a comma or colon to follow the previous identifier. (e.g. VAR X, Y, Z : INTEGER;);

(expected

The compiler is processing a statement that requires arguments to be supplied in round brackets (for example, arguments to a user- declared procedure or function, or arguments to a built-in procedure such as CURSOR). However it cannot find the opening parenthesis.

) expected

The compiler has finished processing a list of arguments and now expects a closing parenthesis, however it cannot find one. This error may appear if an argument is mistyped or a comma which should separate arguments is omitted.

[expected

The compiler is processing an array name (or an array declaration) and now expects a subscript inside square brackets.

] expected

The compiler has finished processing an array subscript and now expects the closing square bracket.

***) expected**

The compiler has reached the end of the program but is still in the middle of processing a comment. The probable cause of this is that in the program a comment has commenced with '(*' but was not terminated with its corresponding '*)'.

. expected

There is no period following the final END in the program.

BEGIN expected

The compiler is processing a block and now expects the word BEGIN to mark the start of the statements in that block. If they have not already been declared this message also means that the compiler will also accept CONST, VAR, PROCEDURE or FUNCTION declarations. This error is usually caused by either forgetting to put the word BEGIN before the statements in a Procedure, Function or main program, or by making a mistake in the CONST or VAR declarations (such as misspelling CONST for example).

Compiler limits exceeded

It is unlikely this error will appear. However if it does simplifiy the statement that it is currently processing.

Constant expected

The compiler is expecting a constant, in other words one of: a number (such as 20), a hex constant (such as $ABCD), a string constant (such as "xyz"), or an identifier declared as a constant in a CONST declaration.

Data type not recognised

The compiler is processing a VAR declaration and has not found either the word INTEGER or CHAR.

DO expected

The compiler is processing a WHILE statement and now expects the word DO. (e.g. WHILE X > 50 DO ...).

Duplicate identifier

The program is attempting to use the same name twice for an identifier under illegal circumstances. It is permitted to use the same name twice if one occurrence is a 'global' declaration (at the start of the program) and another declaration is 'local' (inside a procedure or function). The same name cannot be used twice within the same 'group' of declarations, for example VAR FRED, FRED : INTEGER; would be an error.

Identifier expected

The compiler is processing a CONST, VAR, PROCEDURE, or FUNCTION declaration, or the argument to the ADDRESS function, and is now expecting a user-supplied identifier. (e.g. VAR FRED).

Illegal factor

The compiler is processing an expression and finds an illegal factor. This could be caused by an illegally constructed arithmetic expression (e.g. 5 + *), a missing argument to a procedure or function (e.g. WRITE ()), or a missing expression where one is expected (e.g. IF THEN).

Illegal identifier

An identifier has occurred in a context in which it was not expected.

Incorrect string

The compiler has detected that a string literal is not where it should be or is not terminated. In the case of the LOAD and SAVE statements this error will occur if there is no file name where one is expected, otherwise the error is caused by an opening quote symbol on a line but no corresponding closing quote symbol. (e.g. "HELLO). In this example the upwards arrow would point to the 'H' in 'HELLO'.

Incorrect symbol

The compiler believes it is at the end of the program, however it has found more program following the final period.

Literal string of zero length

A string of zero length (i.e. two consecutive quote symbols: "") was encountered.

Memory full

There is insufficient room for your P-codes to follow the source program. Reduce the size of your program, or place the P-codes at a different address than the default one of at the end of the source program. See the section on the %A compiler directive for details on how to do this.

Number out of range

The compiler is processing a decimal or hexadecimal literal and has found that it is too large (or too small). The allowable range for decimal integers is –8388608 to +8388607. The allowable range for hexadecimal integers is $0 to $FFFFFF.

OF expected

The compiler is processing a CASE statement and now expects the word OF. (e.g. CASE recordtype OF ...).

Parameters mismatched

The compiler is processing the invocation of a user-declared Procedure or Function and has detected that the number of arguments supplied to the Procedure or Function invocation does not agree with the number of arguments declared for the Procedure or Function.

Stack full

The compiler's internal stack has overflowed due to processing too many nested procedures, functions or expressions. This message is very rare, however if it does occur the problem can be corrected by re-writing the program with less 'nesting'. (A nested procedure, for example, is a procedure within a procedure within a procedure etc.).

String literal too big

The compiler has encountered a string literal in an expression, however it is more than 3 characters long. String literals of more than 3 characters are only allowed in the LOAD, SAVE and WRITE statements.

Symbol table full

The program has too many user-declared identifiers (in other words, CONST, VAR, PROCEDURE or FUNCTION names). There are three possible solutions to this problem. First, reduce the number of variables if possible. Second, reduce the length of identifiers (with the Editor's Replace command). Third, make identifiers 'local' to procedures or functions as much as possible. Local identifiers (in other words, CONST, VAR, PROCEDURE or FUNCTION declarations which are *inside* other procedures or functions) only occupy room in the symbol table whilst that procedure or function is being processed.

Each user-declared identifier occupies 10 bytes on the symbol table plus the length of the identifier. In other words, the word FRED would occupy 14 bytes on the symbol table – 10 plus 4 letters in 'FRED'. The total length of the symbol table is 4096 bytes so there is room for 256 identifiers if each is 6 characters long.

THEN expected

The compiler is processing an IF statement and now expects a 'THEN' to follow the conditional expression. (e.g. IF A = 5 THEN ...).

TO or DOWNTO expected

The compiler is processing a FOR statement and now expects the words TO or DOWNTO. (e.g. FOR x := 1 TO 10 DO ...).

Type mismatch

This error rarely occurs because G-Pascal allows mixed type operations in general. For example you may assign a integer variable to a char variable. However this error will occur if the program either: a) attempts to read a string into an integer array, or b) attempts to use a constant name on the left-hand side of an assignment statement.

Undeclared identifier

The identifier to which the upwards arrow is pointing has not been declared. Possibly it is misspelt, or no CONST, VAR, PROCEDURE or FUNCTION declaration exists for it. *All identifiers MUST be declared before they are referenced, so all PROCEDURE and FUNCTION declarations must precede any attempts to refer to them.*

Use of procedure identifier in expression

The compiler is processing an expression and finds the name of a procedure where the name of a variable, constant or function should appear.

RUN-TIME ERROR MESSAGES

G-Pascal has only five run-time error messages – these are explained below. A run-time error message is one that occurs while a program is running rather than compiling. When one of these messages occurs it will be followed by the words: 'Error occurred at P-code xxxx' where xxxx is the address of the P-code (instruction) currently being executed. If the meaning of the error is not immediately obvious, write down the P-code address and then recompile the program, asking for a listing during the compilation. By referring to the P- code addresses listed at the side of the compilation listing you can isolate which statement caused the error message.

Break ...

The program has been aborted by pressing the RUN/STOP key.

Divide by zero

An attempt has been made to divide by zero, which is not permitted. (The MOD operator, which is a form of divide, may also cause this error).

Illegal Instruction

This message should not appear in normal operation. It means that the P-code interpreter has encountered an invalid instruction (P- code). This could be caused by a program self-destructing somehow (perhaps by an array subscript going out of its allowable bounds) or invoking an 'independent' procedure which had not been loaded into memory at the correct address.

Illegal parameter in function call

One of the built-in graphics or sound effects procedures or functions has been called with a parameter (argument) outside its allowable range. For example, referring to sprite 9 in a SPRITE statement will cause this error, or voice 4 in a VOICE statement. Not all parameters are checked in this way – generally speaking the range check is carried out where an incorrect argument could have disastrous effects on the system if it went unchecked. In other cases the supplied value is truncated to fit into the required range – for example the filter frequency specified for the SOUND statement should be in the range 0 to 2047 – if the value exceeds 2047 then the supplied value modulus 2048 is taken.

Stack full

There is no more room on the stack for variable data. The stack consists of 3,790 bytes (this is the equivalent of 1,263 INTEGER variables or 3,790 CHAR variables). This could be caused by either declaring too many variables (e.g. an array of 2,000 integers), or by calling procedures or functions recursively too often. Each time a procedure or function is invoked 6 bytes are reserved on the stack for 'linkage' data (such as the return address – the address from which the procedure was called) plus any 'local' variables declared for that procedure or function.

FILE HANDLING

If you enter 'F' (for Files) from the Main Menu you will see:

```
(L)oad, (A)ppend, (P)rint, (D)os,
(S)ave, (N)oprint, (V)erify, (Q)uit,
(E)dit, (C)atalog, (O)bject?
```

This is called the 'Files Menu'. To choose one, press the letter corresponding to your choice (the letter in brackets). Do *not* press RETURN as well. To leave the Files Menu and return to the Main Menu press 'Q' (for Quit).

The choices are:

Load

This will load a G-Pascal source program from disk or cassette. *The loaded program will replace any currently in memory.* If you do not want to load anything just press RETURN. You will see:

```
(C)assette or (D)isk?
```

Press C for Cassette or D for Disk. Any other character will return you to the Files Menu.

You will then see:

```
File name?
```

Type in the file name of the file to be loaded (the program name, in other words) and press RETURN. When loading, appending or verifying from cassette the file name is optional – if any file will do just press RETURN. The program will now load.

Warning: There is *no* check to see if the program is too big. It is your responsibility to not load programs that are too big. This would not normally happen unless the file was created independently of G-Pascal. If you do load a file that is too big you will probably 'clobber' part of G-Pascal.

Append

This appends a program to the back of one that is currently in memory. If you do not want to append anything just press RETURN. Append is a powerful way of copying useful procedures or functions from one program to another. You could build a 'library' of useful procedures and functions and just Append them at the appropriate points in your program, thus saving a lot of typing and debugging effort. Apart from the fact that the Appended program goes at the end of the one currently in memory, Append works the same as Load so see the 'Load' command for what to type next. Be careful when appending that you do not append too much to fit into memory. G-Pascal does not check when appending that there is sufficient room for the new file.

Save

This saves your program on disk or cassette. *If saving to disk then the saved program will replace any of the same name on disk.* If you do not want to save your program just press RETURN. Apart from the fact that the program is being saved, not loaded, the Save command works the same as the Load command so see the 'Load' command for what to type next.

Verify

This verifies that the program recently saved on disk or cassette has saved properly. It reads the file on disk or cassette and compares it with the image in memory. Naturally for this to work properly it must be done before any changes are made to the program in memory. Apart from the fact that the file is being verified, not loaded, the Verify command works the same as the Load command so see the 'Load' command for what to type next.

Object

This allows you to save your P-codes to disk or cassette. (Compiled files are commonly called 'object' files). If you have not done a error-free compile you will get an error message. You will be asked for your object file name. Just enter its name and press RETURN. If you have selected 'Object' by mistake just press RETURN only. *If saving to disk then the saved object file will replace any file of the same name currently on disk.* Please be careful that you do not save your object under the same name as your 'source' program – that is, your G-Pascal statements. You cannot convert object code back to source code so if you lose your source code you are in trouble. However if you lose your object code you can always recompile from the source code. The object code is completely relocatable so it can be loaded at any address
without change. You would do this for using the code as an independent module (see section on 'Independent Modules') or for use with the run-time system (available separately).

Edit

This takes you directly to the Editor. It is useful to press 'E' while a program is loading then you will be taken to the Editor as soon as the load completes.

Print

This directs output to your printer. Further screen displays will also appear on the printer unless 'Noprint' (below) is selected or you press RUN/STOP/RESTORE. Printing is also cancelled at the end of a program run.

To print your program just press 'P' for print (this option), then E (for Edit) and L (RETURN) to list the program.

If you have no printer plugged into the serial port then you will get an error message.

Noprint

This cancels any previous 'Print' command. If you were not printing or have no printer anyway this has no effect.

Dos

This option allows commands to be sent to the DOS (Disk Operating System) in order to accomplish disk-oriented operations such as deleting files and so on. After selecting this option G-Pascal will reply:

```
Command?
```

At this stage press RETURN if you do not want to send any command to the DOS, otherwise enter the command (as described in the DOS manual), and then press RETURN.

Catalog

This option will produce a catalogue (directory) listing of your disk. In other words, it will display on the screen the names of all the files on your disk. Also displayed will be the size of each file and the amount of room available on the disk.

