QSPL REFERENCE MANUAL

L. P. Deutsch

B. W. Lampson

University of California, Berkeley

Document No R-28

This document is a brief but complete description of a new language invented and implemented by the authors. This language is intended to be a suitable vehicle for programs which would otherwise be written in machine language for reasons of efficiency or flexibility. It is part of a system which also includes a compiler capable of producing reasonably efficient object code and a runtime which implements the input-output and string-handling features of the language as well as a fairly elaborate storage allocator. The system automatically takes care of paging arrays and blocks from the drum if they have been so declared.

The Language:

A QSPL program consists of **statements** separated by semicolons. Carriage returns and blanks have no significance in the language except that they:

1. Act as word (and comment) delimiters.

2. Are taken literally in string and character constants.

Warning: This is one of the many features of the language which can cause trouble for the unwary programmer. It is quite possible to write two statements without the separating semicolon and wind up with something which is legal, but not at all what was intended. It is a general characteristic of QSPL that it is very permissive; many things are legal which are not at all reasonable.

A statement may be:

1. A declaration.

2. A listing control statement.

3. An end statement.

4. A function definition.

5.   A comment,   which   is   a   line   beginning
     (after   a   semicolon   or another comment)
     with an asterisk ("*") and ending with   a
     carriage return (not ";").

Most statements are expressions,   so   we   will   discuss
them first.

also: control statements (IF, FOR, WHILE)
        INCLUDE

Expressions

An expression is made up of operands separated by
operators. Parentheses are allowed to any reasonable depth.
The operators are arranged in a hierarchy of binding
strength or precedence. Those at the top of the following
list are executed latest, so that a+b*c is a+(b*c).

&   denotes sucessive evaluation. The value of the
result is the value of the last expression in
the string. Thus

        A+B & C+D;

or more plausibly

        F(A,B) & G(L,Y);

which causes both functions to be called in the
order in which they are written.

WHERE   is similar to &, but causes the following
expression to be evaluated first. It may not be
iterated. Thus

        F(X,Y) WHERE N+14;

FOR,WHILE
    takes the form

        <expression> FOR <for clause>;

or

        <expression> WHILE <expression>;

The expression is evaluated repeatedly under
control of the for clause (see below for the
syntax of this construct). The final value of
the expression is discarded, and the value of
an expression involving FOR or WHILE is
undefined. Of course, something like

        (A[I])[J]+0 FOR I=1 TO N FOR J=1 TO M;

is legal.

IF   takes the form

<expression> IF <expression> ELSE
<expression>;.

The second expression is evaluated. If it is
non-zero, the first expression is evaluated.
Its value becomes the value of the whole thing,
and the third expression (which, by the way,
may contain another IF) is skipped.   Otherwise
the first expression is skipped, and the third
is evaluated. Thus

    X←4 IF Y=6 ELSE X←5 IF Y>=0 ELSE X←6;

If the final ELSE is omitted, 0 will be
supplied.

←   is the assignment operator. It ranks on the
        same level as '.' for its left-hand operand,
        and just below IF for its right-hand one.   The
        right-hand operand is evaluated, and its value
        becomes the value of the left-hand one.   The
        whole expression is then treated as though only
        the left-hand side had been written.

OR   Is the logical or.   If either operand is a
        relation (or an expression containing logical
        operators connecting at least one relation),
        then the result is 0 or 1 depending on whether
        both operands are true (non-zero).   If both
        operands have ordinary values, these values are
        combined with the machine's MRG instruction.
        Thus

            A<4 OR B<5

        is true if either relation holds;

            A<4 OR X+1

        is true if A<4 or if X+1 is not zero. In both
        these cases, the second operand is not
        evaluated if the first one is true. But

            F(X,Y) OR Z

        is the 24-bit logical or of Z and the value of
        the function call. The operands of an OR are
        never re-ordered.

AND,EOR

AND is the logical and. It is exactly the same
as OR in the way it treats its operands,
differing only in the result.
EOR always converts its operands to values and
uses the EOR instruction.

NOT
is the logical not. If its single operand is a
relation (see the discussion of OR) its value
is inverted (0 becomes 1, 1 becomes 0).
Otherwise, a 24-bit complement is taken (with
EOR =-1).

= # < <= > >=
are the relations. Each one evaluates its
operands and then performs the indicated test.
For these and all the arithmetic operations,
the operands may be re-ordered if it suits the
compiler's convenience.

MOD
A MOD B is the remainder of A/B.

+ -
perform 24-bit integer addition or subtraction.

* / LSH RSH LCY RCY
"*" and "/" perform 24-bit integer
multiplication and division. No test is made
for overflow on division.
The shift operations shift the first operand
the number of places indicated by the second
operand. Vacated bits are replaced by zeros.
The cycle operators do an end-around shift.

+ - GOTO RETURN SRETURN DO (unary operators)
The unary "+" and "-" do the obvious thing. DO
is a noise word and is ignored. It may be
convenient for constructions such as this:

        DO F(X,Y);

GOTO transfers to the address which is the
value of its operand (see the discussion of
labels below).
RETURN and SRETURN evaluate their operand(s),
of which there may be at most 3. They leave
the values in the A, B, and X registers
respectively and return through the return link
of the most recently defined function (see
below). If this is not desired, the RETURN may
be modified by following it with FROM
<expression>. In this case the return is to

the address which is 1 + the value of the
expression. Thus

        RETURN X+Y FROM FCNL;

The programmer should be sure that FCNL Has a
proper return address in it, since the compiler
will not check this. The operand of RETURN may
be omitted.   RETURN is a normal return, or a
no-skip return (failure return) from functions
called with a failure location (see function
calls below). SRETURN is a skip return, or a
normal return from functions called with a
failure location.

()   (function calls).
            The arguments of the function are enclosed in
            the parentheses, separated by commas. Thus

        F(X,Y+5,Z).

Note that the function may be specified by an
expression; thus

        (A+B)(A,Y+5,Z)

is perfectly legal. It causes control to be
transferred to the location which is the value
of the expression A+B with the specified
arguments. Beware. The values of the first
three function arguments are transmitted in the
A, B and X registers respectively.   The
addresses of the values of further arguments
are put into NOP instructions which precede the
function call. The function is called with a
POP which leaves the link in 0 and transfers to
the location addressed by it. Thus

        F(A,Y+5,Z,P+1,Q)

compiles

        LDA P; ADD =1; STA T:+1; LDA Y; ADD =5; CAB;
            LDA A; LDX Z; NOP Q; NOP T:+1; CALL*
            F;.

In addition to 0 or more arguments, a function
call may also have a failure location. This
corresponds to a no skip return (RETURN
operation) from a function which also has a
skip return (SRETURN operation).   The failure

location comes after the arguments and is
preceded by a colon, thus:

F(A1,A2:L).

