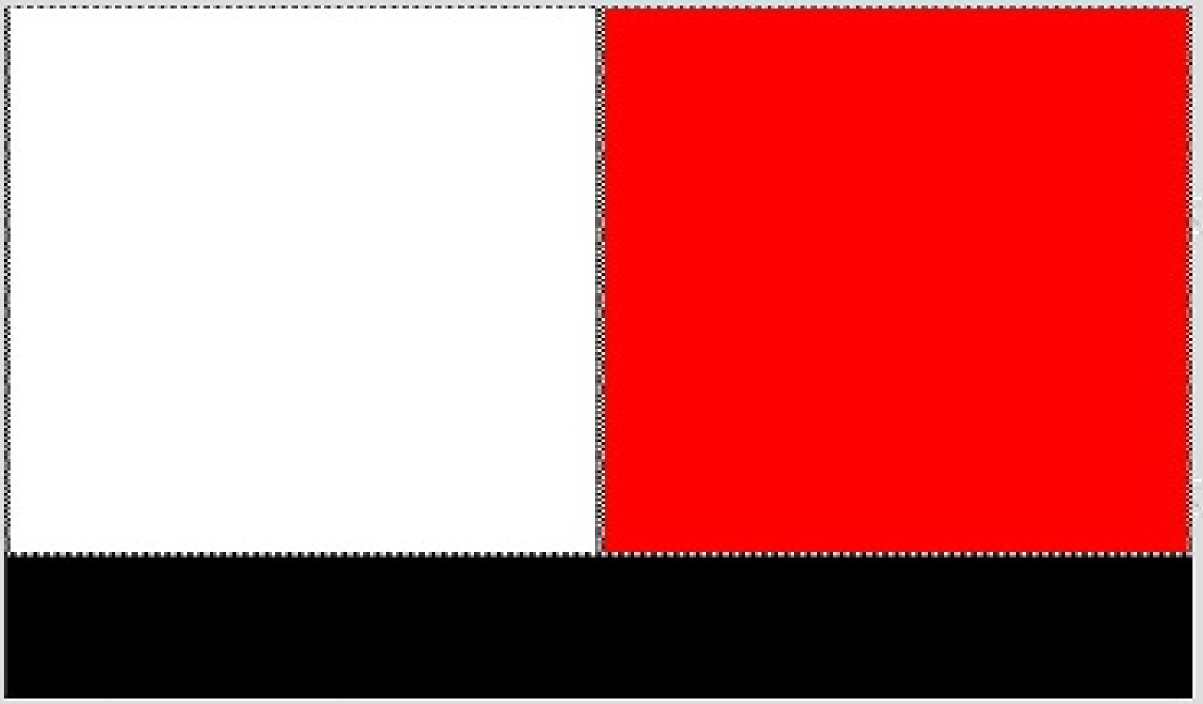
SuperBASIC Source Book



Timothy C. Swenson

SuperBASIC Source Book

(Version 1.0 Jan 1999)

By Timothy Swenson

TABLE OF CONTENTS

INTRODUCTION

This Source Book comes from wanting to know how to do different things while programming in SuperBASIC. This Source Book does not come from expertise (more of a lack of expertise) but from being willing to sit down and document what others and I have learned. I have not really done a lot of SuperBASIC programming, compared to many. A fairly big lazy streak, a house, and family make for a good excuse for not having produced a professional product. (Plus I have contributed to the QL community via the QL Hacker's Journal and articles to QL publications) So, I consider myself a relative beginner at programming really well on the QL, but I do have a drive to want to know how to program really well, and a drive to write down what I have learned. Some of this material may be old hat to some and completely new to others.

This document was originally going to be called the Qliberator Source Book, but since most of the information is not directly related to Qliberator, I decided to change the name.

This document is not meant to be the end-all in covering SuperBASIC and programming SuperBASIC. There are many books on SuperBASIC and on general programming. The focus of this document is on those topics that apply directly to SuperBASIC, the QL, Qliberator, programming tools and toolkits, and creating compiled, professional programs. For information on programming SuperBASIC in general, there are many other sources. Most QL publications have had some form of SuperBASIC tutorial and other programming articles. These publications would include QL World, Quanta, International QL Report (IQLR), QL Today, and many club newsletters. Another source of programming articles is the QL Hacker's Journal, a newsletter solely covering QL programming. All back issues of the QHJ are available electronically. See the section on RELATED DOCUMENTS for details on where to get these back issues and other information.

In covering any tools for creating or assisting in the creating of SuperBASIC program, there was a decision to cover only Freeware tools. As a fan of Freeware and as this document is Freeware, I wanted to only initially cover the Freeware tools. This was partially done to limit the initial scope of the document and because Freeware is easily had and used by programmers.

This document is to be considered a living document. As I have more time or more material become available, I will be adding to this document. Some sections may seem a bit thin on helpful advice on using the different toolkits. This is mostly due to my not having used the specific toolkits. As I experiment more with them, or as others pass along helpful advice, I'll add to those sections.

CREDITS

The biggest credit for this document goes to Dilwyn Jones, who has taken the time (many times) to answer many a question. Dilwyn's answers were always comprehensive. The section on using the DISPLAY toolkit comes directly from Dilwyn and is 100% his writing.

RELATED DOCUMENTS

One of my original ideas for this document was to troll around for other like documents and include them (limit the amount of stuff I have to write). Two key documents that I felt really should be included are either two long or stand well on their own.

Norman Dunbar has written a series of articles, a tutorial, for QUANTA on programming with EasyPTR II. These articles and accompanying example programs have been put into one single zip file and is publicly available via the Internet or through most QL Freeware sources. If you are thinking of using EasyPTR, you really need to get a hold of this document.

Dilwyn Jones has written a short tutorial on using QMENU for QL Today. It, too, has been made publicly available through the Internet or QL Freeware sources. It comes with example programs also. This tutorial is very helpful in figuring out how to use QMENU.

Norman Dunbar has also written an Introductory Guide (Idiot's Guide) to the Pointer Environment (PE). It was created to be distributed with all PE programs to guide users in learning all about the PE, its terms, and how to use it. This is another must-have document.

All of these documents, along with many others, are collected in my QL Documentation Project. The Project is an attempt to collect as many publicly available QL documents and make them available from one location. The focus is on documents that cover the operating system, key extensions, and tutorials. This does not include software/hardware reviews or time sensitive (news) information.

The documents from the Project can be found at my web site: http://www.oocities.org/svenqhj/

QLIBERATOR

Qliberator is written by Ian Stewart and the current version is 3.36.

Known Bugs

At this time there are only 3 known bugs with the 3.36 runtimes. They are:

1. ERNUM/ERLIN Bug.

ERNUM reports the line of the error (ERLIN) instead of the error number (ERNUM).

2. Passing Channels.

The Channel Passing example in the book (Page 10.4) does not seem to work with the Pointer Environment loaded. Cycling through the available jobs (CTRL-C) and back to the example program makes the example work.

3. Overlays.

Make sure all GLOBAL variables are correctly declared in the main file and the overlay(s). If one is missed, a system crash is the likely result.

Fixed Qlib Runtimes

A number of individuals have taken the 3.36 runtimes and applied their own fixes. I know of four different fixed Qlib runtimes. They were done by:

- Phil Borman
- Thierry Godefroy
- Hans-Peter Recktenwald
- Arvid Borestzen & Pal Monstad

Phil Borman's Fixes

1. Fix the well known ERNUM/ERLIN bug.
2. Add extra debugging messages on Qlib errors (e.g. "String is not numeric" shows you the string it was trying to coerce.
3. Add a file for fatal errors instead of the red error window. Built into the runtimes in a Config block, a fatal error causes the job to print its error information to a file and then force remove itself. This prevented fatal errors from stopping Phil's BBS. It comes with the PBOX Distribution.

Thierry Godefroy's Fixes

Thierry's version of the runtimes fixed only the ERNUM/ERLIN bug. It's file name is Qlrun336mod_lzh.

Hans-Peter Recktenwald's Fixes

Hans-Peter has not only fixed a few bugs but has added some additional functionality to the runtime. Due to my rusty High School German, I am not able to translate the document that comes with the runtimes. Below is a list of the keywords that have been modified or added:

```
Q_CMPLD   - New
Q_ERR
Q_JOB     - New
Q_RLIN    - New
Q_CURSOFF
Q_CURSON
Q_ERR_ON Name1$[,Name2$[,...]]
Q_ERR_ON Name1$,type1[,Name2$,type2[,...]]
Q_ERR_LIST
Q_ERR_OFF [Name1$ [,Name2$]...]
Q_PIPE
QR
Q_RSET[ernum%]
```

At the moment I do not have any information about the fourth modified Qlib runtime.

