USER FRIENDLY PROGRAMMING

Another QL trader and myself once lost a skilled programmer to the QL community. He had sent us both a program for evaluation and, independently, we had reached the same conclusion. We had no doubt it did some clever things and had commercial potential, but it was so user unfriendly we had been unable to master it. When we suggested to him it needed some rewriting, he went off in a huff and left the QL community.

It reminded me of the Dutch newspaper I used to read which had a daily cartoon from a reader. Generally speaking a cartoon was either funny but badly drawn or not funny and well drawn. It was rare to have a cartoon that was both funny and well drawn.

It is much the same with software. There are some brilliant programmers who can write clever programs and routines, but who know little about user friendliness and program presentation. Others can produce attractive, easily used programs, but which are limited in what they do and sometimes full of bugs,

This article is not about programming as such, but about good software design. About what you can do to make your programs more attractive and more user-friendly. It is based on my experience of producing Just Words! software.

My first commercial software product, Solvit-Plus 2, was initially distributed by Dilwyn Jones Computing. It was an interesting, and at times humbling, experience to see a amateurish program being whipped into shape by an experienced trader Through this experience my programming and presentation skills improved greatly,

Even after Dilwyn's work Solvit-Plus 2 had shortcomings. Some users were overwhelmed by the number of commands, and others felt some command names were illogical. I learnt from these comments and radically altered my program design. From comments I received from both reviewers and users, it was a successful redesign, although not everyone was happy with it.

SCREEN DESIGN

The image shows the screen design of the GD2 pointer version of QL Thesaurus.

I needed a place for three menus and a large work area. The menus have different functions and are active at different times during the use of the program. One of the menus (SET UP) would be used only occasionally so I could put it in a pull down box, but the other two had to be on the screen alongside the work area. I placed one of these on the left hand side. This is the menu you use to start a search and I gave the three most used menu items extra emphasis by placing them in a box with a different coloured background. The other menu, which becomes active during a search, I placed at the bottom of a screen.

Above the work area I put the program's name and copyright notices. As finishing touches I added my logo to the top right hand corner of the screen, and, very gimmicky, made my pointer the pen nib from the logo.

It should be said that there are some QL-ers who do not like this type of layout. They believe quite strongly that there should be a standard QL layout especially for pointer environment programs. This gives a certainty because they know where to look for menu items. It is a matter of choice.

Now take a look at one of your own programs. Can you describe why you designed it in the way you did? Why you used those colours? Why you placed the menu where you did? Why you put the work and input areas where you did? If you cannot answer these questions maybe it is time for a redesign.

MORE ABOUT MENUS

Do you remember buying your first QL and your first attempts at using Quill? Would it have been easier if, instead of having to remember L for Load, S for Save and P for Print, you had had to remember F5 for Load, F10 for save and Shift + F7 for print? In total 40 combinations of the function keys with Shift, Control and Alt? This is what you would have had to learn if, instead of a QL, you had bought a PC and WordPerfect, the state of the art wordprocessor at the time. WordPerfect must be one of the most user unfriendly commercial programs ever, whereas Quill, if not as sophisticated. was quick and easy to learn.

The length, content and placing of menus determine how easy your program is to use.

First length. The longer your menu the more difficult people will find your program. Submenus improve things a little, but some users also find these complicated. It is a good practice to look at your menu items one by one and ask if each is essential.

Early versions of my program Solvit-Plus 2 had two dictionary commands, one for loading a dictionary in Solvit-Plus format and another for importing a plain text file. When I was writing the pointer version and wanted to reduce the number of menu items, I realised only one command was necessary. The later version first checks the format of a file, and if is not in the Solvit-Plus format warns the user and asks whether he wishes to attempt an import.

Similarly, most word processors have separate load and import commands, but Perfection does not. In contrast Text87 has a complicated menu structure that makes it a slower and more difficult program to use. There are separate load and import commands, and then an import submenu with a choice between ASCII and Quill formats. The menu structure of Text87 could have been much simpler.

Programming is a logical activity and sometimes when we think logically we miss the simple and obvious. When I was designing Style-Check, logic said I would need a load

command followed by a submenu to choose between Quill, Perfection, Text87 and ASCII formats. It was only when I was writing the program that I realised the different file formats were easy to detect automatically. I needed just one load command.

