CHAPTER 17

LANGUAGE

17.1 INTRODUCTION

The Organiser Programming Language (OPL) is a high level language which has developed from a number of other languages:

C
ARCHIVE (The database module in Xchange)
BASIC
FORTH
PL1

The language is designed to be:

Fast
Compact
Flexible
Accurate
Extensible
Simply overlayed

The language is stack based; all code is held on the stack as are all intermediate results. To achieve speed the source code is translated into an intermediate code (Q code) before it is run.

17.2 DEFINITIONS

17.2.1 VARIABLES

All variables in OPL are held in one of three forms:

Integer
Floating pointing
String

All this can either be simple variables or field variables.

All variables are zeroed when declared by a LOCAL, GLOBAL, OPEN or CREATE statements.

17.2.2 PROCEDURES

OPL is a procedure base language, a number of procedures normally go to make up a program. Up to 16 parameters can be passed to a procedure which always returns a variable.

When a procedure is called a header is placed on the stack, followed by space for variables declared and the Q code itself. When a procedure returns all the stack is freed for use by other procedures. This allows overlaying of code so that programs can run which are substantially bigger than the available memory on the machine.

17.2.3 PARAMETERS

Parameters passed to a procedure may be integer, floating point or string. They are passed by value. On the stack they are in reverse order to the order they are input.

For example the statement "PROC:(12,17.5,"ABC")" will generate the following stack entry before the procedure PROC is called:

```
high memory      00 12
                00                      ; Integer type
                00 00 00 00 50 17 01 00
                01                      ; Floating point type
                03 41 42 43             ; "ABC"
                02                      ; String type
low memory      03                      ; Parameter count
```

17.2.4 ADDRESSES

Memory addresses in OPL are held as integers. Pack addresses are held in 3 bytes. In the CM operating system the most significant byte is ignored.

17.2.5 INTEGERS

An integer is a number between 32767 and -32768. It is stored in memory as a single word. In the source code of the language an integer may be input in hexadecimal by preceding the number by a '$', so $FFFF is a valid number and equal to -1. A number in an OPL program will be taken as integer if it is in the integer range with the one exception, -32768 is taken as a floating point number.

The reason for this is that the translator translates a negative number as the absolute value, followed by a unary minus operator. 32768 is outside the range for integers and so is translated as a floating point number. A small increase in speed and compactness can be obtained by writing negative integers in hexadecimal.

It is very important to anticipate what is taken as integer. For example:

```
        30001/2 is the integer 15000
but
        40001/2 is floating point number 20000.5.
```

To ensure that a number is taken as a floating point number just add a trailing period. '2' is an integer, '2.' is a floating point number.

The calculator translates numbers as floating point. If you wish to put an integer into the calculator you must use the function INT. So, for example, from the calculator:

```
                PRICE:(INT(10))
```

passes the integer 10 to the procedure PRICE.

17.2.6 FLOATING POINT

Floating point numbers are in the range +/-9.99999999999E99 to +/-1E-99. They are held in Binary Coded Decimal (BCD) in 8 bytes; 6 bytes for the mantissa, 1 byte for the exponent, and 1 for the sign.

The decimal number -153 is held as:

```
                00 00 00 00 30 15 02 80
```

where the last byte is the sign byte (either 00 or 80) and the preceding byte the exponent.

The decimal number .0234567 is held as:

```
                00 00 00 67 45 23 FE 00.
```

It is possible for the exponent to go out of range, e.g. 1E99*10 or 1E-99/10. This is reported as an EXPONENT RANGE error.

When floating point numbers are translated they are held in a more compact form. The first byte contains both the sign, in the most significant bit, and the number of bytes following. The next bytes are the significant bytes of the mantissa, the final byte is the exponent.

In Q code the decimal number -153 is represented as:

```
                83 30 15 02.
```

The decimal number .0234567 is represented as:

```
                04 47 45 23 FE
```

This compact form is always preceded by a QI_STK_LIT_NUM operator.

17.2.7 STRINGS

Strings are up to 255 characters long, with a preceding length byte. The string "QWERTY" is held as:

```
                06 51 57 45 52 54 59
```

All string variables, except field strings, are preceded by that variable's maximum length, as declared in the LOCAL or GLOBAL statement.

All strings in OPL have this format. For example when using USR$ the machine code should return with the X register pointing at the length byte of the string to be returned.

17.2.8 ARRAYS

One dimensional arrays are supported for integers, floating point numbers and strings. Multi-dimensional arrays can be easily simulated by the use of integer arithmetic.

Like all other variables, arrays are held on the stack. In the case of string arrays the maximum string length is the first byte, the next word contains the array size, this is followed by data. So, for example,

```
                LOCAL A$(5,3),B%(2),C(3)
                A$(4)="AB"
                C(1)=12345
```

initially sets up memory as follows (from low memory to high memory):

```
High memory     00 00 00 00             ; 5st element of A$()
                00 00 00 00             ; 4nd element of A$()
                00 00 00 00             ; 3rd element of A$()
                00 00 00 00             ; 2th element of A$()
                00 00 00 00             ; 1th element of A$()
                00 05                   ; array size of A$()
                03                      ; max string length of A$()
                00 00                   ; 2st element of B%()
                00 00                   ; 1st element of B%()
                00 02                   ; array size of B%()
                00 00 00 00 00 00 00 00 ; 3rd element of C()
                00 00 00 00 00 00 00 00 ; 2nd element of C()
                00 00 00 00 00 00 00 00 ; 1st element of C()
Low memory      00 03                   ; array size of C()
```

After running the procedure it looks like:

```
High memory     00 00 00 00             ; 5th element of A$()
                02 41 42 00             ; 4th element of A$()
                00 00 00 00             ; 3rd element of A$()
                00 00 00 00             ; 2nd element of A$()
                00 00 00 00             ; 1st element of A$()
                00 05                   ; array size of A$()
                03                      ; max string length of A$()
                00 00                   ; 2st element of B%()
                00 00                   ; 1st element of B%()
                00 02                   ; array size of B%()
                00 00 00 00 00 00 00 00 ; 3rd element of C()
                00 00 00 00 00 00 00 00 ; 2nd element of C()
                00 00 00 50 34 12 04 00 ; 1st element of C()
Low memory      00 03                   ; array size of C()
```

The string and array limits are inserted into the variable space after it has been zeroed. This process is referred to as "fixing up" the variables.

Only available memory limits the size of arrays.

17.2.9 TYPE CONVERSION

Automatic type conversion takes place where possible. For instance:

```
                A=10
```
and
```
                A=FLT(10)
```

produce exactly the same Q code. Whereas:

```
                A=10.
```

has different Q code. All three place the floating point number 10 into the variable A.

When expressions are evaluated the standard left to right rule is applied with type integer being maintained as long as possible. So, for example:

```
                A=1000*1000*1000.
```

generates an "INTEGER OVERFLOW" error. But :

```
                A=1000.*1000*1000
```

does not. This applies to any sub-expressions inside brackets, so:

```
                A=1000.*(1000*1000)
```

generates the overflow error.

17.2.10 RECORDS AND FIELDS

A file consists of a file name record with a number of data records. A record contains at least one character and at most 254 characters.

A record may contain up to 16 fields, delimited by the TAB character (ASCII 9).

Strings are held as the ASCII characters, numbers are held in the ASCII form. So for example after:

```
                OPEN "A:ABC",A,A%,B,C$
                A.A=12
                A.B=3.4
                A.C$="XYZ"
```

the file buffer contains:

```
                len      tab         tab
                0A 31 32 09 33 2E 34 09 58 59 5A.
```

When a file is opened the field names are given. The field names and types are not fixed and may be varied from OPEN to OPEN. When a numeric field is accessed the contents are converted from ASCII to integer or floating point. Should this conversion fail the error "STR TO NUM FAIL" is reported.

When searching for a particular field the field name is matched with the field name buffer (see section 6.3.2) and the corresponding field split out of the file buffer using UT$SPLT.

Note that any string can be assigned to a string field but that if it includes a TAB character it will generate an extra field. For example:

```
                OPEN "A:ABC",A,A$,B$,C$
                A.B$="Hello"
                A.A$="AB"+CHR$(9)+"CD"
                PRINT A.C$
                GET
```

will print "Hello" to the screen. The file buffer contains:

```
                0B 41 42 09 43 44 09 48 65 6C 6C 6F
```

Saving data in ASCII is simple but it is easy to see how data can be compressed by using BCD, hexadecimal or other techniques.

17.2.11 VARIABLE SCOPE

When a procedure is loaded all the LOCALs and GLOBALs declared in it are allocated space on the stack. This area is zeroed and the strings and arrays are fixed up. In other words, the maximum length of each string and the array sizes are filled in.

These variables remain in memory at fixed locations, until execution of the declaring procedure terminates. LOCAL variables are valid only in that procedure, whereas GLOBAL variables are valid in all procedures called by the declaring procedure.

See EXAMPLE 1 & 4.

17.2.12 EXTERNALS

If a variable used in a procedure is not declared LOCAL or GLOBAL in that procedure it is taken as external. The Q code contains a list of externals and these are resolved at run time.

Using the frame pointer, RTA_FP - see section 17.2.13, the previous procedures are checked for all entries in the GLOBAL tables. If a match is found the variable address is inserted in an indirection table. If an external is not found it is reported as an error.

See EXAMPLE 4.

Note that neither the LOCAL names nor the parameter names are present in the Q code, but that GLOBAL names are.

17.2.13 LANGUAGE POINTERS

There are three key pointers used by the language:

RTA_SP	Language stack pointer
RTA_PC	Program counter
RTA_FP	Frame (procedure) pointer

RTA_SP points at the lowest byte of the stack. So if an integer is stacked, RTA_SP is decremented by 2 and the word is saved at the address pointed to by RTA_SP.

RTA_PC points at the current operand/operator executed and is incremented after execution - except at the start of a procedure or a GOTO when RTA_PC is set up appropriately.

RTA_FP points into the header of the current procedure.

Each procedure header has the form:

```
                        Device (zero if top procedure)
```

```
                         Return RTA_PC
                         ONERR address
                         BASE_SP
RTA_FP points at:        Previous RTA_FP
                         Start address of the global name table
                         Global name table
                         Indirection table for externals/parameters
```

This is followed by the variables, and finally by the Q code.

RTA_FP points at the previous RTA_FP, so it is easy to jump up through all the procedures above. The language uses this when resolving external references and when handling errors.

See EXAMPLE 4.

17.2.14 ADDRESSING MODES

Local variables and global variables declared in the current procedure are accessed directly. A reference to such variables is by an offset from the current RTA_FP.

Parameters and externally declared global variables are accessed indirectly. The addresses of these variables are held in the indirection table, the required address in this table is found by adding the offset in the Q code to the current RTA_FP.

See EXAMPLE 4.

17.2.15 TOP LOOP

Each procedure consists of two parts, a header and Q code. The Q code contains all the operands and operators in a table that is run by the TOP LOOP.

The TOP LOOP controls the language, it performs the following functions:

Increment RTA_PC by the B register
Test for the ON/CLEAR key, see section 7.2.1
Test for low battery, see section 7.4.3
Load and execute the next operand/operator
Test carry - if set then initiate error handling

17.3 OPERANDS

Each operand stacks either a constant value or a pointer to a variable.

There are a number of types of operands. Operands are named after their type, the types are:

Integer	INT
Floating point	NUM
String	STR
Constants (i.e. not variables)	CON
Arrays	ARR
Simple (i.e. not array)	SIM
Offset from RTA_FP	FP
Indirect offset from RTA_FP	IND
Left side (i.e. assigns)	LS
Field	FLD
Stack byte/word	LIT
Refer to the fixed memories	ABS

Internal Name	Op	+ Bytes	Added to the stack
QI_INT_SIM_FP	$00	2	The integer
QI_NUM_SIM_FP	$01	2	The floating point number
QI_STR_SIM_FP	$02	2	The string

These operands take the following word, add it to RTA_FP (see section 17.2.13) and stack the variable at that address.

Internal Name	Op + Bytes		Stack
QI_INT_ARR_FP	$03	2	Drops element number, adds an integer from the array
QI_NUM_ARR_FP	$04	2	Drops element number, adds a floating point number from the array
QI_STR_ARR_FP	$05	2	Drops element number, adds a string from the array

These operands take the following word, adds it to RTA_FP to get the start of the array. The required element number is dropped off the stack and checked against the maximum size of the array. The address of the element is then calculated and the variable stacked.

Internal Name	Op + Bytes		Added to the stack
QI_NUM_SIM_ABS	$06	1	Floating point number

This operand gives access to the calculators memories, M0 to M9. The operand is followed by the offset to the memory required.

