Question List Processing
Using the QLIST Utilities

Last Edit: 18 November 1985
Transcription: 26 February 2015

Notes:

The original document was prepared for Linear Designware, a company founded by James A. Kuzdrall in 1984 to market design synthesis technology to DARPA. There was no interest.

The code reflects a time when most computers used 8-bit 8080, Z80, 6502, and 6809 processors. Hard disks held 10 megabytes of data.

By:
James A. Kuzdrall
Intrel Service Company
Box 1247
Nashua, New Hampshire 03061

TEL: (603) 883-4815
FAX: (603) 883-4815
iscmail@intrel.com

Table of Contents

1.0 Question Lists

Many ISC programs require the user to answer a series or list of questions during operation. In fact, these prompting sessions are so common that procedures for presenting the questions and getting the answers are formally defined in the User Interface Standard. The Question List or QLIST utilities automatically generate a C language program which presents a question list to the user. The presentation conforms to all standards of the User Interface and the program implements all the user commands defined there.

QLIST questions cover all common data types: integer, long, floating point, and string. In addition, they can request dates, simple yes or no answers, or a choice from a table of selections. The presentation and input functions are not part of the QLIST program, however, but are found in the C Standard Library (usrint, usrlng, usrflt, usrstr, usrdate, yesno, and usrtab). The QLIST program provides the question sequencing, and, most important, the execution of user commands. These commands, mandated by the User Interface Standard, allow the user to review or change previous answers. They also allow skipping forward over previously answered questions and session escape. The QLIST program interprets the user command and gets to the proper question in the list.

The job of the QLIST program would be fairly simple if all question lists were simply sequential. In practice, however, the answer to a question may affect the next information needed. For example, a user may be asked if he wishes to buy corn, bread, or ice cream. If he chooses corn, the questioning proceeds with further choices such as canned, fresh, frozen, or popcorn. The bread choice may be followed by rye, whole wheat, white, or Syrian style. Some breads, but not all, may be available sliced or unsliced. The ice cream may come in 4 flavors, 3 sizes, and for here or to-go. After all this, the user may be asked how he wants to pay for the food, regardless of his choice.

The example illustrates questions which affect subsequent questioning; these are referred to as branching or branch point questions. Other common situations include questions that are repeated in loops and answers that require additional calculations to determine subsequent questioning. All these are handled by the QLIST programs. As you can see, a user's request to go back 7 questions may not be so easy when parallel or looped paths are involved.

The primary advantage of the QLIST concept is uniformity. By limiting inputs to standard data types and formats, the user always sees a familiar presentation, regardless of the application or computer system. The QLIST system adds to this a uniform interpretation and execution of standardized user commands. Consider the testing and verification necessary to assure that each of 67 possible user commands does the right thing at each question in a typical 30 question session. Consider also the hundreds of hours of dull programming which would be duplicated in every

application just to find the right question in response to each user command. We have better uses for our talents and resources. All things considered, it is surprising this system is not an industry standard!

Some other advantages of the QLIST system follow:

Assured compliance with the User Interface Standard.

Reduced code size. Both the QLIST runtime interpreter and the question codes are in very compact form. Surprisingly large programs can be run on common desktop systems.

Greatly simplified programming.

Quick question changes, deletions, and additions using the QLIST editor and compiler.

Flexible programming options allow creative adaptation to many environments and circumstances.

Prompts and help strings may be arbitrarily long. (C imposes a 250 byte limit per string.)

Easy language translation; suitable for non-programmer language specialists.

The QLIST system consist of 3 utility programs and one function called from the Standard Library. The programs are the QLISTE editor, the QLISTC compiler, and the QSUMMARY documentation generator. The library function is an interpreter or processor, qlistp().

1.1 What the QLIST Processor Does

Before starting the question sequence, the Question List Processor (QLISTP) first reads the question information from a disk file. This information is compactly encoded. The question file data is put in RAM using the dynamic memory allocator, malloc(). This memory is freed for other uses when the QLISTP finishes.

The questions are numbered from 0 upward but the first question asked is always 1. Question zero is reserved for session escape. For each question, the processor follows this procedure:

1. Update the question sequence list or trail.
2. Execute a programmer supplied pre-process function (optional).
3. Set the disk-help index, if any.
4. Determine the question type; call the correct library function.
5. Execute a programmer supplied post-process function (optional).

```
6. Determine the next question, perhaps based on a user command.
```

When the question sequence ends, QLISTP releases the memory it got from malloc() and returns the the calling program.

In the above process, QLISTP must store the answers where the program can get them after the questioning is completed. Addresses in the program are not known until the program is linked, however. Since the QLIST is prepared independently, the linked program must communicate the data addresses to QLISTP at runtime. This is done through an extern array of addresses initialized by the linker. The calling program passes the beginning address of the array to QLISTP, and the required addresses are found by index. The source code for this array is prepared by the QLIST compiler to assure that the indexes and data location addresses agree. The array also contains a pointer to the name of the disk file containing the question data; QLISTP is not specific to any particular QLIST.

1.2 When to Use QLISTs

All products marketed must conform to the User Interface Standard. To avoid exhaustive testing, the QLIST system is used for any program with three or more questions. If you need help adapting your program to the QLIST system, see J. Kuzdrall. Since the programming options make the QLIST adaptable to virtually any situation, exceptions (required in writing) will be rare.

1.3 User Commands

The user commands implemented by QLIST are: ^n, ^^, !, !!, @n, and @@. (See User Interface Standard.) If the range of the user command exceeds the QLIST, qlistp() returns an error with the errnr and errdat locations properly set. For example, if the user requests to go back 17 questions (^17) from the 10th question asked, qlistp() returns ERROR with errnr set to USRBAK and errdat set to 7.

It should be noted that the user moves through the questions by displacement relative to the current question. It is not practical to identify questions with specific numbers since there are many potential paths through the questions.