The failure location may be either a label, in
which case control will go there in case of
failure, or RETURN or SRETURN possibly followed
by a _single_ expression, which will be executed
in case of failure. A function normally
returns just one value, which is passed in the
A-register. However, it may return no values,
or up to three: additional values are put in
the B and X registers. The first value is
always the one used for further computation
(e.g. in situations like F(X)+1), but any
subset of the values can be saved by putting a
save list after the failure location, preceded
by another colon, thus: F(X,Y:M:V1,V2,V3) or
even F(P,R::X,Y, Z) if there is no failure
location. The save list must be a list of
simple variables. It is all right to have a
comma with no name: the corresponding return
value just gets lost. See below for a
discussion of function declarations. Note that
this calling convention is _not_ the same as
FORTRAN's. In particular, in the above example
nothing the function does (within reason) can
affect the value of A or Z. It is possible to
transmit the _address_ of A or Z with the
reference operator, however (see below).

. (tailing). The . must be followed by a field
name (see discussion of declarations below).
The resulting object refers to the specified
field relative to the address which is the
value of the first operand. Thus, if we have

DECLARE FIELD A(1), B(2);

and if X contains 143, then X.A refers to
location 144, X.B to 145, X.A.B to 2 + the
contents of location 144. A tailed operand may
appear on either side of an assignment. Cf the
discussion of PAGED declaration for the
treatment of paged blocks.

$ (binary, same precedence as .). The construct
        T$F is almost equivalent to @T.F. I.e., it
        refers to the bits of T (not the word addressed
        by T) selected by F. The word displacement of
        F is ignored, and F must not cross a word
        boundary.

@ $ (reference and indirection).   The reference
        operator "@" takes an operand which must be an
        address (i.e. acceptable on the left side of
        an assignment) and returns this address as its
        value. Note that this implies that iteration
        of the reference operator is illegal (in fact
        it does not make any sense).   The indirection
        operator "$" evaluates its operand and returns
        this value as an address. The sequence "@$" is
        equivalent to no operation, except that "$" on
        an address is compiled with the machine's
        indirect bit, and will therefore be affected by
        the presence of indirect or index bits in the
        contents of address. If we have written

                DECLARE FIELD S(0);

        then <E>.S is equivalent to $<E>, with the
        exception noted above.

[]   (subscripting). A single subscript is allowed.
        As with function calls, the object being
        subscripted may be an arbitrary expression. If
        it has been declared as an array, the compiler
        loads the subscript into the X-reg and compiles
        an indirect reference through the array name.
        I.e., it expects the array name to contain the
        base address of the array with index bit on.
        For any other expression the "[]" operator is
        equivalent to "$"+".". Thus

                (A=B)[C OR D]+1

        compiles as
        LDA C; MRG D; STA T; LDA A; SUB B; ADD T; XXA;
        LDA 0,2; ADD =1;

Primaries

The primaries for expressions may be numbers, names, string constants, or character constants.

numbers

A number may be an integer constant or a real constant. An integer constant is a string of digits, possibly followed by B or D, possibly followed by a single-digit scale factor. B makes the number octal; if it is absent, decimal is assumed. Thus 100D = 1D2 = 144B = 1B2+44B = 100. A real constant is of the form xxx.xxxExxx. Either the dot or the E must be present. If the dot is present, there must be some digits before it; if the E is present, there must be some digits after it. For further details, consult the description in R-21 of the SIC syspr, which is used to convert real constants to binary form.

names

A name is a string of any number of letters and digits beginning with a letter. Only the first six characters of the name are significant. A name must be declared (see below). All names except parameters and fields are treated in exactly the same way when they occur in expressions (except for subscripting). E.g. a string name refers to the pointer to the string descriptor which is the value of the name. Thus, if S is a string

        S ← A+1;

simply stores A+1 into S; this is probably not reasonable. Functions are provided to convert between strings and numbers.

reserved words

There are about 80 reserved words (see Appendix B) which may not be used as names. In addition, about 20 locations in the runtime (see Appendix C) are predeclared as external; attempts to declare them for other purposes will fail.

character constants

A character constant has the form

'<three or fewer pseudo-characters>'

and may be used wherever a constant is used. A pseudo-character is any character other than "&", or "&" followed by one of the following:
1. Another "&" or a "!". The two are equivalent to a single "&" or "!" in the constant.
2. Three octal digits. The number thus defined, truncated to 8 bits, counts as one character.
3. A letter. The ASCII (internal) code for the letter + 100B is the value of the pseudo-character.

The characters are right-justified in the constant, which is filled out with blanks (0) on the left. It is an error to have more than 3 pseudo-characters in the constant.

string constants

A string constant has the form "<any number of pseudo-characters>". It is legal in any context in which a string name is legal. A descriptor will be created which points to the constant string. If the string constant appears alone immediately after a left arrow, the variable to which the constant is being assigned is assumed to hold a pointer to an already existing string descriptor; in all other cases, space for a descriptor will be allocated for the constant by the compiler. In any case, writing into the string will alter the constant.

A variety of operations are provided for converting field names into constants:

1. F(expression) has the value of T after the statements

        T←0; T$F←expression;

   have been executed. F must not cross word boundaries.
2. The function FSHIFT(F) has 23-the rightmost bit position occupied by F as its value. F must not cross word boundaries. The value of FSHIFT is a constant.
3. The function FMASK(F) has as value a constant which has one bits in positions selected by the field as its value. It is equivalent to F(-1). F must not cross word boundaries.
4. A field name F appearing in any context other than
        F(

                    .F
                    $F
            is equivalent to a constant whose value is
            the word displacement of the field.

constant_expressions

        Any expression involving operators of precedence higher
than FOR and constant operands will be evaluated by the
compiler yielding a result which behaves exactly like a
constant.

Declarations

Variables are declared with DECLARE or FUNCTION
statements or by appearing as labels. The syntax of
variable DECLARE is

        DECLARE [FIXED or PAGED] [INTEGER or REAL or
            STRING] [ARRAY] [EXTERNAL or ENTRY or
            LOCAL] <namelist> ;

The stuff after the DECLARE may be repeated as many times as
desired. Once FIXED, PAGED or ARRAY has been used it
remains in effect for the remainder of the current DECLARE
statement. INTEGER is assumed if it is omitted, but once
STRING has been used it remains in effect until INTEGER
appears again. Each name in the namelist may be preceded by
"$" (which makes it an entry) or by "*" (which makes it
external, i.e. prevents storage from being assigned for it).
If FIXED and ARRAY are both is present, a name may be
followed by an expression in parentheses (or brackets).
Thus

        DECLARE FIXED ARRAY A[12], B[X*2+14];

The expression (which must be a constant) will be evaluated
and that many cells assigned for the array at compile time.
The base address of the region assigned, with the index bit
set, will be stored in the name. If a name is declared
ARRAY without any storage being assigned, the system will
assume that its value is a pointer to an array with the
index bit set.
I.e., it will compile

            LDX I; LDA* A; STA B
for   B←A[I].

    Example:
            DECLARE INTEGER A, B STRING D, $G1, G2,
            EXTERNAL G3, G4,
            ARRAY E(X+Y[4]), INTEGER C(10);
declares two scalar integers, one integer array which will
be assigned 10 locations when the declaration is executed,
two local scalar strings (D and G2), one local string array
which will be assigned X+Y[4] locations when the declaration
is executed, one scalar string which is an entry (G1), and
two scalar strings which are assumed to be defined elsewhere
(G3 and G4).

declared_paged

A name or an array may be declared paged by putting the
word PAGED in front of its declaration. This attribute,
once mentioned, applies to all the names declared following
it in the same statement. If an array is declared PAGED
(not a FIXED array, of course), all references to it will
be made to the drum. Correct access to the array will be
obtained only if it is subscripted in the usual way: A[I].
It is not true that (A+1)[I] is equivalent to A[I+1], for
example, as is the case for core arrays.