Internal Name	Op + Bytes		Added to the stack
QI_INT_SIM_IND	$07	2	The integer
QI_NUM_SIM_IND	$08	2	The floating point number
QI_STR_SIM_IND	$09	2	The string

These operands take the following word, add it to RTA_FP, load the address at that address and stack the variable at that address.

Internal Name	Op + Bytes		Stack
QI_INT_ARR_IND	$0A	2	Drops element number, adds the integer from the array
QI_NUM_ARR_IND	$0B	2	Drops element number, adds the floating point number from the array
QI_STR_ARR_IND	$0C	2	Drops element number, adds the string from the array

These operands take the following word, adds it to RTA_FP, loads the address at that address to get the start of the array. The element of the array required is dropped off the stack, it is then checked against the maximum size of the array. The address of the element is then calculated and the variable stacked.

Internal Name	Op + Bytes		Added to the stack
QI_LS_INT_SIM_FP	$0D	2	The address of the integer + field flag
QI_LS_NUM_SIM_FP	$0E	2	The address of the floating point number + field flag
QI_LS_STR_SIM_FP	$0F	2	The maximum size + the address of the string + field flag
QI_LS_INT_ARR_FP	$10	2	The address of the integer from the array + field flag
QI_LS_NUM_ARR_FP	$11	2	The address of the floating point number from the array + field flag
QI_LS_STR_ARR_FP	$12	2	The maximum size + the address of the string from the array + field flag
QI_LS_NUM_SIM_ABS	$13	2	The address of the calculator memory + field flag
QI_LS_INT_SIM_IND	$14	2	The address of the integer + field flag
QI_LS_NUM_SIM_IND	$15	2	The address of the floating point number + field flag
QI_LS_STR_SIM_IND	$16	2	The maximum size + the address of the string + field flag
QI_LS_INT_ARR_IND	$17	2	The address of the integer from the array + field flag
QI_LS_NUM_ARR_IND	$18	2	The address of the floating point number from the array + field flag
QI_LS_STR_ARR_IND	$19	2	the maximum size + the address of the string from the array + field flag

These operands correspond to their right side equivalents. In the case of strings the maximum length is stacked first. Then, in all cases, the address of the variable is stacked. The field flag byte is then stacked, in all these cases it is zero to show that it is not a field reference.

See EXAMPLE 1.

Internal Name	Op + Bytes		Stack
QI_INT_FLD	$1A	1	Drops the field name, adds the integer
QI_NUM_FLD	$1B	1	Drops the field name, adds the floating point number
QI_STR_FLD	$1C	1	Drops the field name, adds the string

These operands are followed by a logical file name, 0,1,2 or 3, which says which logical file to use. First it looks for the field name in the Field Name Symbol Table. If it is found the corresponding field is split from the corresponding File Buffer.

If it is a string it is immediately placed on the stack. If it is numeric it is converted from ASCII to the relevant format and placed on the stack.

Internal Name	Op + Bytes		Stack
QI_LS_INT_FLD	$1D	1	Stacks the logical file name + field flag
QI_LS_NUM_FLD	$1E	1	Stacks the logical file name + field flag
QI_LS_STR_FLD	$1F	1	Stacks the logical file name + field flag

These operands stacks the logical file, the byte following the operand, and the field flag which in this case is non-zero. All the work is done by the assign.

Internal Name	Op + Bytes		Added to the stack
QI_STK_LIT_BYTE	$20	1	The byte
QI_STK_LIT_WORD	$21	2	The word

Stacks the following byte or word. QI_STK_LIT_WORD is identical to QI_INT_CON.

Internal Name	Op + Bytes		Added to the stack
QI_INT_CON	$22	2	Integer
QI_NUM_CON	$23	*	Floating point number (see section 17.2.6)
QI_STR_CON	$24	*	String

Stacks the constant value following.

17.4 OPERATORS

Operators generally do things to the variables already on the stack.

17.4.1 ERRORS, CALLS AND PARAMETERS

In the following section if an operand cannot return an error then no errors are listed.

Any access to a device can result in the following errors. They are no given explicitly as error for that operand/operator:

```
        ER_FL_NP - no pack
        ER_PK_IV - unknown pack
        ER_DV_CA - bad device name
and if the pack was not blank:
        ER_PK_NB - pack not blank
```

When writing to a pack the following are always possible:

```
        ER_FL_PF - pack full
        ER_PK_RO - read only pack
        ER_PK_DE - write error
```

If the operator calls an operating system then that is listed. If no calls are given then the run time code handles it all itself. In general there is no difference between call with a $ and with an _, the $ calls are called through SWIs whereas the _ calls are made directly. Direct calls are faster, but SWIs can be redirected for the addition of extra features. See section 5.1.1 on calling system services.

If there is more than one parameter they are listed. The values are stacked in order. So para1 is stacked before para2 - when the operator is called the last parameter is the one pointed to by the RTA_SP.

17.4.2 LOGICAL AND ARITHMETIC COMPARE OPERATORS

Internal Name	Op	Stack
QCO_LT_INT	$27	Drops 2 INTs, returns 0 or -1 as an INT
QCO_LTE_INT	$28	Drops 2 INTs, returns 0 or -1 as an INT
QCO_GT_INT	$29	Drops 2 INTs, returns 0 or -1 as an INT
QCO_GTE_INT	$2A	Drops 2 INTs, returns 0 or -1 as an INT
QCO_NE_INT	$2B	Drops 2 INTs, returns 0 or -1 as an INT
QCO_EQ_INT	$2C	Drops 2 INTs, returns 0 or -1 as an INT
QCO_ADD_INT	$2D	Drops 2 INTs, returns result as an INT
QCO_SUB_INT	$2E	Drops 2 INTs, returns result as an INT
QCO_MUL_INT	$2F	Drops 2 INTs, returns result as an INT
QCO_DIV_INT	$30	Drops 2 INTs, returns result as an INT
QCO_POW_INT	$31	Drops 2 INTs, returns result as an INT
QCO_UMIN_INT	$32	Drops an INT, returns result as an INT
QCO_NOT_INT	$33	Drops an INT, returns result as an INT
QCO_AND_INT	$34	Drops 2 INTs, returns result as an INT
QCO_OR_INT	$35	Drops 2 INTs, returns result as an INT
QCO_LT_NUM	$36	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_LTE_NUM	$37	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_GT_NUM	$38	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_GTE_NUM	$39	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_NE_NUM	$3A	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_EQ_NUM	$3B	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_ADD_NUM	$3C	Drops 2 NUMs, returns result as an NUM
QCO_SUB_NUM	$3D	Drops 2 NUMs, returns result as an NUM
QCO_MUL_NUM	$3E	Drops 2 NUMs, returns result as an NUM
QCO_DIV_NUM	$3F	Drops 2 NUMs, returns result as an NUM
QCO_POW_NUM	$40	Drops 2 NUMs, returns result as an NUM
QCO_UMIN_NUM	$41	Drops a NUM, returns result as an NUM
QCO_NOT_NUM	$42	Drops a NUM, returns 0 or -1 as an INT
QCO_AND_NUM	$43	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_OR_NUM	$44	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_LT_STR	$45	Drops 2 STRs, returns 0 or -1 as an INT
QCO_LTE_STR	$46	Drops 2 STRs, returns 0 or -1 as an INT
QCO_GT_STR	$47	Drops 2 STRs, returns 0 or -1 as an INT
QCO_GTE_STR	$48	Drops 2 STRs, returns 0 or -1 as an INT
QCO_NE_STR	$49	Drops 2 STRs, returns 0 or -1 as an INT
QCO_EQ_STR	$4A	Drops 2 STRs, returns 0 or -1 as an INT
QCO_ADD_STR	$4B	Drops 2 STRs, returns result as a STR

The compares drop whatever is on the stack and return an integer either TRUE(-1) or FALSE(0).

NOT, AND, and OR are bitwise on INTs, but on NUMs they are logical. So the following equalities are true:

```
NOT(3.0) = 0;   (3.0 AND 5.0) = -1;     (3.0 OR 5.0) = -1;
NOT(3) = -4;    (3 AND 5) = 1;          (3 OR 5) = 7;
```

The string compares are case sensitive.

Divide by zero generates the error ER_FN_BA.

The function X^{Y} will generate ER_FN_BA if X zero and Y less than or equal to zero, X negative and Y non-integer. NOTE VERY WELL: In the calculator all numeric constants are automatically converted to floating point. So in the calculator NOT(3) evaluates to 0, whereas NOT(INT(3)) is -4.

Note also: Outside the calculator a simple number is taken as an integer if it is less than 32768 and more than -32768, so in a procedure 10**10 gives an INTEGER OVERFLOW error.

17.5 COMMAND OPERATORS

17.5.1 QCO_AT

Positions the cursor.

```
OP:     $4C
OPL:    AT
Para1:  New X position (1 to 16)
Para2:  New Y position (1 or 2)
Stack:  Drops the two integers on the stack
Calls:  DP$STAT
Errors: ER_FN_BA - Bad parameter if either parameter out of range.
```

Clears RTB_CRFL, the carriage return flag.

17.5.2 QCO_BEEP

Beeps with a frequency of 460800/(39+para2).

```
OP:     $4D
OPL:    BEEP
Para1:  Integer duration in milliseconds
Para2:  Integer period
Stack:  Drops the two integers
Calls:  BZ_TONE
Bugs:   If para1 is negative BEEP returns immediately.
        Para2 is regarded as an unsigned word.
```

17.5.3 QCO_BREAK

Break the execution of OPL. Note that this is not equivalent to the OPL word BREAK.

```
OP:     $26
Calls:  UT_LEAV
```

17.5.4 QCO_CLS

Clears the screen. The cursor is homed to the top left.

```
OP:     $4E
OPL:    CLS
Stack:  No effect
Calls:  DP_CLRB
```

17.5.5 QCO_CURSOR

Set the cursor on or off.

```
OP:     $4F
OPL:    CURSOR ON, CURSOR OFF
Stack:  No effect
Calls:  DP$STAT
```

Gets byte after operator, sets or clears most significant bit of DPB_CUST.

17.5.6 QCO_ESCAPE

Enables or disables the ON/CLEAR key freeze and quit.

```
OP:     $50
OPL:    ESCAPE ON, ESCAPE OFF
```

Stack: Drops the integer on the stack

Gets byte after operator, sets or clears RTA_ESCF.

17.5.7 QCO_GOTO

Jump RTA_PC to a new location in the same procedure.

```
OP:     $51
OPL:    GOTO, BREAK, CONTINUE, ELSE
Stack:  No effect
```

Adds word after the operator to RTA_PC. See QCO_BRA_FALSE.

17.5.8 QCO_OFF

Turns off the machine. Does not terminate language execution.

```
OP:     $52
OPL:    OFF
Stack:  No effect
Calls:  BT_SWOF
```

This is exactly the same state as when the machine is turned off at the top level. The drain on the battery is minimal. See section 5.4.

17.5.9 QCO_ONERR

Set up error handling.

```
OP:     $53
OPL:    ONERR, ONERR OFF
Stack:  No effect
```

The following word contains the offset to the address to jump to in the event of an error being detected. ONERR OFF is the same operator followed by a zero word. The ONERR address is saved in the header, see section 17.2.13.

17.5.10 QCO_PAUSE

If positive it pauses for that many 50 millisecond units, if negative it pauses for that many 50 millisecond units or until the first key press. If it is zero it waits for the next key press.

```
OP:     $54
OPL:    PAUSE
Stack:  Drops the integer
Bugs:   If a key is pressed it is not removed from the input buffer, so
        it should be read by a KEY or GET function.
```

Uses the 'SLP' processor instruction, so less power is used when PAUSEd compared to normal operation. It does however use more power than being switched off. See section 3.2.

17.5.11 QCO_POKEB

Pokes a byte into memory.

```
OP:     $55
OPL:    POKEB
Para1:  Address to write to
Para2:  Byte to be written
Stack:  Drops the two integers
Errors: ER_FN_BA - Bad parameter
```

Reports an error if para2 is not a byte. If the address is in the protected range $00 to $3F or $282 to $400 then it does nothing.

17.5.12 QCO_POKEW

Pokes a word into memory.

```
OP:     $56
OPL:    POKEW
Para1:  Address to write to
Para2:  Word to be written
Stack:  Drops the two integers on the stack
Errors: ER_FN_BA - Bad parameter
```

If the address is in the protected range $00 to $3F or $282 to $400 then it does nothing.

17.5.13 QCO_RAISE

Generates an error condition.

```
OP:     $57
OPL:    RAISE
Stack:  Drops the integer
Errors: ER_FN_BA - Bad parameter
```

If integer on the stack is not a byte it reports error. Otherwise it has exactly the same effect as if that error was generated. Errors generated by RAISE are handled in the normal way by ONERR.

Using this command and ONERR the programmer can completely take-over the handling and reporting of errors.