The following sections summarize the question sequence effects for each of the user commands listed above.

1.3.a USRBAK, ^n, Moving to a Previous Question

The QLISTP maintains a question sequence number as it goes through the question list. When the user requests a question 5 back from the current (^5), the sequence number is check to see if that question would be in this QLIST. If it is less than the sequence number, the user is given that question.

Getting to the question is not easy, however. One approach would be to maintain a sequence list rather than just the sequence number. Just jumping back to the question would be easy using the list, but could lead to problems. For example, a file may have been opened in question 17 (yes, this is possible), but was closed in question 23. Jumping from question 27 back to question 19 would find the file closed when it was supposed to be open.

To assure that the circumstances or environment are the same when a question is reviewed, QLISTP uses the automode to get to the desired question. The automode feature is built into the library functions for all data types (usrint, usrlng, etc). When the autoflg is greater than zero, the function accepts the default answer for the question (without prompting the user) and decrements the autoflg. This operation is invisible to the user. QLISTP starts at question 1 using this feature and automodes down the list to the desired question. Since the defaults are always the previous answers, this assures that the correct path is retraced without keeping a list.

The above discussion assumes the question requested is in the current QLIST. If a backward move is requested from question 1, qlistp() returns an error with errnr and errdat still set. From any other question, moves larger than the sequence number result in question 1. The reason for this is that leaving the current QLIST by accident then returning requires that the question file be reloaded (all the old answers are retained by the main program, of course). Since it is easy for the user to lose track of his position in a long question list, we give him a second chance to make sure he really wants a previous question.

1.3.b USRTOP, ^ ^, Moving to the Top of the Question List

The USRTOP command get the user to the top of the current structure. If in a loop, it goes to the top of the loop. If at the top of a loop or elsewhere in the question list, it goes to question 1. If at question 1, qlistp() returns an error with the errnr set to USRTOP.

When loops are within other loops, the outer loops are ignored. That is, once at the top of the inner loop, the next stop is question 1. In general, three entries of ^^ will get you to the first question in the program. If the QLISTS are nested or the programs are chained, one additional ^^ may be required.

1.3.c USRFOR, @n, Moving Forward in the Question List

QLISTP takes no action for forward moves requested by the user. The library usr functions simply set the autoflg appropriately which turns on the automode for the requested number of questions. In the automode, the default answer is taken for the question without prompting the user.

Since QLISTP takes no action, the user can automode through questions which he hasn't answered before. It is therefore important that the programmer provide reasonable default answers.

In some cases, it is not possible or practical to predetermine the default. The programmer can stop the automode, requiring the user to answer the question at least once by making the default answer out-of-range. (See User Interface Standard, C Library Standard.) For numerical inputs, the allowable answer range is usually less than the data type range. The default is simply set outside the allowable range. For tables and dates, defaults of zero will stop the automode. For strings, a null string, "", as the default will force an answer. For yesno, -1 or 2 always stops the automode.

1.3.d USRAUT, @@, the Automode

The USRAUT user command is equivalent to requesting a forward move of 16383. It is handled exactly the same as USRFOR discussed above.

If the QLIST completes its questions in the automode, it releases its question file memory and returns to the caller. The balance of the autoflg steps are handled by the caller.

1.3.e USREOF, !, User End-of-Loop/List/File

Interpretation of this command varies with application. If in a loop, it indicates that the user wishes to exit the current question loop. In other cases, it may indicate the current buffer should be stored on the disk. In a way, it is the programmer's wildcard user command.

Because of these variations, any special interpretations are supplied by the programmer's post-process option. The post-process function can intercept the user command, take appropriate action, and reset the command before QLISTP attempts to process it. If QLISTP finds the user command still set, it issues the message "Not allowed here" (NOTACC_) and repeats the question.

1.3.f USRESC, !!, User Session Escape

Question zero is always reserved for session escape. It presents a table with two standard choices, "Resume session", and "Exit". A third choice is allowed as an option when creating or editing a QLIST, "Save session before exit". The programmer must provide the function which saves the data.

If the user chooses to resume the session, he is put at question 1.

1.3.g USRHLP and USRTYP, ? and ??, Help

The help commands are handled completely by the usr library functions themselves with no assistance from QLISTP. The QLISTP does, however, set a disk-help index (hlpinx) at the programmer's option. See User Interface Standard.

2.0 Using the QLIST Editor

The Question List Editor, QLISTE, creates or modifies source files for question lists. These re-editable intermediate data files are also source files for the Question List Compiler, QLISTC, which makes the files actually used by the system.

The programmer gives each new program a unique name, hereafter referred to as <ROOT>. The editor files use a .QLD extension which stands for Question List Data. When an old file is edited, it is renamed with the backup extension, .QXD. For example, when <ROOT>.QLD is edited, the modified file becomes <ROOT>.QLD and the old file is renamed to <ROOT>.QXD. The file name <ROOT> is always in uppercase.

On entering QLISTE, the programmer is given 3 options: create a new QLIST, create a new QLIST based on an old one, or edit an existing QLIST. The second option allows the programmer to use sections of old QLISTs for new applications rather than reprogramming from scratch. The first two options result in a new file, <ROOT>.QLD, with the name given in this session. The third option creates a <ROOT>.QXD file with the previous data.

The editor is completely prompted, and all ISC user commands are valid. Since the current version doesn't use the QLISTP, however, some user command responses are simplified. When entering new questions ("New Question #, appears in the right margin) the commands ^^, ^n, @@, @n, and !! are all valid. The results, however, may seem unpredictable. Some of the question being counted are invisible, since they are not applicable to every question type. These are counted by the question sequencer, nevertheless.

The user commands in the edit mode (table of data presented on screen) access the questions based on question number. The command ^^ always gets q1, and @@

always gets the last question. The @5 command from q3 gets q8, etc. This differs from the table's "next question" option which follows the question linkage which is often not sequential.