When you have decided what to have on your menu, go through the items one by one and assess how often that item will be used. In a word processor loading, saving and printing are among the most used commands. In most word processors these commands are the first on the menu bar. In contrast the help screens may only be used when the user is learning the program. Similarly configuration commands are likely to be used only occasionally, usually when the program is new. Both of these commands can have a relatively low priority in the menu list.

If your program has a print routine, you will need menu commands for printing, setting the baud rate and the printer output device but do not place these together. Print will be used regularly but the other two perhaps only once in the life time of the program. Place them by the configuration command

Press F3 in Perfection or Text87 for good examples of setting priorities on menu lists and in Quill for a bad example.

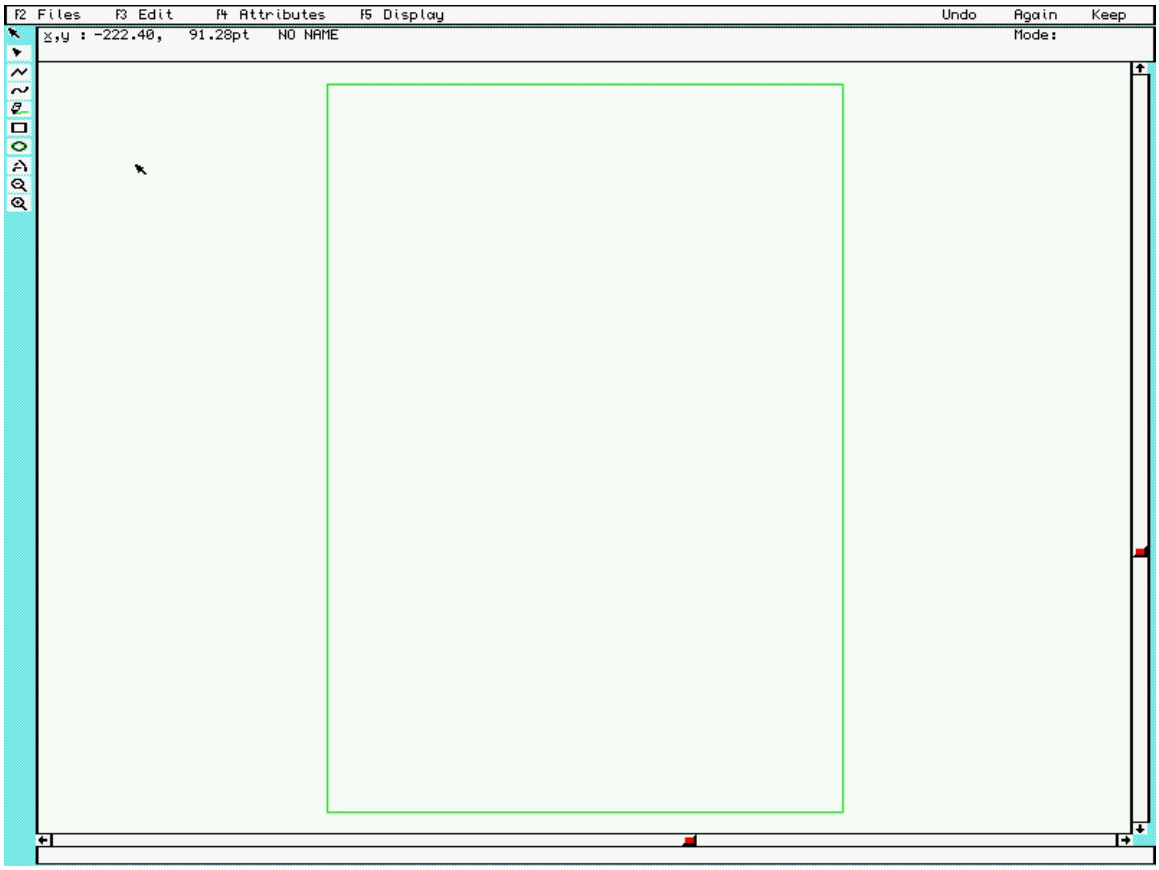
The next thing you have to decide is the placing and form of your menu. The most important thing is that your menu items should stand out from other items on the screen They can be a different colour, a different size, a different font or in a special menu box or window. Good examples of this are Quill and to a lesser extent Perfection A bad example is Text87. Poorly planned, barely legible menus are one of the commonest faults in QL software.

