QUANTA HELPLINE Dilwyn Jones

Members wishing to submit helpline requests via email can use the email address helpline@quanta.org.uk or if you prefer to use traditional post, please send the helpline request to me via the address printed inside the front cover of the newsletter.

Obviously, we cannot guarantee to answer every query we receive, but we will do our best! Where we have been unable to answer the queries, we may print the help request as an open request in the newsletter to ask if any of the readers can come up with a solution. And, of course, if readers feel that they have a better solution than we came up with, or would like to correct any errors we make, please write to us!

_Sav Files

A member on the ql-users mailing list recently raised the issue of decoding tokenised SBASIC programs. These are BASIC program files, saved with the QSAVE command, which are designed to load faster than ordinary plain text BASIC programs saved with the ordinary SAVE command. When I looked into it, I found there wasn't really much by way of publically available documentation on the _sav file format.

QSAVE works by saving the program in pretty much the same way as it is stored in memory, with the procedures, functions, extensions and keywords all stored as tokens. This lets the BASIC interpreter load the program without having to "tokenise" the plain text version of the program - it saves a lot of processing and is particularly effective on slower systems.

QSAVE and QLOAD extensions were originally supplied as a small software package by Liberation Software (publishers of the Qliberator compiler) for the original QL. LOADing SuperBASIC programs was quite slow with long programs on a QL, so Liberation Software released this handy little utility package, which also included a utility called QREF for listing names of variables, procedures, functions, etc. Lots of us still use it today!

A QSAVE _sav file has a table header which includes some name table information at the beginning of the file, which includes information such as the number of name table entries for this program, the name table length and the number of lines in the BASIC program. Then comes the name table itself - the list of names used by the program. The rest of the file is basically a copy of the BASIC program in a very similar format to that in which it is stored in memory.

Acknowledgements

I am very grateful to Per Witte and Norman Dunbar who sent me all sorts of useful information to help me understand the file format. Both sent me programs to help with decoding the programs. Norman wrote a program many years ago to decode these _sav files and as he hadn't updated it for SBASIC, I tentatively volunteered to update it. As it happened, it had been written so well originally that all I had to do was make fairly small changes to allow it to handle two new facilities in SBASIC (integer and hex constants) and change a couple of variable names to avoid clashes with new extension names in SBASIC. Norman kindly gave me permission to publish the new listing. There's also a wealth of information in the Jan Jones book, "QL SuperBASIC - The Definitive Handbook" in Appendix C which lists the tokens used to store a SuperBASIC program.

Tokens

The tokens are represented by special values in memory. For example, the kyword "REMark" is not stored as the word REMark, but rather as a two byte value, hex 811e.

Now this sounds awfully complex, and in one way it is - you have to be able to understand hex numbers to follow much of this, as you have to follow a trail of hexadecimal values to decode a tokenised SuperBASIC program. Or SBASIC program - the basic file format is essentially the same, just that SBASIC has added a few new tokens for binary and hexadecimal constants using the % and $ prefixes for values, e.g. %1111 or $F to represent the decimal value 15.

Don't worry too much if you don't understand all this - you can still use the program and try to follow how it is decoding the file. This might be useful to those who might like to use it to help them write a program to handle and manipulate these programs, e.g. advanced users might like to try to write a program which copies and pastes whole routines, or builds libraries of routines, a kind of basic development environment, even!

The tokens which represent the various keywords, operators, separators, names and so on all start with $8xxx. The second "nybble" in the hex value indicates the group or type of name:

$80xx indicate snumber of running spaces
$81xx keywords
$82xx unused
$83xx unused
$84xx symbol identifier
$85xx operator
$86xx monadic operators
$87xx unused
$8800 xxxx Names - Number of entry in name table
$89xx unused
$8Axx unused
$8B..... strings
$8C00... text
$8D00xxxx Line number
$8Exx Separators
$8F... Floating point value, top 4 bits indicate if a float, binary, or hex representation

These are explained in more detail in the Jan Jones book, but also quite easy to follow in Norman's listing below.