Meaning of error codes

If an error is encountered in loading, saving, verifying etc. an error code number will be returned. The meanings of the common error codes are as follows:

1 – Too many open files
2 – File already open
3 – File not open
4 – File not found
5 – Device not present
6 – File is not an input file
7 – File is not an output file
8 – File name is missing
9 – Illegal device number
10 – Unrecoverable read error / mismatch
20 – Checksum error
40 – End of file
80 – End of tape/Device not present

OPENING AND CLOSING FILES

G-Pascal allows you to open and close files, direct output to a file or obtain input from a file. This is achieved with the following statements:

OPEN (file, device, channel, "file-name")

The file number may be from 1 to 255 and is used to identify the file in subsequent GET, PUT and CLOSE statements.

The device number identifies the type of device that the OPEN applies to. It is normally 8 for a disk, 4 for a printer and 1 for the cassette.

The 'channel' (otherwise known as the 'secondary address') is used to send secondary information to the device. It may be from 2 to 15 for a disk where 15 is the disk 'command' channel. Other numbers may be used for printers for special purposes such as plotting, graphics, lower case and so on.

The "file-name" must be a string of at least one character inside quote symbols. It represents the file name when opening a disk file, or the command when sending a disk command. In the case of printers it is ignored, so just a single space (" ") will do.

The result of the OPEN is returned in the reserved word INVALID – if INVALID is non-zero after the OPEN then an error has occurred and INVALID contains the actual number corresponding to the type of error that occurred.

For example, to open file 5 as the printer (device 4) you would say:

```
OPEN (5, 4, 0, " ");
```

To open a disk file for output as file number 10 you would say:

```
OPEN (10, 8, 2, "0:DATA,S,W");
```

PUT (file-number) N.B. must PUT(0) in between redirecting to another file.

PUT is used to direct output (from WRITE and WRITELN statements) to a previously opened file. All subsequent WRITE and WRITELNs will direct their output to the nominated file an error will occur if the file is not open or is not an output file.

To re-direct output to the TV screen (the default condition) the statement: PUT (0) must be issued.

The result of issuing a PUT is stored in the function INVALID. If INVALID is zero then the PUT was OK, otherwise it contains the number of the error that occurred.

GET (file-number)

GET is used to receive input (to a READ statement) from a previously opened file. All subsequent READs will receive input from the nominated file an error will occur if the file is not open or is not an input file.

To receive output from the keyboard again (the default condition) the statement: GET (0) must be issued.

The result of issuing a GET is stored in the function INVALID. If INVALID is zero then the GET was OK, otherwise it contains the number of the error that occurred.

Both GET (0) and PUT (0) function identically they issue a 'clear channel' command to the Kernal, resetting both input and output to the default operation of keyboard and screen respectively. The files are still open, however and may be re-accessed with further GET and PUT statements.

CLOSE (file-number);

CLOSE is used to close the nominated file. For example, CLOSE (15) closes file number 15. All open files should be closed, although G-Pascal automatically does a 'close all files' at the completion of each run.

THE GRAPHICS COMMAND

The GRAPHICS command is a general-purpose command which accomplishes 18 different actions. The GRAPHICS command is supplied with pairs of arguments, where the first is the action number and the second is the value to be passed to that action routine. For example: GRAPHICS (*multicolour, on, extended background, off*); For ease of comprehension, and to make your programs self-documenting, we strongly suggest that the action numbers and colour names etcetera be defined in a series of CONST definitions at the start of the program, as shown below. If you do this, then the command: GRAPHICS (BORDERCOLOUR, RED) and GRAPHICS (11, 2) function identically, however the former makes the program much easier to read. The exact spelling of the action names is up to you as they are not reserved words, however your spelling must be consistent throughout your program. The descriptions of the various actions over the next few pages use the spelling given below. We suggest that you key in the CONST definitions given below, plus the ones for the SPRITE, VOICE and SOUND functions, and keep them on disk or cassette in a separate file for ease of use in the future. If you wish you can omit the action names for any actions that a particular program is not going to use – for example if you do not plan to use bit-mapped graphics you could omit: BITMAP = 1.

```
CONST
BITMAP = 1;                    MULTICOLOUR = 2;
EXTENDEDBACKGROUND = 3;        COLUMNS40 = 4;
LINES25 = 5;                   DISPLAYSCREEN = 6;
BANKSELECT = 7;                CHARGENBASE = 8;
VIDEOBASE = 9;                 CHARACTERCOLOUR = 10;
BORDERCOLOUR = 11;             BACKGROUNDCOLOUR0 = 12;
BACKGROUNDCOLOUR1 = 13;        BACKGROUNDCOLOUR2 = 14;
BACKGROUNDCOLOUR3 = 15;        SPRITECOLOUR0 = 16;
SPRITECOLOUR1 = 17;            WRITEBASE = 18;
BLACK = 0;      WHITE = 1;     RED = 2;
CYAN = 3;       PURPLE = 4;    GREEN = 5;
BLUE = 6;       YELLOW = 7;    ORANGE = 8;
BROWN = 9;      LIGHTRED = 10; DARKGREY = 11;
MEDIUMGREY = 12; LIGHTGREEN = 13; LIGHTBLUE = 14;
LIGHTGREY = 15; ON = 1;        OFF = 0;
```

GRAPHICS (BITMAP, ON);
Turns bit-mapped (high resolution) graphics mode on.

GRAPHICS (BITMAP, OFF);
Turns bit-mapped graphics off – returns to character graphics mode.

GRAPHICS (MULTICOLOUR, ON);
Turns on multi-colour display mode – can be used with character graphics or bit-mapped graphics.

GRAPHICS (MULTICOLOUR, OFF);
Turns off multi-colour display mode.

GRAPHICS (EXTENDEDBACKGROUND, ON);
Turns on extended background display mode.

GRAPHICS (EXTENDEDBACKGROUND, OFF);
Turns off extended background display mode.

GRAPHICS (COLUMNS40, ON);

Displays 40 columns of text (the default).

GRAPHICS (COLUMNS40, OFF);

Displays 38 columns of text (border contracts). Normally used in conjunction with sideways smooth scrolling.

GRAPHICS (LINES25, ON);

Displays 25 lines of text (the default).

GRAPHICS (LINES25, OFF);

Displays 24 lines of text (border contracts). Normally used in conjunction with vertical smooth scrolling.

GRAPHICS (DISPLAYSCREEN, ON);

Enables normal display of text or graphics on the screen (default condition).

GRAPHICS (DISPLAYSCREEN, OFF);

Blanks out the screen. The border colour is displayed on the whole screen. Results in a faster execution of programs. Normally used to hide the contents of a screen until it is ready to be viewed.

GRAPHICS (BANKSELECT, bank);

Used to bank-select the VIC (Video Interface Chip) to a nominated 16K bank of memory. 'Bank' can be from 0 to 3, where 0 is the normal mode.

Bank	*Range*
0	*$0000-$3FFF*
1	*$4000-$7FFF*
2	*$8000-$BFFF*
3	*$C000-$FFFF*

GRAPHICS (CHARACTERCOLOUR, colour);

Sets the colour for all subsequent characters to be displayed with the WRITE command. The colour definitions are given at the start of this section. Colours range from 0 (black) to 15 (light grey).

GRAPHICS (BORDERCOLOUR, colour);

Sets the border colour to the nominated value.

GRAPHICS (BACKGROUNDCOLOUR0, colour);

Sets the background colour (normal background).

GRAPHICS (BACKGROUNDCOLOUR1, colour);

Sets the colour of extended background colour 1. (Used with extended background mode).

GRAPHICS (BACKGROUNDCOLOUR2, colour);

Sets the colour of extended background colour 2. (Used with extended background mode).

GRAPHICS (BACKGROUNDCOLOUR3, colour);

Sets the colour of extended background colour 3. (Used with extended background mode).

GRAPHICS (SPRITECOLOUR0, colour);

Sets multi-colour sprite colour 0. (Used with multi-colour sprites).

GRAPHICS (SPRITECOLOUR1, colour);

Sets multi-colour sprite colour 1. (Used with multi-colour sprites).

GRAPHICS (CHARGENBASE, base);

Used to nominate where the character patterns for character display mode are to be found. 'Base' should be from 0 to 7, where 2 is normal upper-case and graphics characters, and 3 is upper and lower- case characters. Other values could be used if you set up your own character memory. Each number represents a 2K boundary of memory, so that the locations of character memory are as follows:

Base	*Location of character memory*
0	*$0000-$07FF*
1	*$0800-$0FFF*
2	*$1000-$17FF (ROM IMAGE in BANK 0 and 2 – default)*
3	*$1800-$1FFF (ROM IMAGE in BANK 0 and 2)*
4	*$2000-$27FF*
5	*$2800-$2FFF*
6	*$3000-$37FF*
7	*$3800-$3FFF*

If you are not using Bank 0 then the Bank address (refer to the BANKSELECT description above) must be added to the above.

GRAPHICS (VIDEOBASE, base);

Used to nominate where screen memory starts. 'Base' should be from 0 to 15, where 1 is normal. Each number represents a 1K boundary of memory. You would normally change the location of screen memory if you wanted to do 'page flipping' – that is, animation or other special effects by keeping text or graphics on two or more separate areas of screen memory and flipping from one to the other. *Warning:* this command changes the area of screen memory that is *displayed* – it does not change the area of screen memory that the WRITE command actually puts text onto. To change this you need to use the *GRAPHICS (WRITEBASE, base)* command so that subsequent WRITEs write to the correct screen memory area.

Base	*Starting location*
0	*$0000*
1	*$0400*
2	*$0800*
3	*$0C00*
4	*$1000*
5	*$1400*
6	*$1800*
7	*$1C00*
8	*$2000*
9	*$2400*
10	*$2800*
11	*$2C00*
12	*$3000*
13	*$3400*
14	*$3800*
15	*$3C00*

If you are not using Bank 0, then the bank address (refer to the BANKSELECT description previously) must be added to the starting location.
e.g.

```
GRAPHICS (VIDEOBASE, 15, WRITEBASE, 15);
```

GRAPHICS (WRITEBASE, base);

Used to nominate which piece of screen memory that the WRITE command will write to. 'Base' should be from 0 to 15, where 1 is normal. Each number represents a 1K boundary of memory. You would normally use this command in conjunction with the GRAPHICS (VIDEOBASE, base) command (see previously) in order to write to a different area of memory then usual when doing 'page flipping'. *Warning:* this command changes the area of screen memory that the WRITE command *writes to* – it does not change the area of screen memory that is displayed. To change this you need to use the *GRAPHICS (VIDEOBASE, base)* command so that the correct screen memory area is displayed.

SPRITE PROCESSING OVERVIEW

G-Pascal provides extensive support for sprites. Sprites (otherwise known as 'movable object blocks' – MOBs) are shapes of up to 24 dots across and 21 dots down which can be placed anywhere on the screen very easily without affecting any display underneath. You can have up to 8 sprites displayed at once – all moving independently, having unique shapes, and coloured independently.

First, to show how easy sprites are to program in G-Pascal, try this small program on your Commodore 64. This program contains the essence of successful sprite programming, namely: a) define the shape of your sprite (DEFINESPRITE); b) point a particular sprite to the shape you have defined (POINT); c) allocate a colour to the sprite (COLOUR); and finally: d) move it about on the screen (MOVESPRITE).

```
CONST COLOUR = 1; POINT = 2; YELLOW = 7;
BEGIN
  DEFINESPRITE (128, $FFFFFF, $F0000F, $F0000F, $FFFFFF);
  SPRITE (1, POINT, 128, 1, COLOUR, YELLOW);
  MOVESPRITE (1, 50, 50, 256, 256, 180);
END.
```

This example illustrates how a simple 6-line program is all that is needed to define a sprite and move it around on the screen. Having tried this example, experiment with different colours, different coordinates in the MOVESPRITE command, and different shapes in the DEFINESPRITE command.

Since the sections that follow which describe the various sprite- handling commands and functions may seem a bit daunting at first we will describe briefly the purpose of the various commands and relate them to each other.

You define sprite shapes with DEFINESPRITE. You set up a sprite's colour, point it to a shape definition, activate it, expand it in the x and y directions if desired, and put it in front of or behind the background with the SPRITE command. You position a stationery sprite on the screen with POSITIONSPRITE or move a sprite around automatically with MOVESPRITE. You can make a sprite sequence through different shapes (to give animation effects) with ANIMATESPRITE. You can stop a moving sprite with STOPSPRITE and start it again with STARTSPRITE. You can tell G-Pascal to stop sprites moving if they collide with SPRITEFREEZE and check whether this has happened with FREEZESTATUS. You can also check whether sprites have collided with the SPRITECOLLIDE function, and whether a sprite has hit the background with the GROUNDCOLLIDE function. You can see whether or not a sprite is moving with SPRITESTATUS, and check its coordinates on the screen with the SPRITEX and SPRITEY functions.