If the error is out of the range normally reported by the OS the message "*** ERROR ***" is reported.

RAISE 0 is special as it does not report an error.

17.5.14 QCO_RANDOMIZE

Set the seed of the random number generator. The sequence numbers generated by RND becomes repeatable.

```
OP:     $58
OPL:    RANDOMIZE
Stack:  Drops the floating point number on the stack
Calls:  FN_RAND
```

17.5.15 QCO_SPECIAL

Special operator used to vector to machine code.

```
OP:     $25
OPL:    See below
Stack:  No effect
```

Vectors via the contents of the location RTA_1VCT to machine code. The machine code should return with the carry flag set to report an error.

If the ASCII value 1 is encountered in the OPL source code it is taken to be a SPECIAL call which returns an integer. A 2 is for a floating point return and 3 for a string. It is impossible to get these values into the source code from the editor, it must be generated by another program.

For example if you want to write an evaluator for a spreadsheet and you want to add cell A1 to cell B1 you could poke in:

```
        01 ????
```

17.5.16 QCO_STOP

Stops executing the language.

```
OP:     $59
OPL:    STOP
```

Resets RTA_SP, zeroes the file buffers by calling AL_ZERO and leaves the language.

17.5.17 QCO_TRAP

Disables the reporting of any error arising from the execution of the following operator. Instead the error number is saved in RTB_EROR which can be read by the function ERR.

```
OP:     $5A
OPL:    TRAP
Stack:  No effect
```

Clears RTB_EROR and sets the trap flag RTB_TRAP.

The following operators can be used with TRAP:

APPEND	BACK	CLOSE
COPY	CREATE	DELETE
ERASE	EDIT	FIRST
INPUT	LAST	NEXT
OPEN	POSITION	RENAME
UPDATE	USE	

If no error occurs these operators clear RTB_TRAP.

Most of these are file-related operator. The programmer will frequently either need to report errors arising from the operators himself or handle them in a discriminating way. For example:

```
                TRAP OPEN "B:XYZ",A,A$
                IF ERR
                 TRAP OPEN "C:XYZ",A,A$
                 IF ERR
                  CLS :PRINT "FILE XYZ NOT" :PRINT "FOUND"
                  BEEP 100,100 :GET :STOP
                 ENDIF
                ENDIF
```

INPUT and EDIT are different. TRAP changes the conditions under which they exit. "EDIT A$" will not exit on the ON/CLEAR key, "TRAP EDIT A$" will exit with RTB_EROR set to ER_RT_BK. When inputting a number without the TRAP option, the routine will not exit until a valid number is input; however with TRAP any input will be accepted and the corresponding error condition placed in RTB_EROR.

See QCO_INPUT_INT, QCO_INPUT_NUM, QCO_INPUT_STR, QCO_EDIT.

17.6 FILE OPERATORS

17.6.1 QCO_APPEND

Adds the current record buffer to the current file as a new record.

```
OP:     $5B
OPL:    APPEND
Stack:  No effect
Errors: ER_RT_FC - file not open
Calls:  FL$SETP, FL$RECT, FL$RSET, FL$WRIT
Bugs:   If the current length of the current record is zero, it is
        automatically made non-zero by adding a TAB, the field delimiter.
```

The contents of the file buffer are saved at the end of the current device.
The first byte of the buffer is the length of the buffer.

17.6.2 QCO_CLOSE

Closes the current file.

```
OP:     $5C
OPL:    CLOSE
Stack:  No effect
Errors: ER_RT_FC - file not open
Calls:  FL$SETP, FL$RECT, FL$RSET, AL$ZCEL
Bugs:   After closing the file it looks for another file to make current.
        If several files are open it is unpredictable which will
        become current.
```

CLOSE has no effect on the file itself, it checks that the file is open,
clears the record type in RTT_FIL, and zeroes the two cells.

17.6.3 QCO_COPY

Copies a file from one device to another. If the target already exists the
data is appended.

```
OP:     $5D
OPL:    COPY
Stack:  Drops the names of the two files
Errors: ER_FL_NX - file does not exist
        ER_PK_CH - changed pack
Calls:  fl$copy
Bugs:   You cannot copy to the same device.
```

17.6.4 QCO_CREATE

Creates a file.

```
OP:     $5E
OPL:    CREATE
Stack:  Drops the name of the file to be created
Errors: ER_FL_EX - file already exists
        ER_AL_NR - out of memory
Calls:  FL$CRET, AL$GROW, FL$SETP, FL$RECT, FL$RSET, FL$READ
```

See EXAMPLE 2.

17.6.5 QCO_DELETE

Deletes a file.

```
OP:     $5F
OPL:    DELETE
Stack:  Drops the name of the file to be deleted.
Errors: ER_FL_NX - file does not exist
        ER_RT_FO - file open
Calls:  FL$DELN
```

Checks that the file is not open. Deletes all records, starting with the
first, and finally the file name record of the file.

17.6.6 QCO_ERASE

Erases the current record of the current file.

```
OP:     $60
```

```
OPL:    ERASE
Stack:  No effect
Errors: ER_RT_FC - file not open
        ER_FL_EF - end of file
Calls:  FL$ERAS, FL$SETP, FL$RECT, FL$RSET, FL$READ
Bugs:   The current record becomes the record following the erased record.
        If, after the erase, FL$READ returns an 'END OF FILE', the length
        of the current record is set to zero and the current record number
        set to the number of records (as found  by FL$SIZE) plus one.

        'END OF FILE' error will be generated if already at the end
        of the file. This includes the case of a file with no records.
```

17.6.7 QCO_FIRST

Goes to the first record of the current file.

```
OP:     $61
OPL:    FIRST
Stack:  No effect
Errors: ER_RT_FC - file not open
Calls:  FL$SETP, FL$RECT, FL$RSET, FL$READ
Bugs:   No error reported if there are no records.
```

17.6.8 QCO_LAST

Goes to the last record of the current file.

```
OP:     $62
OPL:    LAST
Stack:  No effect
Errors: ER_RT_FC - file not open
Calls:  FL$SIZE, FL$SETP, FL$RECT, FL$RSET, FL$READ
Bugs:   No error reported if there are no records.
```

17.6.9 QCO_NEXT

Goes to the next record.

```
OP:     $63
OPL:    NEXT
Stack:  No effect
Errors: ER_RT_FC - file not open
Calls:  FL$NEXT, FL$READ
Bugs:   No error reported if at the end of file.
        If FL$READ returns an "END OF FILE" error, the length of the
        current record is set to zero and the current record number set
        to the number of records (as found by FL$SIZE) plus one.
```

17.6.10 QCO_BACK

Steps back one record.

```
OP:     $64
OPL:    BACK
Stack:  No effect
Errors: ER_RT_FC - file not open
Calls:  FL$BACK
Bugs:   No error reported if already on the first record.
```

17.6.11 QCO_OPEN

Open a file.

```
OP:     $65
```

```
OPL:    OPEN
Stack:  Drop the name of the file.
Errors: ER_RT_FO - file open
Calls:  FL$OPEN, FL$SETP, FL$RECT, FL$RSET, FL$READ
```

OPEN has exactly the same form as CREATE. See EXAMPLE 2.

17.6.12 QCO_POSITION

Position at that record.

```
OP:     $66
OPL:    POSITION
Stack:  Drops the integer
Errors: ER_RT_FC - file not open
Calls:  FL$SETP, FL$RECT, FL$RSET, FL$READ
Bugs:   If the FL$READ returns an 'END OF FILE', the length of the current
        record is set to zero and the current record number set to the
        number of records (as found by FL$SIZE) plus one.
```

17.6.13 QCO_RENAME

Renames a file.

```
OP:     $67
OPL:    RENAME
Stack:  Drops the two file names
Errors: ER_RT_FO - file open
        ER_FL_NX - file exists
        ER_FL_NX - file does not exist
Calls:  FL$RENM
```

Erases the file name record and writes a new one.

17.6.14 QCO_UPDATE

Updates a record.

```
OP:     $68
OPL:    UPDATE
Stack:  No effect
Errors: ER_RT_FC - file not open
Calls:  FL$ERAS, FL$WRIT, FL$SETP, FL$RECT, FL$RSET, FL$READ
Bugs:   If the APPEND fails, with 'PAK FULL' for example, the original
        record is already erased.
```

It deletes the current record in the current file and then APPENDs the contents of the buffer.

17.6.15 QCO_USE

Changes the current file.

```
OP:     $69
OPL:    USE
Stack:  No effect
Errors: ER_TR_BL - bad logical name (logical name not in use)
```

Takes the byte following the operator and after checking it makes it the new current logical file. See section 17.11.3.

17.7 OTHER OPERATORS

17.7.1 QCO_KSTAT

Set the shift state of the keyboard.

```
OP:     $6A
OPL:    KSTAT
Stack:  Drops integer
Errors: ER_FN_BA - function argument error
Calls:  KB$STAT
```

Use KSTAT to change the upper/lower alpha/numeric case:

1	alpha, upper case (default setting)
2	alpha, lower case
3	numeric, upper case
4	numeric, lower case

17.7.2 QCO_EDIT

Edits a string.

```
OP:     $6B
OPL:    EDIT
Stack:  Drop the left side reference to string
Errors: ER_RT_BK - ON/CLEAR key pressed
        ER_RT_FC - file not open
        ER_RT_NF - field not found
        ER_RT_RB - record too big
Calls:  ED$EDIT
```

If the string to be edited is a field then the maximum length of the string is 252. Otherwise the maximum length allowed is the length of the string as defined in the LOCAL or GLOBAL statement. The string to be edited is copied into RTT_BUF. Once the string is edited it is assigned to the source.

If the EDIT is preceded by TRAP then the edit will exit on the ON/CLEAR key with the error condition ER_RT_BK. The string remains unchanged.

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

17.7.3 QCO_INPUT_INT

Input an integer.

```
OP:     $6C
OPL:    INPUT
Stack:  Drops the left side integer reference
Errors: ER_RT_BK - ON/CLEAR key pressed
        ER_MT_IS - conversion to number failed
        ER_RT_IO - integer overflow
        ER_RT_FC - file not open
        ER_RT_NF - field not found
        ER_RT_RB - record too big
Calls:  ED$EDIT
```

If the INPUT is preceded by TRAP then the input will exit on the ON/CLEAR key with the error condition ER_RT_BK. It will also exit if an invalid integer is input, e.g. 99999 or $1.

If there is no TRAP then the INPUT will not exit on the ON/CLEAR key and invalid integers generate a '?' on the next line and the INPUT is repeated.

Up to 6 characters, including leading spaces, are allowed.

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

17.7.4 QCO_INPUT_NUM

Inputs a floating point number.

```
OP:     $6D
OPL:    INPUT
Stack:  Drops left side reference to floating point number
Errors: ER_RT_BK - ON/CLEAR key pressed
        ER_MT_IS - conversion to number failed
        ER_RT_IO - integer overflow
        ER_RT_FC - file not open
        ER_RT_NF - field not found
        ER_RT_RB - record too big
Calls:  ED$EDIT
```

If the INPUT is preceded by TRAP then the input will exit on the ON/CLEAR key with the error condition ER_RT_BK. It will also exit if an invalid floating point number is input, e.g. 999999999999999 or $1.

If there is no TRAP then the INPUT will not exit on the ON/CLEAR key and invalid integers generate a '?' on the next line and the INPUT is repeated.

Up to 15 characters, including leading spaces, are allowed.

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

17.7.5 QCO_INPUT_STR

Inputs a string.

```
OP:     $6E
OPL:    INPUT
Stack:  Drops left side reference to string
Errors: ER_RT_FC - file not open
        ER_RT_NF - field not found
        ER_RT_RB - record too big
Calls:  ED$EDIT
```

QCO_INPUT_STR is exactly equivalent to QCO_EDIT with an initial null string.

17.7.6 QCO_PRINT_INT

Prints an integer to the screen.

```
OP:     $6F
OPL:    PRINT
Stack:  Drops the integer
Calls:  UT$DISP
Bugs:   If the number $FFFF is assigned to an integer and then it is
        printed it will be represented as -1.
```

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

17.7.7 QCO_PRINT_NUM

Prints a floating point number to the screen.

```
OP:     $70
OPL:    PRINT
Stack:  Drops the floating point number
Calls:  UT$DISP
```

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared. The format in which a number is displayed is integer, decimal or scientific in that order of precedence.

17.7.8 QCO_PRINT_STR

Print a string to the screen.

```
OP:     $71
OPL:    PRINT
Stack:  Drops the string
Calls:  UT$DISP
```

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

17.7.9 QCO_PRINT_SP

Prints a space to the screen.

```
OP:     $72
OPL:    PRINT
Stack:  No effect
Calls:  UT$DISP
```

This operator is generated by use of the ',' separator in a PRINT statement.