This version is only available from Hans-Peter (phpr@snafu.de)

Creating Background Hidden Jobs

I have experimented with creating background type jobs. These are jobs that do not display any information on the screen, but run in the background and do things. The Hotkey System is a form of background job. Some jobs have background modes, like Screen Snatcher, and "pop up" after a specific key stroke. The TKII command ALARM is a good example of a fully background job.

I wrote a short program to experiment with. I noticed that after I EXECed it, then CTRL-C'd to another job and back to QDOS, the screen that was present when I had EXECed it was saved and would appear as I cycled through the jobs. Well, this was not what I wanted. I puzzled over this until I ran across the page in the Qliberator manual covering the compile time options. The -winds option, when turned on, will automatically open a Channel #0, #1, and #2 for the job. With it is turned off, the job must open its own windows for output.

By default, Winds is turned on. This means that I had my own Channels open. When I EXECed the job, the Pointer Environment saw my Channels open up and saved the screen for me.

I then compiled the job with the winds turned off (un-highlighted in the QLib compiler window) and now when I EXEC it, the Pointer Environment does not save the current screen.

A job can run in the background, open up a window, display output, close the window, and still continue running in the background. When the window is closed, the Pointer Environment will not save the window, but let it disappear.

Reading the Keyboard Buffer

As a continuation of the discussion above, once you have a background job, you may want it to sit quietly in the background and wait for a certain keystroke, then pop up and do something. This means that you need to keep track of what keystrokes are happening.

One way of doing this is the use the KEYROW command. If you pick a keyboard combination that is on the same keyrow, say CTRL-V, then you can keep scanning the keyboard with KEYROW for that key sequence. Below is a simple program to demonstrate this:

```
10 x = KEYROW(7)
20 IF X = 20 THEN       : REM CTRL-V is value 20 on KEYROW 7
30  ....... do something
40  ....... more something
50 END IF
60 GOTO 10
```

Doing something could be a simple as a BEEP or as complicated as opening a window, outputting some data, and closing it again. The limitation doing it this way is that the keystrokes are not captured by your program, but are only looked at, and they get passed on to the underlying program. In this example, if the underlying program does not understand the CTRL-V, then everything is fine.

Another way is to look into the keyboard queue directly. This is done by PEEKing into the System Variables area of memory. Since the System Variables area can change location depending on what version of QDOS is running (JS, Minerva, SMSQ/E) you will have to be careful. There is a command in the DJToolKit called SYSTEM_VARIABLES that finds out what version of QDOS is being run and returns the appropriate location of the System Variables. If you do not have this command, you will have to do the figuring out yourself.

Here is a section of code taken from Screen Snatcher that looks into the keyboard auto repeat buffer, which is similar to the keyboard queue:

```
 REM System Variables = 163840
 REPeat loop
   value = PEEK_W(SYSTEM_VARIABLES + 138)
   IF value = snatch_key OR value = abort_key THEN EXIT loop
END REPeat loop
```

The PEEK_W returns the ASCII value of the key in the auto repeat buffer. In QDOS, all possible key sequences, such as ALT-SHIFT-A, has a specific ASCII value.

Stuffing the Keyboard Buffer

In a further continuation of the above discussion, now that we have a background job that can read the keyboard queue, what about adding characters to the keyboard queue, so that data can be sent to other programs?

One way to do this would be to POKE some data into the keyboard queue. There is a system variable that points to the keyboard queue. You could POKE the text you want into the keyboard queue. Not being an expert in low level QDOS, I do not know if this would work or is safe.

If you have the HotKey System II, you can use some HotKey commands to do exactly the same thing. You can add some text to the Stuffer Buffer by using the command:

```
HOT_STUFF "Some Text"
```

or

```
HOT_STUFF text$
```

Now the user can use ALT-SPACE to place the information in the underlying program (stuffing the keyboard buffer/queue). Or you can do this for the user by using the following command:

```
HOT_DO ' '    : REM that's a space character
```

This will do the ALT-SPACE key.

Another way of doing the same thing is to use the QSEND toolkit. It is discussed below.

Using STDOUT (Passing Channels to Child Job)

On page 10.3 of the Qliberator manual is a section entitled "Passing Channels to Jobs". This section explains how you can pass a job a set of channels (say an open window) and have it redirect its output from its own channels to the channels passed to it.

On the next page is an example program. After some experimentation it has been discovered that the example does not work with the Pointer Environment (at least with the more recent versions of it). The example does not totally fail, the user just needs to cycle through the running jobs (by using CTRL-C) and when it gets back to the job that passed the channels, the screen output is updated.

Where this would be useful is in creating programs to work well with C68 programs. A default behaviour of a C68 program is to see if it has been passed any input/output channels. If it has not, it will open its own Window for its input and output. If it has been passed channels it will use those. This is very useful when running MicroEmacs. From MicroEmacs I can execute a program (say unzip or mtools). Instead of the unzip or mtools opening up a new window, the output from it is put in the MicroEmacs window.

It would be nice to write compiled SuperBASIC programs that have this behaviour so that they could be run in Adrian Ives' SHELL along side with C68 programs. I've never been a great C programmer, but I would like to write programs that work well under SHELL.

Linking in Toolkits

One little hint that applies to using extensions from the SuperBASIC command line, but it may apply to using them in programming: When an extension is called it is started with an EXEC_W. This means that you can't CTRL-C out of it. If you use an exclamation mark (!) at the end of the name of the extension, then it will be started with an EXEC.

An example of this would be this: take the following program:

```
 REMark $$external
 DEFine PROCedure CAT
   DIR flp1_
   PAUSE
END DEFine CAT
```

Compile it with Qliberator, LRESPR it (or OVERLAY it), and run it from the command line by typing CAT. Now run it by typing CAT!. Kind of a neat trick and something nice to know. (Thanks to Ralf Reokoendt)

Creating Toolkits (Loadable Extensions)

One of the things that has always amazed me about the QL was the ability to load a binary file and have a bunch of new keyboards available in SuperBASIC. In most computers that had BASIC built in, the language was static and had no way to extend itself. Other languages (like C, FORTRAN, or Pascal) used libraries of functions and procedures to extend the capability of the library.

The first major loadable extension to the QL was Toolkit. From then on the term toolkit has been used in reference to loadable extensions. Popular toolkits are ToolKit II (TKII), DIY ToolKit, and DJToolKit.

I knew that the first of these toolkits were written in Assembly, but I did not know that they could be created by Qliberator. It seems that QDOS executables and extensions are real close in format and when compiled right, they can be interchangeable. This means that an executable can also be loaded as an extension.

To figure out how to create a toolkit, I grabbed a simple function, Qliberated it, and gave it a try. The function is included below:

```
10 REMark $$external
100 DEFine FuNction upper$(up$)
110     LOCal x, temp
120     FOR x = 1 TO LEN(up$)
130        temp = CODE(up$(x))
140        IF temp > 96 AND temp < 123 THEN up$(x)=CHR$(temp-32)
150     NEXT x
160     RETurn up$
170 END DEFine
```

The function takes any string and converts it to all upper case letters. The $$external is a compiler directive to Qliberator that tells it that the next function or procedure needs to be available outside of the executable. For each procedure or function that you want to turn into an extension, you would have to put the $$external directive in front of it.

If I was to put an additional line in the program,

```
180 PRINT upper$("This is a test")
```

then when I executed the program, line 180 would be executed. If I LRESPRed the program, the keyword upper$ would be made available.

When compiling the program it is a good idea to turn WINDS off, since the extension will have no channels open. Otherwise 3 channels will be opened for it, wasting them. To lower the size of the binary file, turning off NAMES and LINES might be a good idea. Note that the case of the function or procedure will be maintained in the extension. In my example, the name that will show up when entering EXTRAS is "upper$" (all lower case). If I defined the function a UPPER$, then "UPPER$" would show up in the EXTRAS command. By convention, extensions should be done in all upper case.