The SPRITE command

The SPRITE command is a general purpose command that accomplishes 7 different actions relating to sprites, such as controlling a sprite's colour, whether or not it is displayed on the screen, whether to expand it in the x or y direction and so on. The SPRITE command accepts arguments in groups of three – the first is always the sprite number, from 1 to 8. (The Commodore 64 Programmer's Reference Guide – which should be consulted for more details about sprite characteristics – refers to sprites as being numbered from 0 to 7, however G-Pascal numbers sprites from 1 to 8.) The second argument to the SPRITE command is an 'action number' from 1 to 7. In a similar way to the GRAPHICS command, we will assume that the action numbers are defined in a CONST declaration as given below. The third argument to the SPRITE command is the value to be passed to the action routine. For example: SPRITE (2, *colour, green, 2, expandx, on, 2, active, on*);

```
CONST
  COLOUR=1; POINT=2;
  MULTICOLOURSPRITE = 3; EXPANDX = 4;
  EXPANDY = 5; BEHINDBACKGROUND = 6;
  ACTIVE = 7;
  ON = 1; OFF = 0;
```

SPRITE (sprite-number, COLOUR, colour);

Sets the colour of the nominated sprite. The colour definitions are given at the start of the GRAPHICS command section. Colours range from 0 (black) to 15 (light grey).

SPRITE (sprite-number, POINT, position);

Points the sprite to its appropriate pattern definition. This command is used to set or change the appearance of a sprite on the screen. The pattern number must have been previously defined with a DEFINESPRITE command or a random shape will appear. The position ranges from 0 to 255 where each position number represents a 64 byte boundary in memory (in the current bank). If using bank zero then positions less than 16 would not normally be used. Also positions in the range of 64 to 127 clash with the ROM images of the character sets (addresses $1000 to $2000) and should not be used. Normally, sprite definitions would start at position 128 onwards. If using the ANIMATESPRITE command then this command is not necessary, as ANIMATESPRITE overrides the position set by this command.

SPRITE (sprite-number, MULTICOLOURSPRITE, ON);

Enables this sprite to be in multi-colour mode. When using this mode the sprite auxiliary colours are defined using the GRAPHICS command, actions SPRITECOLOUR0 and SPRITECOLOUR1.

SPRITE (sprite-number, MULTICOLOURSPRITE, OFF);

Returns this sprite to single-colour mode. This is the default condition.

SPRITE (sprite-number, EXPANDX, ON);

Doubles the size of this sprite in the X axis (horizontally).

SPRITE (sprite-number, EXPANDX, OFF);

Returns this sprite to its unexpanded display in the X axis.

SPRITE (sprite-number, EXPANDY, ON);

Doubles the size of this sprite in the Y axis (vertically).

SPRITE (sprite-number, EXPANDY, OFF);

Returns this sprite to its unexpanded display in the Y axis.

SPRITE (sprite-number, BEHINDBACKGROUND, ON);

Displays this sprite behind any 'background' data on the screen, such as text or bit-mapped graphics drawings.

SPRITE (sprite-number, BEHINDBACKGROUND, OFF);

Displays this sprite in front of background data.

SPRITE (sprite-number, ACTIVE, ON);

Displays this sprite on the screen so that it can be seen. Note that the sprite may not be visible if its display co-ordinates fall outside the screen area (in other words, if it is in the border area). The sprite should have been positioned on the screen previously with the POSITIONSPRITE command. If you use the MOVESPRITE command (explained further) then activating the sprite with this command is unnecessary.

SPRITE (sprite-number, ACTIVE, OFF);

Turns this sprite off so that it can no longer be seen on the screen. This does not stop a sprite moving if it is moving under MOVESPRITE control - it just makes it invisible.

GENERAL SPRITE COMMANDS

The following commands control other aspects of sprites that are not handled by the SPRITE command, such as defining a sprite's shape, moving from one point on the screen to another and so on. Most of them use a sprite number as an argument, in which case a sprite number in the range 1 to 8 is supplied.

DEFINESPRITE (position, row1, row2, row3 .... row21);

This is used to define the shape of a sprite. It does not use a sprite number as shapes are independent of sprites – all sprites can use the same shape if desired, or perhaps a given shape may not be used by any sprite at a given moment. The 'position' nominates the location of the shape definition in memory. Each sprite definition takes 64 bytes so each 'position' number refers to a 64 byte boundary within the current video bank. To avoid clashes with screen memory, character memory, and system work areas, sprite definitions should normally start at 128 (this is address $2000 in memory). A sprite definition consists of up to 21 rows of sprite shape data. Each row consists of 24 'bits' (dots). As an integer in G-Pascal is three bytes long (24 bits) each row is defined by one integer. You would normally define sprites as a series of hex constants, but decimal constants or even variables could be used if desired. Any unused rows at the end can be omitted – they will be assumed to be zero (background). For example, a definition of a straight line would be:

```
DEFINESPRITE ( 128, $FFFFFF );
```

A simple box shape would be:

```
DEFINESPRITE ( 129, $FFFFFF, $C00003, $C00003, $FFFFFF);
```

POSITIONSPRITE (sprite-number, x, y);

This command positions a sprite at the nominated x and y co-ordinates on the screen. It is used for placing stationery sprites on the screen. If you want the sprite to move, use the MOVESPRITE command instead. If a sprite is moving under MOVESPRITE control when you give a POSITIONSPRITE command then the POSITIONSPRITE will cancel the sprite's movement.

For example, to position sprite 5 at coordinates 100, 140 on the screen:

```
POSITIONSPRITE (5, 100, 140);
```

MOVESPRITE (sprite-number, x, y, x-increment, y-increment, moves);

This command:

a) Positions the nominated sprite at the nominated x and y co-ordinate.
b) Turns the sprite on so that it can be seen.
c) Sets its SPRITESTATUS to 1 to indicate that it is moving.
d) Moves it by the nominated increments 'moves' times.
e) When it has moved the nominated number of times, stops the sprite's movement and sets its SPRITESTATUS to 0 to indicate that it has finished moving.

The sprite should have been pointed to a sprite definition, *or* an ANIMATESPRITE command given for the sprite. The movement of the sprite is carried out by 'interrupt driven' routines asynchronously with your program. In other words, once the sprite has been started in motion by the MOVESPRITE command it will move independently of the program. All the program has to do is check its SPRITESTATUS to find whether it has finished its planned movement or not. It is not necessary to wait for the sprite to stop moving before doing something else with the sprite. It can be stopped with the STOPSPRITE command which will 'freeze' it wherever it currently is on the screen. It can be hidden by inactivating it (e.g. SPRITE (5, ACTIVE, NO);). Alternatively another MOVESPRITE command can be given which will cancel the current one. The x-increment and y-increment are specified in 1/256 of a sprite position. This is to allow very fine tuning of the rate at which a sprite moves. For example, an increment of 256 means the sprite will move exactly one pixel per frame (a frame is about 1/60 of a second). An increment of 1024 means the sprite will jump four pixels per frame (4 times 256 is 1024). This will make the sprite look a bit jerky. An increment of 1 will mean that the sprite will move one pixel every 256 frames (about every 5 seconds). An increment of zero means the sprite will not move in that direction at all. The 'moves' argument specifies the number of frames that the sprite will move before stopping automatically. When the number of moves has been reached the sprite is stopped and its SPRITESTATUS set to zero so that the program knows that the sprite has stopped. Note that this does not mean that the sprite becomes invisible – it just stops moving. The maximum number of moves that can be specified is 32768. The sprite's position at any time can be established by the SPRITEX and SPRITEY functions.

ANIMATESPRITE (sprite-number, frame-count, position1, position2 ...);

This command allows a sprite which is in movement by a MOVESPRITE command to sequence through a series of sprite definitions automatically. A use for this could be to give the appearance of a person running, by having half a dozen (say) different sprite definitions of a person in different stages of running and nominating the order in which they are to be displayed. ANIMATESPRITE is normally given *before* a MOVESPRITE as the actual movement is carried out by the MOVESPRITE command. Up to 16 positions can be nominated. A position of zero should not be used. See the DEFINESPRITE command for more details about the range of numbers that can be chosen for positions. The 'frame-count' nominates the number of 'frames' that are to pass before the next position in sequence is displayed. The higher the frame-count, the more slowly the sprite will change its shape. For experimental purposes we suggest a frame-count in the range of 5 to 10. (Each frame is about 1/60 of a second). The maximum frame count is 255. e.g.

```
ANIMATESPRITE (3, 5, 128, 129, 130, 131, 132);
```

SPRITESTATUS (sprite-number);

This function returns the status of a nominated sprite, namely whether or not it is moving. If the status is zero then the sprite has:

a) Never been moved with a MOVESPRITE command *or*
b) Completed a move specified by a MOVESPRITE command *or*
c) Been stopped with a STOPSPRITE command *or*
d) Been stopped by a collision under SPRITEFREEZE control.

If the status is 1 then the sprite is currently moving under control of a MOVESPRITE command. SPRITESTATUS may be used as a boolean function as it will, in effect, return TRUE if the sprite is moving and FALSE if it is stationery. (G-Pascal considers zero to be FALSE, and non- zero to be TRUE).

SPRITECOLLIDE

This functions returns the contents of the sprite-to-sprite collision register. This is a hardware register in the VIC chip. If the result is zero then no sprites are colliding with each other. If the result is non-zero then two or more sprites are colliding with each other. *You may prefer to let G-Pascal automatically check for sprite collisions for you and automatically stop any sprites involved in a collision. In this case, you should use the SPRITEFREEZE command described further on.* If you wish to establish which sprite is involved then the result must be ANDed with the numbers in the following table. *Warning:* checking the SPRITECOLLIDE status will *clear* the collision register ready for the next collision. If you want to do a series of tests on the result then the result should be saved into an intermediate variable and the test carried out on the variable. You should *not* refer to the SPRITECOLLIDE function if you are using the SPRITEFREEZE command. In this case you should check the FREEZESTATUS function which returns collision bits in the same way as the SPRITECOLLIDE function (i.e. as in the table on the next page).

Sprite	*Collision bit value*
1	*1*
2	*2*
3	*4*
4	*8*
5	*16*
6	*32*
7	*64*
8	*128*

For example, to check whether sprite 1 or 5 was involved in a collision with another sprite (or each other):

```
X := SPRITECOLLIDE;
IF (X AND 1 < > 0) OR (X AND 16 < > 0) THEN
BEGIN
(* GOT A COLLISION *)
END;
```

Alternatively (and more simply) you can add together the values for any sprites you are interested in so the above example could read:

```
IF SPRITECOLLIDE AND 17 THEN
BEGIN
(* GOT A COLLISION *)
END;
```

Note that if more than two sprites are involved in collisions at one time then you cannot tell from the SPRITECOLLIDE function which sprite is colliding with which. (This is a hardware limitation). In that case you must work out which is colliding with which by comparing their co-ordinates on the screen (by using the SPRITEX and SPRITEY functions for each sprite). Of course it may not matter which sprite has hit which, for example if sprite 1 is the player's ship and all other sprites are 'aliens' then it would suffice to detect that sprite 1 is involved in a collision (i.e. IF SPRITECOLLIDE AND 1 THEN ...).

GROUNDCOLLIDE

This functions returns the contents of the sprite-to-background collision register. This is a hardware register in the VIC chip. If the result is zero then no sprites are colliding with the background. If the result is non-zero then two or more sprites are colliding with the background. If you wish to establish which sprite is involved then the result must be ANDed with the numbers in the table above (the same values as for SPRITECOLLIDE). *Warning:* checking the GROUNDCOLLIDE status will *clear* the collision register ready for the next collision. If you want to do a series of tests on the result then the result should be saved into an intermediate variable and the test carried out on the variable.

STOPSPRITE (sprite-number);

This command stops the movement of the nominated sprite, assuming that it has previously been moved by the MOVESPRITE command. If the sprite is not moving already this command will have no effect. STOPSPRITE will set the appropriate SPRITESTATUS to zero. It is advisable to do a STOPSPRITE before checking a sprite's position, as the sprite may keep moving while its position is being calculated, for example after detecting a collision. If it is desired to allow the sprite to start moving again and complete the original move specified by the MOVESPRITE command in the first place, use the STARTSPRITE command. Note that sprites may be stopped automatically if they collide by the SPRITEFREEZE command (described later.)