Before trying to use the editor, the programmer should have some knowledge of the QLIST structures. The basic organization is discussed below. If special applications require additional information, see the QLIST.H header file, the Question List Processor IDS, an existing example, such as RAWPWR, or the source code.

The QLIST file used during program execution has three parts: the question list, the table pointers, and the text section. The question list consists of an array of structs, one for each questions. The struct is named QDATA and defined in the QLIST.H header file. The array of structs is supported by a separate section of pointers and a section of string data. Both the pointer section and string section sizes are variable and not related to the number of questions.

The QDATA structs maintain their constant size by using pointers to the text section for prompts and help messages. In the special case of a choice table, the QDATA pointer is to a NULL terminated array of pointer in the pointer section. Each of these pointers is to the text section. (See usrtab() documentation for table array format.) This system of pointers avoids restrictive and inefficient fixed length text fields in the QDATA structs.

The QDATA structs must also reference functions and data in the compiled program. To do this, an array of address is constructed by the programmer with the help of the QLIST utilities. These addresses are referenced by index in the QDATA struct. Since the array is limited to 256 elements, the indexes are char data types in the QDATA.

2.1 The Extern Address Array

The extern address array is the QLIST's link to the compiled program. It contains addresses of both functions and data structures. The array is always named with the QLIST <ROOT> name mentioned above. The array provides all the information about the program needed by qlistp() and is the only parameter passed to it:

```
qlistp(&root.[0]);
```

The array contains char pointers; all addresses are cast to char pointers for simplicity. The first entry in the array in a pointer to the QLIST name, that is, <ROOT>. This is used by QLISTP to construct the name of the file containing the question data. The second entry is initially set to NULL, but QLISTP will put the address of its question array here for use by the pre-process and post-process functions. The remaining entries in the array are supplied by the programmer.

When the QLISTE requires and external address, it indicates the type (C language type) required and requests a name. This is followed by a request for a comment. This information becomes the C code which initializes the extern address array:

```
(char *)&yourname,        /* q01, your comment */
```

All of the text except "yourname" and "your comment" is automatically supplied by the editor. When the QLIST is compiled, an auxiliary file, named <ROOT>.QLH, helps the programmer remember the data types and construct a list of declarations or definitions. For example:

```
  /* declarations for MFGRQ[] */
extern int    escans;
extern char   fname[];
extern int    clscmp();
extern int    opncmp();
extern IPAR   mode;
extern int    setbufs();
extern char   match[];
```

The file also contains the C statements to create and initialize the extern address array. An example follows:

```
  /* array of addresses needed by qlist */
static char  *MFGRQ[] = {
  "MFGRQ",
  NULL,
  (char *)&escans,                                  /* USRESC answer */
  (char *)&fname,                 /* q01 filename company database */
  (char *)&clscmp,                        /* q01 close company file */
  (char *)&opncmp,     /* q01 open file; announce new file; loadflg */
  (char *)&mode,                     /* q02 add, edit, delete, quit */
  (char *)&setbufs,          /* q03 initialize data buffers for add */
  (char *)&match                  /* q04 mfgr match string or code */
  };
```

QLISTP accesses this array by index, for example MFGRQ[6] to get the address of "mode". Only the index numbers are stored in the QLIST.

2.2 The QDATA Struct

The QDATA struct contains 14 character sized datum and two pointers. The struct definition copied from QLIST.H is as follows:

```
  /* data struct containing information for each question */
 typedef struct {
```

```
char seq;         /* sequential order questions were asked in */
char this;        /* index number of this block */
char prep;        /* index to address of preprocess program */
char type;        /* question type (int, float, table, etc) */
char valptr;      /* index of pointer to place to put answer */
char tsize;       /* size of text string for usrstr */
char list;        /* index to prompt array for usrtab */
char errflg;      /* error result (0= no error) from usr fcts */
char post;        /* index to post process function */
char branch;      /* branch flag; 0= none, val= nr of branches */
char next;        /* index number of next question (0= done) */
char xhelp;       /* non-zero is help index in help file */
char nxt1;        /* auxiliary jump target (question number) */
char nxt2;        /* auxiliary jump target (question number) */
char *prompt;     /* prompt string; table pointer for QTABX */
char *help;       /* help string */
} QDATA;
```

Each variable is briefly discussed below from the editing viewpoint. Use of the variables by the programmer is discussed under "QLIST Strategies" section which follows. The use of some variables by QLISTC and QLISTP is covered in "Question List Processor, Software Design".

2.2.a Variables for Internal Use

Two variables are used at runtime and do not show up during editing. They are seq and errflg.

The variable "this" is the block number; it can't be changed during editing. That would be equivalent to moving questions around in the list, a facility not provided. The editor displays the value of "this" as the question number.

The tsize variable is programmed indirectly when the size of the string data buffer is specified. It is not used for other data types. Likewise, the variable "list" is used only with only one data type, the extern table.

2.2.b Question Type, "type"

There are 9 question types to choose from:

```
Nul          No question asked.
Integer      Integer via usrint()
Long         Long integer via usrlng().
Float        Floating point via usrflt().
Date         Date via usrdate().
Yes/no       Yes or no answer via yesno().
```

Table	Table with heading and choice text maintained by QLIST. Answer via usrtab().
Ext Table	Table heading and choices are in an extern array managed by the host program. Answers obtained via usrtab().
String	Text string via usrstr().

Of the group, Nul, Table, and Ext Table bear more explanation. The Nul question gives the programmer the opportunity to insert a "just process" block in the question list. It usually contains a pre-process or post-process function that does some data crunching or file operation. No question is asked and the block is not counted for forward or backward moves via the user commands.