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

17.7.10 QCO_PRINT_CR

Print a carriage return to the screen.

```
OP:     $73
OPL:    PRINT
Stack:  No effect
Calls:  UT$DISP
```

If a PRINT, INPUT or EDIT statement is not followed by a ';' or ',' then this operator is automatically inserted. It is not acted on immediately; it sets the flag RTB_CRFL.

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

Note that if a carriage return results in scrolling the screen there is an automatic delay; the length of this delay is defined by DPW_DELY which is in 50 millisecond units, the default being 10.

17.7.11 QCO_LPRINT_INT

Sends an integer to the RS232.

```
OP:     $74
OPL:    LPRINT
Errors: ER_DV_NP - device missing
        ER_DV_CS - device load error
```

Exactly as PRINT_INT, except the CR flag is not tested.

17.7.12 QCO_LPRINT_NUM

Send a floating point number to the RS232.

```
OP:     $75
OPL:    LPRINT
Errors: ER_DV_NP - device missing
        ER_DV_CS - device load error
```

Exactly as PRINT_NUM, except the CR flag is not tested.

17.7.13 QCO_LPRINT_STR

Send a string to the RS232.

```
OP:     $76
OPL:    LPRINT
Errors: ER_DV_NP - device missing
        ER_DV_CS - device load error
```

Exactly as PRINT_STR, except the CR flag is not tested.

17.7.14 QCO_LPRINT_SP

Send a space character to the RS232.

```
OP:     $77
OPL:    LPRINT
Errors: ER_DV_NP - device missing
        ER_DV_CS - device load error
```

Exactly as PRINT_SP, except the CR flag is not tested.

17.7.15 QCO_LPRINT_CR

Send a carriage return to the RS232.

```
OP:     $78
OPL:    LPRINT
Errors: ER_DV_NP - device missing
        ER_DV_CS - device load error
```

As PRINT_CR except it is acted on immediately.

17.7.16 QCO_RETURN

Return from a procedure.

```
OP:     $79
OPL:    RETURN
Stack:  Unwinds the procedure
```

This operator follows the operator which stacks the return value.

All procedures return a value. If no explicit value is returned then it will return integer zero for integer procedures, floating point zero for floating point procedures or a null string for string procedures.

See EXAMPLE 5.

17.7.17 QCO_RETURN_NOUGHT

For an integer procedure this is the default return.

```
OP:     $7A
OPL:    RETURN
Stack:  Stack the integer zero and then unwind the procedure
```

Stacks default return value, then exactly the same as QCO_RETURN.

17.7.18 QCO_RETURN_ZERO

For an floating point procedure this is the default return.

```
OP:     $7B
OPL:    RETURN
Stack:  Stack a floating point zero and then unwind the procedure
```

Stacks default return value, then exactly the same as QCO_RETURN.

17.7.19 QCO_RETURN_NULL

For a string procedure this is the default return.

```
OP:     $7C
OPL:    RETURN
Stack:  Adds a null string and the unwinds the procedure
```

Stacks default return value, on the stack, then exactly the same as QCO_RETURN.

17.7.20 QCO_PROC

Call a procedure.

```
OP:     $7D
OPL:    procnam:
Stack:  Initialises procedure
Errors: ER_RT_PN - procedure not found
        ER_RT_NP - wrong number of parameters
        ER_RT_UE - undefined external
        ER_EX_TV - parameter type mis-match
        ER_AL_NR - out of memory
        ER_GN_BL - test explicitly for low battery error
Calls:  PK$RBYT, PK$RWRD, PK$READ, DV$LKUP, DV$VECT
```

First checks to see if a language extension of that name has been booted into memory (see section 11.1.4.3). If not it searches the 4 devices for an OPL procedure of the right name. It starts with the default device. So if the procedure called was on C: then it searches in the order C:, D:, A: and B:.

If a language extension has been found (for example LINPUT) it calls the relevant vector and the device is then responsible for checking the parameters and handling the stack. See section 17.12.2.

If it is an OPL procedure the header information is read in and the memory required checked. The external references are then checked and the fixups on the strings and arrays performed. See EXAMPLE 4.

The Q code is then read in, and RTA_PC and RTA_SP are set to their new values.

17.7.21 QCO_BRA_FALSE

Branches if the integer on the stack is false.

```
OP:     $7E
OPL:    UNTIL, WHILE, IF, ELSEIF
Stack:  Drop the offset
```

Adds the integer following the operator to RTA_PC if the value on the stack is zero.

17.7.22 QCO_ASS_INT

Assign an integer to a variable.

```
OP:     $7F
OPL:    =
Stack:  Drops the integer and the integer reference
```

```
Errors: ER_RT_RB - field too big
        ER_RT_FC - file not open
        ER_RT_NF - field not found
        ER_RT_RB - record too big
```

At the start of the operand the stack looks like:

```
High memory             Address of integer variable
                        0 (field flag)
Low memory              Integer
                or:
High memory             Field name
                        Logical file name (0,1,2 or 4)
                        1 (field flag)
Low memory              Integer
```

If the assign is to a field, it checks that the file is open, checks the field name and saves the value.

If not a field it simply saves the integer to the address.

See EXAMPLE 4.

17.7.23 QCO_ASS_NUM

Assigns a floating point number.

```
OP:     $80
OPL:    =
Stack:  Drops the floating point number and the floating point reference
Errors: ER_RT_RB - field too big
        ER_RT_FC - file not open
        ER_RT_NF - field not found
        ER_RT_RB - record too big
```

Exactly the same as QCO_ASS_INT except it handles floating point numbers. See EXAMPLE 4.

17.7.24 QCO_ASS_STR

Assigns a string.

```
OP:     $81
OPL:    =
Stack:  Drops the string and the string reference
Errors: ER_RT_RT - field too big
        ER_LX_ST - string too long
```

Exactly the same as QCO_ASS_INT except it handles strings. See EXAMPLE 4.

17.7.25 QCO_DROP_BYTE

Drops a byte off stack.

```
OP:     $82
OPL:    -
Stack:  Drops byte
```

17.7.26 QCO_DROP_WORD

Drops a word off the stack.

```
OP:     $83
OPL:    -
Stack:  Drops word
```

Used internally to drop unwanted results off the stack, for example a statement "GET" which translates into RTF_GET,QCO_DROP_WORD.

17.7.27 QCO_DROP_NUM

Drops a floating point number off the stack.

```
OP:     $84
OPL:    -
Stack:  Drops a floating point number
```

Used internally to OPL when, for example, a floating point procedure returns a value that is not required.

17.7.28 QCO_DROP_STR

Drops a string off the stack.

```
OP:     $85
OPL:    -
Stack:  Drops a string off the stack
```

Used internally to OPL when, for example, a string procedure returns a string that is not required.

17.7.29 QCO_INT_TO_NUM

Converts an integer into a floating point number.

```
OP:     $86
OPL:    -
Stack:  Drops an integer, stacks a float
Calls:  MT$BTOF
Bugs:   Integers are always taken as signed. To make unsigned:
                A=I% :IF I%<0 :A=A+65536 :ENDIF
```

Used for automatic type conversion.

17.7.30 QCO_NUM_TO_INT

Converts a floating point number to integer.

```
OP:     $87
OPL:    -
Stack:  Drops float, stacks integer
Errors: ER_RT_IO - integer overflow
Calls:  IM$DINT, IM$FLOI
Bugs:   Always rounds down, 3.9 becomes 3 and -3.9 becomes -4.
```

Used for automatic type conversion.

17.7.31 QCO_END_FIELDS

Indicates where the field names end.

```
OP:     $88
OPL:    OPEN, CREATE
Stack:  No effect
```

Only used internally at the end of an OPEN or CREATE command. See EXAMPLE 2.

17.7.32 QCO_RUN_ASSEM

Runs machine code immediately after operator.

```
OP:     $89
OPL:    -
Stack:  No effect
```

Runs the code immediately after the operator as machine code. On return if there are no errors carry must be clear and the B register must be the number of bytes for RTA_PC to jump. If there is an error carry must be set and the B register should contain the number of the error to be reported.

This cannot be generated from the editor.

17.8 INTEGER FUNCTIONS

These functions return integer values.

17.8.1 RTF_ADDR

Returns the address of a numeric variable.

```
OP:     $8A
OPL:    ADDR
Stack:  Drops the 'left side' reference, stacks the address.
Bugs:   Cannot deal with elements of arrays, though they may be easily
         calculated.
```

In the case of arrays ADDR returns the address of the first element which is immediately after the word giving the size of the array.

So "PRINT PEEKW(ADDR(A%))" is exactly the same as "PRINT A%" and "PRINT PEEKW(ADDR(A%())) is the same as "PRINT A%(1)".

17.8.2 RTF_ASC

Returns the ASCII value of the first character of the string.

```
OP:     $8B
OPL:    ASC
Stack:  Drops the string, stacks an integer
Bugs:   If the string is zero length it returns zero.
```

17.8.3 RTF_DAY

Returns the current day of the month - in the range 1 to 31.

```
OP:     $8C
OPL:    DAY
Stack:  Stack an integer
```

17.8.4 RTF_DISP

Displays a string, a record or the last string displayed, using cursor keys for viewing and waiting for any other key to exit.

```
OP:     $8D
OPL:    DISP
Para1:  Integer: 1 - displays para2
                 0 - redisplays the last DISPed string (ignores para2)
                -1 - displays the current record (ignores para2)
Para2:  String to be displayed
Stack:  Drops the two parameters, stacks the exit key as an integer.
Calls:  UT$DISP
Bugs:   In the case para1 is zero it displays the contents of RTT_BUF.
```

        RTT_BUF is used by a number of other operand/operators, for
        instance by string adds.

The display used is the same as that used by FIND in the top level. Each field, delimited by a TAB character, is on a different line. There is no limit to the number of fields.

17.8.5 RTF_ERR

Returns the current error value.

OP:	$8E
OPL:	ERR
Stack:	Stack the error number as an integer

When the language starts running the value of RTB_EROR is zero. If an error is encountered and handled by a TRAP or ONERR the value remains until the next error or a TRAP command.

17.8.6 RTF_FIND

Finds a string in the current file.

OP:	$8F
OPL:	FIND
Stack:	Drops the search string, stacks the record number.
Bugs:	FIND does not do an automatic NEXT, the correct loop structure is:

```
                DO
                 IF FIND "ABC"
                  statement(s)
                 ENDIF
                 NEXT
                UNTIL EOF
```

If no record is found zero is returned and the current record remains the same as before the FIND.

17.8.7 RTF_FREE

Returns the amount of free memory.

OP:	$90
OPL:	FREE
Stack:	Stack the resulting integer.

Calculates the amount of free memory by subtracting ALA_FREE from RTA_SP and then subtracting $100.

17.8.8 RTF_GET

Get a single character.

OP:	$91
OPL:	GET
Stack:	Stack the character as an integer.
Calls:	KB$GETK
Bugs:	The ON/CLEAR key returns 1. It can be difficult to break out of a tight loop with a GET using the ON/CLEAR, Q keys. With perseverance it is normally possible.

If there is a key in the buffer it gets that key first. If no key is received the Organiser will turn itself off after the timeout. See section 7.4.1 and section 7.4.4.

17.8.9 RTF_HOUR

Returns the current hour of the day - in the range 0 to 23.

OP:	$92
OPL:	HOUR
Stack:	Stack the number as an integer.

17.8.10 RTF_IABS

Does an ABS on an integer.

OP:	$93
OPL:	IABS
Stack:	Leaves the integer on the stack.

Converts a negative integer to a positive integer. If ABS is used in place of IABS the result would be the same but the function would require two unnecessary type conversions. IABS is significantly faster than ABS.

17.8.11 RTF_INT

Converts a floating point number to an integer.

OP:	$94
OPL:	INT
Stack:	Drops float, stacks integer
Errors:	ER_RT_IO - integer overflow
Calls:	IM$DINT, IM$FLOI
Bugs:	Always rounds down, INT(3.9) is 3 and INT(-3.9) is -4.

Identical to QCO_NUM_TO_INT.

17.8.12 RTF_KEY

Returns any key in the input buffer. Zero if no key is waiting.

OP:	$95
OPL:	KEY
Stack:	Stack the integer
Bugs:	Except after an "ESCAPE OFF" statement, KEY cannot pick up the ON/CLEAR key.

17.8.13 RTF_LEN

Returns the length of the string.

OP:	$96
OPL:	LEN
Stack:	Drops string, stacks the length as an integer

17.8.14 RTF_LOC

Locates one string in another, returns zero if not found.

