PMcJ

QSPL REFERENCE MANUAL

L. P. Deutsch

B. W. Lampson

University of California, Berkeley

Document No. R-28

Issued June 12, 1967

Revised March 1, 1968

Contract SD-185

Office of Secretary of Defense

Advanced Research Projects Agency

Washington, D. C.   20325

Everything described in this manual was implemented on March 10, 1968.

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 semi-colons. 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 semi-colon 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 semi-colon or another comment) with an * and ending with a carriage return (not ;).
6. A FOR statement.
7. An IF statement.
8. An expression.

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

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 successive evaluation. The value of the result is value of the last expression in the string. Thus a+b & c+d; or more plausibly f(a,b) & g(1,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 y←14;

FOR takes the form <expression> FOR <for clause>. 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 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=16 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
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 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
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
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 evaluates its operand. It leaves the value in A and returns 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.

(),
(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)(x,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 follow 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) compiles LDA y;ADD=5, CAB; LDA a; LDX z; CALL* f. See below for a discussion of function declarations. The function expects control to be returned to the following location with the value

of the function in A. 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 operator. 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 X 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 the index bit on. For any other expression, the [] operator is equivalent to $ +. Thus (a-b)[c OR d]+1 compiles

```
LDA a; SUB b; STA t; LDA c; MRG d; ADD t; CAX; LDA 0,2; ADD=1;
```

Primaries

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

A number 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 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.

There are about 80 reserved words (see Appendix B) which may not be used as names.

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.

A string constant has the form "<any number of pseudo-characters>". It is legal only in the context <string name> ← <string constant>. A descriptor will be created which points

to the constant string. If the value of the name is 0, space will be allocated for the descriptor. Writing into the string will alter the constant.

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

1. 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.
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 (-1)$F. F must not cross word boundaries.
4. F(expression) has the value of T after the statements

   T←0;   T$F←expression

   have been executed. F must not cross word boundaries.

Any expression involving operators of precedence higher than IF 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 DECLARE is DECLARE [FIXED or PAGED] [INTEGER 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 ARRAY is present, a name may be followed by an expression in parentheses (or brackets). Thus

```
        ARRAY    A[12], B[X↑2+14]
```

If FIXED is absent, this construct makes the DECLARE an <u>executable</u> statement; every time it is executed, the expression will be evaluated and that many cells assigned for the array. The base address of the region assigned, with the index bit set, will be stored in the name. Any previous storage assigned to the name will not be released automatically. The programmer must release it explicitly, if he wants to, with the FREE function. The system does not check to see that an array declaration is executed before the array is referenced, or that the program does not store other things into the array name. If either one of these things happens, a mess will probably result unless the programmer knows what he is doing. 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).

If a name is declared with FIXED ARRAY, thus:

```
        DECLARE FIXED ARRAY A[20], B[10];
```

this causes the number of words specified to be allocated by the <u>compiler</u> and the location addressed by the name to be initialized to the address of the block allocated with the index bit on. This declaration is equivalent to

```
        DECLARE ARRAY A[20], B[10];
```

except that it is not an executable statement but is done once and for all by the compiler.

A name on 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), space will be allocated for it on the drum when the declaration is executed, and all references to it thereafter 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.

c. No arithmetic is done on the address. A construct like (P+2).X is not legal if P is paged. It will result in P being treated as though it were not paged.

Declarations of fields are not affected by PAGED. Indirection ($) should not be used on a PAGED pointer.

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 string descriptors. The address of such a descriptor must be put into the string variable before 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 in 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.

An integer may be <u>initialized</u> 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 <u>not</u> 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 initialized as indicated. The remaining four elements are initialized to 0.

A 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.

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 sub-fields 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) ≡ $(ptr+1) ≡ ptr.b.

The declaration

DECLARE PARAMETER c1←1,c2←2,c3←3; makes the names c1,c2, c3 equivalent in <u>all</u> <u>ways</u> 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. May be redeclared

The declaration

DECLARE INTEGER Q=R, S=T[3] is legal only if R has already been declared and 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].

A function is defined by

FUNCTION name(arglist);

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 f(a, ARRAY b, STRING c) would compile STA a; CBA; MRG = 2B7; STA b; STX c; LDX 0; STX Xf;

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;

No explicit provision is made for recursive functions. However, since the return link is available, the programmer can save and restore it himself. If a function call appears in a complex 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.

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.