Table and Ext Table both use the usrtab() function, but give the programmer an option of specifying the table choices in the QLIST or allowing other functions to set them. For Table, the programmer supplies QLIST with a prompt and the text for each choice. It has two advantages: the choices are available to edit; and the text is kept on the disk rather than as part of the compiled program.

Ext Table is used less often. It specifies an address in the compiled program where the table choice array is. This address is in the address array and its index is kept in "list" of QDATA. Engineering programs use such lists for component choices gathered from component libraries. The text for the choices is made up in buffers by sprintf() at runtime in response to the user's specifications. Neither number of choices nor their text is known beforehand. Although it is possible to use a Table question with long dummy choices in this application, the Ext Table allows the use of malloc() buffers for more memory efficiency. The choices are usually set by a pre-process function.

2.2.c Where the Data Goes, "valptr"

This is the address where the usr function puts the answer obtained from the user. It is coded as an index in the extern address array. The editor prompts for an extern name of the correct data type (IPAR for integer, buffer for strings, etc).

2.2.d Pre- and Post-Process Functions, "prep" and "post"

The QLISTP will execute two functions in each question, at the option of the programmer. If the option is exercised, the editor prompts for the function name. The address of the function is put in the extern address array and its index is saved in the prep or post locations of the QDATA struct (a zero index means no function).

The prep function is executed before the question is asked. The post function is executed after the question is asked but before QLISTP processes any errors or user commands. Both functions are VOID, returning nothing. They receive 2 parameters from QLISTP, a pointer to the extern address array and a pointer to the current QDATA struct. The QDATA array base address is available at index 1 of the extern address array.

QLISTP and the functions communicate error status via errflg in QDATA. If the usr function returned an error, errflg will be set when the post-process function is called. The post-function may modify its process if an error occurred. It may also be programmed to intercept and handle certain errors or user command, in which case it clears errflg before returning. Similarly, the post-process function may signify an error by setting errflg (and errnr). The pre-process function normally does not set errors. If necessary, however, it can set an error and skip the usr function. Either function can cause an error-free exit of QLISTP by setting next to zero.

The QDATA pointer is available to the programmer to allow examination and changes of parameters in the current QDATA struct. Obviously, there is some risk in this.

2.2.e The Next Question Link, "next"

The variable "next" contains the QDATA array index of the next question to ask. It is often the next sequential question, but may have special values. A value of zero indicates the end of a question session; QLISTP returns to the caller. There may be several zero "next"s in a QLIST.

The "next" may also be a previous question. This causes a question loop. The programmer must provide a way for the user to terminate the loop and continue the QLIST in such cases.

2.2.f Branch Flag, "branch"

A non-zero value in "branch" indicates the next question index is calculated from the answer given by the user. Such questions are called "branch point" questions and they initiate a new question series for each possible answer. The editor suggests the branch option only for yesno and table type questions. The variable "branch" is equal to the number of independent question series or branches that follow.

If the branch option is taken, the editor next requires the programmer to program each branch. Each branch is a series of one or more questions. The series is terminated by end-of-series (next=0), a jump to a previous question, or a jump to a common continuation question which follows the branch point and all its branches. (See Question List Arrangement which

follows.) Other than these ending options, the questions in the branch series are no different than any others.

2.2.g The Question Prompt, "prompt"

This is one of two text pointers in each question. The prompt text must be specified for each question except Nul and Ext Table types. For Table type questions, the text for each choice is requested after the prompt (table heading) has been entered.

For new questions, the editor gives a default prompt "No prompt" and allows the programmer to edit it. Each line in the prompt can be up to 78 chars long. Additional lines may be added before or after any line of a multi-line prompt. The "prompt" pointer is passed to the usr functions.

2.2.h The Help Message, "help"

The "help" variable is a text pointer like "prompt". The editor produces a "No help" default for new questions which may be edited into a more meaningful single or multiline message.

The "help" pointer is passed to the usr functions. All data types except Nul require the help message.

2.2.i Disk Help Index, "xhelp"

All usr functions have facilities to get more help information from a disk file. To activate this facility, the extern FILE pointer, helpfp, and the extern integer, hlpinx, must be non-zero (See isc.h). The editor allows the programmer to set hlpinx for each question. If the file has not been opened elsewhere, however, only a User Manual reference (equal to xhelp) is given.

2.2.j Auxiliary Jump Target, "nxt1" and "nxt2"

These variables are actually question array indexes. They are updated by the editor if a question insertion or deletion causes a change in that question number.

"nxt1" and "nxt2" are not used by QLISTP, but are provided for use by post and prep functions. It is common in engineering programs to change the questioning sequence in response to a calculation rather than a user's answer. Three target indexes (next, nxt1, nxt2) are then available to the

function; editing changes in other parts of the QLIST do not require changes it the function's targets.

2.3 Question List Arrangements

It is best to conceptualize the question list as a string of boxes, one for each QDATA or question. The boxes have one exit point, usually linking it to the following box. The entry point, however, may have links from several other question boxes.

Figure 1a, page 16, shows a simple sequential question list. Part 1b of the figure shows a branch point question with two options, the first loops back to a previous question and the second continues along the series.

Questions are sequential as shown in 1a except for branch points. To make branch editing and decision calculations easy, the n questions following each n-branch branch point must be the first questions of their respective series; the remaining questions, if any, are elsewhere. It is also required that a "continuation" question always follows immediately. An n-branch question is followed by n+1 required questions.