Finally your menu items should be clear and unambiguous. Most programs no longer use the function keys, but a mnemonic. Where possible the mnemonic should be the first letter of the menu item. Emphasise this letter by using a capital letter, a different colour or in pointer programs underlining.

If you have two menu items with the same first letter, try to find an alternative name for one of them. In some of my pointer programs **D**ictionary has been changed to word **L**ist because I needed the D for page **D**own. Be careful, however. One reviewer rightly criticised me for using "**N**ame of dictionary" in a program instead of "**L**oad dictionary". When I was developing the program I needed the L for another command, which I later replaced with a different routine. I forgot to change "**N**ame" back to "**L**oad".

If you have to use a letter other than a first letter for the mnemonic, do not use any old letter but think carefully which is the most suitable. In the print submenu of Text87 there are three commands beginning with P Print, Preview and Page. Obviously print itself must have the P The other two are pre**V**iew and pa**G**e, which in both cases is a good choice of alternative letter as these letters are emphasised when the words are spoken. The alternatives of p**R**eview or p**A**ge would have been inappropriate.

One program that has a good screen design is LineDesign. There is a large working area for the page, and the menus do not get in the way of this working area. At the top left hand side of the screen are the main commands. You can access these by either the mouse or a key press. The key press is highlighted in red. Press one of these keys and a submenu appears. Again this can be accessed by either the mouse or a key press, and the key press is denoted by underlining. Available menu items are in black ink, unavailable ones in green. On the left hand side of the screen there are 10 icons. These are compact and well designed. Even if you have only a simple knowledge of the program, it is clear what they do.



COMPATIBILITY

One of the commonest mistakes QL-ers make is to assume that other users have the same equipment configured in the same way as them. A Quanta librarian once told me of a lengthy and vituperative correspondence he had with a member who had submitted a program that only ran on his own system. I had a similar experience, although without the abusive correspondence, when I took a look at a program from a skilled programmer with a view to reviewing it in QL Today. The first thing I discovered was that it made use of the latest commands in SMSQ-E and was incompatible with other systems. Then I discovered it was only usable with the writers choice of colourway.

Making good use of the latest facilities in an operating system is good programming practice for your own use, but it has disadvantages if you want your software to be usable by others. Commercial programmers have to ensure a degree of downward compatibility.

Take monitors for example. In use within the QL community are a wide range of monitors from the CGA green screens to the latest LCDs. It would not surprise me to discover some people still using a TV set. If you are writing software for other users, do not assume that their monitor works like yours.

My first monitor which gave many years of faithful service, gave a sharp contrast that made it unpleasant to work with white letters on a black screen. Where possible I used green letters on a black background, a practice I had to unlearn when I started to write

commercial software. On many monitors green letters are too faint to read easily, making white letters on a black background essential. Consider the possibility of allowing users to configure their own choice of colours.

Mice are another hardware complication. They are not yet an essential QL peripheral, but they are for PCs. QL users tend to be either lovers or haters of mice. Do not assume that all QL-ers use the pointer environment.

Think about your own mouse use, particularly in one of the more complicated PC programs. Do you use the mouse all the time? Do you use key presses all the time? Or do you use a mixture? I suspect most of us do the last of these. Make sure your programs allow a mixture of both mouse and key press use.

How often when you are using a PC program do you lose the pointer? In most PC the pointer is thin and I-shaped and the cursor a thicker vertical line. I find it easy to confuse the two, particularly when I am tired.

A pointer is what it says, a "pointer". The tip is a point, and on the QL only the tip has to be within the input window. Make your pointer a true pointer and make it large enough to be seen. Make sure your pointer is visible on all points of the screen. In practice this means a pointer should always have at least two colours. The Just Words! pen nib pointer looks black, but it has a white "shadow" for use on a black background.

SPEED FREAKS

A user friendly programmer knows all the tricks to improve the speed of his programs.