If you will be running the extension on a system that already has the Qlib runtimes loaded, then compile the program without runtimes. If you don't know if the Qlib runtimes will be available, compile it with the runtimes included. It is a good idea to compile both ways and let the user decide which one they need. The example program when compiled without the Qlib runtimes was 594 bytes. With the Qlib runtimes it was 11,146 bytes. The runtimes take up a fair bit of space.

If you load an extension that does not have the Qlib runtimes on a system where the Qlib runtimes are not loaded, you will not get an error message when you LRESPR the extension. When you call the extension is when the error will occur. The exact error message is:

```
  Error "Runtimes Missing !"
```

Once you have compiled an extension, all that is needed is to LREPSR it and test it out. Remember that you can't LRESPR while any jobs, other than Job 0 (SuperBASIC), are running.

Overlays

When I heard a well-known Qliberator user profess that he was not sure that an overlay was, I thought it would be a good idea to discuss it a bit.

We all know what an Extension is; it is a binary file LRESPR'd in a BOOT program that loads some new SuperBASIC keywords. This is a nice feature of QDOS, but LRESPR (or just RESPR) has one key problem - you can't LRESPR when there are any jobs running. So, if you start up QDOS, load a BOOT program that at the end starts the HotKey System II running, you can no longer LRESPR any extensions. If you want to use an extension, you have to load it an boot time or you have to kill all jobs to load it. So, most users had a tendency to load as many extensions as they can in at boot time. This can really take up bunch of memory, esp. if you only use the particular extension rarely. Another workaround is to create a second boot disk that you use only when you want to use a particular application.

The OVERLAY command is just like LRESPR but it can be used at any time. Any extension OVERLAYed by SuperBASIC can be used by all programs. If a program calls an OVERLAY then only it can use the extension. Only 16 overlays can be used and is limited to a particular process. This means that if SuperBASIC uses 2 overlays and a program uses 3 overlays, they don't count against each other.

Once an extension is OVERLAYed, it can be removed from memory when it is not longer needed by using the UNLOAD command. In this way, OVERLAYs are very similar to Dynamic Linked Libraries, in that they are called when needed and the released when they are done being used. This is all very memory efficient, but it does come at a small cost of the time it takes to load the extension.

The biggest problem with OVERLAYs is that only extension binaries created with Qliberator (version 3.2 or later) can be OVERLAYed. This means that if you want to OVERLAY an extension written directly in assembly, you can't. This is a serious limitation as most of the extensions available were not created using Qliberator.

Given this limitation I can see OVERLAYs being used mostly to load and unload your own extensions. If you compile a number of functions and procedures of a program in separate chunks, then these chunks can be loaded in as needed. If you are writing an especially large program that consists of distinct units, then it might be wise to load each unit as needed. This can allow the program to run on systems with limited memory.

CONFIGURING A PROGRAM

Despite whatever settings you define in your programs, at sometime the user is going to want to change them. Depending upon the nature of your program, the number and type of settings that you will let the user change will vary. Based on this and other programming considerations, how you implement configuring a program will also vary.

Types of Configuration

There are five different ways of configuring a program:

1. User Intervention.
2. Config Blocks.
3. Configuration File.
4. Command line Arguments.
5. Environment Variables.

User Intervention

This was probably one of the first ways of being able to configure a program. Most of the programs we use today have some sort of User Intervention configuration. For example, with Quill, I have menus that let me change the layout of the page. This includes how many lines per page, how many lines for the upper and lower margins, etc. When I start Quill, I have to change these settings as they are only saved if saved in a document.

A lot of times User Intervention does not allow for the savings of the settings.

Positives:

- Write as much help as necessary

Negatives:

- Lots of additional code to write

Config Blocks

Config Blocks are chunks of data appended to an executable that can be accessed by the executable and altered by a standard configuration program. Config blocks are unique to the QL world and are non-transportable. The biggest feature of using Config Blocks is that once a user knows how to use the 'config' program, they can configure almost any executable. There is a simple one-time learning curve. Config Blocks are good for programs that a user will configure once or twice to meet there needs. You would not want to use a Config Block for a program that needs to be changed fairly often.

Positives:

- Standard interface
- Easy to use
- Error checking done by Config

Negatives:

- Have to run a program to change

Configuration File

A Configuration File is probably the most flexible and powerful form of program configuration. A Configuration File can handle so many different configuration items. Some programs, like the editors DME and MicroEmacs, can totally change their behaviour through Configuration Files. The downside to Configuration Files is the code necessary to read in and check the Configuration File for errors (parse). This is a fair amount of code and will change from program to program.

Positives:

- Very flexible
- Good for lots of options

Negatives:

- Error checking can be hard to write (parsing)

Command Line Arguments

Command Line Arguments are probably the quickest and most transient of all forms of program configuration. They are added to a command line after the program name. They will only take effect for the one instance of the program. If a user needs to make a number of configuration changes frequently, then Command Line Arguments might be the way to go. A downside is that if you have more than one command line argument, you have to parse the different arguments to see which ones the user has typed in.

Positives:

- Quick configuration change

Negatives:

- Requires parsing
- Not good for lots of options

Environment Variables

Environment Variables are in that middle ground between the short term Command Line Argument and the more permanent Config Block or Configuration File. They will affect a program for more than one instance, but they can be changed between program instances. In other words, they hang around for as long as you need them, but can be easily changed. Setting Environment Variables in a BOOT program can make them seem permanent.

Positives:

- Quick to change
- Easy to use

Negatives:

- Bad for lots of options
- Requires ENV_BIN
- You do the error checking

Each of the options have their own place and their own benefits and faults. Some are more permanent, like Config Blocks and Config files, while some are very short lived, like User Intervention and Command line Arguments. Environment Variables are in between as they can be set in a BOOT program, but they can be changed by just typing in a new command.

Config Blocks

A Config Block is a chunk of data attached to executable files that have configurable items. The "config" program is used to change these configurable items in the program, without affecting the actual executable section of the code.

When writing a program, there are usually some parameters to the program that are hard-coded. Things like how big an array should be, how much memory to allocate, and so on. There are two ways of letting the user configure these parameters; writing a section of code to do it, or having a configuration file.

Writing a section of the program to allow the user to change various parameters is feasible, but it adds complexity to the program and it takes time. Having a configuration file may seem cleaner but the program would have to worry about the syntax of the file and how to handle errors.

Somewhere in the history of the QL someone came up with a third option, the Config Block. A Config Block is a part of an executable program that can be edited by "config". It is not part of the actual code of the program, but resides apart from it in a known location so that "config" can find it. The data in a Config Block is read by both "config" and the actual code of the program. The programmer writes the program so that configurable parameters are read or grabbed from the Config Block. The Config Block is created separately and linked to the program code to create the final executable.

As far as the operating system is concerned there is nothing special about a program with a Config Block. It is EXECed just like any other executable.

Definition of Config Block

Each Config Block is comprised of the following elements:

- Configuration ID
- Configuration Level
- Software Name
- Software Version
- Config Item(s)

The Config Block only has one Configuration ID, one Configuration Level, one Software Name, one Software Version, and one or more Config Items. The Configuration ID and Level are preset by the program that creates the Config Bock. The Software Name and Version is displayed when using "config" so that the user knows exactly what program they are configuring and can not be changed by the user.

The key part of the Config Block lies in the Config Items. This is the actual data that the user can change. There are seven different Config Item types:

- String
- Long Word
- Word
- Byte
- Select
- Code
- Char

A String is just that, a string. It can contain anything and is used for having default devices (FLP1_, WIN1_), default extensions (_DOC, _TXT, _EXE), or any other character-based default.

Long Word, Word, and Byte are three different Config Items for storing numbers of the three different sizes (8, 16, and 32 bit). The each have attributes of Min and Max values. The value of each item can be any number within the range of the Min and Max values.

Code is a single byte in length and is set to any number in a list of numbers. If you want only the number 10, 15, & 20 to be possible values for an item, Code is what you would use.

Select is the same as Code, but instead of the data being in the Config Block, the value is a pointer to the actual data.

Char is a single character. It is used when only one character is needed and not a whole string. One could use a String Config Item when ever you need a character, but you would have to do the checking yourself to make sure that only one character is used.

Each Config Item has the following Attributes:

- Selection Keystroke
- Description Text
- Default Value
- Option(s)

Selection Keystroke seems to have been designed to access the Configuration Block from within an application program. When using the "config" program, the Selection Keystroke has no meaning.