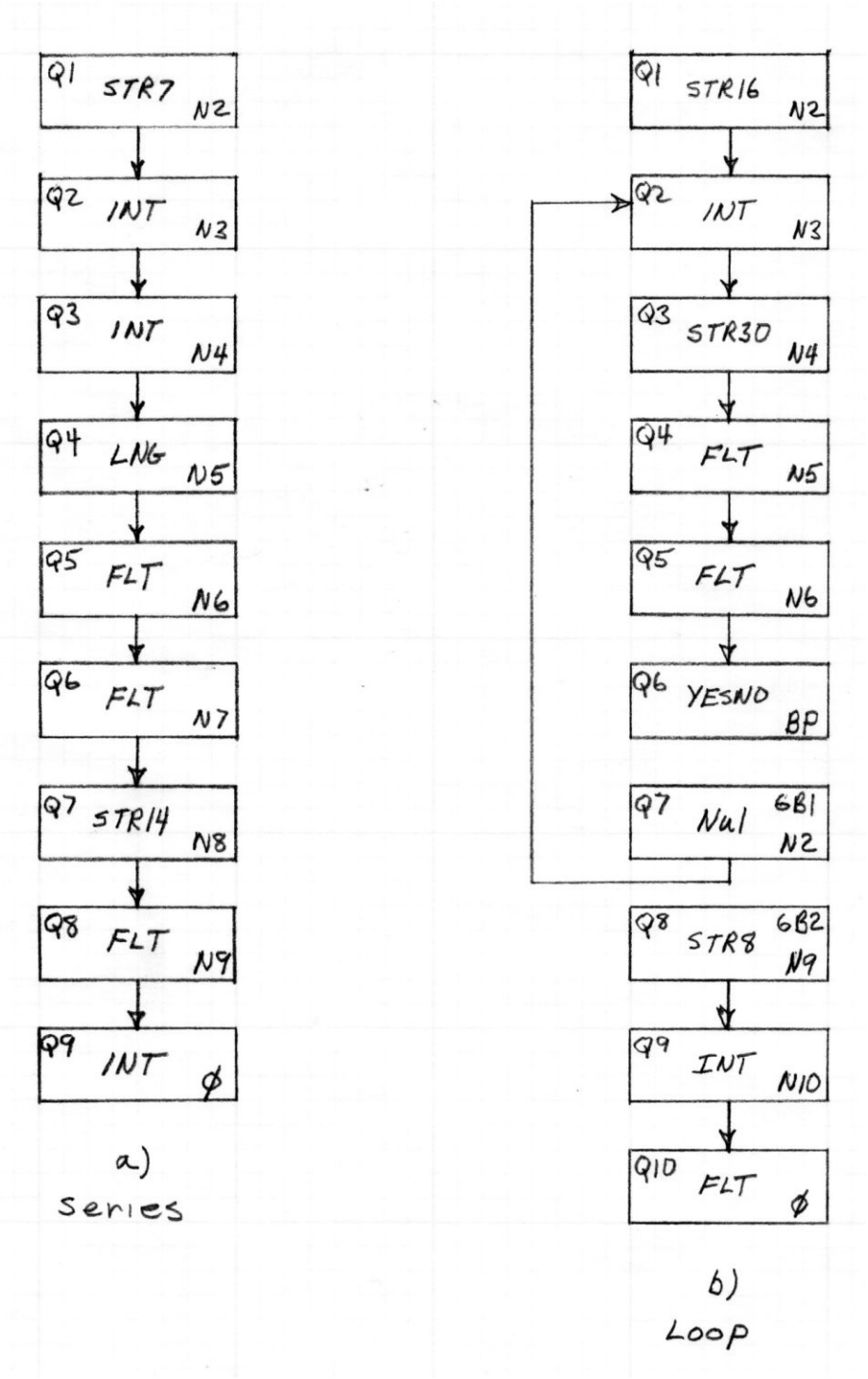

Figure 1: Qlist examples

Where do the additional questions go when a branch series is more than one question? Figure 2, page 18, illustrates the general rule. The additional series questions follow the continuation question in branch order (branch 1 first, branch n last).

A printing function, QSUMMARY gives a one page summary of each question, including its number. This is the easiest way to find out what the question sequence is.

2.4 Editing Functions

The QLISTE editor has two modes of operation, new question and edit. When programming a new question list, the new question mode is invoked. It keeps prompting for required information as new questions are created. When the new question sequence is ended, the edit mode is entered.

The edit mode shows a summary of each programmer selected question and offers an edit function menu. For old QLISTs, the edit mode is entered immediately. If new question series are added, back to the new question mode.

In addition to changing the variables in the QDATA struct, the QLISTE editor provides the following functions:

```
Add a question
Delete a question
Copy the current question to a Nul question
Add branches (choices) in tables
Delete branches (choices) in tables
Edit help and prompt text
Change extern variable names and comments
```

The editor provides a table of these editing options with the presentation of each question summary. The options are completely prompted and explain themselves. Data is not deleted without warning.

The following sections explain some seemingly strange responses to text editing, adding questions, and deleting questions.