If a name declared paged is not an array, the only
effect is that when it is tailed the system will assume it
contains a drum address. Such an address can only be
correctly obtained with PMAKE (see below). It is the
programmer's responsibility to see that:
  a. It does contain a drum address generated with
      PMAKE.
  b. The field name used for tailing has a word
      displacement less than the block size
      specified by the PMAKE. Unpredictable errors
      will occur if this rule is not observed.
Declarations of fields are not affected by PAGED.
Indirection ($) and subscripting ([]) will work properly on
a PAGED pointer. Arithmetic may be done on PAGED pointers
in the usual fashion, provided the result is within a block
allocated by a single call to PMAKE. thus after

        DECLARE PAGED P; FIELD F0(0),F1(1),F2(2);
          P+PMAKE(2);

the statements

        A+P.F1; A+(P+1).F0; A+(P+1)[0];

have the same effect, but

        A+P.F2; A+(P+2)[0];

are all erroneous, since only two words were allocated in
the block pointed to by P.

declared_string

When a name is declared to be a string, a single
storage location is reserved for it unless FIXED has been
used. Strings are specified, however, by four-word
descriptors. The address of such a descriptor must be put
into the string variable befor it is used in any string

operation. For non-FIXED strings, this is usually done with
the SETUP function, possibly preceded by a MAKE;
alternatively, the address of a descriptor obtained in some
other way can be used. If a string variable is not properly
initialized, the consequences of using it any string
operation are likely to be serious.

     If a string declaration is preceded by FIXED, the
four-word descriptor is assigned by the compiler and its
address is the initial value of the string. If a FIXED
STRING is followed by a parenthesized expression, that many
characters are allocated for the string and the descriptor
is initialized to point to the area thus allocated.
Example:

          DECLARE FIXED STRING S,T,U(5),V(240);

allocates string descriptors for S and T; they must be set
up to point to strings by SETUP.   It also allocates 5
characters for U and 240 for V and sets up the descriptors
properly.

initialized declarations

     An integer may be initialized by following its name
with "+" <constant> or "+" <name>. Thus,

          DECLARE A+3,B+14,C+A;

makes 3 the initial value of A, 14 the initial value of B.
Of course, any expression which can be evaluated by the
compiler may be used as a constant. This is not the same as
a PARAMETER declaration (see below).   The use of this
construct is not recommended if the program changes the
values of the variables, since the program must then be
reloaded in order to be restarted.

     A FIXED ARRAY can be initialized in the same way:

          DECLARE FIXED ARRAY A[10]+1,3,5,7,11,13;

The first six elements of A are initialzed as indicated.
The remaining four elements are initialized to 0.

     A fixed string or a fixed string array may be
initialized in the same way, but the initial values must be
string constants. warning: writing into initialized
strings will destroy the contents.

If any declaration causes space to be allocated at the
point in the program where the declaration occurs, a branch
over it is compiled. Declarations may therefore be freely
interpolated in the program.

field declarations

Another form of DECLARE is the following:

    DECLARE FIELD <name>
        (<constant>[:<constant>,<constant>])

which defines a field. Lots of fields can be defined if
desired. The first constant specifies the word displacement
of the field, the other two the bit positions in the word.
Bit positions can take on values between 0 and 47. A field
may span two words, but it may not be more than 24 bits
long, thus:

    DECLARE FIELD A(0),B(1),C(2),C1(2:0,5),
        C2(2:3,20),XYZ(2:12,23);

defines six fields. The last three might be thought of as
subfields of C, but they do not have to be used in this way.
If P were a pointer to a three-word data object, for
example, then P.XYZ would refer to the last 12 bits of the
third word of the object. Such objects can be created from
nowhere with the MAKE function or, of course, may be
allocated by the programmer.

Names declared as FIELD are output to DDT with their
word displacements as value. If they appear not following a
".", they are treated as constants equal to their word
displacements. Thus,

    $(PTR+B) equiv $(PTR+1) equiv PTR.B .

A full-word field may be declare REAL or PAGED. This
means that whenever it is used for tailing, the resulting
quantity is considered REAL or PAGED respectively.

parameter declarations

The declaration

    DECLARE PARAMETER C1+1,C2+2,C3+3;

makes the names C1,C2,C3 equivalent in all ways to the
constants 1, 2,3 for the rest of the program. Any constant
may appear on the right of the "+". Note again that any
constant expression may be used where a constant is
required. Parameters, unlike other names, may be
redeclared.

equivalence declaration

The declaration

DECLARE INTEGER Q=R, S=T[3];

is legal only if T has already been declared as a fixed
array.   It causes Q to be assigned to the same location as
R, S to the same location as T[3].

function definition

A function is defined by

[REAL] FUNCTION or ENTRY [S] name(arglist);

If the word REAL appears, the function is assumed to return
a floating point value; otherwise, it is assumed to return
integer values, if any. If a S precedes the function name,
the name is made an entry. Except when compiling under
NOLIST LOCAL (described below), there is no difference
between FUNCTION and ENTRY. Each argument in the arglist
can be preceded by INTEGER, STRING or ARRAY and is declared
automatically.   INTEGER is assumed unless otherwise
specified. If ARRAY is specified, the index bit will be
merged into the value supplied. A name can be redeclared in
a function definition (this is illegal in any other
context), but only if the redeclaration exactly matches any
previous declaration. The system creates a return link by
prefixing the function name with X. The statement

FUNCTION <FNAME>(A, ARRAY B, STRING C);

would compile

STA A;CBA;MRG =2B7;STA B;STX C;LDX 0;STX
    X<FNAME>;

If additional arguments INTEGER D,E were supplied, the code

LDA* -1,2; STA D; LDA* -2,2; STA E;

would be added.

The function name itself is also declared by this statement.   A storage location is reserved for it, and the address of the first word of the function (STA A above) is put into this address.

The link may be specified explicitly, if desired, as follows:

FUNCTION F(Q,R), LINK W;

recursive functions

A function may be declared recursive by

[REAL] RECURSIVE FUNCTION or ENTRY [S] F(A,
    B), SAVE E;

The effect will be that whenever the function is called the link and the current values of A,B and E will be saved. When the function returns (via a RETURN or SRETURN with no FROM modifier and only one value returned), the saved variable values are restored.

Space for saving the variables is obtained by calling the function in the reserved location RECSTK. This cell is initialized to be a call to MAKE, but the user may supply his own function. The call

TOPRST ← RECSTK(N)

where N contains the number of words required and the function returns the address of the first word.

Space is released by

DO RECUNS(TOPRST);

and RECUNS is initialized to FREE.