STARTSPRITE (sprite-number);

This command reinstates the movement specified for a nominated sprite which had been temporarily stopped by a STOPSPRITE command. Do not give a STARTSPRITE for a sprite which had not been moved originally with a MOVESPRITE command or the results may be unpredictable. The intended use for STOPSPRITE and STARTSPRITE is for in a game where a number of sprites are moving about on the screen and the program is waiting for some event to happen, for example a collision being detected, or the player making a different move with the joystick. In this case the program may want to temporarily 'freeze' all relevant sprites (with STOPSPRITE), calculate their positions, and proceed on the basis of what the sprite positions are. Any sprites not affected (for example, those that are not involved in a collision) can then be started again with a STARTSPRITE so they can proceed on their planned courses. Also see the SPRITEFREEZE command for details about how sprites can be automatically stopped by a collision. In this case you would use STARTSPRITE if you wanted sprites to continue on their courses.

SPRITEX (sprite-number);

SPRITEX is a function that returns the x co-ordinate of the nominated sprite. It is particularly useful when used in conjunction with the MOVESPRITE command, as the program may not know where the sprite is on the screen at a given moment. If the sprite is in motion because of a MOVESPRITE command it would be advisable to stop it temporarily with a STOPSPRITE command or the sprite may have moved from the position that is returned by this function. A powerful use of the SPRITEX and SPRITEY functions is for changing the direction of a sprite. You could use the sprite's current position when specifying a new MOVESPRITE command in a different direction.

SPRITEY (sprite-number);

SPRITEY is a function that returns the y co-ordinate of the nominated sprite. It is normally used in conjunction with SPRITEX. For example:

```
XPOSITION := SPRITEX (4);
YPOSITION := SPRITEY (4);
```

SPRITEFREEZE (mask);

The SPRITEFREEZE command and its associated function, FREEZESTATUS, provide real-time control over collisions between sprites and other sprites. First, some background on the need for such a command...

Most arcade-style games involve objects moving around the screen. Frequently these objects are spaceships, aliens, missiles and bombs. (Please excuse the bloodthirsty nature of these descriptions). Often the player of a game controls one object (his/her ship) and fires missiles at the other objects which are controlled by the program.

Normally the object of the game is to cause objects to collide, such as a missile with an alien, or avoid a collision, such as an alien with the player's ship. Therefore the subject of 'collision detection' is very important. It is important to correctly register collisions between objects – not only the fact that a collision occurred, but correctly decide which objects collided.

We will assume for our discussion that all the moving objects in question will be implemented as sprites, and so the problem is detection of collisions between sprites. The VIC (Video Interface Chip) inside the Commodore 64 has provision for detection of collisions between sprites and other sprites and returns the details of a collision in a 'hardware' register, known within G-Pascal as SPRITECOLLIDE.

One method of checking for collisions is to periodically check whether SPRITECOLLIDE is non-zero. Unfortunately, a finite time must elapse between such checks, as the program has other things to do as well, such as scorekeeping, checking for player input from the joystick, keyboard or paddles, and other tasks such as moving aliens around on the screen. Therefore it is possible for a collision to go undetected too long – in other words, by the time the program notices the collision the two sprites which collided may have separated again.

A further problem is that the VIC chip tells us which sprites have collided, but not which sprites they have collided with. For example, on the left-hand side of the screen a missile may have hit an alien (which we will call a 'genuine' collision), but on the right- hand side of the screen one alien may pass in front of another (which we will call a 'pseudo' collision). We cannot tell just by examining SPRITECOLLIDE the difference between a genuine and a pseudo collision - the only way is to compare the coordinates of each sprite involved in a collision and see which ones overlap. Because of this necessity it is especially important to detect a collision as soon as it occurs.

Fortunately, the SPRITEFREEZE command provides this capability. It works in conjunction with an 'interrupt' routine built into G- Pascal. The way it works is this: at the start of the game issue a SPRITEFREEZE command with a mask which specifies which sprites may be involved in 'genuine' collisions (e.g. the sprite which will be the missile). This is done by adding together the mask values for all relevant sprites. For example, if we want to detect any collisions involving sprites 7 or 8 (with any other sprites) then the mask would be 64 + 128 = 192. So we would say: SPRITEFREEZE (192);

Then the moment any collision occurs involving the nominated sprites an 'interrupt' is generated which *immediately* transfers control to a special routine inside the G-Pascal interpreter, regardless of what else the program is doing. This routine then places the contents of the sprite-to-sprite collision register in FREEZESTATUS, and then inhibits any further such interrupts (this is to give the program a chance to process the first one). *The routine then stops all sprites that were involved in the collision (by effectively doing a STOPSPRITE). This means that they will stop moving so that their coordinates (at the time of the collision) may be examined by the program.*

All the program has to do is periodically examine FREEZESTATUS – as soon as

it becomes non-zero the program knows that one of the nominated sprites has collided with another sprite. Each 'bit' in FREEZESTATUS represents a sprite (from the table below), so that if sprites 1 and 8 collided, for example, then FREEZESTATUS would be 129. The program would then check each relevant sprite's coordinates (by referring to SPRITEX and SPRITEY) and establish whether a genuine collision has occurred and take appropriate action. Any sprites which were not involved in a genuine collision can be restarted with STARTSPRITE. Once the collision has been processed another SPRITEFREEZE command must be issued so that the process can commence again.

Specifying SPRITEFREEZE (0) is a special case which will inhibit any future interrupts due to sprite collisions.

Sprite	*Mask bit value*
1	*1*
2	*2*
3	*4*
4	*8*
5	*16*
6	*32*
7	*64*
8	*128*

Speeding up sprite movement

There are two ways of moving sprites rapidly around the screen. The first is to specify a large increment in the MOVESPRITE command so that sprites will 'jump' a large distance each frame. (A frame is normally 1/60 of a second). For example, specifying an increment of 1024 will result in sprites moving 4 pixels per frame. The drawback of this approach is twofold – first, the sprites look 'jerky'; secondly, a sprite which moves in large increments may jump over an obstacle that it was supposed to collide with.

The other approach is to speed up the rate of interrupts. The reason for this is that MOVESPRITE works by using an 'interrupt routine' in the G-Pascal interpreter which normally gains control every 1/60 of a second. This routine is responsible for moving sprites around. If this routine is accessed more frequently then sprites will move faster. For example, if interrupts are processed every 1/120 of a second, then sprites will move twice as fast as usual. This will also provide smoother animation than the technique of using larger increments described above. The way to speed up interrupts is to change location $DC05 in memory with a MEMC statement. (This is the high-order byte of Timer A in CIA 1). It normally contains $42 (decimal 66). Therefore to makes sprites move twice as fast you would say: MEMC [$DC05] := 33;

There are some drawbacks to speeding up sprites in this way – the first is that the more frequent interrupts mean that less processing power is available for the program. This becomes a greater problem the faster the interrupts. For example, if location $DC05 was changed to 1, then the processor would spend most of its time processing interrupts and very little time running your program.

The other drawback to speeding up sprites is the problem of collision detection (again). The VIC chip only detects a collision between two sprites when the collision is actually drawn on the screen. If two sprites pass through each other very quickly, then they may have separated before the collision is drawn on the screen (TVs redraw the screen about every 1/25 of a second). If the collision is not drawn then it is not detected. This will lead to timing-dependent bugs – sometimes the collision will be detected (because the raster line happens to be drawing that part of the screen at the time), and sometimes it won't.

MISCELLANEOUS GRAPHICS COMMANDS

The commands and functions described below do not specifically apply to sprites – they provide control over other aspects of the Commodore 64 display and graphics capabilities such as making changes to the display between frames (WAIT), smooth scrolling (SCROLL), hi- resolution (bitmapped) graphics (CLEAR and PLOT), positioning the cursor on the screen (CURSOR), reading the paddles (PADDLE), reading the joysticks (JOYSTICK), setting the clock and reading it (SETCLOCK and CLOCK) and so on.

WAIT (raster-number);

This command suspends operation of your program until the raster on the screen reaches the nominated line. (The 'raster' refers to the actual line currently being drawn on the TV screen). The use of this command is to temporarily delay making some change to the display until the TV is between 'frames' and therefore eliminate any flickering that might occur if the change was made in the middle of a frame. For example, page flipping (changing the area of memory from which data is displayed) would best be carried out between frames. Also, changing border or background colour (especially if done in quick succession) looks better if it is done between frames. The last raster line on the screen is normally 250, so saying:

```
WAIT (251);
```

would allow changes in the visible area to be carried out between frames. If you wish to change the border colour you may want to wait until about line 285 to avoid the change appearing on the screen. Note that as the TV screen is refreshed quite quickly it is important to make the change that you want directly after the WAIT command. If too many other instructions intervene, then the TV may have commenced drawing the next frame before you make your change.

The WAIT command can also be used to make a change halfway down the screen as in the following example. However this technique should be used with caution as the program is interrupted by the monitor every 1/60 of a second for keyboard scanning, automatic sprite movement, etc. which means that you cannot be guaranteed that the program will make a change at exactly the same line on the screen every time. Key in the following example and you will see what we mean. This example displays a three-colour border (something which is normally not possible).

```
const bordercolour =   11;
  red =                2;
  green =              5;
  yellow =             7;
  false =              0;
begin
  repeat
  wait (285);
  graphics (bordercolour, red);
  wait (90);
  graphics (bordercolour, green);
  wait (180);
  graphics (bordercolour, yellow);
  until false;
end.
```

If you run this example you will see the border in three different colours. The 'shimmering' effect at the edges of each colour is caused by the program being interrupted every 1/60 of a second by the monitor. The shimmering is relatively infrequent because a lot of the time the interrupts occur when they do no harm, namely when a colour change is not about to occur. The shimmering could be stopped altogether by disabling interrupts, however if this is done then no keyboard scanning takes place, and the MOVESPRITE command will not work.

PLOT (colour-type, x, y);

The PLOT command plots a point in bit-mapped (high-resolution graphics) mode. The program should have selected bit-map mode previously (e.g. GRAPHICS (BITMAP, ON); or unpredictable (and disastrous) results may occur. The x and y co-ordinates of the point to be plotted are given. The y co-ordinate should be in the range 0 to 199 or a run-time error will occur. The x co-ordinate should be in the range 0 to 319 (normal mode) or 0 to 159 (multi-colour mode) or a run-time error will occur. The 'colour-type' refers to the type of plotting that will take place. In normal mode (not multi- colour) the colour-type can only be 0 or 1. To plot a point the colour-type should be 1, to erase a previously plotted point the colour-type should be 0. In multi-colour mode the colour-type can be from 0 to 3. In this case, 0 will show the background colour, 1 the upper 4 bits of the corresponding byte of screen memory, 2 the lower 4 bits of the corresponding byte of screen memory, and 3 the corresponding colour nybble. See the Commodore 64 Reference Manual for more details about how multi-colour mode works.

Prior to using the PLOT command the location of character memory should be set to $2000, bitmap mode selected, and a CLEAR command issued to blank out the high-resolution graphics area. (High resolution plotting actually occurs in character memory, not screen memory). e.g. GRAPHICS (bitmap, on, chargenbase, 4); CLEAR (colour1, colour2); See the example over the page which shows how to get ready for PLOTting.

CLEAR (foreground-colour, background-colour);

The CLEAR command clears the area used by bit-mapped (high-resolution) graphics prior to doing PLOT commands. *WARNING: BEFORE ISSUING A CLEAR COMMAND YOU SHOULD CHANGE THE LOCATION OF CHARACTER MEMORY TO 4.* This is because bit-mapped graphics uses an 8K block of memory for the data about which dot is plotted where on the screen. The 8K block of memory is determined by the location of character memory. If character memory is less than 4 there will be interference between the ROM images and your bit- mapping. If character memory is greater than 4 you will 'clobber' parts of your G-Pascal program. Also, *before issuing a CLEAR command you must be in bit-map mode, otherwise a run-time error will occur.* When issuing the CLEAR command you specify two colours. In normal bit-mapped mode (not multi-colour) the first colour is the foreground colour – that is the colour of any points that are plotted. The second colour is the background colour – that is the colour of any points that are not plotted (or plotted as zero). In multi-colour mode the first colour appears if you plot using a colour-type of 1, the second colour appears if you plot using a colour-type of 2. If you want finer control of colours than that you will have to change screen memory or colour memory yourself using the MEMC command. Here is a simple example illustrating the use of bit-mapped graphics:

```
const bitmap =      1;
  chargenbase =     8;
  black =           0;
  yellow =          7;
  on =              1;
var x : integer;
begin
  graphics ( bitmap, on,
  chargenbase, 4);
  clear (yellow, black);
  for x := 1 to 190 do
  plot (on, x, x);
  repeat until getkey;
end.
```

This example will plot a yellow line on a black background and then wait until a key is pressed before finishing. You can experiment with different colours by changing the CLEAR command.