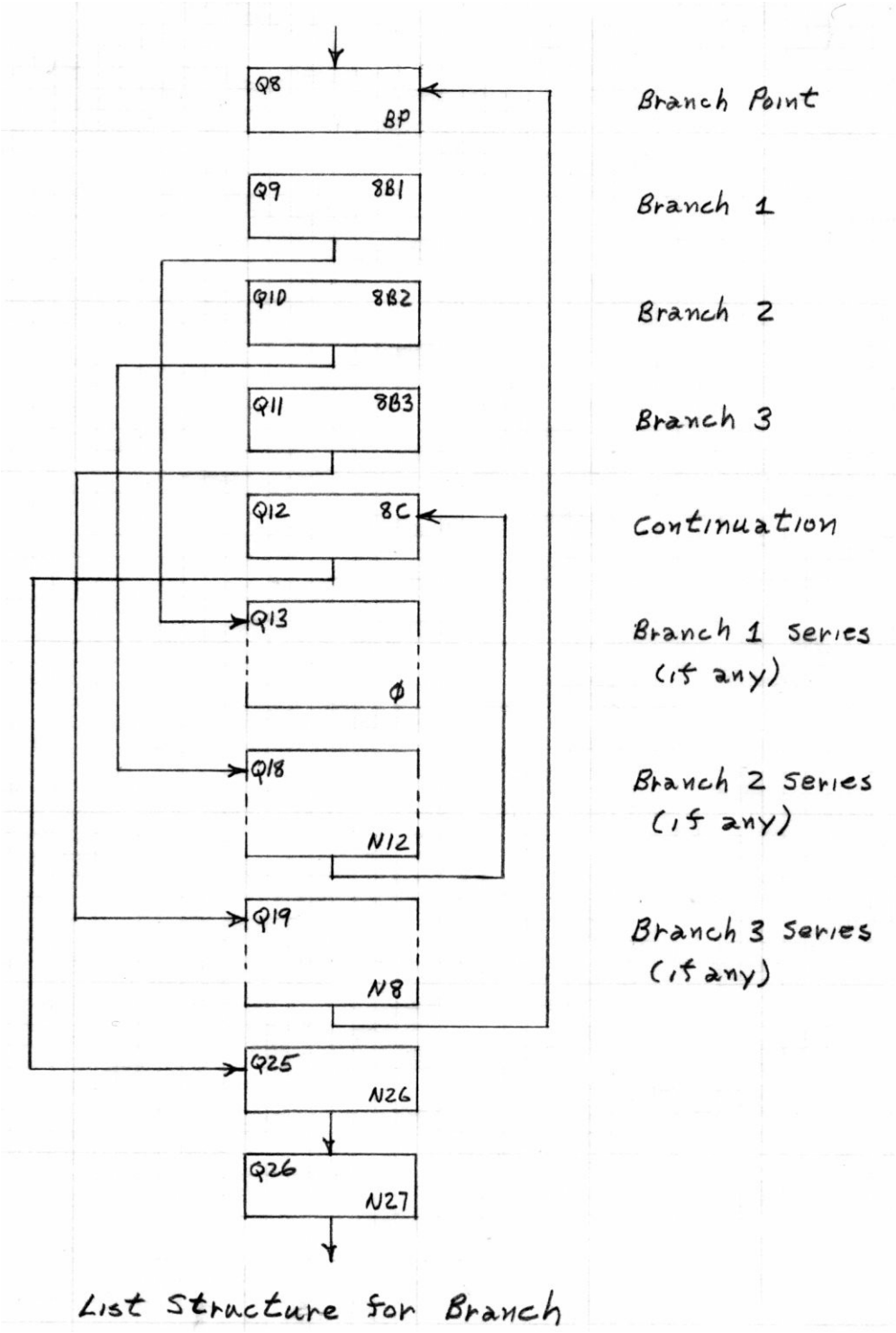

Figure 2: Branching Qlist example

2.4.a Editing Text

Multiline prompt and help text is a feature of the QLIST system. When entering the text edit facility, first the entire text is shown; this is followed by editing choices for each line. The lines are then presented one-at-time with options to change, not change, or delete. Also presented is the option to add a line before or after the current line. The programmer must pay close attention to the screen to determine which line is up for editing.

Choice tables also use a similar format, except each line is a choice. If the table question is a branching type, entering and deleting choices implies adding and deleting questions. This operation is fully prompted.

2.4.b Adding and Deleting Questions

Question in a simple sequence may be added and deleted without restriction. When a question is added, it is linked into the question chain. That is, the previous question's "next" becomes its next, and the previous question links to the new one.

When deleting in a simple sequence, the previous question links to the deleted question's "next".

For branching questions, things are not as simple. First, you cannot delete the branch point question or its continuation. To get rid of such a series, delete the branches one by one. When deleting one of two remaining branches, QLISTE changes the question to non-branching (with linkage to the remaining question).

Questions may be added "after" any of the branches, but they will not be put directly after. Likewise, questions may be added after the continuation question but these may not follow it directly either. In each case, the rules for building the question list structure are being followed.

Adding a question directly after a branch point is not permitted, but new choices may be added anywhere in the choice list.

2.4.c Copying Questions

The copy facility allows the programmer to move a question to a new spot in the list without reprogramming it. First, create a Nul question where the old question will be moved. Any Nul question already in place may also be used, but it must not contain either post or prep functions. Copy the old question to the Nul question, then delete the old question.

3.0 Using the QLIST Compiler

The QLIST Compiler, QLISTC, uses the question list source file, <ROOT>.QLD, produced by the editor. It produces two files for use by the program, <ROOT>.QLH and <ROOT>.QLP. The first is a header file, <ROOT>.QLH (Question List Header). It contains the extern address array used by qlistp(), and must be included in the C module calling qlistp(). The second is a binary file, <ROOT>.QLP (Question List Processor). It contains the question list array and text read into memory by qlistp(). This file is called from the system directory or drive of DOS.

3.1 The Question List Header File

The question list header file is formatted which the Isc CPROLOG and is almost ready to go. It contains declarations for all extern variables and functions referenced in the extern address array. These are only declarations! The programmer must still create the functions an data structures somewhere. The declaration list is used to assure the variables created are of the type expected in the QLIST.

The header file actually creates the extern address array. For this reason, this header should only be included in the C module actually using this QLIST. If it is used elsewhere, a redundant array will be created. As mentioned in the preceding section, The Extern Address Array, the array is defined as:

```
char  *<ROOT>[] = { };
```

where initializing addresses are included in the brackets.

3.2 The Question List Processor File

The .QLP file is a binary file using Isc standard data types, namely integers and characters. The file begins with three integers which give the size of the QDATA array in questions, the number of table choice pointers, and the number of bytes of text. The data follows in that order, QDATA structs, pointer arrays, and text.

It is not possible to put pointers in the disk file, of course, because the RAM storage address obtained by qlistp() is not known until runtime. Instead, all pointers are integer offsets into their respective sections. Note that the offsets into the table choice pointers must be in pointer size units since the size of a pointer in the target system is not known beforehand.