OP:	$97
OPL:	LOC
Para1:	String to be searched
Para2:	String to locate
Stack:	Drops the two strings, stacks the resulting position as an integer

17.8.15 RTF_MENU

Gives a menu of options.

```
OP:     $98
OPL:    MENU
Stack:  Drops the string, stacks the exit item as an integer
Calls:  MN_AXDP
Errors: ER_RT_MU - menu error
        ER_FN_BA - bad argument
Bugs:   In the input string the menu items are delimited by commas.
        Before MN_AXDP is called the string is converted to individual
        strings each terminated by a null word.  It is possible to
        have too many items.
        Don't have spaces or tabs as part of menu items, they can have
        unpredictable effects.
```

The normal input is a string with each menu item delimited by a comma. An item is selected either by a unique first letter or by positioning on that item and pressing the EXE key. If the menu exits by the ON/CLEAR key it returns zero.

17.8.16 RTF_MINUTE

Returns the current minute of the hour - in the range 0 to 59.

```
OP:     $99
OPL:    MINUTE
Stack:  Stack the number as an integer.
```

17.8.17 RTF_MONTH

Returns the current month of the year - in the range 0 to 11.

```
OP:     $9A
OPL:    MONTH
Stack:  Stack the number as an integer.
```

17.8.18 RTF_PEEKB

Peeks a byte at the given address.

```
OP:     $9B
OPL:    PEEKB
Stack:  Drops the address, stacks the result as an integer
```

If the address is in the ranges $00-$3F and $282-$400 then it returns zero. These ranges are the processor registers and the custom chip's control addresses. See section 9.3.2 for more details. The informed user may access these addresses via machine code.

17.8.19 RTF_PEEKW

Peeks a word at the given address.

```
OP:     $9C
OPL:    PEEKW
Stack:  Drops the address, stacks the result as an integer
```

See the comments after RTF_PEEKB.

17.8.20 RTF_RECSIZE

Returns the size of the current record.

```
OP:     $9D
```

```
OPL:    RECSIZE
Stack:  Stack the size as an integer.
Bugs:   The maximum size of a record is 254, this includes the field
         separators.
```

See 17.2.10 for more details.

17.8.21 RTF_SECOND

Returns the current second of the minute - in the range 0 to 59.

```
OP:     $9E
OPL:    SECOND
Stack:  Stack the number as an integer.
```

17.8.22 RTF_IUSR

Calls machine code.

```
OP:     $9F
OPL:    USR
Para1:  Address of the machine code
Para2:  The value to be passed in the D register
Stack:  Drops the parameters, stacks the X register on return
```

17.8.23 RTF_SADDR

Returns the address of a string.

```
OP:     $C9
OPL:    ADDR
Stack:  Stack the result
```

Returns the address of the length byte, the byte after the maximum length.

In the case of an array it returns the address of the length byte of the first element of the array. So "ADDR(A$())-2" is the address of the size the array (a word) and "ADDR(A$())-3" is the address of the maximum string length (a byte).

17.8.24 RTF_VIEW

View a string, or the last string viewed.

```
OP:     $A0
OPL:    VIEW
Para1:  Line on which to view (1 or 2)
Para2:  String to be viewed
Stack:  Drops the parameters, stacks the exit character as an integer
```

If the string is null it re-displays the last string VIEWed (which is held in RTT_BUF).

17.8.25 RTF_YEAR

Returns the current year - in the range 0 to 99.

```
OP:     $A1
OPL:    YEAR
Stack:  Stack the number as an integer
```

17.8.26 RTF_COUNT

Returns the number of records in the current file.

```
OP:     $A2
OPL:    COUNT
Stack:  Stack the result as an integer
Calls:  FL$SIZE
```

17.8.27 RTF_EOF

Returns TRUE if the position in the file is at the end of file. If the current record is the last record of the file, EOF returns FALSE.

```
OP:     $A3
OPL:    EOF
Stack:  Stack result as an integer
Errors: ER_RT_FC - file not open
Bugs:   If there are no records this returns true.
```

Returns TRUE if the current record buffer is zero. When OPL appends a record with zero length it adds a TAB ($09) character so that it never actually saves a null string.

17.8.28 RTF_EXIST

Returns TRUE is the file exists.

```
OP:     $A4
OPL:    EXIST
Stack:  Drops string, stacks result
Calls:  FL$OPEN
```

17.8.29 RTF_POS

Returns the current record number in the current file.

```
OP:     $A5
OPL:    POS
Stack:  Stack the result
Calls:  FL$SETP, FL$RECT, FL$RSET
Errors: ER_RT_FC - file not open
Bugs:   If no records still return 1.
```

17.9 FLOATING POINT FUNCTIONS

These functions return a floating point value.

17.9.1 RTF_ABS

Does an ABS on a floating point number.

```
OP:     $A6
OPL:    ABS
Stack:  Leaves the floating point number on the stack.
Calls:  FN_ABS
```

17.9.2 RTF_ATAN

Returns the arctangent of the input in radians.

```
OP:     $A7
OPL:    ATAN
Stack:  Drops the input floating point number, stacks the result
Calls:  FN_ATAN
Bugs:   Returns values in the range plus or minus pi/2
```

17.9.3 RTF_COS

Returns the cosine of the input, the input being in radians.

```
OP:     $A8
OPL:    COS
Stack:  Drops the input floating point number, stacks the result
Calls:  FN_COS
Errors: ER_FN_BA - bad argument if the absolute value is greater than
                   3141590.
```

17.9.4 RTF_DEG

Converts the input from radians to degrees.

```
OP:     $A9
OPL:    DEG
Stack:  Drops the input floating point number, stacks the result
Calls:  FN_DEG
Bugs:   All this does is multiply the input by 57.29...
```

17.9.5 RTF_EXP

Returns the value of e raise to the specified power.

```
OP:     $AA
OPL:    EXP
Stack:  Drops the input floating point number, stacks the result
Calls:  FN_EXP
Errors: ER_FN_BA - bad argument if the absolute value is greater than 229.
```

17.9.6 RTF_FLT

Converts an integer to floating point format.

```
OP:     $AB
OPL:    FLT
Stack:  Drops the input integer, stacks the result
Calls:  MT$BTOF
Bugs:   Integers are always taken as signed. To make unsigned:

                A=I% :IF I%<0 :A=A+65536 :ENDIF
```

Exactly the same effect as QCO_INT_TO_NUM.

17.9.7 RTF_INTF

Rounds a floating point number down to a whole number.

```
OP:     $AC
OPL:    INTF
Stack:  Drops the input floating point number, stacks the result
Calls:  IM$DINT, IM$FLOI
```

Essential to use INTF rather than INT if the number is out of the integer range.

17.9.8 RTF_LN

Returns the natural logarithm of the input.

```
OP:     $AD
OPL:    LN
Stack:  Drops the input floating point number, stacks the result
Errors: ER_FN_BA - bad argument
Calls:  FN_LN
Bugs:   The input must be greater than 0.
```

17.9.9 RTF_LOG

Returns the base 10 logarithm of the input.

```
OP:     $AE
OPL:    LOG
Stack:  Drops the input floating point number, stacks the result
Errors: ER_FN_BA - bad argument
Calls:  FN_LOG
Bugs:   The input must be greater than 0.
```

17.9.10 RTF_PI

Returns the number pi = 3.14159265359.

```
OP:     $AF
OPL:    PI
Stack:  Stack the result
Calls:  FN_PI
```

17.9.11 RTF_RAD

Converts the input number to radians. The inverse of DEG.

```
OP:     $B0
OPL:    RAD
Stack:  Drops the input floating point number, stacks the result
Calls:  FN_RAD
Bugs:   All this does is divide the input by 57.29...
```

17.9.12 RTF_RND

Returns a pseudo-random number in the range 0(inclusive) to 1(exclusive).

```
OP:     $B1
OPL:    RND
Stack:  Stack the result
Calls:  FN_RND
```

17.9.13 RTF_SIN

Returns the sine of the input, the input being in radians.

```
OP:     $B2
OPL:    SIN
Stack:  Drops the input floating point number, stacks the result
Calls:  FN_SIN
Errors: ER_FN_BA - bad argument if the absolute value is greater than
                        3141590.
```

17.9.14 RTF_SQR

Returns the square root of the input.

```
OP:     $B3
OPL:    SQR
Stack:  Drops the input floating point number, stacks the result
Calls:  FN_SQRT
Errors: ER_FN_BA - bad argument if negative
```

17.9.15 RTF_TAN

Returns the tangent of the input, the input being in radians.

```
OP:     $B4
OPL:    TAN
Stack:  Drops the input floating point number, stacks the result
Calls:  FN_TAN
Bugs:   At the discontinuities in TAN, pi/2, 3*pi/2, etc, the values
        returned are either greater than 1E10 or less than -1E10.
```

17.9.16 RTF_VAL

Returns the input string as a number.

```
OP:     $B5
OPL:    VAL
Stack:  Drops the input string, stacks the result
Errors: ER_MT_FL - conversion to number failed
Calls:  MT_BTOF
Bugs:   This routine insists that the whole string is used in the
        conversion, so VAL("12.34 ") generates an error.  The null string
        also gives an error.
```

17.9.17 RTF_SPACE

Returns the amount of space on the current device.

```
OP:     $B6
OPL:    SPACE
Stack:  Stack the result as floating point number
Calls:  FL$SIZE
Errors: ER_RT_FC - file not open
Bugs:   This may be longer than a word!
```

17.10 STRING FUNCTIONS

17.10.1 RTF_DIR

Returns the name of the first/next file on a device.

```
OP:     $B7
OPL:    DIR$
Stack:  Drops the input string, stack the resulting string
Calls:  FL$CATL
Errors: ER_FN_BA - bad argument
```

If the string is non-null it checks that it is of the form "A:" or "A". It splits out the device name and returns the first file name preceded by the device name. If the string is null it returns the next file name, on the device already specified. When there are no more file it returns a null string.

17.10.2 RTF_CHR

Converts the integer input to a one character string.

```
OP:     $B8
OPL:    CHR$
Stack:  Drops the input integer, stacks the resulting string
Errors: ER_FN_BA - bad argument if out of the range 0-255
```

17.10.3 RTF_DATIM

Returns the date-time string in the form:

```
        "TUE 04 NOV 1986 10:44:29"
```

```
OP:     $B9
OPL:    DATIM$

Stack:  Stacks the resulting string
```

17.10.4 RTF_SERR

Returns the error string associated with the integer error number.

```
OP:     $BA
OPL:    ERR$
Stack:  Drops the input integer, stacks the resulting string
Errors: ER_FN_BA - bad argument (if not a byte)
Calls:  ER$LKUP
Bugs:   Returns "*** ERROR ***" if less than the lowest recognised
        error number.
```

17.10.5 RTF_FIX

Returns the floating point number as a string with a fixed number of decimal places.

```
OP:     $BB
OPL:    FIX$
Para1:  The floating point number
Para2:  The require number of decimal places
Para3:  The field size
Stack:  Drops input parameters, stacks the resulting string
Calls:  MT_FBDC
Bugs:   If the number does not fit, '*'s are inserted
```

17.10.6 RTF_GEN

Returns the floating point number as a string. This is the same format as used by QCO_PRINT_NUM.

```
OP:     $BC
OPL:    GEN$
Stack:  Drops the floating point number
Calls:  MT_FBGN
Bugs:   If the number does not fit, '*'s are inserted
```

The format in which the number is displayed is integer, decimal or scientific in that order of precedence.

17.10.7 RTF_SGET

Get a character and return it as a one character string.

```
OP:     $BD
OPL:    GET$
Stack:  Stack the resulting string
Calls:  KB$GETK
Bugs:   The ON/CLEAR key returns a valid string.  It can be difficult to
        break out of a tight loop with a GET$ using ON/CLEAR, Q keys.
        With perseverance it is normally possible.
```

17.10.8 RTF_HEX

Converts the integer into a hexadecimal string.

```
OP:     $BE
OPL:    HEX$
Stack:  Drops input integer, stacks resulting string
Calls:  UT_XTOB
Bugs:   Input must be in the integer range.
```

17.10.9 RTF_SKEY

Returns any keys in the input buffer as a string. Returns the null string if no key is waiting.

```
OP:     $BF
OPL:    KEY$
Stack:  Stack the string
Calls:  KB$TEST, KB$GETK
Bugs:   Except after an "ESCAPE OFF" statement, KEY cannot pick up the
        ON/CLEAR key.  ON/CLEAR key normally suspends OPL execution.
```

17.10.10 RTF_LEFT

Returns the first n characters of the string.

```
OP:     $C0
OPL:    LEFT$
Para1:  The string
Para2:  Number of characters to keep
Stack:  Drops the input parameters, stacks the resulting string
Bugs:   If the string is shorter than the number of characters the entire
        string is returned.
```

17.10.11 RTF_LOWER

Converts the string to lower case.

```
OP:     $C1
OPL:    LOWER$
Stack:  Drops the input string, stacks the result
```

17.10.12 RTF_MID

Returns the middle of a string.

```
OP:     $C2
OPL:    MID$
Para1:  The string
Para2:  The start character
Para3:  The number of characters to be kept
Stack:  Drops the input parameters, stacks the resulting string
Bugs:   If there are insufficient characters the rest of the string is
        returned.
```

You can get all the characters after the nth by the statement:

```
MID$(a$,n,255)
```

17.10.13 RTF_NUM

Converts a number to an integer string.