SCROLL (x-offset-pixels, y-offset-pixels);

The SCROLL command achieves smooth scrolling by changing the hardware scroll registers in the VIC chip. To provide full-scale smooth scrolling, especially in the sideways direction requires machine-code subroutines and falls beyond the scope of G-Pascal, particularly as requirements vary from game to game. However the SCROLL command can be used for some special effects, as well as smooth scrolling of text onto the screen as in the example below. In this example the procedure 'scrollit' is called when smooth scrolling is required at the end of a line instead of the standard carriage return.

```
const home = 147;
  lines25 = 5;
  false = 0;
  off = false;
procedure scrollit;
var pixel : integer;
begin
  if cursory = 25 then (* only if at bottom of screen *)
  begin
    for pixel := 6 downto 0 do
    begin
    wait (251);
    scroll (0, pixel);
    end;
  wait (251);
  scroll (0, 7);
  end;
writeln
end;
begin
  write (chr(home));
  graphics (lines25, off);
  cursor (25, 1);
  repeat
    write ("here is a scrolling demo");
    scrollit;
  until false
end.
```

Other special effects can be achieved by using the SCROLL command to 'jiggle' the screen in the x and y directions, for example during an 'explosion'. Note the use of the WAIT command in the above example which forces the scrolling to occur during screen frames, otherwise the scrolling may be jerky and thin black lines may appear on the screen.

SCROLLX

The SCROLLX function returns the current amount of scrolling in the x direction, in pixels.

SCROLLY

The SCROLLY function returns the current amount of scrolling in the y direction, in pixels.

CURSOR (line, column);

The CURSOR command is used to position the cursor on the screen in preparation for a READ or WRITE command. The line should be in the range 1 to 25, and the column should be in the range 1 to 40, or a run-time error will occur.

CURSORX

CURSORX is a function which returns the current column of the cursor.

CURSORY

CURSORY is a function which returns the current line of the cursor.

SETCLOCK (hours, minutes, seconds, tenths);

To set the built-in time-of-day clock use the SETCLOCK command. The clock keeps accurate time to the nearest tenth of a second, regardless of what else the program is doing. You can use the clock to keep track of the actual time of day (in which case the operator would need to enter the time initially), or just as an elapsed time counter, in which case you would just need to set the clock to zeroes initially. The time is stored in 24-hour format, so to set the time at 1.30 p.m. you would say:

```
SETCLOCK (13, 30, 0, 0);
```

CLOCK (whichtime);

The CLOCK function returns the contents of the time-of-day clock. 'Whichtime' is an argument specifying which part of the time is required, as follows:

Whichtime	*Time returned*
1	*Tenths of a second*
2	*Seconds*
3	*Minutes*
4	*Hours*

In order to facilitate accurate time reading, the output of the clock is 'frozen' when the hours are read, and resumes when the tenths of a second are read. (Although the clock itself continues to keep accurate time). Therefore you should read the hours first and the tenths last. If you only want to time short intervals (e.g. 30 seconds) then you can ignore this feature – just read the seconds. Here is an example of using the clock:

```
const false = 0;
  tenths = 1;  seconds = 2;
  minutes = 3; hours = 4;
begin
  setclock (4, 30, 25, 0);
  repeat
    cursor (25, 1);
    write (clock (hours),":",
      clock (minutes),":",
      clock (seconds),":",
      clock (tenths),"   ");
  until false;
end.
```

PADDLE (gameport);

The PADDLE function returns the value of an analogue paddle or analogue joystick plugged into the nominated game port (1 or 2). As two paddles plug into one game port there are in fact two values returned, both from 0 to 255. The way this is done is that one of the values is multiplied by 256 and added to the second value. The example below shows how to extract both values:

```
const false = 0;
var paddle1, paddle2, result : integer;
begin
  repeat
  cursor (25, 1);
  result := paddle (1); (* read gameport 1 *)
  paddle1 := result and $ff;
  paddle2 := result shr 8;
  write (paddle1," ",paddle2," ");
  until false;
end.
```

JOYSTICK (gameport);

The JOYSTICK function returns a value corresponding to which direction a digital joystick is pointing and whether the joystick fire button is pressed or not, or whether or not the buttons on the paddles are depressed. 'Gameport' is specified as 1 or 2, depending on which port the paddles or joystick are plugged into. It is preferable to use port 2, as port 1 is shared with the keyboard – in other words, using a joystick plugged into port 1 makes the operating system think you have typed a key on the keyboard as you operate the joystick (this is undesirable in programs that use both the joystick or paddles and the keyboard). When using a digital joystick the values returned are as follows:

Up: 1
Down: 2
Left: 4
Right: 8
Fire button: 16

It is possible for more than one switch inside the joystick to be activated at once in which case the values are added together (for example pushing the joystick up and to the right returns the value 9). Also if the 'fire' button is pressed then the value 16 will be added to any other values returned. Individual values can be isolated by ANDing the desired value with the JOYSTICK function. For example, to see whether the fire button on joystick 2 is pressed, regardless of which direction the joystick is pointing:

```
IF JOYSTICK (2) AND 16 THEN
```

If paddles are plugged into the game port then the paddle fire buttons return values as follows:

First paddle: 4
Second paddle: 8

G-PASCAL SOUND EFFECTS

G-Pascal provides extensive support for the SID (Sound Interface Device) chip inside the Commodore 64. There are two main commands that are used for this purpose – the SOUND command which controls all voice-independent actions (such as overall volume, filtering, and delays between notes), and the VOICE command which controls all voice- dependent actions (such as a voice's frequency, waveform, ADSR envelope etc.).

The SOUND command

The SOUND command is a general-purpose command which accomplishes 9 different actions. The SOUND command is supplied with pairs of arguments, where the first is the action number and the second is the value to be passed to the action routine. For example: SOUND (*filterfreq, 1000, volume, 15, bandpass, on*);

For ease of comprehension, and to make your programs self- documenting, we strongly suggest that the action numbers be defined in a series of CONST definitions at the start of the program, as shown below. (See the discussion under 'The GRAPHICS command' for more detail about this technique).

```
CONST CLEARSID = 1; FILTERFREQ = 2;
  DELAY = 3; VOLUME = 4;
  RESONANCE = 5; LOWPASS = 6;
  BANDPASS = 7; HIGHPASS = 8;
  CUTOFFVOICE3 = 9;
  ON = 1; OFF = 0;
```

SOUND (CLEARSID, 0);

This command clears the SID (Sound Interface Device) chip, resetting all parameters (volume, frequencies, waveforms etc.) to zero. It is primarily used to 'start from scratch'. G-Pascal automatically clears the SID chip at the start and the end of each run. The second argument to the CLEARSID action is a 'dummy' (in other words it is ignored) so just specify it as zero.

SOUND (FILTERFREQ, frequency);

Used to specify the cut-off frequency of the filter. The frequency ranges from 0 to 2047.

SOUND (DELAY, period);

Used to cause the program to wait (delay) for the nominated period of time. The period is specified as 1/100ths of a second. When a DELAY command is issued the program is suspended (does nothing) until the nominated period elapses (although any sprites moving under MOVESPRITE control will continue to move). *While the program is suspended by a DELAY the RUN/STOP key will not stop the program, nor can Debug or Trace mode be initiated. The only way to stop the program is by pressing RUN/STOP and RESTORE simultaneously.*

The normal use for the DELAY command is to separate notes in a musical piece by precise intervals, however it could also be used for other special effects unrelated to music, such as slowly displaying text on the screen.

The example below illustrates playing voice 1 for a second:

```
VOICE (1, PLAY, ON);
SOUND (DELAY, 100);
VOICE (1, PLAY, OFF);
```

SOUND (VOLUME, volume-level);

Used to set the overall volume output. Volume ranges from 0 (no sound) to 15 (full volume).

SOUND (RESONANCE, resonance-level);

Used to set the level of resonance of the filter. Resonance ranges from 0 (no resonance) to 15 (full resonance).

SOUND (LOWPASS, ON);

Directs the output of selected voices through the lowpass filter. Whether a particular voice is filtered or not is controlled by the VOICE command. This is used in conjunction with the FILTERFREQ and RESONANCE actions described above. Can be used in conjunction with BANDPASS and HIGHPASS filtering.

SOUND (LOWPASS, OFF);

Turns off the lowpass filter.

SOUND (BANDPASS, ON);

Directs the output of selected voices through the bandpass filter. Whether a particular voice is filtered or not is controlled by the VOICE command. This is used in conjunction with the FILTERFREQ and RESONANCE actions described above. Can be used in conjunction with LOWPASS and HIGHPASS filtering.

SOUND (BANDPASS, OFF);

Turns off the bandpass filter.

SOUND (HIGHPASS, ON);

Directs the output of selected voices through the highpass filter. Whether a particular voice is filtered or not is controlled by the VOICE command. This is used in conjunction with the FILTERFREQ and RESONANCE actions described above. Can be used in conjunction with LOWPASS and BANDPASS filtering.

SOUND (HIGHPASS, OFF);

Turns off the highpass filter.

SOUND (CUTOFFVOICE3, ON);

Disconnects Voice 3 from the audio path. Used if Voice 3 is only being used for special effects such as random number generation and is not intended to be heard.

SOUND (CUTOFFVOICE3, OFF);

Reconnects Voice 3 to the audio path (the default condition).

```
CONST FILTERFREQ = 2; VOLUME = 4; RESONANCE = 5;
  HIGHPASS = 8; ON = 1; OFF = 0;
  FREQUENCY = 1; WIDTH = 2; FILTER = 3;
  SUSTAIN = 6; PLAY = 8; PULSE = 13;
VAR FREQ : INTEGER;
BEGIN
  SOUND (VOLUME, 15, RESONANCE, 15, HIGHPASS, ON);
  VOICE (1, FREQUENCY, 8583, 1, WIDTH, 2048,
    1, FILTER, ON, 1, SUSTAIN, 15,
    1, PULSE, ON, 1, PLAY, ON);
  FOR FREQ := 0 TO 2047 DO SOUND (FILTERFREQ, FREQ);
  FOR FREQ := 2047 DOWNTO 0 DO SOUND (FILTERFREQ, FREQ);
END.
```

This program plays 'middle C' and then changes the filtering while the note is playing, to demonstrate the effects of different filter frequencies.

The VOICE command

The VOICE command provides control over the characteristics of each voice. It is a general-purpose command which accomplishes 15 different actions. The VOICE command accepts arguments in groups of three – the first is always the voice number, from 1 to 3. The second is the action number and the third is the value to be passed to the action routine. For example: VOICE (*1, frequency, 4291, 1, triangle, on, 1, play, on*);

For ease of comprehension, and to make your programs self- documenting, we strongly suggest that the action numbers be defined in a series of CONST definitions at the start of the program, as shown below. These would normally be added to the definitions for the SOUND command. (See the discussion under 'The GRAPHICS command' for more detail about this technique).

```
CONST FREQUENCY = 1; WIDTH = 2;
  FILTER = 3; ATTACK = 4;
  DECAY = 5; SUSTAIN = 6;
  RELEASE = 7; PLAY = 8;
  SYNC = 9; RINGMOD = 10;
  TRIANGLE = 11; SAWTOOTH = 12;
  PULSE = 13; NOISE = 14;
  TEST = 15;
  ON = 1; OFF = 0;
```

VOICE (voice-number, FREQUENCY, freq);

Used to define the frequency at which this voice will play. The frequency ranges from 0 to 65535. The frequencies of the top 12 'standard' musical notes are given below. All of the other frequencies can be derived by simply dividing the values given by the appropriate power of 2. In other words, each time the frequency is divided by 2 it drops an octave. A simple (and quick!) way of dividing by a power of 2 is to use the SHift Right operator (SHR). For example, to produce Middle C (which is 3 octaves below the value of C given below), just say: *VOICE (1, FREQUENCY, 34334 SHR 3);* This means that any program that needs to use all octaves of notes need only contain the 12 frequencies given below and quickly calculate all the others at the start of the program (or as required).

Note	*Frequency*
C	*34334*
C sharp	*36376*
D	*38539*
D sharp	*40830*
E	*43258*
F	*45830*
F sharp	*48556*
G	*51443*
G sharp	*54502*
A	*57743*
A sharp	*61176*
B	*64814*

VOICE (voice-number, WIDTH, pulse-width);

Used to define the pulse width for the pulse waveform. Not needed (and ignored) if the pulse waveform is not in use. The pulse width ranges from 0 to 4095. A pulse width of 2048 gives a square wave (which has a rich sound). A pulse width of 0 or 4095 will give an inaudible sound. The further that the pulse width goes from 2048 (higher or lower) the 'thinner' the sound will become – it will be less rich in harmonics.