The procedure called Decode_Header decodes the short header with name table information. It tries to identify the file as a _sav file by looking at the first four bytes of the file in line 310, but two of these bytes can vary depending on how old the particular file version is - if you know of any other values to check for here, let me know! Then, the Decode_Name_Table routine wades through the name table list, and finally the Decode_Program routine steps through the program changing it back to plain text. Which is essentially what this program does - decodes a _sav file into an ordinary untokenised BASIC program.

The program is well commented (thanks, Norman!) to help you follow what it does.

After the Decode_Program routine comes a series of routines such as Multi_Spaces and Keywords which show how to handle the various tokens. At the end of the listing comes an initialisation routine which has a list of the keywords, operators and separators corresponding to token values.

```
100 REMark _SAV file decoder
```

```
110 :
120 CLS
130 PRINT 'SAV File Decoder'\\
140 INPUT 'Which _sav file ? ';sav$
150 IF sav$ = '' THEN STOP: END IF
160 :
170 initialise
180 decode_header
190 IF NOT _quit
200    decode_name_table
210    decode_program
220 END IF
230 RELEASE_HEAP float_buffer
240 CLOSE #3
250 :
260 DEFine PROCedure decode_header
270   LOCal head$(4), name_table_length
280   _quit = 0
290   OPEN_IN #3,sav$
300   head$ = FETCH_BYTES(#3, 4)
310   IF (head$ <> 'Q1' & CHR$(0) & CHR$(0)) AND (head$ <> 'Q1' &
CHR$(2) & CHR$(192)) AND (head$ <> 'Q1'& CHR$(3) & CHR$(128))
320   PRINT head$, head$(1);head$(2)!!CODE(head$(3))!CODE(head$(4))\
330      PRINT sav$ & ' is not a SAV file, or has a new flag.'
340      CLOSE #3
350      _quit = 1
360      RETurn
370   END IF
380   name_table_entries = GET_WORD(#3)
390   name_table_length = GET_WORD(#3)
400   program_lines = GET_WORD(#3)
410   max_name_size = name_table_length - (4 * name_table_entries) /
name_table_entries
420   :
430   PRINT sav$
440   PRINT 'Number of name table entries : '; name_table_entries
450   PRINT 'Name table length            : '; name_table_length
460   PRINT 'Number of program lines      : '; program_lines
470   PRINT
480   :
490   DIM name_table$(name_table_entries -1, max_name_size)
500   float_buffer = RESERVE_HEAP(6)
510   _quit = (float_buffer < 1)
520 END DEFine decode_header
530 :
540 DEFine PROCedure decode_name_table
550   LOCal x, name_type, line_no, name_length, name$, lose_it$(1)
560   LOCal num_procs, num_fns
570   num_procs = 0
580   num_fns = 0
590   FOR x = 0 TO name_table_entries -1
600     name_type = GET_WORD(#3)
610     line_no = GET_WORD(#3)
620     name_length = GET_WORD(#3)
630     name$ = FETCH_BYTES(#3, name_length)
640     IF name_length && 1
650        lose_it$ = INKEY$(#3)
660     END IF
670     IF name_type = 5122 THEN num_procs = num_procs + 1
680     IF name_type >= 5377 AND name_type <= 5379
690        num_fns = num_fns + 1
```

```
700     END IF
710     PRINT x;'  Name type = '; HEX$(name_type, 16) & '  ';
720     PRINT 'Line number = '; line_no & '  ';