The cell TOPRST, which is also reserved (i.e. built into the runtine) contains the address of the current top of the recursion stack. Its old value is saved in the second word of the stack entry.
If a function call appears in a compiled expression, it is not safe to re-execute the expression inside the function, since the expression may use temporary locations which are not saved when the function is called. Beware.

declaration_of_labels

A symbol is declared as a label by writing it at the
beginning of a statement followed by a colon. It is treated
exactly like a function name: a storage location is reserved
for it and initialized to the address of the first
instruction of the statement. Any statement can be labeled.
A label is assumed to be an integer scalar. If we have
A:... ; GOTO A; this will compile

        :A BSS 0; ...;BRU* A; ...;A ZRO :A;

so that the right thing happens. If the symbol is preceded
by a $, the label is made an entry.

These conventions for arrays, strings and labels make
it very easy for them to be transmitted as arguments.

Real (floating-point) numbers

Real numbers occupy two words of storage rather than
one and therefore have a somewhat anomalous status in OSPL,
which otherwise takes the position that any kind of quantity
only occupies a single word (integers, strings, labels,
functions, and arrays all have this property). We define a
real operand as a real name (possibly subscripted if an
array), a real constant, a real function, a real expression,
or an expression tailed by a real field. A real expression
may be formed in the following ways:

> 1. By combining two real operands with any of
> the following binary operators: +, -, MOD, *,
> /. If any of these operators is applied to a
> real operand and an integer constant, it will
> convert the constant to a real number. A real
> operand and any other kind of integer operand
> will produce an error.

> 2. By unary + or - applied to a real operand.

> 3. By the construct <real operand> IF <integer
> expression> [ELSE <real operand>].

In addition, two real operands may be compared by any of the
relational operators (=, #, >, <, >=, <=). The test is made
by doing a floating subtraction and testing the result
against zero: beware of round-off error in testing for
equality. Also, a real operand may appear in a RETURN or
SRETURN provided it is the only argument of the operation.
There is no restriction on mixing real and non-real
arguments of functions: however, the types of the actual
arguments in a call must correspond to those in the function
definition. The compiler does not check this, and an error
will probably cause chaos at run time.

Various special functions are available for doing the
same things to real numbers that one can do to integers.
RIN and ROUT provide floating point input/output; CSR and
CRS provide conversion between reals and strings; FIX and
FLOAT convert between reals and integers. These are all
discussed in detail in the later sections on special
functions. There is a library of mathematical routines with
OSPL-compatible calling sequences available, including SIN,
COS, TAN, ATAN, EXP, LOG, LOG10, and random number
generation: this is described in a separate document.

Control Statements

the IF construct

The construction

        IF <expression> DO;
            .
            .
        ELSEIF <expression> DO; (0 or more ELSEIFs
        allowed)
            .
            .
            .
        ELSE DO;
            .
            .
            .
        ENDIF;

is legal with the obvious meaning. Any sequence of
statements balanced with respect to IF and ENDIF may appear
in place of the dots. Of course, IF may be nested. Proper
use of indentation is strongly recommended. The final ELSE
may be omitted.

the FOR and WHILE constructs

The construction
        FOR <for clause> DO;
            .
            .
            .
        ENDFOR;
is also allowed. The arbitrary sequence of statements
balanced with respect to FOR and ENDFOR which is symbolized
by the dots is executed repeatedly under control of the for
clause, whose syntax has three forms:

        <name>←<expression> WHILE <expression>

which causes the value of the first expression to be
assigned to the name and the second expression tested each
time around the loop. When the test fails (value of the
expression=0) repetition stops. The assignment and test are
performed once before the loop is executed;

<name>←<expression> [BY <expression>] TO
    <expression>

with the obvious meaning. If the BY is omitted, an
increment of 1 is assumed. Repetition continues until the
value of the name is greater than the TO expression, unless
the latter is a negative **constant**, in which case it
continues until the name is less. A test is performed
before the loop is executed for the first time. The special
cases

        I←<expression> BY 1 TO N

and

        I←<expression> BY -1 TO 0

are recognized and compiled more efficiently.

    The similar construction

        WHILE <expression> DO;

            •
            •
            •
        ENDFOR;

is also allowed. The body of the loop is executed
repeatedly as long as evaluation of the expression yields a
true (nonzero) result. The expression is evaluated once
before the loop is entered for the first time.

Miscellaneous Statements

Listing may be controlled with the statements LIST and
NOLIST.   Either may be followed by SOURCE, CODE, or BINARY,
and turns on or off the specified form of output. It is not
a good idea to turn binary output on and off, since this
will in general produce an unloadable result.

Two special options concerning allocation of variables
are   also   controlled   by NOLIST. NOLIST FREE will prevent
ZRO's for uninitialized scalars from appearing on the
assembly-language listing; this may be useful if re-entrant
programs are desired. NOLIST EXTERNAL will cause undeclared
variables to be treated as external; normally they are
treated as errors and space is assigned for them.

NOLIST PAGED puts the compiler into a mode where the
code produced for tailing a paged pointer no longer assumes
that the block lies within a single page.   It is intended
for programs where the user is allocating paged storage
himself without regard to page boundaries.

The statement

    INCLUDE "<file name>",

has the effect of placing the entire contents of the named
file in the program at that point. This process may be
nested, i.e. the file being inserted may itself contain
INCLUDEs. Note that since the file is inserted verbatim, it
should **not** end with an END statement. The INCLUDE feature is
meant   primarily   for   groups of programs with common
declarations. To this end, the statement NOLIST INCLUDE is
provided with the following effect: if it occurs in the
original source file, it has no effect, while if it occurs
in an INCLUDEd file, it terminates processing of that file.
Thus a main program could have the form

            .

          . (declarations)

            .

          . NOLIST INCLUDE;

            .

          . (remainder of program)

            .

. END;

and any subprogram could use its declarations by INCLUDEing
it.

The statement

IDENT <name>;

will cause the name to be output to DDT as the program name.
No more than one IDENT may appear in a program.

A program should be terminated by an END statement,
i.e.

END;

Macro Facility

The format of a macro definition is:

```
DECLARE MACRO
     <name>(<dummies>)+<definition>;
```

where <name> is the name of the macro being defined,
<dummies> is the list of dummy argument names, and
<definition> is the definition. <name> must be hitherto
unmentioned identifier. <dummies> may be an empty list; if
it is not empty, it is a sequence of identifiers separated
by commas. These identifiers serve only to indicate the
place within the definition where actual arguments are to be
substituted: their use here does not conflict with their
previous or subsequent uses for any purpose. The
<definition> is any sequence of tokens (identifiers,
numbers, operators, character constants, or string
constants) not including a semicolon. It need not be a
legal statement, expression, or anything else.

A macro call looks almost like a function call, i.e.
has the form <name> (<arguments>); However, the <arguments>
are not required to be legal expressions: they need only be
sequences of tokens balanced with respect to parentheses,
not containing semicolons, and delimited by commas which are
not enclosed in inner parentheses. For example, STRING
X(20), *, and (B,C) are legal arguments. The effect of the
macro call is that the definition, with the actual
arguments, replaces the call before any further processing
is done on the statement. A macro call may appear anywhere
in the statement, not just where a function call would be
legal. Macros may call other macros. If listing is being
done, statements will be listed before macro substitutions
have been performed; this is also true when a statement is
listed in response to an error.

A word of warning for those accustomed to the NARP
macro facility. Since substitutions are performed on the
basis of tokens rather than characters, no substitution
occurs within character or string constants in the
definition., e.g.

```
DECLARE MACRO S1(X)+"X"
```

will not cause a substitution. Also, concatenation is not
available. Finally, each dummy argument has a name of its
own and the proper number of arguments must be supplied at
each call.