Description Text is what is printed to the user when "config" is changing that Config Item. You would enter a line or two telling the user what the Config Item is used for, what values are valid, suggested values, etc.

Default Value is the value that is originally assigned to the Config Item.

Options are possible limitations are the Config Items. For String there is an Option of maximum length of the string. For Long Word, Word, and Byte there are maximum and minimum limit Options. For Char there are Options upper or lower case, printable or non-printable, and cursor characters are allowed. Each Config Item has its own set of options based on its data type.

Config Levels

There are two different versions or levels of Config Blocks: Level 1 and Level 2, or just called Config 1 and Config 2. Config 1 was created first and is the most common Config Block. Config 2 is relatively newer and just starting to be used by software developers. Config 1 is modified by "config" and Config 2 is modified by "MenuConfig".

- Can MenuConfig edit a Config 1 Block?
- What happens when 'config' tries to edit a Config 2 Block?

BASCONFIG - Adding Config Blocks to Qlib Compiled Programs

There is a utility called BasConfig that allows Config 1 Blocks to be built for programs compiled by Qliberator. BasConfig creates a file that has both the Config Block and the BasConfig extensions needed to get the data out of the Config Block. This file is then linked into the main program by Qliberator and becomes one executable.

- Three versions - Oliver Fink, Norman Dunbar, Dilwyn Jones.

Programming with BASCONFIG

Since configuration data is stored in the Config Block there has to be a way for the SuperBASIC program to read it and use it. BasConfig comes with a number of SuperBASIC extensions that read the Config Items from the Config Block. The extension could possibly be loaded into memory to be used during program testing, but since there is no Config Block present, I doubt they would work. There are two Config Item types that are not supported by BasConfig; Select and Long Word.

The Config Block and BasConfig extensions are linked to the executable program by Qliberator when it compiles the program. The following compiler directive is used:

```
REMark $$asmb=ram1_file_ext,0,10
```

Because the extensions are not loaded in memory, Qliberator will have a problem finding them. An EXT_FN line is needed in the program to let Qliberator know that there are external functions and that they will be resolved at link time. The following lines are needed right after the above REMark statement. You need only EXT_FN those extensions that you use.

```
EXT_FN "C_STR$"
EXT_FN "C_WORD"
EXT_FN "C_BYTE"
EXT_FN "C_CODE"
EXT_FN "C_CHAR"
```

The BasConfig extensions are:

```
C_STR$(n)    -  String
C_WORD(n)    -  Word
C_BYTE(n)    -  Byte
C_CODE(n)    -  Code
C_CHAR(n)    -  Character
```

Each extension is used to retrieve a different Config Item data type. The extensions return the nth Config Item of the specific data type. As example is:

```
$var = C_CHAR(2)
```

This will put the second CHAR Config Item in the variable $var.

Here is an example program:

```
100 REMark $$asmb=ram1_test_cfg,0,10
110 EXT_FN "C_BTYE"
120 OPEN #3,scr_100x100a50x50
130 PAPER #3,0: INK #3,4: CLS #3
140 item1 = C_BYTE(1)
150 item2 = C_BYTE(2)
160 PRINT #3,"Item #1 =";item1
170 PRINT #3,"Item #2 =";item2
180 PAUSE 500
190 CLOSE #3
```

The program will take the two Byte Config Items and print them out to the screen.

Since you can not run a program with the BasConfig extensions in the SuperBASIC interpreter, here is a way to get around this problem and limit the effect on your code testing. For each BasConfig extension call, put in a LET statement along with a REMark of the call. The LET statement works essentially as a definition of a constant. Here is an example:

```
100 REMark C_CHAR(1) has a default value of "b"
110 REMark $var = C_CHAR(1)
120 LET $var = "b"
130 REMark C_BYTE(1) has a default value of 30
140 REMark x = C_BYTE(1)
150 LET x = 30
```

When it comes time to compile the final program, either delete the LET statements or change them to REMarks, then un-REMark the extension lines. As long as you have the right data in the Config Block, the program should run exactly as tested without the Config Block.

- What happens if there is no String Item but a call is made to C_STR$()?
- What happens if there a call is made to any of the functions to n+1 items (1 more than the number of config items of a specific type)?

Using BasConfig to Create Config Blocks

Before running BasConfig, the programmer needs to determine the number of Config Items needed and the type of each item.

Below is a listing of all of the questions asked by BASCONFIG.

Number of Items:

Enter the total number of Config Items that you are putting in this Config Block.

Enter Software Name:

This is the name of the program that the Config Block will go into.

Enter Software Version:

This is the software version of the program that the Config Block will to into.

Select Type for Item # :

Use the left and right arrow keys to toggle through the selections until you reach the one you want. Hit return to accept the selection.

At this point the program changes and asks different questions for each of the different Config Type.

String

Type Selection Keystroke for Item:

Enter any key as it is not used by "config".

Enter Max Length of the String:

Enter Default String:

Enter Description Text for Item:

This is where you tell the user what this Config Item is used for.

Special Characteristics:

Strip Spaces / Do Not Strip Spaces

The left arrow is used to toggle between the two selections, Strip Spaces and Do Not Strip Spaces.

Char

Type Selection Keystroke for Item:

Type in Default Character:

Enter Description Text for Item:

Select Possible Characteristics:

Non-Printable Characters Allowed: (Y/N)

Digits Allowed: (Y/N)

Lower Case Letters Allowed: (Y/N)

Upper Case Letters Allowed: (Y/N)

Other Printable Characters Allowed: (Y/N)

Cursor Characters Allowed: (Y/N)

Non-Printable Characters Allowed covers characters that are not normally printed on the QL. This would included characters like Control Characters (CTRL-A, CTRL-B, etc.), the ESC character, etc. Digits Allowed allows you to limit the character to only be a letter. Lower Case and Upper Case Letters Allowed lets you define the case of the character. Other Printable Characters Allowed ..... Cursor Characters Allowed .....

Code

Type Selection Keystroke for the Item:

Enter Default Code Value (0-255):

Enter Description Text for Item:

Allow Selection Keystroke: (Y/N)

Not applicable, as using "config" the Selection Keystroke is not used.

Enter Code (0-255) for Selection #1

Or Press ESC to Finish Code List

Enter String for this Code:

With Code, you are creating a list of possible values that the Item can have. For each possible value there is a description string. The list can be composed of 2 or more possible values. When you have entered all of the possible values, the ESC key is hit to let BasConfig know that you are done with the list.

Byte

Type Selection Keystroke for the Item:

Enter Intial Value for Byte (0-255):

Enter Description Text for Item:

Enter Min Value Allowed:

Enter Max Value Allowed:

Word

Type Selection Keystroke for the Item:

Enter Intial Value for Word (0-65535):

Enter Description Text for Item:

Enter Min Value Allowed:

Enter Max Value Allowed:

Updated Versions of BasConfig

Both Norman Dunbar and Dilwyn Jones have updated the BasConfig program. Norman was the first to update the program and then Dilwyn took Norman's version and further updated it. The program still remains public domain.

Norman Dunbar Update

Norman Dunbar fixed a few unnamed software bugs and included the support for Long Config Items. This includes the C_LONG() function that returns the Nth Long Config Item. The questions asked by BasConfig for a Long are:

Type Selection Keystroke for the Item:

Enter Intial Value for Long Word:

Enter Description Text for Item:

Enter Min Value Allowed:

Enter Max Value Allowed:

Dilywn Jones Update

Dilwyn Jones added two functions: C_NAME$ and C_VERS$. C_NAME$ returns that Software Name from the Config Block. C_VERS$ returns the Version Number from the Config Block. The gives the programmer the option of getting the program name and version from the Config Block instead of hard coding it into the program. Remember that the user can not change either of these values when using the "config" program.

Environment Variables

The concept of Environment Variables comes from the Unix world. They are used slightly in MS-DOS, but not at all to the same extent as UNIX. For the QDOS world, the file ENV_BIN provides a number of extensions that allow the use of Environment Variables on the QL.

Essentially an environment variable is a variable that can be "seen" by executable programs. In SuperBASIC we can set up all kinds of variables, but if we execute a program from SuperBASIC, these programs can not "see" these variables and get data from them.

The purpose of environment variables is to change the configuration of a program. They function like Config Blocks, but don't require running a program to make the change.

The ENV_BIN file comes with 4 extensions. They are:

SETENV

Defines an environment variable.