When qlistp() reads the .QLP file it converts all the integer offsets to real pointers by adding them to base addresses in RAM.

3.3 QLIST Summary Listing

The program QSUMMARY prints a listing of each question for documentation. On entry, the programmer can print all questions or only selected questions. The listing is rather long since it is printed with only one question per page. This, however, seems to be the best way to keep the documentation up-to-date without reprinting the whole list after each change.

4.0 Host Program Structure

There is no required host program structure, that is left to the imagination and cleverness of the programmer. The QLIST was designed with an environment in mind, however, and an explanation of it may trigger some ideas.

As mentioned before, the qlistp() function is called with one argument:

```
ques= qlistp( &<ROOT>[0] );
```

where <ROOT> is the extern address array produced by the header file. The variable ques is an integer. When finished, qlistp() returns the total number of questions processed or the constant ERROR (-1). If the returned value is not ERROR (>= zero), the processing ended normally (perhaps the autoflg is set, however). If ERROR is returned, an error occurred which stopped the question processing. The error may have resulted from a user command such as USRBAK or USRTOP, from a disk file error encountered in redirected input to a usr function, or from a prep- or post-process function. In any case, errnr contains the error code.

The questions-processed may seem like a strange value to return at first glance, but it is intended for environments where several QLISTs are processed in succession. If a USRBAK user command requests a question which is 5 back (5 from the end) of the previous QLIST, the number of total questions previously processed must be known to calculate how far to automode forward in that QLIST. That is, if 23 questions had been processed when the previous QLIST was executed, it would be re-entered with autoflg set to 23-5=18. This would get the user to the 5th question from the end of the QLIST.

4.1 Single QLIST Programs

Consider an example where a user must add a new record to an information file. This commonly occurs when new components are entered into libraries. Very often, the inputs are measurement data but the output to the library must be curve fit coefficients or some other processed result. In such cases, it is not practical to enter the information directly into the library. Instead, a

preprocess program prepares an input program to run with the library manager in the redirected input mode.

A simplified mainline for such a program might look like this:

```
main(argc,argv)
  int   argc;     /* parameter count */
  char  *argv[];  /* command line parameters */
  {
  int   err;      /* return from qlistp */

  /* get the new information; do calculations */
  do {
    if( (err= qlistp(&PWRRES[0])) == ERROR ) {
      if( errnr != USRBAK  &&  errnr != USRTOP )
        errxit()
      }
    } while( err == ERROR );

  /* add new data (in externs) to redirection file */
  if( filecat() )   /* standard file name PWRRES.INP */
    errxit();

  exit(0);
  }
```

In this example, the number of questions processed is of no particular use. It is, however, important to do something sensible if the user requests the top of the question list or requests to go back beyond the first question. In either case, qlistp() will return, not knowing it contains the first question. The solution is to simply return to qlistp(). Another solution would be to intercept these two user commands using a post function in question 1.

The file is written after the question session, taking advantage of the fact that all of the gathered data is still in externs. The filecat() function is assured that qlistp() finished normally by the error checking in the do-loop.

4.2 Programs with Multiple QLISTs

Why would multiple QLISTs be used? One reason might be that the session was too long for the 8K buffer limit or the 100 question limit of QLIST; this is not too likely, however. A more common reason is to group question session into subjects for easier maintenance. The engineering programs are a good example.

All markets need the design specifications; the commercial market needs additional information about production costs and yields; the military market need reliability and standards information in addition to both production and specifications data.

By using 3 QLISTs in the engineering programs, changes in the specifications questions need be made in only one QLIST not 3. Changes in production questions need be made in only one QLIST not 2. Clearly, such subdivision aids software and product management. The following example shows a simplified mainline using 5 QLISTs. Although the QLIST processor can move back and forth in its own prompts, there is no structure in it for moving to other groups of prompts. This structure allows several sessions with the qlist() to be inter-mixed with other types of processing. It requires that one array be defined, q[], and that the following program structure be used:

```
/* definitions */
#define  MAXLEV   4     /* number of levels, including default */
static int  q[MAXLEV];  /* array must be zero when prog begins */

/* initializations before entering loop */
level= 0;     /* level may be an automatic */

/* processing loop */
do {
  switch( level ) {
    case  0: q[0]= qlistp( &pro1 );    /* normal qlistp call */
             break;
    case  1: prep(arg1,arg2);         /* pre process before */
             q[1]= qlistp( &pro2 );    /* then call qlistp */
             break;
    case  2: q[2]= ques();            /* other question prog */
             post();                  /* some post processing */
             break;
    case  3: fct(arg3);               /* no questions asked */
             break;
    default: errchk();                /* process errors */
             save();                  /* save data perhaps */
             /* some exit routine */
             if( yesno("Again", &help1) == YES && !errnr )
               level= 0;
             else
               level= -1;             /* exit flag */
    }
  qcheck( &q, &level );
  } while( level > 0 );
}
```

4.3 Chaining with QLISTs

It is possible allow the user to traverse chained files containing QLISTs. An elegant system would consider each chained section as each QLIST was treated. That is, each would pass along the number of questions it and its predecessors had processed. The current program could chain back based on the size of the backward move requested, perhaps skipping some chain links.

A much simpler and entirely adequate way is to simply chain back to the preceding link regardless of the size of the backward travel requested. If the user is not satisfied when he gets there, he can enter USRBAK or USRTOP again.

5.0 QLIST Strategies

This section assumes the reader understands the normal operation of the QLIST. It covers techniques and tricks for unusual situations. Since all variations from normal processing are accomplished through the pre- and post-process functions, some general tips on these are given first.

When creating these functions, be mindful that they will be executed in the automode whenever the user command USRBAK (^n) is given. Try to avoid opening and closing files under automode conditions; they cause a long delay. In general, three conditions must be handled:

1. What is the appropriate action when autoflg is set.
2. What if the function's operation produces an error.
3. What should be done if errflg is set (post-process only).