OP:	$C3
OPL:	NUM$
Para1:	The floating point number
Para2:	The maximum size of the string
Stack:	Drops the input parameters, stacks the resulting string.
Calls:	MT_FBIN
Bugs:	If the number does not fit, '*'s are inserted
	The number does not have to be in usual integer range.

17.10.14 RTF_RIGHT

Returns the last n characters of a string.

OP:	$C4
OPL:	RIGHT$
Para1:	The string
Para2:	The number of characters wanted
Stack:	Drops the input parameters, stacks the resulting string
Bugs:	If the string is shorter than the number of characters the entire string is returned.

17.10.15 RTF_REPT

Repeats the string n times.

OP:	$C5
OPL:	REPT$
Para1:	The string
Para2:	The repeat count
Stack:	Drops the integer and input string, stacks the result
Bugs:	If the repeat count is zero no error is given.
Errors:	ER_MT_FN - function argument error
	ER_LX_ST - string too long

17.10.16 RTF_SCI

Returns the floating point number as a string in scientific form.

OP:	$C6
OPL:	SCI$
Para1:	The floating point number
Para2:	Number of decimal places required
Para3:	Field width
Stack:	Drops the floating point number, stacks the result
Calls:	MT_FBEX
Bugs:	If the number does not fit, '*'s are inserted

17.10.17 RTF_UPPER

Converts the string to upper case.

OP:	$C7
OPL:	UPPER$
Stack:	Drops the input string, stacks the result

17.10.18 RTF_SUSR

Calls machine code.

```
OP:     $C8
OPL:    USR$
Para1:  Address of the machine code
Para2:  The value to be passed in the D register
Stack:  Drops the parameters, stacks the string pointed at by the
         X register
```

17.11 FILES

17.11.1 CREATING

Before a file is created a check is made that no file exists with the specified name on that device. The first unused record number over $90 is assigned to the file and the file name record is written to the device. The process then continues in the same way as opening a file.

The file name records are type $81. The file name record for a file called "AMANDA", with record file type $95 looks like:

```
                09 81 41 4D 41 4E 44 41 20 20 20 95
```

17.11.2 OPENING

First the file name record is located to ensure that the file exists. The file record type and the device on which the file was found are saved in the file block (RTT_FILE). The field names are saved in the allocator field name cell corresponding to the logical name and the file buffer cell is expanded to 256 bytes. The record position is initialised to 1 and the first record, if it exists, is read.

If the file has just been created or the record is empty the current record will be null and the EOF flag is set.

See section 6.5.1.4.1 for the format of the file blocks.

17.11.3 LOGICAL FILE NAMES

Up to 4 files may be open at one time; to distinguish between then logical file names are used. The 4 logical file names: A,B,C, and D, are used to determine which file is to be operated on by the file commands.

This means that you can open files in any order but have a constant way of referring to them. The USE operator selects which file is affected by the following commands:

```
APPEND          BACK            CLOSE           ERASE
FIRST           NEXT            LAST            POSITION
UPDATE
```

and the following functions:

```
COUNT           DISP            EOF             FIND
POS             RECSIZE         SPACE
```

17.11.4 USING FILES

There is no functional difference between the logical file names.

When opening a file the file name record and the first record are located; two cells, one a buffer and one for the field names are grown. Closing a file entails the two cells being shrunk.

All references to fields must include the logical file name. This serves two purposes; it allows statements such as "A.MAX=B.VALUE" and it allows the language to distinguish between ordinary variables and field names.

17.12 PROCEDURE CALLS

To write compact, fast code it is important to understand the way procedures are loaded and automatically overlaid.

A procedure call consists of a procedure name followed by up to 16 parameters. The procedure name may include an optional '$' or '%' but must terminate with a ':'. If parameters are supplied they must be separated by commas and be enclosed in brackets.

There are two main types of procedure. In standard OPL procedures the Q code is loaded onto the stack and then executed. The second type are known as a device procedure or language extensions; they are identical to standard procedures in appearance, but differs in that it is recognised by the device lookup and runs as self-contained machine code.

17.12.1 STANDARD PROCEDURES

When a QCO_PROC operator is encountered the parameters will already be on the stack, along with the parameter count and the parameter types. After the operator is the name of the procedure.

The following list of actions are then carried out:

1. Check if it is a language extension/device call
2. Search for the procedure starting with the default device
3. Check that there is sufficient memory
4. Set new RTA_SP, RTA_FP
5. Check the parameter count
6. Check the parameter types
7. Set up a table of variables declared GLOBAL
8. Set up the parameter table
9. Resolve the externals, build an externals table
10. Zero all variable space
11. Fix-up strings
12. Fix-up arrays
13. Load the code
14. Set new RTA_PC

The code is loaded every time a procedure is called. This means that recursive procedures are allowed but that the stack will grow by the size of the Q code + data space + overhead for each call. On an XP, following a Reset, the procedure:

```
RECURS:(I%)
IF I%
 RECURS:(I%-1)
ENDIF
```

allows values up to 315 before an 'OUT OF MEMORY' error is given.

See EXAMPLE 3.

17.12.2 LANGUAGE EXTENSIONS

Language extension are also referred to as device procedures. Examples are LINPUT, LSET and LTRIG in the RS232 interface.

To test if a procedure is a language extension, call DV$LKUP. This looks through the devices loaded in order of priority. If a language extension is found it returns with carry clear, the device number in the A register and the vector number in the B register, suitable for an immediate call to DV$VECT to run the code.

The machine code should check that any parameters that have been passed are correct, do whatever it has to do, add the return variable to the stack and return. It is essential to return the right variable type. If the extension name terminates with a '$' it must return a string, if with a '%' it requires an integer, otherwise an 8 byte floating point number.

Note that a variable number of parameters can be passed to a device.

As a simple example, consider a language extension to add two integers without giving an error if the sum overflows. If only one parameter is given the value is simply incremented, again without giving an error. The assembler for this extension called "ADD%" is:

```
XADD:
        LDX     RTA_SP:
        LDA     A,0,X
        BEQ     1$                      ; wrong number of parameters
        DEC     A
        BEQ     INCREM                  ; increment 1 parameter
        DEC     A
        BEQ     XXADD                   ; add the two
1$:
        LDA     B,#ER_RT_NP             ; wrong number of parameters
        SEC                             ; bad return
        RTS
INCREM:
        LDA     A,1,X                   ; load parameter type
        BNE     WRGTYP                  ; branch if not integer
        LDD     2,X
        ADDD    #1
EXIT:
        DEX
        DEX
        STX     RTA_SP:
        STD     0,X                     ; save return value
        CLC                             ; good return
        RTS
XXADD:
        LDA     A,1,X
        BNE     WRGTYP                  ; branch if not integer
        LDA     A,4,X
        BNE     WRGTYP                  ; branch if not integer
        LDD     2,X                     ; and add the two integers
        ADDD    5,X
        BRA     EXIT
WRGTYP:
        lda     b,#ER_FN_BA             ; report wrong parameters type
        SEC                             ; bad return
        RTS
```

See chapter 11 for the necessary pack header.

17.13 WRITING OPL

Like any programming language there is an infinite number of approaches to every problem. The aim should be to produce fast, compact Q code that runs in a minimum of memory but is also easy to write and understand. These aims inevitably conflict with each other; the correct balance varies from application to application.

For example, the decision to use a separate procedure, rather than writing the code in line, is a matter of considering the difference in Q code size, the extra stack required at run time, the time overhead required to load and return from a procedure and finally style.

It is impossible to give definitive rules on writing code but it is worth taking the following points into account.

17.13.1 COMPACT Q CODE

1. Only use procedures where appropriate
2. If it makes no difference, use LOCALs instead of GLOBALs
3. Use short field names
4. Use short global names
5. If you repeatedly use a CHR$ with the same value, assign it to a variable
6. Use "RETURN" instead of "RETURN 0" or "RETURN """
7. Use hexadecimal integers instead of negative integers

17.13.2 COMPACT ON RUN TIME

1. Write short Q code (as above)
2. Use a small main procedure to call several small procedures.
3. Use integers instead of floating point numbers
4. Use short field names
5. Use short global names
6. Check the deepest part of the code by adding, temporarily, PRINT FREE :GET. Then consider restructuring the procedures to decrease the amount of stack used.

17.13.3 FAST CODE

Each operand/operator has an overhead of .05 ms. Most integer based operands/operators are very fast and run in less than .1 ms.

The following timings are rough and should only be used as a guide:

OPERAND	Time (ms)
RND	10
AT	.15
PRINT a string	.5
INT_TO_NUM	2.5
NUM_TO_INT	2
SIN/COS	150
TAN	350
ATAN	170
SQR	240
EXP	130
LOG/LN	200
Integer add/subtract	.1
Integer multiply/divide	1
Floating point add/subtract	3
Floating point multiply	10
Floating point divide	20
Accessing a field	5

PRINT_CR has a default delay of 500 milliseconds. This value can be altered by poking the value in DPW_DELY.

1. don't use too many procedures, regard them as being similar to overlays
2. place the procedures at the beginning of the pack, with the most frequently used at the start
3. Use LOCALs or GLOBALs rather than field variables
4. Don't use procedures inside time critical loops, write the code in-line
5. Use integers instead of floating point numbers
6. Write short Q code (less code to load)
7. Use LOCALs instead of GLOBALs

17.13.3.1 PROCEDURES

The smallest time overhead on loading, and returning form a procedure is 8 ms. This overhead increases if the procedure follows other blocks or records on the device. It also increases if the procedure is not on the same device as the top level procedure (as it will have to search that device first). See chapter 12 for a full explanation of the storage mechanism.

17.13.3.2 FILES

Some of the file operators have to count up the pack each time they are used. For the sake of speed NEXT remembers its position on each of the packs. However it only remembers one position on each pack so:

```
                USE B
                NEXT
                A.MAX=B.VAL
                USE A
                APPEND
```

where file A is on B: and file B on C: is significantly faster than if they are both on the same device.

BACK however always has to count up the pack to locate a record and this can take a noticeable time. Remember that erased records, as well as readable ones, will slow down the location of a record.

17.13.4 CODE STYLE

Before starting to write a program (which normally will consist of a number of procedures) first decide the relative importance of speed of execution, compactness of the Q code and the amount of stack used.

Then rough out the procedure structure. For example, in the case of the finance pack the main procedure is called FINS:

```
fins:
local i%,j%
do
  i%=menu("BANK,EXPENSES,NPV,IRR,COMPOUND,BOND,MORTGAGE,APR,END")
  if     i%=1 : bank:
  elseif i%=2 : expenses:
  elseif i%=3 : npv:
  elseif i%=4 : irr:
  elseif i%=5
    do
      j%=menu("VALUE,FUTURE,PAYMENT,DURATION,INTEREST,END")
      if     j%=1 : value:
      elseif j%=2 : future:
      elseif j%=3 : payment:
      elseif j%=4 : duration:
      elseif j%=5 : interest:
      endif
    until j%=0 or j%=6
  elseif i%=6 : bond:
  elseif i%=7 : mortgage:
  elseif i%=8 : apr:
  endif
until i%=0 or i%=9
```

Your style may vary if you are writing on the emulator or the ORGANISER itself. On the ORGANISER it is worth, as a general rule, making only limited use of the ':' option to have more than one statement on a line. On the emulator you may prefer to write multiple statements on a line. The procedure above was written using a full screen editor which is reflected in the elegant use of non-functional spaces.

It is very helpful to indent the code by logical function. This is very useful in matching IF/ENDIF and loop commands.

Comment the code. The logic may seem very obvious when you write it but other people may want to read it, or you may return to the code after several months. In most cases the extra space taken by the comments is well worth it. Remember that comments make no difference to the Q code size.

Use brackets if you are unsure of the operator precedence. This adds nothing to the Q code size but makes your intentions absolutely clear.

When using the ':' separator it is not necessary to precede it by a space when the preceding characters cannot be taken as a variable name. So "A%=1:B%=2" is valid but "A%=B%:B%=C%" gives a syntax error. It can, however, save time and make the code more readable if you always proceed the ':' separator with a space.

17.14 TRANSLATOR

The translator scans the source code, statement by statement, translating it into Q code. All expressions are converted to reverse polish (postfix) form so that, at run time, the operators can be executed as soon as they are encountered.

It is beyond the scope of this document to describe the detailed working of the translator. Fortunately, such a description is not necessary in order to understand either the execution of the code or the writing of efficient code.