ENV_LIST

Lists all defined environment variables.

ENV_DEL

Deletes an environment variable.

GETENV$

Gets the value of an environment variable.

The two key commands are SETENV and GETENV$. SETENV is used like this:

```
SETENV "VARIABLE=value"
```

SETENV takes a string argument of the type "XXXXX=YYYYY" where XXXXX and YYYYY are two strings separated by an equals sign. Any space before the equals sign is treated as a part of XXXXX and any space after the equals sign is treated as a part of YYYYY. The case of each string is important as "VARIABLE" is different from "variable". By convention, upper case is used for variables.

The SETENV is done either in a BOOT or setup program or by the user. The function for an executable to get the contents of an environment variable is GETENV$. The function is used like this:

```
a$ = GETENV$("VARIABLE")
```

In this case a$ will be assigned the value of "value". This comes from our previous SETENV command above. If the Environment Variable "VARIABLE" is not set (does not exist) then GETENV$ would return a Null string ("").

Now I did say the executables use GETENV$ and not SuperBASIC programs. Since variables are already used in SuperBASIC, we would not gain much in using environment variables. Where the commands are used is in compiled SuperBASIC programs, which are executables.

I see the purpose of using Environment Variables as adding to the flexibility of programs that are using Config Blocks. Both Config Blocks and Environment Variables are really designed to change default program settings. User Intervention and Command Line Arguments are designed to tell the program what the data is. Using environment variables allows the user the ability of making a temporary change to the default options of a program, without having to go through the trouble of using "config". Environment variables would be used to change a setting for a single session and that's it.

Getting Config Blocks and Environment Variables to work together is not difficult. The program would first get its default settings from the Config Block. It would then check to see if there are any Environment Variables set. If there are, then the Environment Variable settings would override the Config Block settings.

FREEWARE PROGRAMMING EXTENSIONS

Adrian Ives Extensions

Adrian Ives has made a number of SuperBASIC extensions available on his web page. Of the four extensions, two are applicable to programmers. These extensions are freeware and may be redistributed with your programs. Since these extensions are rather small, it may be easier just to link them into the final executable instead of having the user LRESPR them.

IsRes_Rext

ISRES() is used to test for the presence of an extension. With it you can tell if the QLib or Turbo runtimes are loaded. You can tell if Environment Variable support is loaded. Once you know if an extension is loaded, you can alter your program to either support the extension or not. If a program supports Environment Variables and, through ISRES(), it finds that ENV_BIN has not been loaded, it can avoid making any ENV calls and not crash.

OS$_Rext

OS$() returns the a string identifying the version of QDOS running. OS$() returns the following values:

"QDOS" = Standard QDOS

"SMSQ" = Any version of SMSQ

"Minerva" = Any version of Minerva

Your application can change behaviour based upon what version of QDOS it is being run under. This function allows a program to use the expanded facilities of SMSQ and Minerva while not requiring it to be run on SMSQ or Minerva.

Display Code

This is a small suite of 9 basic extensions written in assembler designed to help SuperBASIC or SBASIC programmers cope with extended display modes on more recent hardware and emulators such as Aurora, QPC and QXL.

Although extensions are built into SMSQ allowing easy checking of such details as screen size and location in memory, programs using those extensions are limited to running on systems with SMSQ and SBASIC.

These extensions will work on SMSQ and QDOS, providing a means of consistently returning the required information, allowing programs a means of working on all systems. Graphical applications often need to write direct to memory, or at least to know the screen size and location details. That's the sort of information this code will allow you to extract from the system.

Over the years, many programs were written which were later found not to work on displays other than the original 512x256 QL screen. The forward-thinking designers of the QL had actually allowed for the possibility of larger screen sizes by including information in the system which was available to machine code programs, but not to SuperBASIC programmers. As the information about this was not readily documented and available or widely understood in those early days, many programs just assumed the original QL display and hence could not work properly when the size and location of the screen memory changed - I know, I wrote such programs myself! This set of extensions won't fix those early programs of course, but does give a simple means of extracting this information for SuperBASIC and SBASIC programmers so that new programs at least won't be guilty of the same sins. To be fair, with the benefit of hindsight it is easy to refer to those old programs as being sinful, though at the time the necessary information was neither readily accessible nor widely understood.

The files are on the cover disk - look for DISPLAY_CDE and DISPLAY_ASM. You can simply LRESPR FLP1_DISPLAY_CDE if Toolkit 2's LRESPR command is available (remember it may give a 'not complete' error if there are any jobs running at the time), or add this to your boot program:

```
base=RESPR(598):LBYTES FLP1_DISPLAY_CDE,base:CALL base
```

I hope that the names I have given the extensions don't clash with anything you already have installed on your system. If there is a clash, you'll need to either hack the names of the extensions in DISPLAY_CDE using your preferred editor, or reassemble the source code file DISPLAY_ASM after altering the names in the dc.b statements in the table.

There are eight functions and one procedure:

ADDRESS

LET adr = ADDRESS(#channel)

This function returns the base address of the display for the given window channel. Normally you'd use #0. For a 512x256. Normally, you'd only need this value if you were writing programs which wrote directly to the screen area, e.g. using LBYTES to load a screen direct to the video area of memory, using a command such as LBYTES filename,ADDRESS(#0) assuming the screen being loaded was the same size as the current display.

BYTES

LET bytes_per_line = BYTES(#channel)

The display is organised as a series of horizontal lines, with each line being a given number of bytes wide. In several display sizes, the exact width of these lines in bytes happens to be the number of pixels DIV 4, but this is a dangerous assumption to make - many programs made this assumption and fell over when the Aurora came along, as some of its display modes use a fixed line width, irrespective of the number of pixels on a line, meaning that some of the bytes used to store each line are actually unused. So if you are writing individual lines to the screen, as you would for video effects for example, you need to take account of how many bytes there are between the start of one line and the next. That's the purpose of this function - it tells you how many bytes lie between where one line starts and the next line starts. Users of version JM or earlier ROMs should note that this information is not available, as the area which holds this value is used for something else, so a version JM machine always assumes it has a 128 byte line width.

DMODE

LET display_mode = DMODE

Returns the mode number of the current display. This would usually be 0 for the 4 colour modes, and 8 for the 8 colour modes. As the routine uses the mt.inf system trap, it ought to handle the extended colour mode drivers, or monochrome modes on certain emulators (both cases untested at the time of writing). I am not sure what will happen if this function is used while one of the old mixed mode screen displays are used (there are extensions in Quanta library I think which allow part of the screen to be in MODE 4 and part in MODE 8, for example)

SYS_VAR

LET system_vars = SYS_VAR

Tells you at what address you can find the system variables. Now you can PEEK and POKE to your heart's content if you really need to!!!

FLIM_X

LET x_origin = FLIM_X(#channel)

FLIM_Y

LET y_origin = FLIM_Y(#channel)

FLIM_W

LET wide = FLIM_W(#channel)

FLIM_H

LET high = FLIM_H(#channel)

The FLIM_n extensions return information about the maximum sizes or limits of a screen window size. As it uses the iop.flim trap, it means the information can't be extracted if this is not implemented on your system. But I think I'm right in assuming that if iop.flim is not on the system for whatever reason, the system can't use extended displays anyhow. If it can't get the required information, it assumes you're running a 512x256 QL screen rather than unhelpfully causing an error report. If you supply a primary channel window, the values returned will be the maximum possible size for that window (essentially the full display width and height), whereas if you supply a secondary channel number the values returned refer to the outline area for the primary. If you don't know what this means, supply the lowest window channel number opened, e.g. #0 for BASIC. The first two extensions return origin details, whereas the other two return the width and height details.

MOVEMEM

MOVEMEM from_address, to_address, number_of_bytes

This procedure lets you move the content of memory around. Simply tell it where to move from, where to move it to, and how many bytes. Negative values will cause errors. There is no check on overlaps etc so with care you can use this to fill memory areas by writing the first byte value with a POKE, for example, then moving this up to fill the required number of bytes. It always moves from lower addresses first - there is nothing particularly intelligent about this command in terms of working out the best way to move things. It is quite slow by comparison with similar commands in other toolkits.