Prep functions often return immediately if the autoflg is set. It is of no use to prepare for a question that won't be asked. The situation is different if a default is being reset. If the prep function does operate in the automode, make certain that any output to STDERR or STDOUT is suppressed. If a file is to be opened by the prep function, the programmer might arrange to always clear the file pointer when it is closed. This way, the function needn't open the file again if the pointer is non-zero. (Re-opening open files will keep using up memory with new file control blocks until a MEMERR occurs.) To save some checking, note that the fclose() function will "close" a NULL file pointer without error (it ignores it).

If the prep function produces an error in its processing, there are at least two possibilities for handling it:

1. Call errxit() and leave program (fatal error).
2. Set errflg; set errnr so that either the post function or the QLISTP handles it.

For example, assume that a file doesn't open in a prep functions. If the error is RDERRR there is not much choice but to errxit(). If the error is NOTFND, however, the function may execute errrpt(), set errflg, set errnr to USRBAK, and set errdat to 1. This would repeat the previous question which got the file name.

Post-process functions must always look at the errflg. If it is set, the data they intend to process is not valid. Post functions may also be used to intercept a user command for special handling. In such cases, they must clear errflg before returning.

Post functions are the best places to open user specified files. If the file doesn't open, the appropriate message is given (perhaps using errrpt()). To repeat the question, errflg can be set with errnr set to USRBAK and errdat set to zero. This causes and automode processing of all the preceding questions, however. A trickier way is to save the current value of "next" in a static the first time the post function is executed. If the error occurs, set qp->next to qp->this; if not, set qp->next to the saved value.

QLIST functions are programmed as any other C functions would be. They can do simple things such as initializing a variable or complex things involving many functions and perhaps other QLISTs (nesting QLISTs, while possible, defeats the memory efficiencies envisioned). The declarations for a QLIST function require that the QLIST.H header be included in the module:

```
int  postfct(addr,qp)
  char    *addr[];   /* extern address array */
  QDATA   *qp;       /* current QDATA struct */
  {
  QDATA   *q;        /* pointer to QLIST */

  q= addr[1];        /* initialize local array address */
  /* any code */
  }
```

Access to the entire QDATA array is obtained through addr[1] as shown in the beginning of the function. This may be necessary to determine the previous question (via seq) or other answers.

5.1 Must-Answer Questions

Sometimes it is not possible or practical to find an appropriate default answer for a question. An example might be where the user must supply his name or a file name. In such cases, the programmer can force an answer for the first time the question is asked by making the default answer out-of-range or, for strings, a null string "".

In rare situations, the default is never valid, even after one answer is given. These cases can be handled by a prep function which either puts the default out-of-range again or clears the autoflg. The user may be surprised, however, when he asks to go back one question and if forced to answer a question 10 back again.

5.2 Input Data Types

The variety of data types has been carefully selected and is adequate for the vast majority of applications. One unusual feature, however, is the use of answer structs: IPAR, LPAR, FPAR, and DATE. What if the answer location is a just-plain long integer?

First, the primary value of the answer structs is to assure the answers are in the range expected by the program. Any bypass of the answer struct risks losing this feature. If it must be done, however, the easiest way is to use a prep function which switches the address in the address array and a post function that switches it back. The new address is a properly initialized static IPAR common to both the prep and post functions. Once created, this function pair may be used in several questions where a similar situation exists. The procedure is:

```
Prep function:
  1. Initialize the static LPAR limits, if necessary.
  2. Copy the destination address to a local static (char *).
       adrtmp= addr[qp->valptr];
  3. Copy the LPAR's address into the address array:
       addr[qp->valptr]= (char *)&lngpar;

Post function:
  1. Restore the destination address:
       addr[qp->valptr]= adrtmp;
  2. Transfer the data:
       *(long *)adrtmp= lngpar.v;
```

5.3 Branches

Sometimes questioning must branch on the basis of a calculation rather than directly on an answer. In this case, the post function must do the branching operation itself by changing qp->next.

There are several points to remember in such operations. First, it might be important to save the value of the original next. It is more efficient to branch based on qp->this, however.

Maintaining the branch locations through editing can also be a problem. For normal branch types, QLISTE warns if an attempt is made to delete a branch, updates the branch count, and reduces the number of choices. QLISTE will not allow additional questions to be inserted between branches either (see Question List Arrangement, above). Such safeguards are not available to the do-it-yourselfer.

One method of checking the integrity of the branches is to record the first and last branch number in nxt1 and nxt2. If the separation is greater or less than expected, there is a problem.

5.4 Loops

Loops are created by making a previous question the next question. There may be loops within loops. A question may call itself next, making a one question loop. In all cases, the programmer must provide at least one way of breaking out of the loop.

A straight-forward exit technique is to ask the user "Want to do another?" via a branching yesno question. The default should always be reset to no by a prep function. The yes choice can be a Nul question which jumps to the top of the loop.

Particular care is necessary in preparing loops for possible automode execution. The exit question or condition must always default to exit. Otherwise, a USRAUT request from a previous question may get hung up in the loop for 16000 questions. If the answers collected from the loop questions are destined for a disk file or buffer, a save/don't-save choice must be given with the default always don't-save. Otherwise, a redundant record will be added to the buffer or disk every time the automode passes through the loop.

5.5 QDATA Struct Tricks

The QDATA struct can be cajoled into many unanticipated operations by a programmer who thoroughly understands its operation. There is, of course, always some risk in such deviations from normal processing. Some possibilities not discussed so far are:

Changing the data type (perhaps to Nul).

Making runtime changes in prompts or help messages.

Borrowing help messages from other questions. (The editor doesn't currently provide this feature).

Varying the size or starting point in a string buffer.

Force an error free exit by setting qp->next to zero.

Simulating user commands to move in the question list or exit the question list.

6.0 Copyright Notice