17.15 SYSTEM SERVICES INTERFACE

17.15.1 RM$RUNP

VECTOR NUMBER:	100
INPUT PARAMETERS:	X register - points at the name of the procedure
	B register - if set then runs the calculator
OUTPUT VALUES:	None

DESCRIPTION

Runs the language by loading and running the OPL procedure. The procedure can not have any parameters.

EXAMPLE

For example, to run a procedure called BOOT:

```
                LDX     pname           ; address of the name of the
procedure
                LDA     B,#BLANTYP
                OS      FL$BOPN         ; test that procedure exists
                BCS     2$
                LDX     pname           ; address of the name of the
procedure
                CLR     B
                OS      RM$RUNP
                BCC     1$
        2$:
                OS      ER$MESS         ; report error
        1$:
                RTS
```

ERRORS: Any error is possible

BUGS

If the procedure does not exist the error message will contain a garbage name.

Every time RM$RUNP is run the language re-initialises, it resets RTA_SP to BTA_SBAS, zeroes all the file cells and close all the files.

17.15.2 LN$STRT

```
VECTOR NUMBER:          079
INPUT PARAMETERS:       B register -
                                0 translating language procedures
                                1 translating CALC expressions
                                2 locating errors in CALC
                                3 locating errors in language procedures

                        X register - offset in Q code to run time error
                                ignored if B register 0 or 1.

OUTPUT VALUES:          translated result, if successful, in OCODCELL
                        If an error is detected:
                                X register - offset to error in TEXTCELL
                                B register - error number
```

DESCRIPTION

Runs the translator.

EXAMPLE

```
                CLR     B               ; translate language procedure
                OS      LN$STRT
                BCC     1$
                OS      ER$MESS         ; report error
        1$:
                RTS
```

ERRORS: Many

BUGS

If the B register is 2 or 3 and the value of X is greater than the length of the Q code, in other words you are asking for an error past the end of the code, the effect is unpredictable.

17.16 MACHINE CODE INTERFACE

From the information in this chapter, the programmer knows exactly where everything is on the stack.

When variables are declared they are used in order, so:

```
                LOCAL A%,B%
                PRINT ADDR(A%)=ADDR(B%)+2
                GET
```

will print -1, i.e. TRUE.

See section 6.5.2.2 for details of where machine code can be permanently hidden.

For short machine code routines you can use this crude, but effective, procedure:

```
                LOADR:(ADDR%,CODE$)
                LOCAL A%,B1%,B2%,I%
                A%=ADDR%
                I%=1
                WHILE I%<LEN(CODE$)
                 B1%=ASC(MID$(CODE$,I%,1))-%0
                 IF B1%>9 :B1%=B1%-7 :ENDIF
                 B2%=ASC(MID$(CODE$,I%+1,1))-%0
                 IF B2%>9 :B2%=B2%-7 :ENDIF
                 POKEB A%,B1%*16+B2%
                 A%=A%+1
                 I%=I%+2
                ENDWH
```

When calling this procedure you must pass the machine code in digital form and the address where to put the machine code. It is essential that the programmer ensures there is enough room for the machine code at the address given.

A calling sequence might look like:

```
                MAIN:
                GLOBAL MC%,MC$(10)
                MINIT: :REM Initialise the machine code
                ..
                CELL%=USR(MC%,100) :REM GRABs a cell of size 100
                IF CELL%=0
                 PRINT "No cell free"
                 GET :RAISE 0
                ENDIF
                ..
                RETURN


                MINIT:
                A$="3F012403CE000039"
                IF LEN(A$)/2>LEN(MC$)
                  PRINT "Not enough room for MC"
                  GET :RAISE 0
                ENDIF
                MC%=ADDR(MC$)
                LOADR:(MC%,A$)
```

The machine code is:

```
                OS      AL$GRAB
                BCC     1$
                LDX     #0
        1$:
                RTS
```

17.17 EXCEPTION HANDLING

17.17.1 ERROR HANDLING

When an error is first detected the following actions are taken:

1. The error saved in RTB_EROR
2. If the TRAP flag is set then the language continues
3. The ON_ERR address for that procedure and each procedure above is tested. If one is found to be non-zero, RTA_PC is set to that value and RTA_SP set to the BASE_SP for that procedure. The language then continues on.
4. If no error handling is detected then the error is reported along with the name of the procedure in which the error was detected and the language exits.

If the error is ER_RT_UE (undefined external) then the externals which are undefined are displayed with DP$VIEW.

If the error is ER_RT_PN (procedure not found) then the name of the procedure not found is displayed (as well as the procedure where it was called).

17.17.2 OUT OF MEMORY

Every time round the top loop the difference between RTA_SP and ALA_FREE is calculated. If this difference is less than 256 bytes, "OUT OF MEMORY" is reported. Note that no operand or operator can grow the stack by more than 256 bytes.

The filing system can also generate the "PACK FULL" error if it detects that after an operation fewer than 256 bytes will be free on device A. In this case it means essentially the same thing as "OUT OF MEMORY".

The only time when OPL uses memory, other than on the stack, is when it opens files. See section 17.11.2.

17.17.3 LOW BATTERY

If the voltage goes below the threshold value (5.2 volts) while the language is running, it is detected either in the top loop or during the execution of an operator. In either case it is treated as a standard error. If no error handling is in force, the error is reported and the machine turns off.

If the error is handled by an ONERR, the low battery error number is saved in RTB_EROR. It is not reported again by the top level until the battery voltage has gone back above the minimum voltage. This allows the procedure to take some action (e.g. to turn the organiser off). If the procedure just continues on the battery will eventually die completely and there is a risk of having to cold boot the machine.

Note that the battery is more likely to drop below the threshold voltage when devices, such as the packs or the RS232 interface, are switched on because they drain substantially more current than the Organiser by itself. See section 3.2 for more details of the power drain of different devices. Also note that a battery naturally recovers some of its power after being turned off for a while.

17.17.4 ON/CLEAR KEY

In normal operation pressing the ON/CLEAR key results in the execution of the language being frozen until another key is pressed. If the key pressed is 'Q','q' or '6' it creates an error condition ER_RT_BK. If there is no user error handling, execution of the language will terminate.

If ESCAPE OFF has been executed then the ON/CLEAR key has no special effect.

In an input statement then the ON/CLEAR key acts in one of 3 different ways:

1. If there is any input it is cleared
2. Or if the TRAP option has been used then the input exits with the error condition ER_RT_BK
3. Otherwise it is ignored

See QCO_INPUT_INT, QCO_INPUT_NUM, QCO_INPUT_STR, RTF_GET, RTF_SGET, RTF_KEY and RTF_SKEY.

17.17.5 WARNING

OPL is a powerful flexible language and as such it has the potential to crash the operating system or get into an infinite loop. This is particularly unfortunate in the case of the ORGANISER because all the data held in device A: is lost when the machine re-boots. For extensive development of 'dangerous' routines a RAMPACK has a lot to recommend it.

There are trivial ways to crash such as poking system variables or using USR function with wrong addresses or bad machine code. It is impossible to describe all the other ways in which such problems can arise. The examples listed below show the most obvious ways in the simplest possible form.

```
     1.  ESCAPE OFF :DO :UNTIL 0        :REM Impossible to get out

     2.  WHILE GET :ENDWH               :REM Hard to get out of

     3.  DO :KEY :UNTIL 0               :REM Hard to get out of

     4.  A::  ONERR A::  :RAISE 0       :REM Impossible to get out

     5.  A::  ONERR A::  :DO :UNTIL 0   :REM Impossible to get out
```

Error handling is best added at the end of a development cycle. Turning

ESCAPE OFF substantially increases the chances of getting into an infinite loop from which there is no exit.

17.18 INDEX OF OPERANDS

00	QI_INT_SIM_FP	0D	QI_LS_INT_SIM_FP	1A	QI_INT_FLD
01	QI_NUM_SIM_FP	0E	QI_LS_NUM_SIM_FP	1B	QI_NUM_FLD
02	QI_STR_SIM_FP	0F	QI_LS_STR_SIM_FP	1C	QI_STR_FLD
03	QI_INT_ARR_FP	10	QI_LS_INT_ARR_FP	1D	QI_LS_INT_FLD
04	QI_NUM_ARR_FP	11	QI_LS_NUM_ARR_FP	1E	QI_LS_NUM_FLD
05	QI_STR_ARR_FP	12	QI_LS_STR_ARR_FP	1F	QI_LS_STR_FLD
06	QI_NUM_SIM_ABS	13	QI_LS_NUM_SIM_ABS	20	QI_STK_LIT_BYTE
07	QI_INT_SIM_IND	14	QI_LS_INT_SIM_IND	21	QI_STK_LIT_WORD
08	QI_NUM_SIM_IND	15	QI_LS_NUM_SIM_IND	22	QI_INT_CON
09	QI_STR_SIM_IND	16	QI_LS_STR_SIM_IND	23	QI_NUM_CON
0A	QI_INT_SIM_IND	17	QI_LS_INT_SIM_IND	24	QI_STR_CON
0B	QI_NUM_SIM_IND	18	QI_LS_NUM_SIM_IND		
0C	QI_STR_SIM_IND	19	QI_LS_STR_SIM_IND		

17.19 INDEX OF OPERATORS

25	QCO_SPECIAL	47	QCO_GT_STR	69	QCO_USE
26	QCO_BREAK	48	QCO_GTE_STR	6A	QCO_KSTAT
27	QCO_LT_INT	49	QCO_NE_STR	6B	QCO_EDIT
28	QCO_LTE_INT	4A	QCO_EQ_STR	6C	QCO_INPUT_INT
29	QCO_GT_INT	4B	QCO_ADD_STR	6D	QCO_INPUT_NUM
2A	QCO_GTE_INT	4C	QCO_AT	6E	QCO_INPUT_STR
2B	QCO_NE_INT	4D	QCO_BEEP	6F	QCO_PRINT_INT
2C	QCO_EQ_INT	4E	QCO_CLS	70	QCO_PRINT_NUM
2D	QCO_ADD_INT	4F	QCO_CURSOR	71	QCO_PRINT_STR
2E	QCO_SUB_INT	50	QCO_ESCAPE	72	QCO_PRINT_SP
2F	QCO_MUL_INT	51	QCO_GOTO	73	QCO_PRINT_CR
30	QCO_DIV_INT	52	QCO_OFF	74	QCO_LPRINT_INT
31	QCO_POW_INT	53	QCO_ONERR	75	QCO_LPRINT_NUM
32	QCO_UMIN_INT	54	QCO_PAUSE	76	QCO_LPRINT_STR
33	QCO_NOT_INT	55	QCO_POKEB	77	QCO_LPRINT_SP
34	QCO_AND_INT	56	QCO_POKEW	78	QCO_LPRINT_CR
35	QCO_OR_INT	57	QCO_RAISE	79	QCO_RETURN
36	QCO_LT_NUM	58	QCO_RANDOMIZE	7A	QCO_RETURN_NOUGHT
37	QCO_LTE_NUM	59	QCO_STOP	7B	QCO_RETURN_ZERO
38	QCO_GT_NUM	5A	QCO_TRAP	7C	QCO_RETURN_NULL
39	QCO_GTE_NUM	5B	QCO_APPEND	7D	QCO_PROC
3A	QCO_NE_NUM	5C	QCO_CLOSE	7E	QCO_BRA_FALSE
3B	QCO_EQ_NUM	5D	QCO_COPY	7F	QCO_ASS_INT
3C	QCO_ADD_NUM	5E	QCO_CREATE	80	QCO_ASS_NUM
3D	QCO_SUB_NUM	5F	QCO_DELETE	81	QCO_ASS_STR
3E	QCO_MUL_NUM	60	QCO_ERASE	82	QCO_DROP_BYTE
3F	QCO_DIV_NUM	61	QCO_FIRST	83	QCO_DROP_WORD
40	QCO_POW_NUM	62	QCO_LAST	84	QCO_DROP_NUM
41	QCO_UMIN_NUM	63	QCO_NEXT	85	QCO_DROP_STR
42	QCO_NOT_NUM	64	QCO_BACK	86	QCO_INT_TO_NUM
43	QCO_AND_NUM	65	QCO_OPEN	87	QCO_NUM_TO_INT
44	QCO_OR_NUM	66	QCO_POSITION	88	QCO_END_FIELDS
45	QCO_LT_STR	67	QCO_RENAME	89	QCO_RUN_ASSEM
46	QCO_LTE_STR	68	QCO_UPDATE		

17.20 INDEX OF FUNCTIONS

8A	RTF_ADDR	A0	RTF_VIEW	B6	RTF_SPACE
8B	RTF_ASC	A1	RTF_YEAR	B7	RTF_DIR
8C	RTF_DAY	A2	RTF_COUNT	B8	RTF_CHR
8D	RTF_DISP	A3	RTF_EOF	B9	RTF_DATIM
8E	RTF_ERR	A4	RTF_EXIST	BA	RTF_SERR
8F	RTF_FIND	A5	RTF_POS	BB	RTF_FIX
90	RTF_FREE	A6	RTF_ABS	BC	RTF_GEN
91	RTF_GET	A7	RTF_ATAN	BD	RTF_SGET
92	RTF_HOUR	A8	RTF_COS	BE	RTF_HEX
93	RTF_IABS	A9	RTF_DEG	BF	RTF_SKEY

94	RTF_INT	AA	RTF_EXP	C0	RTF_LEFT
95	RTF_KEY	AB	RTF_FLT	C1	RTF_LOWER
96	RTF_LEN	AC	RTF_INTF	C2	RTF_MID
97	RTF_LOC	AD	RTF_LN	C3	RTF_NUM
98	RTF_MENU	AE	RTF_LOG	C4	RTF_RIGHT
99	RTF_MINUTE	AF	RTF_PI	C5	RTF_REPT
9A	RTF_MONTH	B0	RTF_RAD	C6	RTF_SCI
9B	RTF_PEEKB	B1	RTF_RND	C7	RTF_UPPER
9C	RTF_PEEKW	B2	RTF_SIN	C8	RTF_SUSR
9D	RTF_RECSIZE	B3	RTF_SQR	C9	RTF_SADDR
9E	RTF_SECOND	B4	RTF_TAN		
9F	RTF_IUSR	B5	RTF_VAL		

17.21 EXAMPLES

In these examples all values are given in hexadecimal; word values are given as 4 digits, bytes as 2 digits each one separated by a space. If values are undefined they are written as **.