EXAMPLES OF USE

It may be more instructive to list a few simple examples of how to use these extensions for simple applications. These routines are on the cover disk, in a file called DISPLAY_bas. Note that they use the Toolkit 2 extensions ALCHP and RECHP for allocating and deallocating temporary buffer areas in the common heap area of memory. Most systems these days have Toolkit 2 or equivalent commands so this should not be a problem.

1. FULL SCREEN WINDOW

This routine shows you how to set a window to occupy the full screen, or the full outline area available to it if it is a secondary window. To make channel #0 occupy the full area of the screen, enter the command FULL_SCREEN #0.

If you tried the same thing on channel #2 on a VGA display, for example, #2 would be set to the maximum possible area as covered by the outline for the primary channel, in this case, #0, which would normally cover #0 and #1 and #2 in BASIC.

The procedure leaves the actual values in the variables dw and dh (width and height) and dx and dy (origin co-ordinates). Note that this routine doesn't actually do anything visibly, you may need to do a CLS command on the channel concerned, for example, to see its effect.

2. STORE A SCREEN IN MEMORY

This routine sets up an area in the common heap to store a copy of the current screen. The address of this area is given by the variable "area". We work out the start address of where to copy from the screen using the ADDRESS function, and the total number of bytes to copy is calculated by the product of the number of bytes per line (given by the BYTES function) and the height of the screen (given by the FLIM_H function). Note that when using 512x256 mode on the Aurora, for example, this routine will save the whole memory used for the screen, not just the visible area, as Aurora uses a fixed line length unrelated to the actual number of pixels used on the current display, meaning that although you only see 512 pixels across, for example, the line used to hold this display is 1024 bytes wide, but only 512 used and visible, so the calculation is not as obvious as might be thought at first. A typical application of this little routine might be to store a graphical screen in memory while a menu is superimposed on the picture. The variable "screen_length" holds the actual length (in bytes) of the screen saved.

3. RESTORE A SCREEN FROM MEMORY

This routine restores the screen saved by the previous procedure, and releases the memory area used to store it, by using the RECHP command from Toolkit 2.

4. MERGE SCREEN

There is a large number of clipart screens available for the QL, mostly as 512x256 QL screens. In the old days, when each QL had the same size screen, it was easy enough to load these direct to the screen with a simple LBYTES filename,131072 command. Not only doesn't this work on modern large displays, it might actually crash the system in some cases, since the area of memory previously used by the screen may now be used by something else. This routine tackles this problem by loading the 32k (512x256 pixels) screens into a buffer area in the common heap, then copies it line by line into the top left corner of the display. Note how two variables are used to keep track of where each line starts. With old 512x256 screens, we know they are 128 bytes wide, so it is easy enough to step through them 128 bytes at a time. The other variable is incremented by the width of each display line, given by the BYTES function. Of course, writing direct to the screen is not the done thing, and the picture may well be ruined if another job is writing to the screen at the same time! Finally, when the transfer is complete, the heap memory is released with the RECHP command.

5. FILL MEMORY

A task which arises now and again in programming is to fill a given area of memory with a particular value. This routine takes advantage of how the MOVEMEM command works. The command should be issued in this form: FILL_MEM start_address,how_many_bytes,what_value The routine works by setting the first byte of the area to be filled, using the POKE command. Then, it copies this up one byte with the MOVEMEM command, which repeatedly copies each byte up one address, thus the byte copied is always the value of the previous byte and the area is filled with the value of the first byte fairly quickly.

Another example: if you wished to turn the entire display black, then you could issue the following command. This is quite a naughty way of doing things, but it serves to illustrate how the command works: FILL_MEM ADDRESS(#0),BYTES(#0)*FLIM_H(#0),0

6. SYSTEM_VALUE

This routine reads a value from the system variables. You don't need to supply the absolute address, just the offset as published in several QL technical manuals. The routine adds the offset to the base address, peeks a value from there and returns it as the value of the function. LET value = SYSTEM_VALUE(offset) For example, PRINT SYSTEM_VALUE (140) prints the value of the auto repeat delay, while PRINT SYSTEM_VALUE(55) prints the network station number.

7. SET MODE

Some programs which need to switch between 4 colour and 8 colour mode often set the screen mode (causing an irritating flashing) even if the screen was already in the required mode. A short routine like this can check the current mode and only change it if it is the wrong mode, thus preventing the flashing of windows you get when changing mode to the same mode.

```
REMark example routines for use with DISPLAY_CDE
REMark written by Dilwyn Jones, June 1997
```

```
   DEFine PROCedure FULL_SCREEN (channel)
      dw = FLIM_W(#channel) : dh = FLIM_H(#channel)
      dx = FLIM_X(#channel) : dy = FLIM_Y(#channel)
      WINDOW #channel,dw,dh,dx,dy
   END DEFine FULL_SCREEN

   DEFine PROCedure STORE_A_SCREEN
      screen_length = BYTES(#0)*FLIM_H(#0)
      area = ALCHP(screen_length)
      IF area < 0 THEN PRINT #0,'Unable to reserve memory.' : RETurn
      MOVEMEM ADDRESS(#0) TO area, screen_length
   END DEFine STORE_A_SCREEN

   DEFine PROCedure RESTORE_A_SCREEN
      IF area > 0 THEN
        MOVEMEM area TO ADDRESS(#0), screen_length
      END IF
      RECHP area
   END DEFine RESTORE_A_SCREEN

   DEFine PROCedure MERGE_SCREEN (filename$)
   REMark merges a 32K 512x256 screen onto the top left corner of a
   larger display
      area = ALCHP (32768)
      IF area < 0 THEN PRINT #0,'Unable to allocate memory.' : RETurn
      LBYTES filename$,area
      from_address = area
      to_address = ADDRESS(#0)
      FOR a = 1 TO 256
        MOVEMEM from_address TO to_address,128
        from_address = from_address + 128
        to_address = to_address + BYTES(#0)
      END FOR a
      RECHP area : REMark finished with it
   END DEFine MERGE_SCREEN

   DEFine PROCedure FILL_MEM (addr,no_of_bytes,byte_value)
      POKE addr,byte_value
      MOVEMEM addr TO addr+1,no_of_bytes-1
   END DEFine FILL_MEM

   DEFine FuNction SYSTEM_VALUE (what_offset)
      RETurn PEEK(SYS_VAR+what_offset)
   END DEFine SYSTEM_VALUE

   DEFine PROCedure SET_MODE (mode_number)
      IF DMODE <> mode_number THEN MODE mode_number
   END DEFine SET_MODE
```

Database Handler (DBAS)

DBAS is a collection of extensions that support Database actions. Instead of creating a whole new language to support databases, the DBAS extensions have been created to support databases in SuperBASIC. With DBAS a programmer has all of the tools for creating a database application, including creating and editing a database and report generation.

DBAS is not to be used by a general user, as Archive is, but to be used by a programmer to construct an application based around a database. The DBAS extensions are designed to let the programmer handle databases as channels (or files).

PROCEDURE SUMMARY

```
ADD_FIELD      Add a new field
APPEND         Add a new record
COPY_DATA(_O)  Copy a database
CREATE(_O)     Create a database
CLOSE_DATA     Close a database

EXCLUDE        Deselect records
EXCLUDE        Deselect all records
EXPORT(_O)     Export a database
FIND(C)        Find by INSTR
IMPORT(_O)     Import a database

INCLUDE        Select records
LOAD_NAMES     Reload name list
LOCATE         Find by ORDER parameters
```

OPEN_DATA	Open database - modifiable
OPEN_DDIR	Open database - directory
OPEN_DIN	Open database - read only
ORDER	Unorder a database
ORDER	Order a database
REMOVE	Delete a record
REMOVE_FIELD	Remove a field
REMOVE_NAMES	Remove field names
RESET	Reset record selection
RPOSAB	Goto absolute position
RPOSRE	Goto relative position
SAVE_NAMES	Save field names
SCPTR	Set compare table
SEARCH	Find by INCLUDE parameters
SET	Set a field
SEXTRA	Set extra information
SUBF_ADD	Add subfields
SUBF_REM	Remove subfields
SUBF_SET	Set a subfield by number
SUBF_SETO	Set a subfield by offset
SORDKEY	Set order key length
STNAME	Set a field name
UPDATE	Update a record

FUNCTION SUMMARY

COUNT	Get record count
DBADDR	Get the control address
FETCH	Get field contents
FEXTRA	Get extra information
FLLEN	Get field length
FLNAME	Get a field name
FLNFIND	Get number of field name
FLNUM	Get field count
FLTYP	Get field type
FOUND	Get found flag
RECNUM	Get current record number
SUBF_FETCH	Get subfield
SUBF_FCURR	Get subfield by offset
SUBF_FNEXT	Get next subfield
SUBF_NUM	Get subfields count/number

DBAS is a very complex package with lots of power. A tutorial on using it is beyond the scope of this document and does require a whole book of its own. It has been added here to bring awareness to it and for completeness. DBAS is probably one of the most powerful and under utilized extension packages in QDOS.