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

Control Statements

The construction

```
IF expression DO;
  :
ELSEIF expression DO;  }
  :                    } repeat 0 or more times
ELSE DO;               }
  :                    } optional
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 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 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>, <expression> WHILE <expression>
```

which is the same as the first form except that the first expression is used for the first assignment, the second one thereafter;

```
<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 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
I ←<expression> BY -1 to 0
```

are recognized and compiled more efficiently.

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 result in an unloadable result.

A program should be terminated by an END statement.

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(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 be 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); in this case no prefix word is created. 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.

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.

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.

To copy one block of storage into another one of equal size, use BCOPY(expression,expression). The first expression is a pointer to the source, the second to the destination. These must be pointers acquired by 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.

2. Paging facilities and functions

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 all 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 access machinery, BRSs 124-127, is to be used for storing paged data, or a negative number if a random file called /$QPDATA 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.

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 X 2^{NPL} cells just below (ESTORG) -2^{NPL} and BPUT and BGET may only be used on addresses 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 an INIT, while BSTORG and ESTORG bracket the core available to it before an INIT.

3. String handling functions

A string is described by a four word descriptor which specifies the beginning and end of the area assigned to the string, the reader pointer, and the writer pointer. The function SETUP(string name, size) will obtain a block of the specified size and set up the descriptor pointed to by the string name to point to that block. If the name contains 0, a descriptor will also be created. 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 invoked (except maybe to create the descriptor); it is the programmer's responsibility to ensure that the proper amount of space is in fact available.

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 pointer 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.

To get the next character from a 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, expression) which evaluates the specified expression on failure. Often it will be a GOTO, but it need not be. This convention is also used for the next four functions. GCD(name) reads a character from the end of the string and decrement 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, and fails if there is not 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.

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 can be used, since string descriptors are just 4 word blocks: BCOPY(b,a). To copy the string, use SCOPY(b,a). a will be initialized first as though SETS(a,0,0) had been executed.

To convert a string S to a number, write CSN(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.

4. File-naming functions

A file is opened for input with INFILE(string name, expression); the string contains the full name of the file. This function requires the presence of an expression which is evaluated in case of failure. Its value is the file number. OUTFILE(name, expression[,expression]) does the same thing for output. The second expression is the option word which BRS 16 takes in A. 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 word returned by the BRS is in location ERROR.

To acquire file names, use INNAME(name, expression) and OUTNAME(name, expression), both of which collect the name from the teletype and write it on the end of the string supplied.

Both evaluate the expression in the event of failure, and have the terminating character as value.

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

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 string, 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.

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.

6. Miscellaneous functions

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 GO command automatically sets up BSTORG to point 100B cells beyond the end of the program. If you want some space for patches or whatever it is all right to increase it.

Since the GO command does not call INITIALIZE, the compiler provides an INITIALIZE as the first instruction of the user's

<LOAD name, name,...,name.
loads the specified files with DDT after installing the QSPL runtime first. The files should be ordinary legal DDT binaries; they need not have been produced by QSPL. When the last file has been loaded, the remainder of core is automatically assigned to the invisible storage allocator array called SARRAY. It should *not* be used by the programmer. Note that QSPL binaries can be loaded by an independent DDT if desired. If they use no runtime features they will run without difficulty. Alternatively, the runtime can be supplied manually. Appendix A explains how to do this. Note that runtime features are invoked by every built-in function except CIN, COUT, WIN, WOUT, CLOSE and CLOSALL, by the use of strings for arithmetic, and by array declarations. Except for these features, only the call pop need be supplied. It is the first one; a BRU* 0 in location 100 will suffice.

<GO
transfers control to DDT. If the program does not call INITIALIZE, *do not forget to do* INIT; U before running it.

<EDIT file name
transfers control to QED after reading in the specified file. Thus, an edit, compile and load sequence can be achieved without ever leaving the shelter of the QSPL command language.

The compiler contains a number of internal tables whose overflow is not checked for. These tables have been allocated rather generously, but could be overwhelmed by an excessively grandiose statement. To avoid such a disaster, it would be wise to limit the length of statements to 2 or 3 lines.

APPENDIX A
Runtime Details

At the end of this appendix is a complete list of the runtime pops: opcodes, mnemonics, and calling sequences. The body of the appendix is devoted to a description of QSPL conventions for strings, core allocation, and drum allocation. Note that programs which do not use:

strings

any special functions other than CIN, COUT, WIN, WOUT, CLOSE, CLOSALL

declarations of non-fixed arrays or PAGED quantities

can run without any of the runtime except the CALL pop, which is opcode 100. Putting BRU* 0 into location 100 will take care of it.