One line of code in my program Solvit-Plus took over 30 hours to write. If this statement conjures up a vision of me sitting snug in my anorak nerdishly tapping on my keyboard into the early hours, then you would be wrong. For most of the time I was eating, drinking and making merry and the poor QL was doing all the hard work.

The line of code it produced is not particularly spectacular:

strg$= "eianrtsoldcugmphbfvkzywxqj"

The QL was going through long lists of words for the main European languages and working out the relative frequency in which each letter appears - 'e' is the most common, 'j' is the least common. In some searches Solvit-Plus looks through the search word to find the least common letter and then examines only words containing that letter This can more than treble the speed of some searches but the user is not aware of it, because it is used only in the slowest searches.

This 30 hours of work also proved invaluable when I was part of the QWord writing team. QWord is a word game that is a cross between Scrabble, Tetris and "Find the hidden words" puzzles.

The program generates a grid in which the player has to find words. The grid is tailor made to the language being played because the frequency in which a letter appears in the grid is the same as the frequency of the occurrence of that letter in the language. The maximises the number of words that can be found in the grid.

The easiest way of improving the speed of a program is to compile it. The QL has two compilers Turbo and QLiberator, and there are advocates of both. Early Just Words! programs were compiled with Turbo and some users expressed their distaste that they had to "dirty" their machines by having the Turbo Toolkit installed. In general Turbo is the faster of the two, but QLiberator the more flexible. Eventually QLiberator gained the upper hand because it was more compatible with the pointer environment and later Just Words! programs were compiled using QLiberator.

QLiberator remains a commercial program and is unlikely to be further updated, but Turbo is now public domain and has had more recent revisions to improve its flexibility and compatibility with the pointer environment.

Both programs can now incorporate machine code extensions into compiled programs and this you should always do so that a user does not have to "dirty" his machine with extensions essential to the program.

If your program has to do a lot of calculations that take time, there is a danger that the user can think there has been a program crash. The golden rule is never to leave a screen blank while your program is doing its work. Always give the user something to look at, even if it is only a row of dots being printed to the screen. My program QL-2-PC Transfer prints a backward or forward slash alternately every time a full stop occurs in the text it is transferring so that the user can see the program is still running. This is much better than a simple statement "please wait".

A good programmer knows how to fool the user that his program is working faster than it is. Some users have expressed surprise at how quickly my Style-Check program analyses a sentence, but this is an illusion. The analysis is a slow process that is being done and stored while the sentence is printing to the screen. This slows the printing of the sentence, but few people notice it as the printing is continuous. What they notice is the instant printing of the stored analysis when the sentence is complete.

ERRORS

There is one part of you program that users will not notice if you have done it well, but will soon notice if you have done it badly.

It is almost midnight. You have just finished typing the 2,000 word document you have promised your boss for tomorrow. Before printing you decide to save it, but have not noticed that the disk is write protected. The computer freezes, and you have to start typing again.

In practice this situation rarely arises. All word processors will warn you that something has gone wrong, and most will tell you precisely what. They will give the chance to correct your mistake. It is something we take for granted. If, however, you make a mistake in printing, your word processor may not be so helpful. The stories of people who have made a mistake when printing from the QL to a Windows printer are legion.

User friendly programs are 'error trapped'. They try to stop your mistakes or their own internal shortcomings from causing a fatal crash on your computer If a fatal crash does occur they will tell you what has gone wrong .

QL error trapping has received little attention in QL publications, probably because it was introduced haphazardly. From the start error trapping was planned for QL SuperBasic, but it was not implemented until the JS ROM, and then not effectively. The first trustworthy ROM error trapping came with the Minerva.

In the meantime other forms of error trapping had been introduced. Toolkit 2 contains some commands to trap errors when loading and saving to disk (Section 10.2), and both the Turbo and QLiberator compilers contain their own error trapping. QLiberator even has three different types. Its internal system, the red windows that appear when something goes wrong; general error trapping for use in programs; and support for the error trapping built into the Minerva ROM and SMSQ(-E). The last of these is very useful because if you write a program using Minerva or SMSQ(-E) error trapping and then compile it with Qliberator using one of these systems, the error trapping is also compatible with earlier ROMs.

My personal opinion is that the best error trapping is contained in the Turbo Toolkit, because it encourages the user to think at two levels. It also has numerous commands to trap disk operation errors.