QSEND

QSEND is a package of functions created by Hans-Peter Recktenwald that allows one program to control another program by simulating keyboard input. With QSEND you could have a program automate another program like Quill.

The QSEND package consists of the following functions:

KEYQ(jobnumber)

Get "QDOS-Channel" of a job

QKEYQ

Get "QDOS-Channel" of active job

QSEND([#]channel,text$)

Send string to job at channel

PCONNECT([#]send_chan,[#]receive_chan)

Pipe between jobs

Before sending any data from one job to another, you have to get the "QDOS-Channel" of the job you want to send the data to. KEYQ takes an argument of a QDOS job number and returns the "QDOS-Channel" number. It is used like this:

```
   LET qchan = KEYQ(2)
```

If you are writing a program that another user may use, you may not want to have the user find out what job number the receiving job is. The best way to do this is to find the job number via the job name. Unfortunately I am not aware of an extension that will return a job number given a job name.

QKEYQ works similar to KEYQ but it returns the "QDOS-Channel" of the currently active job. This means that when QKEYQ is called and Quill is the program with the cursor, then Quill's "QDOS-Channel" will be returned.

Now that you have the "QDOS-Channel" of the job you want to send data to, the next step is to use the QSEND function to send the data. QSEND is used like this:

```
   LET text$ = "This is a test"

   LET result = QSEND(qchan,text$)
```

The variable 'result' will return either an error code or how many bytes where sent. The return codes are:

result > LEN(text$)
all data was sent.

result > 0
the amount of bytes sent.

result = 0
no data was sent.

result < 0
a QDOS error occurred.

Using QSEND with the comma (,) separating the "QDOS-Channel" and the string, not all of the string is guaranteed to be sent. So, to make sure that all data is sent, the following construct can be used:

```
   sent = 1
   length = LEN(text$)
   REPeat loop
     result = QSEND(qchan,text$( sent TO))
     sent = sent + result
     IF result < 0 OR sent > length THEN EXIT loop
   END REPeat loop
   IF sent < 0 THEN PRINT "ERROR #";sent
```

Another way is to use QSEND with a exclamation mark (!) separator instead of the comma. This will tell QSEND to use an unlimited timeout and not return until all of the data is sent. Using QSEND will look like this:

```
   QSEND(qchan ! text$)
```

One thing to remember with QSEND, any programs that are not the current job will not have their screen's output updated until they become the current job. If you run a quick test with QSEND and Quill, you will notice that the text sent to Quill will not show up in Quill until you CTRL-C to Quill, where it will quickly appear. If Quill is the current job, and the QSEND program is running in the background, then the text will show up immediately.

A string sent via QSEND can be any string, including control characters. To control Quill and send an F3 key (to get to the menu), you can put the F3 key (code 240) into a string by doing the following:

```
   F3$ = CHR$(240)
```

Now if you send F3$ to Quill via QSEND, it would be just like hitting the F3 key. In this manner, all keystroke combinations can be sent via QSEND.

PCONNECT is used to connect two pipes or pipe sender with a CONsole receiver. An example is using it to also send data to Quill.

```
   pipe_chan = FOPEN("PIPE_200")
   qsend_chan = KEYQ(quill_job_number)
   pcon_chan = PCONNECT(#pipe_chan,qsend_chan)
   IF pcon_chan < 0 THEN error
   PRINT #pcon_chan,text$
```

This routine opens a QDOS pipe. It then gets the "QDOS-Channel" of Quill. It then connects the QDOS pipe channel to the Quill "QDOS-Channel". Now you can treat Quill as another channel. Now instead of using QSEND to send data to Quill, you all you have to do is use PRINT.

What good is QSEND? You can use it to write some fairly complex macros for programs that don't have a macro language. You can use it to write a program to create an auto-running program demo.

DIY Toolkit

The DIY Toolkit comes from a long running series of articles from "QL Word" magazine by Simon Goodwin. The toolkit comes with extensions from each article in a separate file and with separate documentation. The whole toolkit is fairly large.

1. ALIAS and DEF$

Lets you add extra names for a given extension and choose procedures and functions by name from a menu, for editing or programming.

2. Basic Tools

Explore SuperBASIC interpreter memory; ten demos e.g. FORGET (for keywords), POP (RETurns), and WHY, for structured debugging. Extensions: BPOKE, BPOKE_W, BPOKE_L, BPEEK%, BPEEK_W%, BPEEK_L%.

3. Channels

USE sets SuperBASIC default channel; CHAN functions access window settings. Extensions: CHAN_B%, CHAN_W%, CHAN_L%, USE.

4. Error Control

Full editing of default in limited space; avoid coercion errors; zap all tasks. Extensions: PURGE, CHECK%, CHECKF, EDLINE$.

5. File Tools

CustomKit builds toolkits from _CODE & _BIN files. EPROM compiler makes ROM images. Task Commander turns compiled tasks into multi-threaded SuperBASIC extensions. File Header extensions: GetHEAD, SetHEAD.

6. Graphics

Faster & more precise, to suit all QLs and Thor MODE 12 bonus, as well as several decorative demos. Extensions: PLOT, DRAW, PIXEL%.

7. Heap and Horology

Four 20ms stopwatches; SHOW_HEAP RAM explorer and info. MAPper for memory. Extensions: LINKUP, RESERVE, DISCARD, T_ON, T_OFF, T_START, T_STOP, T_COUNT.

8. Serial Mouse

Routines to support a two or three button Microsoft or Mouse Systems "PC" serial mouse. Compatible with the Amiga QDOS 'bus' mouse driver at the SuperBASIC keyword level.

9. Jobs

Multi-tasking the DIY way, with excellent compatibility without gobbling RAM; Task Control toolkit plus TASKFORCE (names task taking input), PSION_PATCH, TASK_RENAME, and ADDNAME programs.

10. Hewlett Packard PCL Printer Routines

Adjustable resolution shaded dump routines for the HP PCL printers (Deskjet, Laserjet, etc.). Also documents PACKBITS compression and colour separations. Plus routines to do interesting things with text printed on an HP printer.

11. MultiBasic

MultiBasic compatible multiple SuperBASICs for any QDOS system. The new version 4.0 can swap screens as well as programs and variables. Extensions: UNLOAD, RELOAD, RESAVE, REMOVE, SCR_SAVE.

12. Networking

Three improved NETPALs issue commands remotely on QL, Minerva, or Thor. MEM - a new QDOS device to share memory between tasks or computers.

13. Pipes and Parameters

Machine code power from SuperBASIC extensions, plus SEARCH_PROG and CAT columnar directory demos. Extensions: PARTYPE, PARSEPA, PARHASH, PARNAME$, UNSET, QLINK, QCOUNT%, QSPACE%, QSIZE%.

14. Queues and QDOS

QUEUE% types into programs; QLIST scans keyboard queues; CHLIST name all open channels. Extensions: CHBASE, SYSBASE, QUEUE%.

15. Replace

Exchange variable, array and device names, or look up details; case converters. Extensions: LOWER$, UPPER$, NEWCHAN% LOOKUP%, REPLACE.

16. Qlipboard

Scan characters in any size or colour into SuperBASIC strings or type them into any other task. Full task including text editor.

17. Traps

QDOS, Minerva, and Argos system calls ('traps') documented and demonstrated with easy-to-use SuperBASIC extensions and assembly code.

18. Environment Variables

Share strings and numbers between concurrently running tasks and SuperBASIC. Also includes documentation and a demonstration of 'User Heap' management.

19. More

Page through files or memory quickly and easily, with indication of your current position.

20. Windows

How to save, restore, and move images in QDOS windows.

21. Msearch & Vocab

Look through the vocabulary of your SuperBASIC interpreter for variables, resident commands and functions, or SuperBASIC DEFinitions. MSEARCH is a fast, flexible search routine.