VOICE (voice-number, FILTER, ON);

Directs this voice through the filter. The filter characteristics, frequency and so on are selected through the SOUND command. If no filtering has been selected by the SOUND command then turning filtering on here will effectively silence the voice.

VOICE (voice-number, FILTER, OFF);

Bypasses the filter for this voice. The voice will be heard regardless of the filter settings.

VOICE (voice-number, ATTACK, rate);

Sets the attack rate for this voice. The attack rate is the rate at which the volume level rises when the voice is played. It ranges from 0 (2 milliseconds) to 15 (8 seconds).

VOICE (voice-number, DECAY, rate);

Sets the decay rate for this voice. The decay rate is the rate at which the volume level drops to the sustain level once the attack cycle is complete. It ranges from 0 (6 milliseconds) to 15 (24 seconds).

VOICE (voice-number, SUSTAIN, level);

Sets the sustain level for this voice. The sustain level is the volume level of this voice once the attack and decay cycles have completed. The voice will remain at this level until it is released.

VOICE (voice-number, RELEASE, rate);

Sets the release rate for this voice. The release rate is the rate at which the volume level drops (to nil) once the note is released. It ranges from 0 (6 milliseconds) to 15 (24 seconds).

VOICE (voice-number, PLAY, ON);

Plays this voice. As soon as this command is executed the attack cycle for this voice will commence.

VOICE (voice-number, PLAY, OFF);

Releases this voice. As soon as this command is executed the release cycle for this voice will commence (assuming that it was previously played).

VOICE (voice-number, SYNC, ON);

Synchronizes the fundamental frequency of the nominated voice with another one. If the voice number is 1, it is synchronized with voice 3. If the voice number is 2, it is synchronized with voice 1. If the voice number is 3, it is synchronized with voice 2.

VOICE (voice-number, SYNC, OFF);

Turns off synchronization between this voice and another.

VOICE (voice-number, RINGMOD, ON);

Ring modulates the triangle waveform output of the nominated voice with another one. For ring modulation to be audible this voice must have triangle waveform selected, and the other voice must have a non-zero frequency. If the voice number is 1, it is ring modulated with voice 3. If the voice number is 2, it is ring modulated with voice 1. If the voice number is 3, it is ring modulated with voice 2.

VOICE (voice-number, RINGMOD, OFF);

Turns off ring modulation between this voice and another.

VOICE (voice-number, TRIANGLE, ON);
Selects the triangle waveform for this voice.

VOICE (voice-number, TRIANGLE, OFF);
De-selects the triangle waveform for this voice.

VOICE (voice-number, SAWTOOTH, ON);
Selects the sawtooth waveform for this voice.

VOICE (voice-number, SAWTOOTH, OFF);
De-selects the sawtooth waveform for this voice.

VOICE (voice-number, PULSE, ON);
Selects the pulse waveform for this voice. The size of the pulse (pulse width) is controlled by the WIDTH action described above.

VOICE (voice-number, PULSE, OFF);
De-selects the pulse waveform for this voice.

VOICE (voice-number, NOISE, ON);
Selects the noise waveform for this voice. Can be used to produce rumbling sounds or 'white noise' depending on the frequency selected for this voice. Also, noise must be selected for voice 3 for the RANDOM function to correctly return random numbers. Noise should not be selected whilst other waveforms are active or the noise generator may 'lock up'. In this case the only way to re-activate noise is to select the TEST mode described below.

VOICE (voice-number, NOISE, OFF);
De-selects the noise waveform for this voice.

NOTE: It is NOT recommended that more than one waveform be selected at once. To change from triangle to pulse, for example, we recommend de-selecting triangle first, then selecting pulse.

VOICE (voice-number, TEST, ON);
Selects 'test' mode for this voice – effectively silencing it. This action would not normally be used. The normal way to silence a voice is to allow its ADSR envelope to silence it, or possibly to de- select all waveforms.

VOICE (voice-number, TEST, OFF);
De-selects 'test' mode for this voice (the default condition).

```
CONST VOLUME = 4; CUTOFFVOICE3 = 9;
  ON = 1; OFF = 0;
  FREQUENCY = 1; SUSTAIN = 6; PLAY = 8;
  SAWTOOTH = 12; NOISE = 14;
VAR FREQ : INTEGER;
BEGIN
  SOUND (VOLUME, 15, CUTOFFVOICE3, ON);
  VOICE (3, FREQUENCY, 10, 3, NOISE, ON,
    1, SUSTAIN, 15, 1, SAWTOOTH, ON, 1, PLAY, ON);
  REPEAT
    VOICE (1, FREQUENCY, RANDOM * 200)
  UNTIL 0;
END.
```

This program plays random frequencies at the rate of 10 per second. To change the rate of frequency changes modify the '10' in the clause: VOICE (3, FREQUENCY, 10, to something else.

Sound effect functions

RANDOM

The RANDOM function returns the current output of voice 3 (upper 8 bits). This results in a number from 0 to 255. The character of these numbers is directly related to the waveform selected for voice 3. If 'triangle' waveform is selected the number will increment from 0 to 255, then decrement back to 0. If 'sawtooth' waveform is selected the number will increment from 0 to 255 then jump back to 0. If 'pulse' waveform is selected the number will jump between 0 and 255. If 'noise' waveform is selected the numbers will vary randomly. In all cases, the rate at which the numbers change is dependent on the frequency of voice 3. The output of RANDOM will be valid, regardless of whether or not voice 3 is actually gated (playing). In other words, you do not have to say: VOICE (3, PLAY, ON) for RANDOM to contain valid data.

The most common use of this function is as a random number generator (hence its name) however by selecting, say, a triangle waveform and using the result to modify the frequency of another voice special effects such as a 'siren' sound could be achieved.

If using RANDOM for random numbers the minimum 'conditioning' required for their generation is:

```
VOICE (3, NOISE, ON, 3, FREQUENCY, 10000);
```

If the noise is not intended to be heard by the user then SOUND (CUTOFFVOICE3, ON) could be selected as described under the SOUND command (alternatively just leave the overall volume at zero). However if you are writing a game with sound effects and random numbers then use voice 3 for noise (such as explosions, footsteps etc.) which will automatically provide random numbers through RANDOM at the same time.

The only caution in using RANDOM for random numbers is that a high enough frequency is selected. The random numbers change at the nominated frequency, so if a frequency of 1 is chosen, for example, the random numbers would only change once a second.

ENVELOPE

ENVELOPE returns the output of the voice 3 envelope generator (ADSR envelope). This will return a number from 0 to 255 reflecting the current volume of voice 3 as controlled by the Attack, Decay, Sustain and Release (ADSR) parameters. Voice 3 must be played in order to trigger the ADSR cycle. The output from ENVELOPE can be added to the filter frequency or fundamental frequency of other voices for special effects.

```
CONST FREQUENCY = 1; NOISE = 14; ON = 1;
BEGIN
  VOICE (3, NOISE, ON, 3, FREQUENCY, 50000);
  REPEAT
    WRITELN (RANDOM + RANDOM SHL 8)
  UNTIL 0;
END.
```

This program generates random numbers in the range 0 to 65535 and displays them on the screen.

INDEPENDENT MODULES

A powerful feature of G-Pascal is the ability to compile procedures and functions independently and 'link' them together at run time. This is made particularly easy because the P-codes generated by the compiler are completely 'relocatable'. That is to say that they may be run at any address, regardless of where they were compiled, without change.

Advantages of independent modules

1. As P-codes are typically only a half to a third of the size of the corresponding source code, larger programs may be run if they are compiled in 'pieces'.

2. Groups of logically related subroutines may be placed in an independent module and then used by a number of other programs – for example you could put all your 'file handling' routines in an independent module. Then if you needed to change the way you access files you only have to change one module rather than perhaps dozens of programs.

3. You can implement an 'overlay' structure – in other words, various different modules can be loaded to the same address (not all at the same time of course!) thereby saving memory space. For example, if you are implementing a big adventure game you might have the first half of the game in module 'A' which loads at address $1000, and then at the appropriate time load module 'B' to address $1000 instead.

How to implement independent modules

1. Compile the module or modules and save them to disk or cassette using the <O>bject option in the files menu.

2. Compile the 'main' program and include 'dummy' procedure declarations for the independent modules in the form:

```
PROCEDURE EXTRAROUTINES (ARG1, ARG2, ARG3); $1003;
```

This tells the compiler that the named procedure will be loaded as an independent module, and to execute it at address $1003.

3. The main program should load the module's P-codes before invoking the module. Alternatively, the independent modules could be compiled at the required address directly by using the %A compiler directive. The module should be loaded or compiled at an address *three bytes below* where it is to be executed at. The reason for this is that a G-Pascal program *always* starts with a 3-byte 'jump' instruction to the 'main line' – that is, the first instruction to be executed. Therefore the first *procedure* in the program starts 3 bytes in from the actual address at which the program is loaded. The examples following should clarify this point.

Multiple procedures in one module

As procedures can be nested within procedures a 'module' can consist of many actual procedures or functions. In this case a typical approach is to supply at least one argument which is a 'procedure number' which would be used in a CASE statement to direct control to the correct sub-procedure.

EXAMPLES OF INDEPENDENT MODULES

We will illustrate the use of independent modules with a simple example, and then a more complicated example which will make greater use of the potential of these modules.

Example 1

The independent module

```
(* %A $1000 *)
PROCEDURE PLAYSOMEMUSIC (PITCH, DURATION);
CONST DELAY = 3; VOLUME = 4; FREQUENCY = 1;
SUSTAIN = 6; PLAY = 8; TRIANGLE = 11;
ON = 1; OFF = 0;
BEGIN
    SOUND (VOLUME, 15);
    VOICE (1, FREQUENCY, PITCH,
    1, TRIANGLE, ON,
    1, SUSTAIN, 15,
    1, PLAY, ON);
    SOUND (DELAY, DURATION);
    VOICE (1, PLAY, OFF)
END;

BEGIN (* start of 'main line' - this will not be executed *)
END.
```

The above module is compiled first – the %A option automatically puts it at location $1000.

The 'Main Program'

```
CONST FALSE = 0;

VAR I, J : INTEGER;

PROCEDURE PLAYSOMEMUSIC (ARG1, ARG2); $1003;
BEGIN
J := 10;
REPEAT
  FOR I := 1 TO 10 DO
    PLAYSOMEMUSIC (I * 1000, J);
  FOR I := 10 DOWNTO 1 DO      PLAYSOMEMUSIC (I * 1000, J)
UNTIL FALSE;
END.
```

(You may want to try this example yourself. As well as illustrating independent modules it plays quite an interesting tune.)

The above illustrates a couple of important points:

1. The module was compiled 3 bytes below where it was called.
2. The 'dummy' declaration for the module in the main program had the same number of arguments (2) as the actual declaration in the module.

Example 2

Now we'll illustrate some more complicated aspects of independent modules, namely:

1. 'Common' data areas.
2. Nested procedures.
3. One independent module calling another.

Module 1

```
VAR A, B, C, D : INTEGER;
FRED : ARRAY [50] OF CHAR;

PROCEDURE MODULE1;
BEGIN
(* This module has no arguments *)
  A := B; (* code for module 1 *)
END;
BEGIN END.   (* dummy mainline *)
```

This module would be compiled and the object saved to disk or cassette using the (O)bject option in the Files Menu as "MOD1.OBJ".