What do we mean by two levels of error trapping? I sometimes call them 'global' and 'tailor-made'. A similar division would be 'non-recoverable' and "recoverable' errors, or 'unforeseeable' and 'foreseeable' errors. Thinking at two levels encourages us to look at the purpose of error trapping.

No computer program is perfect. Just Words! programs are relatively simple when compared to say LineDesign or Text87, but they contain 1,000 - 2,000 lines of code. I would be kidding myself if I thought there were no errors and no shortcomings in that code. This is where global error trapping is important. Error trapping tells me the line where the error, or rather the crash, occurred and the nature of the error. Usually this will not be of much help to the user because the program will have crashed, but it gives program's author vital information about why and where the program went wrong.

A first essential is that error messages make sense. On my PC the Solitaire game provided with Windows became corrupt with the error message, 'This program has performed an illegal operation and will be shut down.' If I ask for details, I am told that 'SQL caused a general protection fault in module SOL.EXE at 0001:0000060d', and I am given the advice to contact the 'program vendor'. How unlike the QL, where you would

get a ‘bad medium’ message, and it would be a simple matter to recopy the corrupted file from the master disk.

Make sure your error messages are meaningful to the user. User friendly software gives the user a clear indication of what has gone wrong when a crash occurs. It should provide him with an idea of what he can do to correct the error or if this is not possible, information he can pass on to the author. If, as a user you have a reason to complain about a program, the worst thing you can do is to say ‘it doesn’t work’ or ‘it was non too successful’, as this provides the author with no information about the cause of the problem. Always say where and how the program went wrong, and if possible give details of the error message.

Tailor made error trapping is more difficult to write than global error trapping. It will mainly be needed to cover situations where the error is caused by the user. For example, saving to a write protected disk; attempting to print to a parallel port using _ser or inputting a letter when only numerals are permitted. It can include other situations such as carrying out an operation for which the computer has insufficient free memory. In these situations the normal QL error messages are often too superficial, and you need to give the user more guidance about what he can do to recover from the error

Writing customised error trapping has its own dangers, because the author can make errors in his error trapping routines. My program QL-Thesaurus has little used routines for sending the results of a search to a printer or to a disk. These two routines initially share the same error coding, but there is some extra coding for the disk routine to prevent an existing file being overwritten. In an early version of the program I made a mistake in this part of the error trapping, and any user trying to print out the results of a search would have seen the error message, “_SER1 already exists. Delete (Y/N)”

UPSETTING PEOPLE

No matter how good your software is you are going to upset some people. This is especially so if your program has ancillary files. The days when the QL had simple device names like mdv1_ and flp1_ have long since gone and now your program is likely to find itself somewhere in a subdirectory on a hard disk.

There are many ways to pass parameters to a program, including DATA_USE, DEV_USE and SUB. Some of these are designed for use with specific hardware or for older software and are best avoided in your programs. It is also possible to pass parameters in an EX (Toolkit 2) or EXECUTE (Turbo Toolkit) command. A configuration block can also be used.

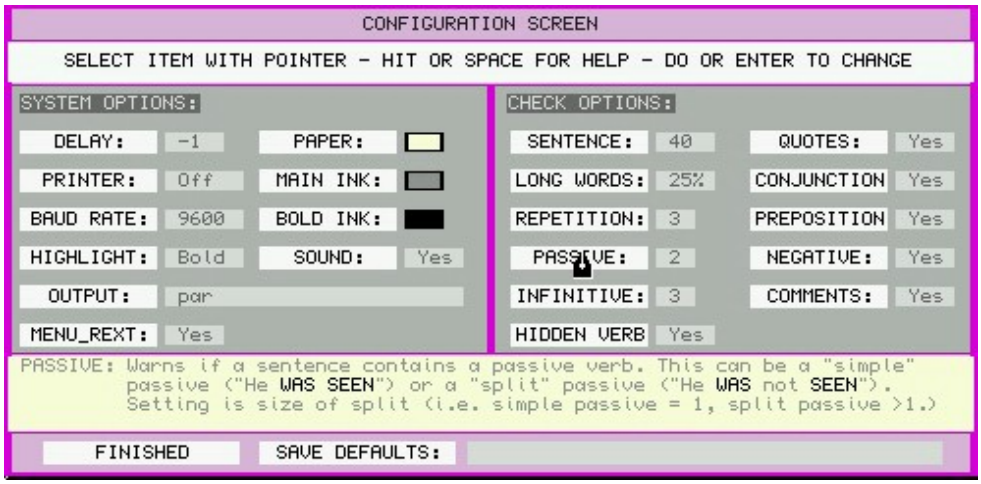
In its early days JUST WORDS! tried to be all things to all men, and ended up pleasing few. It offered more than one way of telling a program where to look for ancillary files, but the first way interfered with the second way and neither worked satisfactorily. As a contributor to QL Today put it, I had “got into a terrible scramble”. However his solution of parameters passed after the file name would mean starting Style-Check 3 with a command like