22. Flexynet

A simple replacement drive for QLAN (the Sinclair network) which can go faster than the original and adjust to mis-matched system speeds.

23. Array Search Routines

SuperBASIC numeric array and string array search routines in assembler, with source code and explanations.

The DIY toolkit and its accompanying articles is a great place for exploration and learning.

Client Server Manager

Client Server Manager is a toolkit from Jonathan Hudson that supports true client/server communications on a single QL. The traditional view of client/server consists of a server process (program) that waits to handle requests from client processes (other programs). The client does not necessarily need to be on the same computer as the server. A web server and web browser are examples of client/server programs.

Jonathan has used CSM in the creation of his applications (QEM, QFAX, QTPI, etc.). All of these have a CSM server component. The user can then send CSM client requests to control the applications. A combination of SuperBASIC with CSM client commands form a macro language for these applications. This saved Jonathan the time and effort in writing a macro language (including a parser) for each application.

How CSM works is like of like this: Before CSM can be used, the CSM toolkit is loaded (LRESPR'd). Part of the toolkit is a Thing. When a server program is started it registers itself with the CSM Thing. When a client program starts, it declares which server it wants to talk to. The CSM Thing handles the message passing between the client and server, since it knows where both are. The client makes a request to the server, the server gets the request, figures out what to do with it, and send a reply. The actual contents of the request and the reply are up to the programmer and is not defined in CSM.

The following keywords are added by CSM:

SERVER 'name' [,buffer]

Declares program as server 'name'

CLIENT 'name' [,buffer]

Declares program as client 'name'

LISTENER 'name' [,buffer]

Declares program as listener 'name'

REQUEST 'name','command' [,res$]

Client request to server 'name' of type 'command'.

LSTNPOLL ('name', bro$ [,timeout])

Listener polls for a broadcast message from server.

SRVPOLL ('name',req$ [,timeout])

Server polls for request from client 'name'.

SRVREPLY context,res$

Server reply.

SRVBRO 'name', bro$

Server broadcast to LISTENERs.

FREECLIENT 'name'

Releases client from server.

FREELSTN 'name'

Releases listener from server.

FREESERVER 'name'

Terminates server.

FINDSERVER ('name')

Check for presence of server 'name'.

SVRPEND ('name')

Check for pending requests.

GetStuffed

With tongue firmly in cheek, Jonathan Hudson has written a one function toolkit that does one thing, it gets the contents of the Hot Key System II Stuffer Buffer. The function is a called GET_STUFFED, hence the tongue in cheek. The function is used like this:

```
   a$ = GET_STUFFED
```

Now a$ will contain whatever was in the Stuffer Buffer. If the buffer was empty, then a$ will contain the empty string (""). That is all that the function does. As simple as it is, it is a very handy tool and may be one of these functions that can make a program work.

Tools To Assist in Programming

Structured SuperBASIC

Structured SuperBASIC (SSB) is a tool that was originally designed to add line numbers to a SuperBASIC program created using a text editor. It has progressed to allow a number of options in writing SuperBASIC programs. SSB takes a program, written in the SSB style and converts it into a form that can be loaded into SuperBASIC. SSB processes the code and depending on commands in the code does a variety of things. For those familiar with the term, SSB can be thought of as a form of preprocessor.

SSB supports the following options:

Include Statements
Add in code from another file.

Conditional Compilation
Depending on Define statements, different code parts can be either passed through SSB or not.

Blank Lines
Blank lines between statements. No longer a need for lines with just colons (:) on them.

Labels
Since lines are not numbered, labels are used for GOTO and GOSUB statements.

Conditional Remarks
Some Remark statements are not passed through SSB.

Long Lines
A single SuperBASIC line can extend across two lines.

With Structured SuperBASIC you don't need to worry about line numbers, esp. when adding code from other sources. If you like to create libraries of SuperBASIC code in separate files, with SSB you don't need worry about what line number are in each file. You just use the Include statements to add the code to a program. SSB takes care of the rest.

Because SSB allows you to easily not use line numbers, other tools can then be used in maintaining your code. A good tool is the Revision Control System (RCS), a tool for keeping track of different versions of source code.

ProcMan

This program allows you to extract selected procedures or functions from SuperBASIC programs. It requires the Pointer Environment and the QMENU extensions.

The program can be configured using the standard program "config". You can change the default drive for loading, saving, default file extension for SuperBASIC programs, and the default printer device.

When you first execute the program, you are given three choices; Continue, Quit, or Button. If you choose Button, the program will become a button in the Button Frame. Continue goes on with the program.

Once into the program, a file select menu pops up. From here you choose which SuperBASIC program you want to extract some code from. The program will then read the SuperBASIC file, collecting all of the names of the Procedures and Functions. The next step is to select which Procedure or Function you wish to extract from the file. You may select one or more items. Once you are done selecting, click on OK to continue.

Now you can either print the selected Procedures and/or Functions or send them to a file. If you select File, you need to enter a file name. The program will now extract the source code you selected and put it into the new file.

Once this is done, you can either Continue (and do it all over again) or Quit.

Editors

SuperBASIC (and QDOS) has a built in editor (albeit limited) for editing SuperBASIC programs. Toolkit II expands the editing capabilities with the ED command, giving a full screen editing capability to SuperBASIC. One of the biggest advantages of the built-in editing capabilities of SuperBASIC is its automatic syntax checking. As you enter each line, SuperBASIC checks the syntax and complains about an error if the syntax is bad.

As useful and handy the built-in editing is, for longer SuperBASIC programs, it leaves much to be desired. A number of programmers have turned to text editors to make the creating of SuperBASIC programs easier. Each editor approaches the task of editing differently, thereby giving each editor its own personality, just like each programmer has their own personality. Sometimes these personalities match, making a good working relationship between programmer and editor, and sometimes they don't. Feelings toward editors are almost as strong as feelings toward languages.

When using text editors, the idea is to type in as much of the program as you can before you need to actually test the code. Since most programmers code on the fly, using an editor allows you to move quickly in the code, adding in lines where ever necessary. It also allows you to write some of the program in pseudo-code, coming back later to fill it in will real code. At this stage, your program could be considered a rough draft. Only when it is nearer its final draft stage is it ready to be loaded in to SuperBASIC and run.

When writing SuperBASIC code using an text editor, you have the option of including the line numbers yourself, having the editor do it, or using another utility (like Structured SuperBASIC) to do it for you. Some editors have a macro language that will allow you to write a sort routine to automatically put line numbers in each text line.

QED

QED, by Jan Bredenbeck(sp?), is a close of Metacomco's editor ED. It is a fairly quick little editor that has a limited macro ability. QED commands are entered on a command line at the bottom of the screen. Besides having commands for moving the cursor, it also has commands for Searching (Find) and handling Blocks of text. A number of commands can be entered on the command line including a repeat command. This can automate some simple functions, making for a form of macros. QEDs biggest advantage is that it is small, quick, and fairly simple to use.

MicroEmacs

MicroEmacs comes from the Unix and MS-DOS worlds with ports to other operating systems (Amiga, Atari ST, etc.). It is a very powerful editor with a full set of commands and a built in macro language, including most of the control constructs of regular languages. The command set is a bit on the complex side and can take a while to learn. Fortunately, the commands can be changed by the user in a configuration file. The user can totally remap all commands to new keyboard combinations. There is even a configuration file to make MicroEmacs work like the popular MS-DOS editor, BRIEF.

MicroEmacs allows cut-n-paste, editing more than one file (via multiple windows), and some complex editing commands. Another advantage is that, because it is available on different operating systems, you don't have to learn a new editor for each different OS. Thierry Godefroy has added Pointer Environment support (including PE menus) and a link to QTYP II.

If you want more power in your editor that QED, then MicroEmacs is a good choice. There can be a stiff learning curve, but you will benefit in having more power with the editor.

Elvis

Elvis is a clone of the standard Unix editor, VI (pronounced vee-eye). Elvis is almost as powerful as MicroEmacs, but it is a whole lot more obtuse. The commands can look like gibberish and be hard to learn. It is an editor built by Unix hackers for use by Unix hackers. It's key advantage is that once learned, you can use it on any Unix system and any other system have has a VI clone. There is even a number of different books covering VI, including one "Learning the VI Editor" that covers Elvis and a few other VI clones. I see this editor as being on the QL to be used by those that already know VI or those persons that want to learn VI as they learn Unix. I don't recommend it for an average user.