To load the QSPL runtime with an independent DDT, rather than with the LOAD command in QSPL, simply load the file ()QRUN with ;T like any other binary file. Before running the program, put into the cell SARRAY (declared external in the runtime) 1+ the address of the first available cell of core, into ESTORG the last available cell of core. When the INIT pop is executed, the QSPL allocator will take over all of core between (SARRAY) and (ESTORG). If the program does not call the INITIALIZE function, be sure to do INIT;U before starting it up if you make any use of the storage allocator.

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; 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 (non-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 FIXED ARRAY declaration or a two-argument call of MAKE do not have this 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 sized 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-free 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 smallest possible block size is 2.

The situation is illustrated in Figure 1.

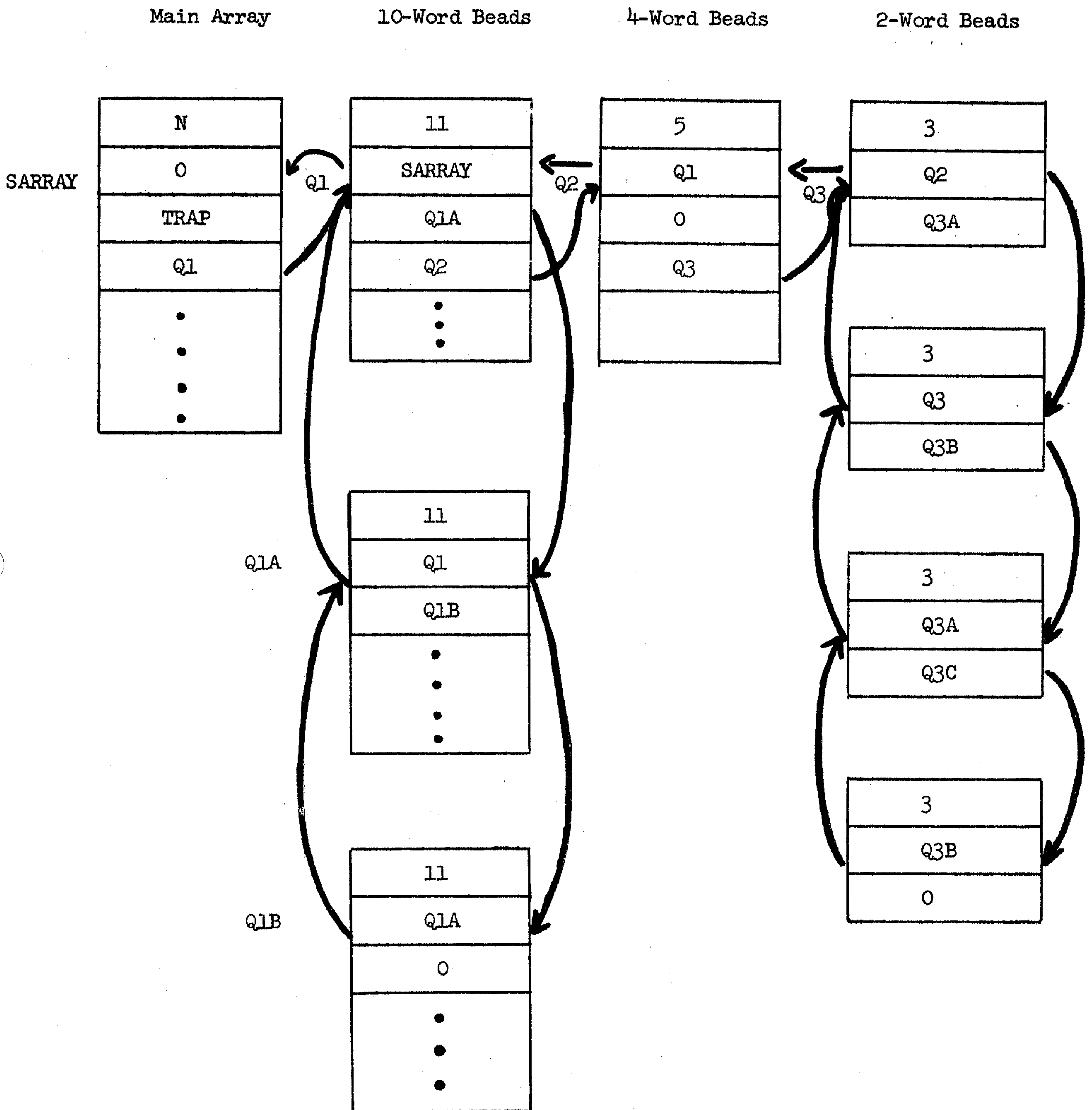

Figure 1: Pointer structure for an array allocating variable size blocks. The top row is the master free list, the columns slave free lists.