Two examples of useful macros:

DECLARE MACRO INC(X)←X←X+1;

causes INC(A) to be equivalent to A←A+1;

DECLARE MACRO TWO(X)←(X)*2;

causes TWO(X+Y) to be equivalent to (X+Y)*2. Note that if
the definition had been simply X*2, then TWO(X+Y) would have
been equivalent to X+Y*2, which is presumably not what is
wanted.

Special Functions

The following special functions are a standard part of the language. They provide all the built-in storage allocation, string handling and input-output facilities. If more elaborate facilities are required, recourse may be had to machine-language routines. The necessary linkages are described under function calls and declarations above.

1. Storage allocation functions

MAKE, SETARRAY

MAKE(<expression>)

creates a block of storage of the length specified by the expression (but of at least two cells) and returns a pointer to this block as its value. In fact, one extra cell is assigned by the system; the user should keep his hands off this cell which is the one before the one pointed to by the value of the MAKE function. An alternate form is

MAKE(<expression>,<array name>)

which assigns the block out of the specified array, which must have been properly initialized beforehand by a call of

SETARRAY(<expression>,<array name>);

Only blocks of the size specified in the call of SETARRAY can be assigned in this way. Blocks of any size can be assigned by a simple MAKE.

PMAKE

To allocate space on the drum the function PMAKE should be used. It is exactly like MAKE, except that the second argument, if present, should be a paged pointer to an object near which the new space should be assigned if possible. Proper use of this feature will greatly improve the efficiency with which paged objects are accessed. See the discussion of the PAGED declaration for further information about the proper use of addresses obtained from PMAKE.

FREE

To release a block of storage, do

FREE(<expression>) or FREE(<expression>,
      <array name>)

where the value of the expression is a pointer to the block.
The function has no meaningful value. The storage allocator
will attempt to coalesce freed blocks, but since it cannot
move blocks around, it is possible to fragment storage
hopelessly by acquiring and releasing blocks of many
different sizes in an indiscriminate manner. If the system
runs out of space, it will complain and quit. Note that

      FREE(MAKE(4))

acquires and immediately releases a block of four words. It
is exactly equivalent to NOP (except for timing). FREE also
works for drum space.

BCOPY

     To copy one block of storage into another one of equal
size use

      BCOPY(<expression>,<expression>).

The first expression is a pointer to the destination, the
second to the source. These must be pointers acquired my
MAKE (or carefully fabricated) since the length of the block
is determined from the contents of the extra hidden word
provided by MAKE. The source block must have been created
by a MAKE with a single argument. If the source block does
not have the hidden word,

      BCOPY(<expression>,<expression>,
        <expression>)

may be used, where the third argument specifies the number
of words to copy.

2.   Paging facility

NPG,PM,/SQPDATA,NPB,PCAT

The paging facilities provide a means for the user to
allocate and access a large (up to 2↑19 words) address
space, by buffering parts of this address space between core
and drum in fixed-size pages. The user can specify the page
size, the amount of core space to allocate for buffers
(which can be changed dynamically during execution), and the
size of the address space; individual pages may be locked
into core for a time and later allowed to be swapped out
again; the user's paged data may be divided into a number of
categories, which allows more efficient allocation of space
by grouping objects of the same category on the same page.

At the time that INIT is called (see the INITIALIZE
function in section 6), certain cells in the runtime are
examined to determine the setup of the paging logic.   The
names of these cells are (are) pre-declared EXTERNAL. The
cell NPL contains the page size as a power of 2, which must
be between 8 and 11. The cell NPG contains the size of the
desired address space as a multiple of 2↑NPL; the size
cannot exceed 2↑19. If NPG contains a zero, it is assumed
that no use will be made of the paging logic, and any calls
on it will produce error comments. The cell NPB contains
the number of core buffers to be provided. If it contains
0, all available space will be used for buffer. The cell
NPC contains the highest category number which will be used.
The cell PM contains a positive number if the direct drum
accell machinery, BRS 124-127, is to be used for storing
paged data, or a negative number if a random file called
/SQPDATA is to be used; the former is somewhat more
efficient, especially if the address space is large, but the
latter can be accessed by other programs via the ordinary
file machinery whereas the former cannot.

A few other cells are of interest. The cell PCAT is
examined whenever a call is made to PMAKE. If it contains a
non-zero number, the new block will be allocated on a page
reserved for data of the designated category.   If it
contains a zero, the new block will be allocated on some
convenient page without reference to category. A call of
PMAKE with a valid drum address as the second argument takes
precedence over the setting of PCAT.

LOCK,UNLOCK

A page may be locked into core with

LOCK(X)

where X is a drum address; the value is the corresponding
core address, which is guaranteed to remain valid until the
page is unlocked. The function

UNLOCK(A)

where A is a core address, stores the corresponding drum
address in a cell called PADDR and returns the old lock
count (which is incremented by LOCK and decremented if
non-zero by UNLOCK) as value; it is all right to unlock an
unlocked buffer. The cell NUP always contains the number of
buffers which are not locked at the moment.

Page buffers are allocated downwards (towards
low-numbered addresses) from the initial setting of a cell
called ESTORG; the bottom of the buffer area is put into the
cell EARRAY by the INIT operation. If the user wants to
reduce the amount of space available for buffers, he may use
BPUT(X), where X is a core address in a buffer. The buffer
will be returned to the pool of space available to the core
allocator (MAKE). The converse operation is BGET(X), which
restores the buffer for use by the paging logic. Note that
the buffer area is defined at INIT time (as the NPB * 2↑NPL
cells just below (ESTORG) - 2↑NPL and BPUT and BGET may only
be used on address in this range. INIT allocates space up
from BSTORG for tables for the drum allocator, leaving the
first unused cell in SARRAY. Thus SARRAY and EARRAY
bracket the core not used by the paging logic after and
INIT, while BSTORG and ESTORG bracket the core available to
it before an INIT.

3.   String handling functions

string descriptors. SETUP

A string is described by a four word descriptor which
specifies the beginning and end of a the area assigned to
the string, the reader pointer, and the writer pointer. The
function

        SETUP(<string name>,<size>)

will obtain a block specified size and set up the descriptor
pointed to by the string name to point to that block. The
name must already contain a pointer to a descriptor; if it
contains a 0 a runtime error will result. The alternate
form

        SETUP(<string name>,<size>,<expression>)

will make a descriptor which points to the specified number
of characters starting with the word pointed to by the
expression. The storage allocator is not involked; it is
the programmer's responsibility to create the descriptor);
it is the programmer's responsibility to ensure that the
proper amount of space is in fact available.

SETS. SETR. SETW. LENGTH

To set the reader and writer pointers of a string, use

        SETS(<name>,<expression>,<expression>).

The first expression specifies the reader pointer, the
second the writer pointer (which must be greater; if it is
not, the reader is set equal to the writer pointer).
Characters are numbered starting at 0. To set the reader
pointer only, use

        SETR(<name>,<expression>).

To set the writer pointer only, use

        SETW(<name>,<expression>).