EX win1_style_style3_obj; ‘win1_style_,4,0,9600,“Bold”,“par”,7,0,2,1,0,40,25,3,3,3,1,1,1,1,1,1

Not exactly user friendly.

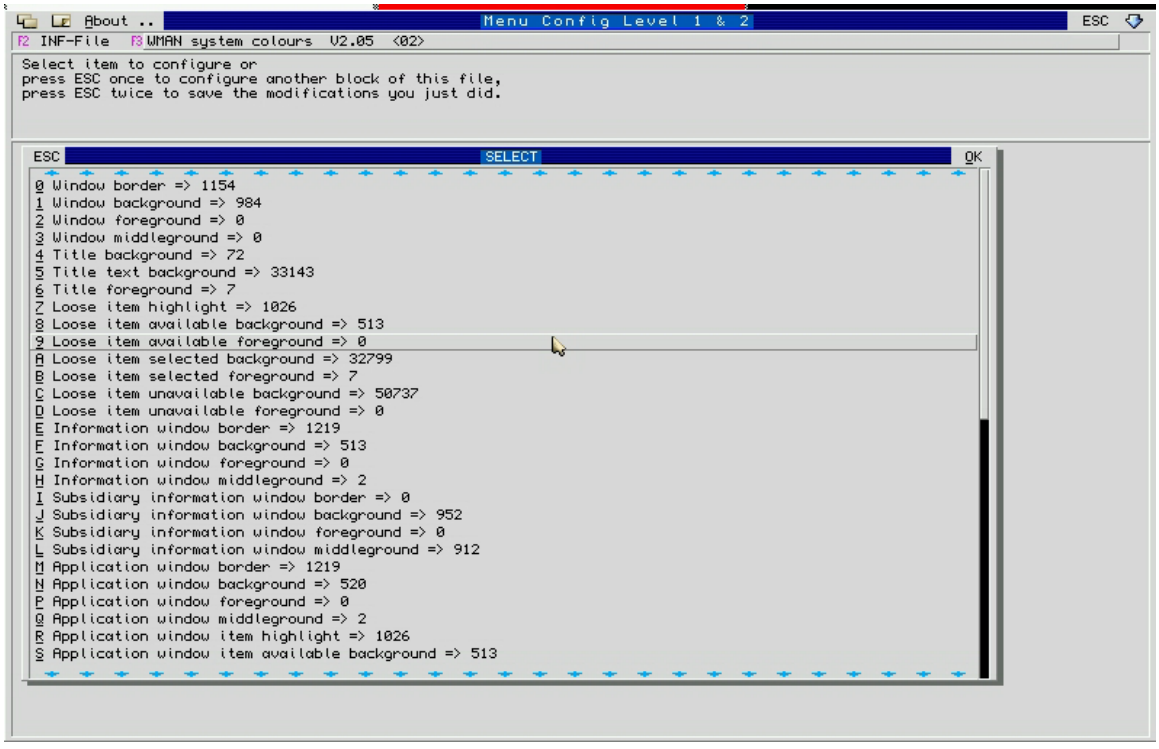
Eventually Just Words! opted for configuration blocks. Formerly these were controversial partly because they could not be incorporated into Turbo compiled programs, but now Turbo has been modified to allow this. Configuration block are currently the most

common way of passing parameters to programs.
Even then there is the question of what you include in a configuration block. Just Words! configuration blocks are controversial because they contain just one piece of information and that is the location of ancillary files including a configuration file. This upsets the purists who want a configuration block to contain everything.