QSPL RUNTIME POPS

* 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) complies LDA A; LDB B; CALL* F
101	*NSC	Numeric to string conversion. (A) = original integer, (Q) = string description address. CNS(A,S) compiles LDA A; NSC S
102	+MSG	Print string starting at Q on teletype with BRS 34. Not output by compiler
103	+FIO	Output integer to file. (Q) = file number, (A) = signed integer, (B) = radix, (X) = number of characters to output (-1 means free format) IOUT(A,F,R,G) compiles LDA A; LDB R; LDX G; FIO F
104	*FII	Integer input to A, terminating character to B. (Q) = file number, (A) = radix. A← IIN(F,R,) compiles LDB R; FII F; STA A.
105	*SGO	String output. (Q) = string descriptor address, (A) = file number. SOUT(S,F) compiles LDA F; SGO S
106	*AIF	Accept input file name. (Q) = string descriptor address. The file name is written on the end of the string. Return terminating character to A. No skip if name not recognized. T← INNAME(S,GOTO F) compiles AIF S; BRU* F; STA T
107	*AOF	Accept output file name. See AIF. T← OUTNAME(S,GOTO F) compiles AOF S; BRU* F; STA T
110	*OIF	Open input file. (Q) = string descriptor address. The string should contain the name. Return file number to A, type to cell FTYPE. No skip if file cannot be opened, error code to cell ERROR N← INFILE(S, GOTO F) compiles OIF S; BUR* F; STA N.
111	*OOF	Open output file. See OIF, and (A) = option word for BRS 16 (See R-21). N← OUTFILE(S,GOTO F, Z) compiles LDA Z; OOF S; BRU* F; STA N.

112	*CPY	(Q) = destination bead address, (A) = source bead address. Copies words from (A) to (A)+((A)-1)-1 inclusive into words from (Q) to (Q)+((A)-1)-1. COPY(A,B) compiles LDA A; CPY B
113	*SNC	String to numeric conversion. (Q) = string descriptor address. Converts the (possibly signed) decimal number on the front of the string to binary and leaves it in A. The string descriptor points to the first character after the number. A CSN(S) SNC S; STA A.
114	RERR	Runtime error. Q (<u>not</u> (Q)) is the error number.
115	*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
116	*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,GOTO F) compiles RCS S; BRU* F
117	*WCS	Write character from string. See RCS, but writes character from A. Skip if space left in string. WCI(C,S,GOTO F) compiles LDA C; WCS S; BRU* F
120	*RCB	Read character backwards. See RCS, but reads the character which would have been written by the last WCS. GCD(S,GOTO F) compiles RCB S; BRU* F.
121	*WCB	Write character backwards. See RCS but writes (A) into the string so that it will be read by a following RCS. WCP(C,S,GOTO F) compiles LDA C; WCB S; BRU* F
122	+ISD	Initialize string descriptor. (Q) = 0 (in which case allocate a four word block) or string descriptor address, (A) = address or 0 (in which case (B)+2/3 words will be allocated automatically), (B) = number of characters. Sets up string descriptor at location addressed by (Q) (allocating the necessary 4-word block and putting its address into Q if necessary) pointing to a(B) character string starting at (A). SETUP(S,Z,W) compiles LDA W; LDB Z; ISD S
123	+RSD	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

124	*LNG	Length of string. (Q) = string descriptor address. Number of Characters between read and write pointers (i.e. number of RCS operations which can be done without a no skip return) returned in A. T ← LENGTH(S) complies LNG S; STA T
125	+CPS	Copy string. (Q) = string descriptor address for destination, (X) for source. Copies source string to destruction string. Skip return if there is enough room. Source string is not altered. APPEND (A,B,GOTO F) compiles LDX A; CPS S; BRU* F
126	*AFB	Allocate fixed block. (Q) = address of array, (A)= block size. Allocates a block of size (A) from the array, which must previously have been initialized by IFB for blocks of this size. Error if the block size is wrong. Returns address of block in A. T ← MAKE(S,A) compiles LDA S; AFB A; STA T
127	+RFB	Release fixed block. (Q) = address of array, (A)= block address. Inverts AFB. FREE(T,A) compiles LDA T; RFB A
130	+IFB	Initialize fixed array. (Q) = address of array, (A) = block size. Sets up the specified array (lenght must be in (Q)-1) so that AFB and RFB can allocate and free blocks of the specified size. SETARRAY(S,A) compiles LDA S; IFB A.
131	+INIT	Initialize runtime.
132	*AVB	Allocate variable block. (Q) = address of array, (A) = block size. Allocates a block of specified size from the array, which must be set up properly. INIT sets up SARRAY, which is the only array the compiler will address with the pop. T ← MAKE(S) compiles LDA S; AVB SARRAY; STA T
133	+RVB	Release variable block. Same as RFB for SARRAY. Inverts AVB. FREE(T) compiles LDA T; RVB SARRAY.
134	CRLF	Generate (A) carriage returns and line feeds on file (Q). Clears A and B but preserves X. CRLF(N,F) compiles LDA N; CRLF F
135	+RSR	Reset string read pointer. Same as RSD for read pointer only. SETR(S,R) compiles LDA R; RSR S
136	+RSW	Reset string write pointer. Same as RSR for write pointer. SETW(S,W) compiles LDA W; RSW S
137	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.