730     PRINT 'Name length = '; name_length; '  ';
740     PRINT 'Name = <' & name$ & '>'
750     name_table$(x) = name$
760   END FOR x
770   PRINT 'There are ' & num_procs & ' PROCs'
780   PRINT 'There are ' & num_fns & ' FNs'
790 END DEFine decode_name_table
800 :
810 :
820 DEFine PROCedure decode_program
830   LOCal x, type_byte, program_line
840   :
850   REMark WORD = size change
860   REMark LONG = $8D00.line number
870   REMark rest of line
880   :
890   REPeat program_line
900     IF EOF(#3) THEN EXIT program_line: END IF
910     line_size = line_size + GET_WORD(#3)
920     IF line_size > 65536 THEN line_size = line_size - 65536: END
IF
930     IF GET_WORD(#3) <> HEX('8d00')
940        PRINT 'Program out of step.'
950        CLOSE #3
960        STOP
970     END IF
980     PRINT GET_WORD(#3); ' ';
990     line_done = 0
1000     REPeat line_contents
1010       type_byte = CODE(INKEY$(#3))
1020       SELect ON type_byte
1030         = HEX('80'): multi_spaces
1040         = HEX('81'): keywords
1050         = HEX('84'): symbols
1060         = HEX('85'): operators
1070         = HEX('86'): monadics
1080         = HEX('88'): names
1090         = HEX('8B'): strings
1100         = HEX('8C'): text
1110         = HEX('8E'): separators
1120         = HEX('D0') TO HEX('DF') : floating_points 1 : REMark %
binary number
1130         = HEX('E0') TO HEX('EF') : floating_points 2 : REMark $
hex number
1140         = HEX('F0') TO HEX('FF') : floating_points 3 : REMark
floating point
1150       END SELect
1160       IF line_done THEN EXIT line_contents: END IF
1170     END REPeat line_contents
1180   END REPeat program_line
1190 END DEFine decode_program
1200 :
1210 :
1220 DEFine PROCedure multi_spaces
1230   :
1240   REMark $80.nn = print nn spaces
1250   :
1260   PRINT FILL$(' ', GET_BYTE(#3));
```

```
1270 END DEFine multi_spaces
1280 :
1290 :
1300 DEFine PROCedure keywords
1310   :
1320   REMark $81.nn = keyword$(nn)
1330   :
1340   PRINT keyword$(GET_BYTE(#3));' ';
1350 END DEFine keywords
1360 :
1370 :
1380 DEFine PROCedure symbols
1390   LOCal sym
1400   :
1410   REMark $84.nn = symbol$(nn)
1420   :
1430   sym = GET_BYTE(#3)
1440   PRINT symbol$(sym);
1450   line_done = (sym = 10)
1460 END DEFine symbols
1470 :
1480 :
1490 DEFine PROCedure operators
1500   :
1510   REMark $85.nn = operator$(nn)
1520   :
1530   PRINT operator$(GET_BYTE(#3));
1540 END DEFine operators
1550 :
1560 :
1570 DEFine PROCedure monadics
1580   :
1590   REMark $86.nn = monadic$(nn)
1600   :
1610   PRINT monadic$(GET_BYTE(#3));
1620 END DEFine monadic
1630 :
1640 :
1650 DEFine PROCedure names
1660   LOCal ignore
1670   :
1680   REMark $8800.nnnn = name_table$(nnnn)
1690   :
1700   ignore = GET_BYTE(#3)
1710   ignore = GET_WORD(#3)
1720   IF ignore > 32768 THEN ignore = ignore - 32768: END IF
1730   PRINT name_table$(ignore);
1740 END DEFine names
1750 :
1760 :
1770 DEFine PROCedure strings
1780   LOCal delim$(1), size
1790   :
1800   REMark $8B.delim.string_size = 'delim'; string; 'delim'
1810   :
1820   delim$ = INKEY$(#3)
1830   size = GET_WORD(#3)