To obtain the length of a string (writer pointer-reader
pointer) use

        LENGTH(<name>).

None of these functions except LENGTH has a meaningful
value.

GCI, GCD, WCI, WCD, APPEND, GC

       To get the next character from as string and increment
the reader pointer, use

       GCI(<name>).

If there is no next character, there will be an error
comment and a halt. To avoid this, use the alternate form

       GCI(<name>:<failure location>)

(see discusion of failure locations below). This convention
is also used for the next five functions.

       GCD(<name>)

reads a character from the end of the string and decrements
the writer pointer.

       WCI(<expression>,<name>)

writes the character specified by the expression on the
string specified by the name. It fails if there is no room.

       WCD(<expression>,<name>)

writes the character on the front of the string, at the
location of the reader pointer, and fails for the same
reason. These functions have the character written as their
value.

       APPEND(<name>,<name>)

appends the second string to the first one.   It fails if
there is not enough room. It has no meaningful value.

       GC(<name>)

yields the next character of the string, but does not
advance the reader pointer. It never fails, but yields junk
if the string is empty.

string_moving, BCOPY, SCOPY

The expression A+B (where A and B are string names)
simply moves the contents of B (presumably a pointer to a
descriptor) into A. To copy the descriptor, the BCOPY
function might be used, since string descriptors are just 4
word blocks:

        BCOPY(B,A).

Be sure to read the section on BCOPY above. To copy the
string, use

        SCOPY(B,A).

The effect is that of SETS(B,0,0) followed by APPEND(B, A).
SCOPY, like APPEND, fails if there is not enough room in B.

string/number_conversion, CNS, CSN, CSR, CSR

To convert a string S to a number, write

        C$N(S).

To convert a number N to a string S, write

        CNS(N,S).

This converts a signed number to its decimal representation,
producing only enough digits to accurately represent the
number. Extra arguments may be supplied which specify radix
(10 assumed) and the number of characters in the string
version (-1 or free format assumed). Corresponding
operations for floating point numbers are

        C$R(S)

and

        CRS(R,S).

4.   File-naming functions

INFILE, OUTFILE, FTYPE, ERROR

A file is opened for input with

    INFILE(<string name>:<failure location>).

the string contains the full name of the file. This
function requires the presence of a failure location to
which control transfers in the case the function fails. Its
value is the file number.

    OUTFILE(<name>,<expression>[:<failure
        location>])

does the same thing for output. The expression is the
option word which BRS 16 takes in the A-reg. It will be
assumed to be 0 if not supplied. Both of these operations
leave in the location FTYPE the type word returned by the
BRS, in case of failure, the error returned by the BRS is in
location ERROR.

INNAME, OUTNAME

To acquire file names, use

    INNAME(<name>:<failure location>)

and

    OUTNAME(<name>:<failure location>)

both of which collect the name from the teletype and appends
it to the string supplied. Both transfer to the given
location in the event of failure, and have the terminating
character as value.

CLOSE, CLOSALL

To close a file, do

    CLOSE(<expression>).

The expression's value should be the file number. To close
all files, do

    CLOSALL();

5. Input-output functions

CIN, COUT, WIN, WOUT, SOUT, CRLF

To read a character, use

        CIN(<expression>);

The value of the expression should be the file number. This
function simply does a CIO. Its value is the character
read. To write a character, use

        COUT(<expression>[,<expression>]);

File 1 is assumed if not specified. This function has the
character written as argument. To read and write a full
word, use WIN and WOUT in exactly the same way. To write a
string, use

        SOUT(<name>[,<file>]).

To write carriage returns, use

        CRLF(<expression>[,<file>]);

The expression specifies how many should be written.

IIN, IOUT, RIN, ROUT

To read a number, use

        IIN(<file>[,<radix>]).

Decimal radix is assumed. To write a number, use

        IOUT(<expression>).

Extra arguments, in order, are the file (1 assumed), the
radix (10 assumed) and the number of characters to be
written (-1 or free format assumed).   Characters   are
discarded from the left; the number is filled out on the
left with blanks. A sign is supplied if the number is
negative.   Corresponding operations for floating point
numbers are

        RIN(<file>[,<format>])

for floating input, and

ROUT(<expression>,<file>[,<format>])

for floating output. The format word is explained in R-21:
if it is omitted, it will be taken as zero, leading to free
format output.

6. Miscellaneous functions

FIX, FLOAT

Two functions are provided for converting between integer and floating point. To convert a floating point number to an integer by truncation, use

    FIX(<expression>).

To convert to an integer by rounding, use FIX(X+0.5). To convert an integer to floating point, use

    FLOAT(<expression>).

INITIALIZE

There are three argumentless special functions of general interest. INITIALIZE() initializes the QSPL storage allocator, taking all the space between the contents of BSTORG and the end of core for itself.

The compiler provides an INITIALIZE as the first instruction of the user's program. If the program starts somewhere other than at the beginning and does not begin with an INITIALIZE, the user may say CALL* INIT;U to DDT before starting the program. Failing this, there will be a disaster as soon as the program calls on any runtime feature.

HALT, EXIT

The other two functions are

    HALT(),

which halts, and

    EXIT(),

which does a BRS 10. The compiler generates a BRS 10 automatically at the end of every program.

BRS, SBRM, POP

To execute a BRS, do BRS(N,A,B,X) which sets up the A,
B, and X registers and does the BRS numbered N. Trailing
arguments may be omitted, and the adjacent commas may be
used if one of the registers does not need to be set up. If
the BRS is expected to skip, a failure location may be used,
which will be used if the BRS does not skip. The value of
BRS is the contents of A when the BRS returns: the
registers may be saved by a save list as for ordinary
function calls.

To execute a SBRM do SBRM(N,A, B, X). Conventions are
exactly the same as for BRS.

Arbitrary machine instructions may be generated with
POP(OP, N, A,B,X). This works like BRS and SBRM except that
the opcode is the value of OP, which must be a constant.
Thus, BRS(31,X) and POP(573B,31,X) are equivalent.

The Compiler

     A program is compiled with following set of tty
actions:

@QSPL.
7/14/69   [the date of the last QSPL assembly]
SOURCE FILE: <file name> <terminating character>
     [the terminating character is either "." - assume
     default file for BINARY and LISTING, or "," or ";" -
     demand further file names]
BINARY FILE: <file name> <terminating character>
     [default is NOTHING]
LISTING FILE: <file name> .   [default is NOTHING]

2 SEC 1 ERRS 243 CELLS   (A:n1,L:n2(n3),I:n4,n5,T:n6,M:n7)
@

     The "n ERRS" does not appear if there were no errors.
The numbers printed out in the compilation summary have the
following significance:

A:n1,   number of symbol table cells remaining at end of
          compilation. The symbol table has about 4900 cells.
          Symbols take 5 cells, constants take 4 cells.

L:n2   number of literals (constants) in the literal table.
          This includes constants actually written in the
          program, temporary and final values of constant
          expressions, and other compiler-generated constants
          such as field masks.

(n3),   number of literals which were never referenced in an
          instruction.

I:n4,   number of indirect cells generated for labels.

n5,   number of indirect cells generated for arrays and
          functions.

T:n6,   total number of arithmetic temporary cells
          generated.