140	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.
141	CEI	Compute effective address, indexed. Same as CEA except that (X) is added to (Q) to get drum address. Use CEIS if object is to be modified. A←P[I] compile LDX I; CEI P; LDA 0,2: STA A
142	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
143	CEIS	Compute effective address, indexed for store. Same as CEI, but for storing into array. P[I]←A compile LDA A; LDX J; CEIS P; STA 0,2.
144	*APB	Allocate paged block. (Q)=drum address near which block will be assigned if possible, (A) =block size. Address of block returned to A. If the block size is less than the page size-5, the block will lie entirely on one page; hence, the core address can be used directly as a base address to access all the words of the block. Otherwise, a separate CEA or CEI is required for each reference. A PMAKE(N,X) compiles LDA N; APB X; STA A
145	+MRF	Miscellaneous runtime functions. The effective address Q (<u>not</u> (Q)) determines the function. The following values of Q are currently in use:

1 (LOCK) — (A)=drum address. The page on which this address lies is brought into core if not already there, and a lock count on the page buffer is incremented, preventing the page from leaving core. The core address is retruned in A. X ← LOCK(Y) compiles LDA Y; MRF I; STA X

		2 (UNLOCK)	(A) =core address. The lock count on the buffer is decremented; the corresponding drum address is stored in PADDR and the old value of the lock count is returned in A. XEUNLOCK(Y) compiles LDA Y; MRF 2; STA X
		3 (BPUT)	(A) =core address. The page buffer is returned for use by the in-core storage allocator. All registers are cleared. BPUT(X) compiles LDA X; MRF 3
		4 (BGET)	(A) = core address. The page buffer is taken back from the in-core storage allocator for use by the paging logic. All registers are cleared. BGET(X) compiles LDA X; MRF 4
146	QBRS	This is exactly the same as the BRS SYSPOP, but also stores the final contents of the central registers in cells SYSA, SYSB, and SYSX. BRS(12,1,,-1) compiles LDA =1; LDX=-1; QBRS 12.	
147	QSBRM	This is exactly the same as the SBRM SYSPOP, but also stores the final contents of the central registers in cells SYSA, SYSB, and SYSX. A← SBRM(F,X) compiles LDA X; QSBRM F; STA A	

APPENDIX B

Reserved Words.

AND
APPEND
ARRAY
BCOPY
BGET
BINARY
BPUT
BRS
BY
CIN
CLOSE
CLOSALL
CODE
COUT
CNS
CRLF
CSN
DECLARE
DO
ELSE
END
ENTRY
EOR
EXIT
EXTERNAL
FIELD
FIXED
FOR
FREE
FROM
FUNCTION
GC
GCD
GCI
GOTO
HALT
IDENT
IF
IIN
INFILE
INITIALIZE
INNAME
INTEGER
IOUT
LCY
LENGTH
LINK
LIST
LOCAL
LOCK
LSH
MAKE
MOD
NOLIST
NOT
OR
OUTFILE
OUTNAME
PARAMETER
PMAKE
RCY
RETURN
RSH
SBRM
SCOPY
SETARRAY
SETR
SETS
SETUP
SETW
SOURCE
SOUT
STRING
THROUGH
THRU
TO
UNLOCK
WCD
WCI
WHERE
WHILE
WIN
WOUT

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 paging. 0 = all available space.
NPC	Number of categories for paging.
NPG	Desired size of drum address space/2^{NPL}. NPG $\leq 2^{19-NPL}$.
NPL	Page size as power of 2. $8 \leq$ 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 if paging logic uses NRH, < 0 if it uses the random file /$QPDATA
SARRAY	Address of second word not used for page buffers or tables after INIT.
SYSA	Saved A register after BRS or SBRM.
SYSB	Saved B register
SYSX	Saved X register