Module 2

```
VAR A, B, C, D : INTEGER;
FRED : ARRAY [50] OF CHAR;

PROCEDURE MODULE1; $1003; (* refer to first module *)
PROCEDURE MODULE2 (ACTION, ARG1, ARG2, ARG3);
PROCEDURE FIRSTACTION;
BEGIN
  MODULE1; (* call other independent module *)
  A := B – C
END;
PROCEDURE SECONDACTION (X, Y);
BEGIN
D := X * Y
END;
BEGIN (* actual start of module 2 code *)
  CASE ACTION OF (* choose a sub-module *)
  1 : FIRSTACTION;
  2 : SECONDACTION (ARG1, ARG2) (* pass parameters *)
  END;      (* of case *)
END; (* of module 2 procedure *)
BEGIN END. (* dummy mainline *)
```

This module would be compiled and the object saved to disk or cassette using the (O)bject option in the Files Menu as "MOD2.OBJ".

```
CONST DISK = 8; LOADFLAG =0;

VAR A, B, C, D : INTEGER;
FRED : ARRAY [50] OF CHAR;
PROCEDURE MODULE1; $1003; (* refer to module 1 *)
PROCEDURE MODULE2 (ACTION, ARG1, ARG2, ARG3); $2003;
          (* refer to module 2 *)
BEGIN (* start of actual program *)
  LOAD (DISK, $1000, LOADFLAG, "MOD1.OBJ");
  LOAD (DISK, $2000, LOADFLAG, "MOD2.OBJ"); (* load modules *)
  MODULE1; (* invoke module 1 *)
  MODULE2 (1, 5, 6, 0) (* invoke module 2 *)
END. (* of program *)
```

This illustrates one module calling another. When using modules it is obviously important to load them at the address that you have told the compiler that you are going to (minus 3) otherwise strange results may occur. Modules may share 'common' (global) data provided that *the same data declarations occur in each module and the calling program.* This works because the compiler allocates all variables as 'stack relative' – in other words their actual addresses in memory are not known until it knows where the run-time stack is going to be. Therefore identical data declarations in different modules will result in the compiler allocating identical stack relative addresses within each module – and the modules can then each refer to each other's data areas. If a particular module needs more 'work areas' then these should be allocated after the procedure declaration (local data) and these will only be in force during the invocation of the procedure.

HOW TEXT IS STORED BY G-PASCAL

This section describes the format of G-Pascal source files for those users that wish to use them with other editors or word processors. Each source line is stored in sequence, and ends with a carriage return (hex 0D). Line numbers are not stored in the program – these are automatically generated by the Editor and Compiler when listing the program. The end of the program is indicated by a 'null' following the last carriage return. In other words, the last two bytes of the program are hex 0D00.

Occurrences of two or more spaces are converted to a special code. This consists of a 'dle' character (hex 10) followed by a one- byte space count with the high-order bit *set*. The high order bit is set so that the editor does not confuse 13 spaces with a carriage return. For example, 4 consecutive spaces would be stored as hex 1084. (hex 10 followed by hex 04 + hex 80).

All reserved words (e.g. BEGIN, END, DEFINESPRITE, CURSOR etc.) are 'tokenized' as they are keyed in and stored as a single byte in order to save space. Also, the system assumes that all reserved words are followed by a space, so the space is not stored and a space is always displayed after a reserved word. This makes it impossible to have punctuation directly following a reserved word. (You may key in the punctuation directly after the word, for example: END; however it will be displayed as END ;).

This tokenization means that PROCEDURE occupies one byte instead of 10 bytes (allowing for the space that follows the word PROCEDURE). If you wish to 'de-tokenize' your program (perhaps to use with an external word processor or editor) then run the following program which reveals the token equivalents (in decimal) of each reserved word:

```
var word : integer;
begin
memc [ $49 ] := 0; (* expand reserved words *)
for word := $81 to $FF do
if (word < $B0 ) or (word > $DE) then
  writeln ("equivalent of ", word,
     " is ", chr(word))
end.
```

Any user-written detokenization program should make allowance for the fact that tokenization is not done within quotes as the characters used for reserved words are in some cases the same as the graphics symbols. For example, try entering a line in the Editor by pressing the 'Commodore' key and 'T' simultaneously. It will display a graphics character (thin horizontal bar). Now list that line and the word 'cursor' will appear. Now repeat the process, this time putting the graphics symbol inside quotes. This time it will list correctly. This demonstrates that the internal listing routines display the same character differently depending on whether or not it is inside quotes.

Programs that have been entered using an independent editor (not the G-Pascal built-in Editor) *will successfully compile as the G-Pascal compiler will recognize either its internally tokenized reserved words or the same words spelt in full.* Programs containing reserved words which are not tokenized will be automatically tokenized (and multiple spaces reduced to the 2-byte code) as soon as a 'Replace' command is done in the Editor which has one or more spaces in its replacement string. In other words, to force full tokenization of a program just enter:

```
R 1. . .
```

This process takes about one second per 100 lines of program code.

Idiosyncrasies of tokenization

Tokenization of the source code has the benefits of increased compile speed, reduced program size and consequently faster loading and saving from disk or cassette. It also has the advantage that spaces can freely be used within the program to aid readability as 10 spaces take up no more room than 2 spaces. However under certain (rare) circumstances the tokenization process can cause strange behaviour by the Editor which could lead to confusion. These are described below ...

- A space will *always* be displayed after a reserved word.
- Reserved words will always be displayed in lower case, regardless of how they are entered.
- As reserved words are stored internally as one byte, the Find and Replace commands in the Editor *cannot locate part of a reserved word*. For example, you cannot find the 'BEG' in BEGIN, or 'PROC' in PROCEDURE. In order to correctly locate reserved words, they must be spelt in full. To obtain a list of all reserved words run the small program described in the previous section.
- It is quite permissable to locate part of a non-reserved word, unless that part is itself a reserved word. In other words, attempting to locate 'FR' will sucessfully locate the word 'FRED', however trying to locate 'TO' will *not* locate the word 'TOOL' as 'TO' is a reserved word.
- As multiple spaces are stored as a 2-byte code, the Find and Replace commands can only match on the *exact* number of spaces (remembering that the space which is displayed following a reserved word is not actually stored in the file and should not be counted).
- A line containing mismatched quote symbols may display strangely. For example, type in the following line as line 1: BEGIN END REPEAT WHILE FOR DO. Then use the Replace command to change BEGIN to a quote symbol: R 1.BEGIN.".

  The reserved words will have been changed to inverse letters! Now get rid of the quote symbol by saying: R 1."..

  The reserved words will re-appear!

CONVERTING FROM OTHER PASCALS

As G-Pascal is a subset of Pascal, not all of the constructs from 'full' Pascal are available, however most of them are either already in G-Pascal or can be 'simulated'. This section contains hints for converting published programs so that they will work in G-Pascal.

PROGRAM statement

The first word in a full Pascal program is:

```
PROGRAM programname (input, output);
```

G-Pascal does not need this, so omit it.

Type BOOLEAN

G-Pascal does not provide a BOOLEAN type, however if you declare:

```
CONST TRUE = 1; FALSE = 0;
```

and change the word BOOLEAN to CHAR where it appears then Booleans will work as normal. If the result of a conditional test is zero G- Pascal considers it to be false, otherwise G-Pascal considers it to be true.

TYPE declaration

G-Pascal does not support the TYPE declaration. However, where full Pascal might say:

```
TYPE COLOUR = (RED, GREEN, BLUE);
VAR FRED : COLOUR;
BEGIN FRED := GREEN END.
```

just say in G-Pascal:

```
CONST RED = 0; GREEN = 1; BLUE = 2;
VAR FRED : INTEGER;
BEGIN FRED := GREEN END.
```

which will have the same effect.

Quotes

G-Pascal expects (") as its string delimiter rather than (') which is different from most other Pascals.

Type REAL

There is no REAL (floating point) type in G-Pascal. However as INTEGERs are 3 bytes and therefore provide at least 6-digit accuracy then these suffice for most applications.

For example, a program that deals in dollars and cents could hold money amounts internally as cents, and then print them with the decimal point in the right place like this:

```
write (AMOUNT / 100,".")
if AMOUNT mod 100 < 10 then write ("0");
writeln (AMOUNT mod 100);
```

Passing procedure and function arguments by address

By using the ADDRESS construct with the MEM construct it is possible to pass parameters to a procedure or function by 'address' (which is not directly supported by G-Pascal).

For example:

```
VAR I, J, K: INTEGER;

PROCEDURE ADD (A, B, C);
BEGIN
MEM [C]:= A + B;
END

BEGIN
I := 2; J := 3;
ADD (I, J, ADDRESS(K) );
WRITELN ("ANSWER WAS: ",K)
END.
```

As the example shows, although all parameters are passed by 'value', in the case of 'K' the value is, in fact, the address of 'K' (by using the word ADDRESS), so that the ADD procedure, by using the supplied address in a MEM statement, can return its result to the desired variable.

READLN, WRITELN etc.

See sections on READ, WRITE and WRITELN, under the section 'Beginner's Guide to Pascal' for G-Pascal formats.

LABEL, GOTO

Not supported. Use other programming techniques.

DEBUGGING

A 'bug' is where a program compiles without errors, but when run does one of the following:

a) Produces incorrect results;
b) Aborts with an error message (e.g. Divide by zero);
c) Goes into a 'loop' and appears to do nothing;
d) 'Crashes', possibly destroying itself and G-Pascal.

Below are suggested methods of dealing with each, however the important thing about debugging is to keep an open mind about possible problems – 'expect the unexpected' so to speak. Try to refrain from giving up and blaming G-Pascal if your program does not work – G- Pascal has been extensively tested with many large and small programs. The likelihood of your uncovering some obscure bug in G-Pascal is pretty remote.

Programs that produce incorrect results

If you can't work out why your program is not working exactly as designed try displaying debugging information at pertinent spots in your program. You could make the displays conditional on a 'debugging flag' at the start of the program so you can easily enable or disable this debugging information by changing one line. e.g.

```
CONST TRUE = 1; FALSE = 0;
  DEBUGGING = TRUE;

  (*... and further on in the program ...*)

  IF DEBUGGING THEN WRITELN ("J5 IS NOW: ", J5);
```

By making 'DEBUGGING' false then your debugging WRITEs would be suppressed.

Programs that abort with an error message

All aborts (including pressing RUN/STOP) display the P-code address which shows where the program was when it aborted. By referring to the P-code addresses displayed by the Compiler you can easily locate the problem. Note that after an abort GPascal waits for you to press a key before returning to the Main Menu this is to give you a chance to read the error message.

Programs that loop or hang and appear to do nothing

If you want to know what part of your program is executing press COMMODORE/T (T = Trace) and a Trace will start. Press COMMODORE/N (N = Normal) to cancel the Trace. Whilst tracing you will see something like the following:

```
(097A) 3C 7D 09 3B
(097D) 3B 0D 00 01
(0980) 08 81 32 00
(0981) 81 32 00 F4
(0982) 32 00 F4 FF
(0986) E4 2A 2C 00
(0987) 2A 2C 00 F4
(0988) 2C 00 F4 FF
```

The addresses in brackets are the P-code addresses (which you can relate to the addresses in brackets which are listed if you get a listing during a Compile). By doing this you can find which procedure or group of instructions are being executed which should help track down the cause of the loop. The data to the right of the P-code addresses are the actual P-codes. Not all P-codes are 4 bytes long in which case disregard some of them. For an explanation of the meaning of the P-codes see 'Meanings of P-codes'.

Programs that 'crash'

There are two likely causes of a program going completely berserk:

1) An array subscript being too big or too small;

2) Pokeing (with the MEM or MEMC arrays) into a piece of memory that you shouldn't have.

If a program is behaving strangely check that all subscripts cannot exceed their declared array bounds.

Other debugging techniques

As a last resort you can select 'Debug' mode by either selecting 'Debug' from the Main Menu or pressing COMMODORE/D (D Debug) while the program is running.

Using 'Debug' mode

Debug mode displays information like the following:

```
(0981) 81 32 00 F4
  Stack: 95F2 = 04 01 00 00 14 00 00 05
  Base: 95FF = 05 04 04 05 05 04
(0982) 32 00 F4 FF
  Stack: 95EF = 01 00 00 04 01 00 00 14
  Base: 95FF = 05 04 04 05 05 04
(0986) E4 2A 2C 00
  Stack: 95F2 = 04 01 00 00 14 00 00 05
  Base: 95FF = 05 04 04 05 05 04
```

The debug information appears in groups of three lines.The first line is identical to the information displayed during a Trace and consists of the P-code address followed by the actual P-code being executed and its operands. The next line ('Stack:') shows the top 8 bytes on the stack – this would frequently be the data being worked on - for example if you said 2 +4 in your program you would see the '2' on the top of the stack (leftmost 3 bytes) in the order: 02 00 00 and then the '4' would be pushed onto the stack so that you would then see: 04 00 00 02 00 00 and then the 'add' P-code would be processed, leaving '6' on the top of the stack. The third line ('Base:') shows the 'stack frame linkage data' namely:

a) Procedure return address (leftmost 2 bytes).
b) Stack frame dynamic link (middle 2 bytes) – this is the address of the stack frame of the last activated procedure or function (prior to the current one).
c) Stack frame static link (rightmost 2 bytes) – this is the address of the stack frame of the last procedure in the lexical order of the program.

Note that the stack and base linkage data is that *before* the displayed pcode is executed.