M:n7   smallest amount of symbol table space encountered at
          any point during compilation. This number is smaller
          than the "A" number by roughly the number of tokens
          in the longest statement in the program. It is the
          best measure of how close you are to overflowing
          QSPL's symbol table.

The Runtime

A subsystem called QRUN contains the QSPL runtime.
When called, it puts the runtime code into page 7,
read-only, and initializes the pop transfer vector in page
0. It leaves ;F set after its storage and converts itself
into DDT. Dumps and continues may be performed exactly as
though it were DDT. The user may set up SARRAY to the first
unused cell before calling INIT; otherwise, SARRAY gets set
to 100k beyond the last non-zero word of the program.

APPENDIX A

Runtime Details

Strings

A QSPL string descriptor consists of four words, each
of which is a character pointer (3* word address + 0, 1 or
2).
They are:
    pointer to character before first character of space
      allocated to string.
    reader pointer for string.
    writer pointer for string.
    pointer to last character of space allocated to string.
ISD creates such a descriptor. RSD, RSR AND RSW set reader
and writer pointers. Characters are counted from 0. RCS
reads the characters between reader and writer pointer, WCS
writes characters between writer and end pointers. RCB
Reads characters between writer and reader pointers. WCB
writes characters between reader and beginning pointers. A
variable declared STRING must contain the **address** of a
descriptor when it is used in a string operation.

Paging Logic

A valid drum address has bit 3 off and bit 4 on; bits
0-2 are ignored and bits 5-23 comprise the actual virtual
address. CEA and CEI are used to translate such addresses
into core addresses into core addresses; if the desired page
is not in core, it is read in (which usually involves
writing out some other page). CEAS and CEIS do the same,
except that they also set a flag associated with the buffer
to ensure that the page will be rewritten on the drum before
a new one is brought into the buffer.

Core Storage Allocation

A block allocated by a (fixed) array declaration or by
a single- argument call of MAKE contains one more word than
was requested by the user. The extra word, which is the one

immediately preceding the zeroth word of the block, contains the total length of the block, including the extra word. The top two bits are used by the storage allocator:

bit 0 is on if the block is free.

bit 1 is on if the next lower block is free.
Blocks allocated by a two-argument call of MAKE do not have htis extra word.

An array being used for storage allocation (i.e. one set up by SETARRAY, or the SARRAY array) has the following form:

Word_____Contents

-1   Length + flag bits, see above

0   Bead size, or 0 for an array which allocates variable sized beads (or blocks).

1   Address of routine to call when free space is exhausted.   This   word   may   be   set   by   the programmer. The system does a CALL* through it

2   Pointer to master free list (or just to free list for arrays allocating fixed size blocks).

3   Free space to be allocated.

The free list for a fixed block size array starts at the second word of the array, is linked through the first word of each free block, and terminates with a zero.

The master free list for a variable block size array uses one block for each block size. Three words of this block are used.

-1   Length + flag bits.

0   Back-pointer. Terminates at 0th word of array.

1   Pointer to slave-fee list for this block size.

2   Pointer to next block on master free list.

The blocks on a slave-free list are all of the same size. Two words of each are used.

-1   Length + flag bits.

0   Back pointer on slave-free list.

1   Forward pointer on slave-free list, or 0.

The last entry on the master free list may be for block
size 2.   In this case the third word is not available, but
it is not needed, since the master free list is sorted by
decreasing block size, and the the smallest possible block
size is 2.

QSPL RUNTIME PCPS

* on mnemonic means that all central registers not used to
return results are destroyed.

+ on mnemonic means that all central registers are cleared.

Code	Mnemonic	Function
100	CALL	Function call. The definition is just BRU* 0. Thus F(A,B) compiles LDA A; LDB B; CALL* F.
101	+ NSC	Numeric to string conversion. (Q) = original integer, (A) = string description address, (B) = radix, (X) = number of characters to write (-1 means free format). CNS(A,S) compiles LDA S; LDB =10; LDX =-1; NSC A.
102	+ MSG	Print string starting at A on teletype with EBS 34. Not output by compiler.
103	+ FIO	Output integer to file. (A) = file number; (Q),(B),(X) as for NSC. IOUT(A,F,R,G) compiles LDA F; LDB R; LDX G; FIO A.
104	RERR	Runtime error. Q (not (Q)) is the error number.
105	* RCN	Read character, no motion. (Q) = string descriptor address. Reads the character following the one addressed by the descriptor to A. The descriptor is not changed. GC(S) compiles RCN S.
106	* RCS	Read character from string. (Q) = string descriptor address. Reads the character following the one addressed by the descriptor into A, increments the descriptor to point to the character. Skip if string is not empty. GCI(S:F) compiles RCS S; BRU* F.
107	* WCS	Write character (Q) on string (A). See RCS, but writes character from Q. Skip if space left in string. WCI(C,S:F) compiles LDA S; WCS C; BRU* F.

110   * RCB   Read character backwards. See RCS, but
                reads the character which would have been
                written by the last WCS. GCD(S:F) compiles
                RCB S; BRU* F.

111   * WCB   Write character backwards. See RCS but
                writes (Q) into the string so that it will
                be read by a following RCS. WCD(C,S:F)
                compiles LDA S; WCB C; BRU* F.

112   + R$D   Reset string descriptor. (Q) = string
                descriptor address, (A) = character number
                to set read pointer to, (B) = character
                number to set write pointer to. SETS(S,R,
                W) compiles LDA R; LDB W; RSD S.

113   * LNG   Length of string. (Q) = string descriptor
                address. Number of characters between base
                and write pointers (.ie. number of
                characters of useful information) returned
                in A. T+LENGTH(S) compiles LNG S; STA T.

114   + R$R   Reset string read pointer. Same as RSD for
                read pointer only. SETR(S,R) compiles LDA
                R; RSR S.

115   + R$W   Reset string write pointer. Same as RSR
                for write pointer. SETW(S,W) compiles LDA
                W; RSW S.\

116   + ESC   Establish string constant. (Q) as for ISD.
                The word after the ESC contains a character
                count, the following words the characters
                packed 3/word. The string descriptor is
                set to point to this string and control
                returns to the word following the last word
                of the string. S+"ABCD" compiles ESC S;
                DATA 4; ASC 2,ABCD.

117   CEA   Compute effective address for paged object.
                (Q) = drum address. Core address of object
                returned in X. A preserved, B destroyed.
                The validity of the core address is
                guaranteed only until the next paged
                storage POP. Use CEAS if object is to be
                modified. A+P.X compiles CEA P; LDA X,2;
                STA A.

120   CEI   Compute effective address, indexed. Same
                as CEA except that (A) is added to (0) to
                get drum address. Use CEIS if object is to
                be modified. A←P[I] compiles LDX I; CEI P;
                LDA 0,2; STA A.

121   CEAS   Compute effective address for above. Same
                as CEA, but for storing into object. P.X←A
                compiles LDA A; CEAS P; STA X,2.

122   CEIS   Compute effective address, indexed into
                array. P[I]←A compiles LDA A; LDX J; CEIS
                P; STA 0,2.