If we look at the configuration screen of Style-Check you can see the logic behind the Just Words! policy. Style-Check has 21 user configurable items. It is easier for the user to adjust these at runtime when help information is available than adjusting a configuration block with no help available.

If you are not yet convinced ask yourself how you configure QPC2. The later versions have some 122 configurable items divided over 9 sections. 60 of these can be configured by the initial configuration screen, but the remaining 62, 59 of which govern your choice of colours, cannot.



The configuration screen of QPC2 crams a massive amount of configurable information in a small space and is highly user friendly. Modifying the colours, which can only be done via the configuration block is a lengthy process mainly of interest to the enthusiasts and experts.

THE GREAT UNREAD

Many people argue that a good computer program does not need a manual. If software is well written, you should be able to use it intuitively and help will be provided on-screen to get you over the difficult bits. It is a powerful argument. When I test or review software, I initially keep the manual firmly shut to see how far I can use the program without looking at it. It is a good test of a program's user friendliness.

The problem is that there is a limit to what you put on a screen and still retain the user's interest.

At the start of this article I told the story of how another trader and I lost a skilled programmer to the QL community because we found his program to be user unfriendly. He had not written a manual, but gave all the details on screen.

After loading the program it started by asking two questions. The first warned about memory, ram disk use and extensions that should be present. The second asked whether or not I was using QPC. Were these questions really relevant? A plus point was that the program correctly adjusted to the screen resolution, but then the opening screen was almost blank. Just a request to enter the parameter file plus a few instructions.

My immediate reaction was, "What is the parameter file?" I pressed D for the defaults. The next screen was a list of 17 parameters, which could obviously be amended, and the statement "action?". Again I was uncertain, but there was a small amount of help, so I pressed I for information. At this stage I started to despair. The first information screen told me nothing about how to use the program. It was a long statement of its history and the system requirements. The next screen listed the files on the disk and what the program did, but not how to use it. After reading four screens I was still no wiser I then discovered another set of information screens, all 6 of them. The author had written some clever routines for displaying text on screens of different resolutions and was showing these off, but they were irrelevant to the working of the program. I was soon wanting to scream out loud, "Just tell me what to do". I pressed ENTER to proceed, and was told the dictionary could not be found. When I finally got the program working, it was almost an anticlimax, leaving me with a feeling of "Was that it?". When I left the program, I discovered it had left all its temporary files on my hard disk.

Now compare that with the experience of a QL Today reviewer writing about a manual:

"It's laid out logically and allows you to enter it at different levels and stages. So it's easy to find the bit you need if you get stuck. Or it's easy to blitz through the Quick Start section if you can't wait to get your hands on the program. Or it's easy to quickly go to the Customisation section if you don't like the results with the current configuration.

Essentially it's difficult to fault. It told me everything I needed to know, gave me no difficulties in finding the relevant section, explained the menu structure concisely, gave numerous examples in the glossary and covered everything."

OK, it's confession time. Just Words! is blowing its own trumpet. Our manuals were

regularly praised by users and reviewers.

What were Just Words! manual concepts?

Firstly design was important. A manual must look attractive, and be laid out in such a way that the reader can quickly find what he is looking for.

Early Just Words! manuals were A5 booklets, but as the market became smaller they were changed to A4 folders, with two columns of text, which is easier to read than one column. There was a bold header at the top of each page describing its contents. Within the text, important items such as menu commands were highlighted by being printed in bold capital letters in a slightly larger font than the main body of the text. When a user thumbed through the manual, these items stood out, making it easier for him to find what he was looking for. If he could not find it, there was an index at the back to help him, and there were also illustrations of the main menus.

If you are writing for the freeware market, your manual will often be either a Quill or text file, and you will have fewer design possibilities, but it is worthwhile to look at the layout of your document to ensure it is as user-friendly as possible. It is also worthwhile considering producing a manual as both a text document and as a PDF file.

Secondly a Just Words! manual had three main sections, designed for different stages of experience in using the program.

The first section was Basic Information. It was what you needed before buying the program and during installation. It gave information people needed when browsing through the manual at a show. It told them what the program did, what the system requirements were, what files were on the disk and how to install the program on a hard disk.