How to start Debug or Trace from within your program

To start debug mode from within the program:

```
MEMC [$31] := 1; MEMC [$67] := 1;
```

To start trace mode from within the program:

```
MEMC [$31] := 1; MEMC [$67] := $80;
```

To stop debug or trace from within the program:

```
MEMC [$67] := 0;
```

CHANNEL NUMBERS

G-Pascal assumes that the disk drive responds to device number 8 and the printer to device number 4. If your disk or printer hardware responds to different device numbers then enter, compile and run the following program prior to attempting to access the disk or printer:

```
BEGIN
MEMC [ $8010 ] := 6; (this is the printer device number *)
MEMC [ $8011 ] := 9; (this is the disk device number *)
END.
```

The actual numbers used in the assignment statements above will depend on what your printer and disk channels are. Obviously if only one device number is wrong then you need only enter the appropriate assignment statement.

MEMORY MAP

This section describes the various memory addresses used by the G-Pascal system. It is particularly important to be aware of memory allocation when:

- Using DEFINESPRITE to place sprite shape definitions in memory.
- Controlling the location of P-codes with the %A compiler directive.
- Using bit-mapped graphics.
- Doing page-flipping – that is, using more than one area of memory for screen memory.
- Placing machine-code subroutines in memory.
- Compiling independent modules.

$0000 to $03FF – used for 'zero page' system and compiler work-areas, system and compile-time stack, other system variables and system jump vectors. This area should not be used except by experienced machine-language programmers who are aware of the ramifications of changing these locations.

$0400 to $07FF – used for the 1K 'primary' screen memory area. This area is in constant use by the compiler for displaying the text that appears on the TV screen. While a program is running the WRITE statement normally causes text to appear in this area.

$0800 to $0FFF – spare (not used by G-Pascal). This 2K area is available for machine-code subroutines, other screen memory pages (numbers 2 and 3), user-defined character sets (character memory base number 1), DEFINESPRITE statements (pointers 32 to 63), or P- codes (by using the %A $800 compiler directive). Of course, it cannot be used for all of these purposes at the same time. The addresses occupied by shapes defined with DEFINESPRITE are the pointer number multiplied by 64 – e.g. pointer 32 is address $800 (32 * 64 = 2048 which is $800 in hexadecimal).

$1000 to $1FFF – spare (not used by G-Pascal). This 4K area is available for machine-code subroutines or P-codes. Because the hardware 'maps' the character generator images into this area during the video display phase of the system clock it is *not suitable* for DEFINESPRITE images, screen memory areas, or bit- mapped graphics areas.

$2000 to $3FFF – spare (not used by G-Pascal). This 8K area is available for machine-code subroutines, P-codes, DEFINESPRITE statements (pointers 128 to 255), screen memory pages (numbers 8 to 15), bit-mapped graphics (by setting character memory base number to 4), or user-defined character sets (character memory base numbers 4 to 7). Again, this area cannot be used for all these purposes at once, however it may be possible, for example, to have some addresses allocated to sprite shapes, some for P-co' _s, and some for extra screen memory pages, provided care is taken that these areas do not overlap.

$4000 to $7FFF – this 16K area is used by G-Pascal to store the source program (i.e. the G-Pascal text as it is typed in or loaded from disk or cassette). Unless the %A compiler directive is used this area is also used to store the P-codes during the compilation process – they are placed directly after the end of the source program.

$8000 to $BFFF – this 16K area used to hold the G-Pascal compiler itself and is not available for other purposes.

$C000 to $CFFF – this 4K area is used for the compiler's symbol table (during compilation) and the G-Pascal run-time stack (storage area for variable data) during running. During running the 'top' 304 bytes are used for sprite-related functions (MOVESPRITE, ANIMATESPRITE and so on) thus leaving 3792 bytes for the run-time stack. The run-time stack actually starts at $CED0 (at time of publication) and grows downwards.

$D000 to $D7FF – this 2K area is used for I/O functions (VIC chip and SID chip).
$D800 to $DBFF – this 1K area is used for Colour RAM nibbles. It is normally updated by writing to the screen.
$DC00 to $DFFF – this 1K area is used for I/O functions (CIA chips and future expansion).
$E000 to $FFFF – this 8K area is used for the Kernal ROM (that is, the Commodore 64 operating system which handles screen editing, loading and saving files, power up activities and so on).

MACHINE LANGUAGE SUBROUTINES

The novice user should skip this section.

Machine-language subroutines may be called by:

CALL (*address*)

Remember that hexadecimal constants must be preceded by a '$'.

It is possible to set up the A, X, and Y registers and the condition codes prior to the call by loading certain reserved memory locations (see table below). G-Pascal loads the contents of those locations into the appropriate registers prior to calling the subroutine with a JSR instruction and places the contents of the A, X, Y registers and condition codes into those locations after the subroutine has exited. The machine-code subroutine should end with an RTS instruction.

Care should be taken not to set either the 'decimal' flag or the 'interrupt inhibit' flag in the condition codes register or G-Pascal will not function correcty.

The example below sends the letter 'A' to logical device 4.

```
const AREG = $2B2; XREG = $2B3;
YREG = $2B4; CCODES = $2B1;
CHKOUT = $FFC9; CHROUT = $FFD2;
begin
  memc [ XREG ] := 4;
  call ( CHKOUT );
  memc [ AREG ] := 'A';
  call ( CHROUT );
end.
```

Warning

The subroutine (if you write one yourself) should not change addresses used by G-Pascal or unpredictable results will occur. The addresses listed in the table below are used by the G-Pascal interpreter to maintain important information about the state of a program as it runs and should not be changed (apart from the 4 addresses used by the CALL instruction). Other addresses which are not used by the Kernal may be used by machine-code programs (such as the ones allocated to Basic) however they should be regarded as 'scratch' areas, subject to use by various G-Pascal instructions. (In other words, they may change from CALL to CALL). For permanent work areas for machine code programs refer to the section entitled 'Memory Map' and choose a suitable area of memory from that.

Addresses of interest to machine-language programmers:

Address	*Meaning*
$2B2	A-register save/restore for CALL instruction
$2B3	X-register save/restore for CALL instruction
$2B4	Y-register save/restore for CALL instruction
$2B1	Condition codes save/restore for CALL instruction

$04 Start address of machine code called by CALL
$0B/$0C Address to return to in event of error
$26/$27 Address of next P-code to execute
$28/$29 Address of start of program (first P-code)
$31 Display P-code flag (0 = no, 1 = yes) – used in conjunction with debug flag ($67) – to start a trace progammatically move 1 to $31 and $80 to $67. (i.e. memc [$31] := 1; memc [$67] := $80;)
$45/$46 Current stack frame base
$49 Program running flag (0 = compiling, 12 = running)
$53 Valid compile flag (0 = no, 1 = valid)
$54/$55 Address to transfer to if RUN/STOP pressed
$56/$57 First free location past P-codes. (End of program + 1). Can be used for temporary data or machine code subroutines.
$67 Debugging flag (0 = none, 80 = trace, 1 = debug) (see description for $31 above)
$6C 'Invalid' flag (returned by INVALID function)
$6D Collision mask (set up by SPRITEFREEZE)
$6E Collision register (returned by FREEZESTATUS)
$C3/$C4 Current runtime stack top (last used location)
$2C5/$2C6 Address of current P-code
$2F8/$2F9 Address of Kernal interrupt routine. G-Pascal's interrupt processor transfers to here after processing MOVESPRITE and ANIMATESPRITE commands.
$2FA to $2FC Used by interrupt routines

All 2-byte locations which contain an address hold it in the normal 6502 format of low-order byte first, then high-order byte.

MEANINGS OF P-CODES

Here are what the P-codes mean. You will not normally need this information. Many P-codes do not have any operands – that is, they are just a single byte. In this case their operands are on the top of the stack, wherever that is at the time. For example, 'ADD' adds together the two top integers on the stack, replacing them with the result of the addition. Some P-codes (such as Load and Store) are followed by the frame displacement and stack relative address of the data to be loaded or stored (in other words, the frame-relative address of the variable). 'Jumping' P-codes are followed by a relative address of the location to jump to. WRITE (string), LOAD and SAVE P-codes are followed by a string length and then the string itself.

Hex P-code Function

00 Load constant
01 DEFINESPRITE
02 Negate (sp)
03 PLOT
04 Add (sp) to (sp – 1)
05 PLOT (same as PLOT) (not currently used)
06 Subtract (sp) from (sp – 1)
07 GETKEY
08 Multiply (sp) * (sp – 1)
09 CLEAR
0A Divide (sp – 1) / (sp)
0B MOD (sp – 1) MOD (sp)

0C Address of integer
0D Address of character
0E Address of integer array
0F Address of char array
10 Test (sp – 1) = (sp)
11 Stop run – (end of program)
12 Test (sp – 1) <> (sp)
13 Cursor position
14 Test (sp – 1) < (sp)
15 Not implemented
16 Test (sp – 1) >= (sp)
17 Input hex number
18 Test (sp – 1) > (sp)
19 Test (sp – 1) <= (sp)
1A OR (sp – 1) with (sp)
1B AND (sp – 1) with (sp)
1C Input number
1D Input character
1E Output number
1F Output a character
20 Not (sp) (Reverse true/false)
21 Output hex number
22 Shift left (sp) bits
23 Output string
24 Shift right (sp) bits
25 Input string into array
26 Increment (sp) by 1
27 Relative Procedure/function call
28 Decrement (sp) by 1
29 Procedure/function return
2A Copy (sp) to (sp + 1)
2B Call absolute address
2C Load integer onto stack
2D Load character onto stack
2E Load absolute address integer
2F Load absolute address character
30 Load integer indexed
31 Load character indexed
32 Store integer
33 Store character
34 Store integer absolute
35 Store character absolute
36 Store integer indexed
37 Store character indexed
38 Absolute Procedure/function call
39 WAIT
3A XOR (exclusive or) (sp – 1) with (sp)
3B Increment stack pointer
3C Jump unconditionally
3D Jump if (sp) zero
3E Jump if (sp) not zero
3F SPRITE
40 POSITIONSPRITE
41 VOICE
42 GRAPHICS
43 SOUND
44 SETCLOCK
45 SCROLL
46 SPRITECOLLIDE
47 GROUNDCOLLIDE
48 CURSORX
49 CURSORY
4A CLOCK
4B PADDLE
4C SPRITEX
4D JOYSTICK
4E SPRITEY
4F RANDOM
50 ENVELOPE
51 SCROLLX
52 SCROLLY
53 SPRITESTATUS
54 MOVESPRITE
55 STOPSPRITE
56 STARTSPRITE
57 ANIMATESPRITE
58 ABS (take absolute value of (sp))
59 INVALID
5A LOAD
5B SAVE
5C SPRITEFREEZE
5D FREEZESTATUS
5E Output a carriage return

80 to FF: Load short literal: P-code – $80 e.g. 80 (hex) means load 0, 81 (hex) means load 1 etc.

HOW TO LOAD G-PASCAL FROM DISK

Once you have turned on the power to your disk drive and Commodore 64, insert the G-Pascal diskette and type:
LOAD "GPASCAL",8

G-Pascal will take about 45 seconds to load. When it has loaded type:
RUN

G-Pascal will now display its 'Main Menu' as described in the Manual.

In the event of a load error there is a second, identical, copy of G-Pascal on the diskette called 'G-PASCAL BACKUP'.

On the disk is also a demonstration program written in G-Pascal, called 'DEMO'. Once G-Pascal is running just Load DEMO, Compile it, and Run it.

We recommend that you *do not* remove the 'write protect' tab from your diskette. Once you have loaded G-Pascal, remove the G-Pascal diskette and store it in a safe place. Then insert a 'work' disk that will be used to store your Pascal programs.

HOW TO LOAD G-PASCAL FROM CASSETTE

Once you have turned on the power to your Commodore 64, insert the G-Pascal cassette into the cassette player and type:
LOAD

When the screen displays "FOUND GPASCAL" press the Commodore logo key (bottom left-hand corner of keyboard).

G-Pascal will take about five and a half minutes to load. When it has loaded type:
RUN

G-Pascal will now display its 'Main Menu' as described in the Manual.

(A simpler method of loading G-Pascal from cassette is to just press the SHIFT and RUN/STOP keys simultaneously. This will automatically load and run G-Pascal).

In the event of a load error, a second copy of G-Pascal is on the reverse side of the cassette. If you are having loading problems make sure that the cassette player's read/write heads are clean, and that the cassette player is not too near the TV set (TV sets generate strong magnetic fields).