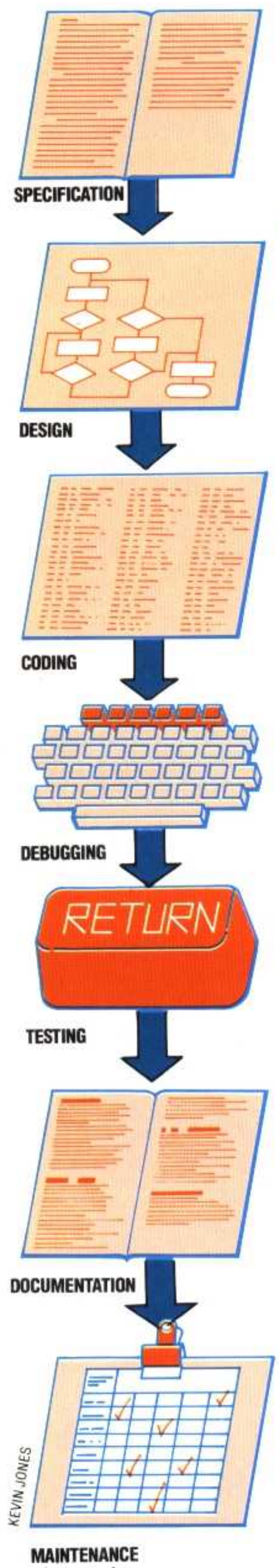
DESIGN SENSE



Design Counts
Observing the rules of good structure is difficult in machine code programming. Developing machine code programs according to the rules of good design is not difficult, however, and pays extra dividends in clarity of design and debugging time saved

In the course so far, we have concentrated on looking at the 6809's instruction set and seeing how a few of these instructions can be put together to form simple routines. However, writing larger, more ambitious programs is a far more complex task. We consider some techniques to give structure to larger Assembly language programs.

We have talked a lot in the course about the benefits of proper program design, modular construction and structured programming in the context of high-level languages. The difficulties of programming, and the benefits of good technique, are greatly magnified at the lower level. In Assembly language, there are usually no convenient control structures, such as BASIC's `WHILE. . . WEND` and `IF. . . THEN. . . ELSE`, to enforce at least some sort of structure on the code. There are also no convenient notations, no data-typing of variables, and, to make it worse, you can expect an Assembly language program to be between six and ten times the size of a high-level program — in terms of the number of instructions. Above all, it is far easier to make errors, and these may have disastrous consequences — it is possible to wipe out all the data on a disk with an error in a single byte. To help make 6809 Assembly language programming less daunting, we consider here the most productive way to approach it.

There's nothing particularly new about structured programming or software engineering: experienced programmers have always known that forethought and clarity of approach were the ground rules for a successful programming style. What makes it seem new and original is the fact that the world of microcomputing has been largely amateur and hobbyist, but it is now becoming both more professional and more appreciative of the professional virtues. Nothing makes this point more clearly or memorably than your first attempt at debugging an undocumented, unstructured, hand-assembled machine code program that you created months ago and put aside. Good design and working methods mean good programming.

STAGES IN PROGRAM DESIGN

• Problem Specification: In this stage, the Assembly language programmer must pay particular attention to the specification of input and output. Often peripheral devices are being controlled directly — especially the keyboard and screen — so the actual signals used must be considered. There may be timing constraints as well. You may not have any convenient routines available that convert the string of bytes that come in or go out into the form in which the program reads the data — for example, converting a string of ASCII characters into a decimal number in binary form. It is important, therefore, to specify not only the form in which the data arises but also the form in which it is required by the rest of the program.

• Program Design: We must now consider the processes that will turn the program's specified input into its specified output. These should be grouped where possible into logically self-contained modules, along with the data that each process requires. There are two main techniques for 'decomposing' a program into modules: *bottom-up*, where you collect a set of what would appear to be useful modules in the context of the program and then try to fit them together; and *top-down*, where the program is successively decomposed into smaller and smaller units, concentrating on the function of each unit rather than how it is to be achieved, until the process cannot usefully be continued. Only at that point do you start considering how each module can be assembled into code.

Bottom-up design has the great advantage of using library modules, which are easy to put together, and the end result is likely to be more efficient in memory usage. The disadvantages are that the program as a whole is likely to prove more difficult to debug and test, and will not be so comprehensible. Top-down design leads to better structured programs, and each stage in the process can be tested separately by means of 'stubs', which are short routines that take the place of as yet unwritten modules by simply accepting input and providing output in the correct form without doing any processing. The disadvantages are that the programs will tend to use more memory and the routines developed are unlikely to have any immediate use elsewhere.

Within each module the data requirements, data structures and algorithms must be specified. A flowchart is useful at this level for representing algorithms, but many people find it much easier to work in a loose kind of high-level language called a pseudo-code. PASCAL is usually used as the basis for this pseudo-code, but there is no reason why BASIC cannot be used. This enables us to design algorithms and data in a way that is familiar to us, and confines the lower-level work to the relatively simple task of translating the algorithm from pseudo-code into Assembly language. This is much easier than trying to design and code in Assembly language at the same time.

• Coding: If the routines have been well designed