The second section was the Quick Start. This proved to be the most popular feature of the manuals. It was a simple step by step tutorial to illustrate the main features of the program, and could usually be followed without further reference to the manual.

The third section was the remainder of the manual. Some users would read this, and others not. If a user wanted to read the manual from cover to cover it would tell him about the program in some depth. Alternatively if he did not want to read it, he could use it as a reference work for when he got stuck. Hence the importance of a design that made things easy to find.

To misquote Orwell manuals are the great unread, but some are more unread than others.

FRIENDLY TO YOU?

An important aspect of user friendly programming is that they are friendly to you as author. If your program is a success and is well used sooner or later someone will find a bug or ask for some enhancement or improvement. How easy will it be for you to make the changes?

A programmer who writes sloppy spaghetti type code will soon find it backfires on him. A bad dream will start at the first bug report, and turn into a full scale nightmare when the program needs upgrading.

In true schoolmasterly style I am going to refer to the 3 Rs. In this case REMarks, Readability and Reference.

REMark statements should liberally pepper your programs. These help you to remember what each section of the program does. You could, for example, put a REMark statement at the start of every procedure describing its purpose and use. You could place them at strategic places in a program to remind you about the meaning and purpose of a variable. They are also helpful in SELect ON - END SELect loops to remind you what each item in the loop does.

In my Style-Check program the processing of a text file is done in SELect on - END SELect loops. You will find that it contains lines like:

```
=       65 to 90 : REMark Upper case letters
=       97 to 122 : REMark Lower case letters
=       44, 58, 59 : REMark Punctuation marks.
```

REMark statements are particularly helpful in pointer programs where you have to use numbers for your menu items:

```
= -4 : REMark L - Load document
= -5 : REMark I - Information screens
= -6 : REMark S - Save file
```

I was once approached by an ex QL-er who is now a Mac user who asked me if I would be prepared to release the source code of Style-Check to some Mac programmers he knew. In that situation REMark statements would be essential for them to understand the code.

A typical Just Words! program covers 30 A4 pages when printed and that is using a small type face. With a document of such length it is essential that you can quickly find your way around it. As with most printed documents readability is enhanced by a simple rule, "Use plenty of white space".

One of the most useful lines in basic does nothing at all. It is simply the line number followed by a colon:

```
1000 :
```

If you place this line at the end of each procedure or function, you will quickly see where one procedure or function ends and another begins. You can go a stage further and use a colon line to separate different parts within a procedure. The menu loop is on obvious place. Separate the code for each menu item and you will find your menu loop easier to follow.

Another important use of white space is in indentation. Make sure your code is indented in all types of loops (eg. SELect On, REPeat, FOR n =):

```
1000 DEFine PROCedure countdown
1010   REPeat loop
1020     If count=3 : EXIT loop
1030     count = count - I
1040   END REPeat loop
1050 END DEFine
```

If you indent your code, you will find it is much easier to follow, particularly if you are fond of using nested loops.

The good news is that you do not have to indent the code yourself as there are many

programs to do it for you. These include Mark Knight's File Utilities on QUANTA library disc UG13 and a listing by Dilwyn Jones published in QL Today Volume 3 Issue 2 page 54.

The final R is REFERENCE. A long basic program will have numerous variables, procedures and functions, and it is easy to lose sight of the structure of the program, particularly if you have not looked at it for sometime. To understand your program you need more than a simple listing.

You could, of course, manually make a note of every variable and the program's structure, but how much easier it would be if this could be done automatically.

Here is more good news. There is a program that does this. QREF analyses and prints the details of a basic program. It first of all lists every procedure giving the line number where it starts and a list of other procedures called from within it. Next it prints out the structure of the program giving the level at which each procedure is used. Then it prints out every variable, making a note of its type and every line in which it is used with the lines in which it is assigned highlighted. It also prints out every command and function used and the number of times it is used. Finally it gives a warning of items defined but not used and of estimated data space requirements.

Dilwyn Jones introduced this program to me when I first started to write commercial software and I have found it a valuable aid every since. It saved much time when someone reported a bug or a program needed an upgrade.

Happy programming!