17.21.1 EXAMPLE 1

Source code:

```
                EX1:
                LOCAL A$(5)
                A$="ABC"
```

The Q code header is:

```
High memory     0009            size of the variables on stack
                000A            length of Q code
                00              number of parameters
                                  type of parameter
                0000            size of global area
                                  global name
                                  global type
                                  offset
                0000            size of externals
                                  external name
                                  external type
                0003            bytes of string fix-ups
                FFF7              string fix-up offset (from FP)
                05                max length of string
Low memory      0000            bytes of array fix-ups
                                  array fix-up offset (from FP)
                                  size of array
```

The Q code is:

```
                0F FFF8         QI_LS_STR_SIM_FP
                24              QI_STR_CON
                03 41 42 43     "ABC"
                81              QCO_ASS_STR
                7B              QCO_RETURN_ZERO
```

If this program is run on a CM the stack looks like:

```
        Initially           Left Side   Constant    Assign      On Return
3EFF    '1'                 '1'         '1'         '1'         '1'
3EFE    'X'                 'X'         'X'         'X'         'X'
3EFD    'E'                 'E'         'E'         'E'         'E'
3EFC    ':'                 ':'         ':'         ':'         ':'
3EFB    'A'                 'A'         'A'         'A'         'A'
3EFA    05                  05          05          05          05
3EF9    00   - Top proc     00          00          00          00
3EF8    00   - No. paras    00          00          00          00
3EF6    3EF9 - Return PC    3EF9        3EF9        3EF9        3EF9
3EF4    0000 - ONERR        0000        0000        0000        0000
3EF2    3EDB - BASE SP      3EDB        3EDB        3EDB        3EDB
3EF0    0000 - FP           0000        0000        0000        0000
3EEE    3EEE - Global table 3EEE        3EEE        3EEE        3EEE
3EED    00                  00          00          00          00
3EEC    00                  00          00          00          00
3EEB    00                  00          'C'         'C'         'C'
3EEA    00                  00          'B'         'B'         'B'
3EE9    00                  00          'A'         'A'         'A'
```

```
3EE8    00                  00          03          03          03
3EE7    05                  05          05          05          05
3EE6    **                  **          **          **          **
3EE5    **                  **          **          **          **
3EE4    QCO_RETURN_ZERO    7B           7B          7B          7B
3EE3    QCO_ASS_STR         81          81          81          81
3EE2    'C'                 'C'         'C'         'C'         'C'
3EE1    'B'                 'B'         'B'         'B'         'B'
3EE0    'A'                 'A'         'A'         'A'         'A'
3EDF    03                  03          03          03          03
3EDE    QI_STR_CON          24          24          24          24
3EDC    FFF8                FFF8        FFF8        FFF8        FFF8
3EDB    QI_LS_STR_SIM_FP    0F          0F          0F          0F
3EDA    **                  3EE8        3EE8        **          00
3ED9    **                  05          05          **          00
3ED8    **                  00          00          **          00
3ED7    **                  **          'C'         **          00
3ED6    **                  **          'B'         **          00
3ED5    **                  **          'A'         **          00
3ED4    **                  **          03          **          00
3ED3    **                  **          **          **          00
3ED2    **                  **          **          **          00

FP      3EF0                3EF0        3EF0        3EF0
PC      3EDB                3EDE        3EE3        3EE4
SP      3EDB                3ED8        3ED4        3EDB
```

17.21.2 EXAMPLE 2

When a file is created the operator QCO_CREATE is followed by the logical name to use and the field type and names. The list is terminated by the operator QCO_END_FIELDS.

For example:

```
                CREATE "B:ABC",B,AAA$,B%,CC
```

is translated as the Q code:

```
                24                        QI_STR_CON
                05 42 3A 41 42 43         "B:ABC"
                5E                        QCO_CREATE
                01                        Logical name B
                02                        Type string
                04 41 41 41 24            "AAA$"
                00                        Type integer
                02 42 25                  "B%"
                01                        Type floating point
                02 43 43                  "CC"
                88                        QCO_END_FIELDS
```

17.21.3 EXAMPLE 3

The recursive example given in 17.12.1:

```
                RECURS:(I%)
                IF I%
                 RECURS:(I%-1)
                ENDIF
```

Looks like this on the stack:

Address	Contents	Description
3D5A	0010	Parameter
3D59	00	Parameter type
3D58	01	Number of parameters
3D57	41	Device A
3D55	3D6D	Return RTA_PC
3D53	0000	ONERR address
3D51	3D29	BASE SP
3D4F	3D82	Previous FP
3D4D	3D4D	Globals start address

```
        3D4B            3D5A            Indirect address to parameter
        3D49            **
        3D48            7B              QCO_RETURN_ZERO
        3D47            84              QCO_DROP_NUM
        3D40            "RECURS"
        3D3F            7D              QCO_PROC
        3D3E            01
        3D3D            20              QCO_STK_LIT_BYTE
        3D3C            00
        3D3B            20              QCO_STK_LIT_BYTE
        3D3A            2E              QCO_SUB_INT
        3D38            0001
        3D37            22              QI_INT_CON
        3D35            FFFC
        3D34            07              QI_INT_SIM_IND
        3D32            001B
        3D31            7E              QCO_BRA_FALSE
        3D2F            FFFC
        3D2E            07              QI_INT_SIM_IND
        3D2C            000F            Parameter for next call
        3D2B            00              parameter type
        3D2A            01              parameter count
```

Note that the top 4 byte and the bottom 4 bytes are almost identical, this is shown at the point where the procedure is about to be invoked:

```
        RTA_PC          3D3F
        RTA_SP          3D2A
        RTA_FP          3D4F
```

17.21.4 EXAMPLE 4

Source code:

```
                EX4:(PPP$)
                LOCAL A$(5)
                GLOBAL B,C%(3),D$(5)
                J$=PPP$
```

The Q code header is:

```
                0035            size of the variables on stack
                0008            size of Q code length
                01              number of parameters
                02                type of parameter
                0011            size of global area
                01 42             global name
                01                global type
                FFE1              offset
                02 43 25          global name
                03                global type
                FFD9              offset
                02 44 24          global name
                02                global type
                FFD3              offset
                0004            bytes of externals
                02 4A 24          external name
                02                external type
                0006            bytes of string fix-ups
                FFCB              string fix-up offset (from FP)
                05                max length of string
                FFD2              string fix-up offset (from FP)
                05                max length of string
                0004            bytes of array fix-ups
                FFD9              array fix-up offset (from FP)
                0003              size of array
```

The Q code is:

```
                16 FFE9         QI_LS_STR_SIM_IND
                09 FFEB         QI_STR_SIM_IND
                81              QCO_ASS_STR
                7B              QCO_RETURN_ZERO
```

If this program is run on a CM from the procedure:

```
                XXX:
                GLOBAL J$(3)
                EX4:("RST")
```

The stack looks like:

```
        3EFA            "A:XXX"
        3EF9            00                      Number of parameters
        3EF8            00                      Top procedure
        3EF6            3EF9                    Return PC
        3EF4            0000                    ONERR address
        3EF2            3ED1                    BASE SP
        3EF0            0000                    FP
        3EEE            3EE8                    Start of global table
        3EEC            3EE4                    Address of global
        3EEB            02                      Global type
        3EE8            "J$"                    Global name
        3EE3            03 00 00 00 00          Global J$
        3EE1            **
        3EE0            7B                      QCO_RETURN_ZERO
        3EDF            84                      QCO_DROP_NUM
        3EDB            "EX4"
        3EDA            7D                      QCO_PROC
        3ED8            20 01                   QI_STK_LIT_BYTE
        3ED6            20 02                   QI_STK_LIT_BYTE
        3ED2            "RST"
        3ED1            24                      QI_STR_CON
        3ECD            "RST"                   Parameter
        3ECC            02                      Parameter type
        3ECB            01                      Number of parameters
        3ECA            00                      Device A:
        3EC8            3EDA                    Return PC
        3EC6            0000                    ONERR
        3EC4            3E83                    BASE SP
        3EC2            3EF0                    FP
        3EC0            3EAF                    Start global table
        3EBE            3E95
        3EBD            02
        3EBA            02 44 24                Global D$
        3EB8            3E9B
        3EB7            03
        3EB4            02 43 25                Global C%()
        3EB2            3EA3
        3EB1            01
        3EAF            01 42                   Global B
        3EAD            3ECD                    Indirection to PPP$
        3EAB            3EE4                    Indirection to J$
        3EA3            00 00 00 00 00 00 00 00         GLOBAL B
        3E9B            00 03 00 00 00 00 00 00         GLOBAL C%()
        3E94            05 00 00 00 00 00 00            GLOBAL D$
        3E8D            05 00 00 00 00 00 00            LOCAL A$
        3E8B            **
        3E8A            7B                      QCO_RETURN_ZERO
        3E89            81                      QCO_ASS_STR
        3E87            FFEB
        3E86            09                      QI_STR_SIM_IND
        3E84            FFE9
        3E83            16                      QI_LS_STR_SIM_IND
```

When running EX4 the offset FFE9 is added to RTA_FP (3EC2) to give 3EAB. The address at 3EAB is 3EE4 which is the address of the global J$. This address with a non-field flag is stacked. Similarly FFEB is added to RTA_FP to give 3EAD, which contains the address 3ECD, the address of the parameter PPP$.

17.21.5 EXAMPLE 5

Source code:

```
                TOP:
                PRINT ABC:(GET)
                GET


                ABC:(N%)
                RETURN(N%*N%)
```

At the point when ABC: has just been called the stack looks like:

```
3EFA            "A:TOP"
3EF9            00              NO. of parameters
3EF8            00              Top procedure
3EF6            3EF9            Return PC
```

```
3EF4           0000           ONERR address
3EF2           3EDD           BASE SP
3EF0           0000           FP
3EEE           3EEE           Global table
3EEC           **
3EEB           7B             QCO_RETURN_ZERO
3EEA           83             QCO_DROP_WORD
3EE9           91             RTF_GET
3EE8           73             QCO_PRINT_CR
3EE7           70             QCO_PRINT_NUM
3EE3           "ABC"
3EE2           7D             QCO_PROC
3EE0           20 01          QI_STK_LIT_BYTE
3EDE           20 00          QI_STK_LIT_BYTE
3EDD           91             RTF_GET
3EDB           0020
3EDA           00             Integer
3ED9           01             No. parameters
3ED8           41             Device A:
3ED6           3EE2           Return PC
3ED4           0000           ONERR
3ED2           3EC1           BASE SP
3ED0           3EF0           FP
3ECE           3ECE           global table
3ECC           3EE4           Address of N%
3ECA           **
3EC9           79             QCO_RETURN
3EC8           86             QCO_INT_TO_NUM
3EC7           2F             QCO_MUL_INT
3EC4           07 FFF7        QI_INT_SIM_IND
3EC1           07 FFF7        QI_INT_SIM_IND
3EBF           0020                   0400                    0300
3EBD           0020                   **                      1024
3EBB           **                     **                      0000
3EB9           **                     **                      0000

PC             3EC7                   3EC8                    3EC9
```