123   RCAL   Recursive call entry. Q (not (Q)) gives
                number of cells to allocate on stack for
                arguments and SAVEd variables. RECURSIVE
                FUNCTION F(X,Y), SAVE 2 compiles STA
                RECRG1; LDA 0; RCAL 3; LDA X; STA 2,2; LDA
                Y; STA 3,2; LDA Z; STA 4,2; LDA RECRG1; STA
                X; STB Y.

124   RRET   Recursive function exit. Removes block from
                stack and returns.

125   DBLX   Double X register. Used for floating point
                arrays. If A is a REAL ARRAY, then B←A[I]
                compiles LDX I; DBLX; LDF← A; STF B.

126   ROUT   Real output. (Q),(Q+1) is the number. (A) =
                file, (B) = format. ROUT(R,N,Q) compiles
                LDA N; LDB Q; ROUT R.

127   CRS   Convert real to string. (Q),(Q+1) is the
                number. (A) = string descriptor address,
                (B) = format. CRS(R,S,Q) compiles LDA S;
                LDB Q; CRS R.

APPENDIX B

Reserved Words

AND	FREE	OUTNAME
APPEND	FROM	PAGED
ARRAY	FSHIFT	PARAMETER
BCOPY	FUNCTION	PPLUSH
BGET	GC	PMAKE
BINARY	GCD	POP
BPUT	GCI	RCY
BRS	GOTO	REAL
BY	HALT	RECURSIVE
CIN	IDENT	RETURN
CLOSE	IF	RSH
CLOSALL	IIN	SAVE
CODE	INCLUDE	SERM
COUT	INFILE	SCOPY
CNS	INITIALIZE	SETARRAY
CRLF	INNAME	SETR
CSN	INTEGER	SETS
DECLARE	IOUT	SETUP
DO	LCY	SETW
ELSE	LENGTH	SOURCE
ELSEIF	LINK	SOUT
END	LIST	SRETURN
ENDFOR	LOCAL	STRING
ENDIF	LOCK	TO
ENTRY	LSH	UNLOCK
EOR	MACRO	WCD
EXIT	MAKE	WCI
EXTERNAL	MOD	WHERE
FIELD	NOLIST	WHILE
FIXED	NOT	WIN
FMASK	OR	WOUT
FOR	OUTFILE	
CRS	FLOAT	SCALAR
CSR	RIN	
FIX	ROUT	

APPENDIX C

Standard External Symbols

BSTORG   First word of storage available to INIT.

EARRAY   Last word not used for page buffers or tables
         after INIT.

ERROR   Error codes left here by INFILE and OUTFILE.

ESTORG   Last word of storage available to INIT.

FTYPE   File type left here by INFILE and OUTFILE.

NPB   Number of core buffers for pagins. 0 = all
         available space.

NPG   Desired size of drum address space(2↑NPL).
         NPG<=2↑(19-NPL). 0 = paging will not be
         used.

NPL   Page size as power of 2. 8<=NPL<=11.

NUP   Number of unlocked pages.

PADDR   Drum address of unlocked page.

PCAT   Category to be used by PMAKE. 0 = don't
         care.

PM   >=0: use NRH for paging logic. 4B7: use file
         /SOFDATA. 4B7+F: use file no. F. Any of
         above + 2B7: recover old state from file.

SARRAY   Address of second word not used for page
         buffers or tables after INIT.

APPENDIX D

Some of the

BNF Syntax of QSPL


```
    <expr> = <xwhr> $('& <xwhr>);

    <xwhr> = <xforx> [WHERE <xforx>];

   <xforx> = <xcond> $(FOR <xforc> / WHILE <xwhile>);

   <xforc> = <identifier> ('= / '+) <xcond> ([ ', <xcond>]
             WHILE <xcond> / [BY <xcond>] TO <xcond>);

  <xwhile> = <xcond>;

   <xcond> = <xor> [IF <xor> [ELSE <xcond>]];

     <xor> = <xand> $(OR <xand>);

    <xand> = <xnot> $((AND / EOR) <xnot>);

    <xnot> = [NOT] <xrel>;

    <xrel> = <xmod> [('= / '# / '> / '>= / '< / '<=) <xmod>];

    <xmod> = <xadd> $(MOD <xadd>);

    <xadd> = <xmul> $(('+ / '-) <xmul>);

    <xmul> = <xsign> $(('* / '/ / LSH / RSH / LCY / RCY)
             <xsign>);

   <xsign> = ['+ / '- / GOTO] <xtail> / (RETURN / SRETURN)
             <xsgn17>;

  <xsgn17> = [<xor> ', <xor> ', <xor> / <xor> ', <xor> / <xor>]
             [FROM <xtail>];

     <xfc> = <xtail> ['( <xfcn> ')];

    <xfcn> = [<expr> $(', <expr>)] [': [<failure>] ['
             [<identifier>] $(', [<identifier>])]];

 <failure> = <identifier> / (RETURN / SRETURN) [<xor>];

   <xtail> = <xref> $(('. / '$) <afield>) [<xsubs>] ['+ <xor>];

    <xref> = $('$) ['@] <xprim> <xsubs>;
```

```
    <xsubs> = $('[ <expr '});

    <xprim> = '( <expr> ') / <isc> / <xsf> / <identifier> /
              <constant>;

      <isc> = '" $<psch> '";

     <psch> = '& ('& / '' / '" / <letter> / <octal> <octal>
              <octal>) / <otherchar>;

  <constant> = <octal> $<octal> B [<digit>] / <digit> $<digit> [D
               [<digit>]] / <charcon>;

<identifier> = <letter> $(<letter> / <digit>);

  <charcon> =  '' <psch> [<psch> [<psch>]] '';

    <octal> = '0 / '1 / '2 / '3 / '4 / '5 / '6 / '7;

    <digit> = <octal> / '8 / '9;

   <letter> = 'A / ... / 'Z;

<otherchar> = <any character other than '&, '', or '">;

     <stat> = (DECLARE <xdec> / RECURSIVE <function> <xfcr> /
              <function> <xfdn> / LIST  <listop> / NOLIST
              <nolistop> / END / IDENT <identifier> / FOR
              <xforc> DO / WHILE <xwhile> DO / ELSE DO /
              ENDIF / INCLUDE <isc> / <expr>) '; / ['$]
              <identifier> ': ;

 <function> = FUNCTION / ENTRY;

   <listop> = SOURCE / BINARY / CODE;

 <nolistop> = <listop> / FREE / EXTERNAL / LOCAL / PAGED /
              INCLUDE;

     <xfdn> = <fdec> [', LINK <addr>];

     <xfdr> =  <fdec> [', SAVE <vblist>];

     <fdec> = ['$] <identifier> '( [<vblist>] ');

   <vblist> = <grwfd> $(', <grwfd>);

    <grwfd> =  $(INTEGER / STRING / ARRAY / PAGED / REAL)
              <identifier>;

    <token> = <isc> / <identifier> / <constant> / <pmark>;
```

<pmark> = <any of: "#S&()*+,-.:<=>@[]+">;

<xmd> = <identifier> ['([<identifier> $(', <identifier>)]
         ')] '+ S<token>;

<fielddec> = <identifier> '(<icon> [': <icon> ', <icon>] ');

<paradec> = ['S] <identifier> '+ <icon>;

<icon> = <expr which evaluates to a constant>;

<Xdec> =   . . .