1840   PRINT delim$; FETCH_BYTES(#3, size); delim$;
1850   IF size && 1
1860      size = GET_BYTE(#3)
1870   END IF
```

```
1880 END DEFine strings
1890 :
1900 :
1910 DEFine PROCedure text
1920   LOCal size
1930   :
1940   REMark $8C00.size = text
1950   :
1960   size = GET_BYTE(#3)
1970   size = GET_WORD(#3)
1980   PRINT FETCH_BYTES(#3, size);
1990   IF size && 1
2000      size = GET_BYTE(#3)
2010   END IF
2020 END DEFine text
2030 :
2040 :
2050 DEFine PROCedure separators
2060   :
2070   REMark $8E.nn = separator$(nn)
2080   :
2090   PRINT separator$(GET_BYTE(#3));
2100 END DEFine separators
2110 :
2120 :
2130 DEFine PROCedure floating_points (fp_type)
2140   REMark modified for % and $ SBASIC values 22.01.10 - DJ
2150   LOCal number$(6),fpt
2160   fpt = fp_type : REMark to avoid SEL ON last parameter issue
later
2170   :
2180   REMark fp_type=...
2190   REMark $Dx.xx.xx.xx.xx.xx - %binary number
2200   REMark $Ex.xx.xx.xx.xx.xx - $hex number
2210   REMark $Fx.xx.xx.xx.xx.xx - need to mask out the first $F !
2220   :
2230   MOVE_POSITION #3, -1: REMark back up to the first byte
2240   number$ = FETCH_BYTES(#3, 6)
2250   number$(1) = CHR$( CODE(number$(1)) && 15)
2260   POKE_STRING float_buffer, number$
2270   SELect ON fpt
2280     =1 : PRINT '%';LTrim$(BIN$(PEEK_FLOAT(float_buffer),32));
2290     =2 : PRINT '$';LTrim$(HEX$(PEEK_FLOAT(float_buffer),32));
2300     =3 : PRINT PEEK_FLOAT(float_buffer);
2310   END SELect
2320 END DEFine floating_points
2330 :
2340 DEFine FuNction LTrim$(str$)
2350   REMark added 22.01.10 for % and $ values - DJ
2360   REMark remove leading zeros from binary or hex strings
2370   LOCal a,t$
2380   t$ = str$ : REMark full length by default
2390   FOR a = 1 TO LEN(t$)
2400     IF t$(a) <> '0' THEN t$ = t$(a TO LEN(t$)) : EXIT a
2410   NEXT a
2420     t$ = '0' : REMark in case it was all zeros
2430   END FOR a
2440   RETurn t$
2450 END DEFine LTrim$
2460 :
2470 DEFine PROCedure initialise
```

```
2480   LOCal x
2490   :
2500   _quit = 0
2510   last_line_size = 0
2520   line_size = 0
2530   name_table_entries = 0
2540   :
2550   RESTORE 2580
2560   DIM keyword$(31, 9)
2570   FOR x = 1 TO 31: READ keyword$(x): END FOR x
2580   DATA 'END', 'FOR', 'IF', 'REPeat', 'SELect', 'WHEN', 'DEFine'
2590   DATA 'PROCedure', 'FuNction', 'GO', 'TO', 'SUB', '', 'ERRor',
''
2600   DATA '', 'RESTORE', 'NEXT', 'EXIT', 'ELSE', 'ON', 'RETurn'
2610   DATA 'REMAINDER', 'DATA', 'DIM', 'LOCal', 'LET', 'THEN',
'STEP'
2620   DATA 'REMark', 'MISTake'
2630   :
2640   DIM symbol$(10)
2650   symbol$ =  '=:#,(){} ' & CHR$(10)
2660   :
2670   DIM operator$(22, 5)
2680   FOR x = 1 TO 22: READ operator$(x): END FOR x
2690   DATA '+', '-', '*', '/', '>=', '>', '==', '=', '<>', '<=', '<'
2700   DATA '||', '&&', '^^', '^', '&', 'OR', 'AND', 'XOR', 'MOD'
2710   DATA 'DIV', 'INSTR'
2720   :
2730   DIM monadic$(4, 3)
2740   FOR x = 1 TO 4: READ monadic$(x): END FOR x
2750   DATA '+', '-', '~~', 'NOT'
2760   :
2770   DIM separator$(5, 2)
2780   FOR x = 1 TO 5: READ separator$(x): END FOR x
2790   DATA ',', ';', '\', '!', 'TO'
2800   :
2810 END DEFine initialise
```