Proceedings of The Institute of Acoustics

System Design with Microprocessors

C. Pinches

Leeds University

Introduction

The impact of microprocessors on the measurement based sciences, although substantial, has been greatly reduced by the failure of researchers to adopt a sound engineering approach when tackling applications. Far too many microprocessor based systems have become qualified failures in as much as they were not available on time and did not fulfil the specification to which they were designed. Such catastrophies can, however, be avoided by the application of sound engineering techniques to both software and hardware design.

The Design Process

The typical design procedure for a microprocessor based application is shown in Figure 1. The starting point of any design should be a statement of user requirements. Of all the stages in the design process this is the most often omitted in practice, usually with dire consequences. It is too easy to fall into the trap of assuming that the user requirements are 'too obvious for words' and to feel that the process of committing such requirements to paper is a trivial, avoidable exercise. Unfortunately as we shall see, errors have a nasty habit of growing in cost rapidly as the design process proceeds and a statement of user requirements agreed between the end user and the designer is a key document in eliminating errors as early as possible.

Out of the statement of user requirements a specification can be formulated. This is a much more detailed document than the earlier statement. At this stage the designer should attempt to define the performance of the system in precise, numerical terms. Every parameter of the system should be assigned some numerical values and a tolerance. A designer who at this stage finds it impossible to assign such precise descriptions to the system should pause and ask why? The usual reason for not being able to express the system performance in numerical terms is lack of understanding of the required function of the system. Such gaps in the designers knowledge must not be allowed to propagate into the next critical phase, that of system design. The designer should resist the temptation of ploughing on regardless, and instead move back to the user requirements to find out why a particular parameter cannot be quantified. It is an observed fact that as designers become more experienced the proportion of total project time devoted to these early stages increases. The newcomer to microprocessor design by contrast is often so eager to get to grips with the 'real problem' that these early stages are often skimped on or omitted altogether. The result of this omission, combined with the lack of experience which is inferred by it, often means in practice that a project is doomed <u>before</u> system design begins.

The next stage in the design process, that of system design, is the point at which the designer should move from <u>what</u> is to be done to <u>how</u> it is to be done. Of all the activities in the overall design stage, that of system design can benefit most from experience. It is extremely difficult to provide any guidelines other than those of a most general nature since each problem will present unique difficulties at this stage. In recent years, however, there has been a move, fuelled by every rising project costs, to impose some sort of systematic structure on the design process. Out of a wide range of proposed techniques one, that of 'top-down design' has emerged as the most promising of all strategies. Rather than being a set of detailed rules, top-down design imposes a philosophy on the designer. Broadly speaking the approach is to proceed from a general statement of the problem to be solved to the detailed solution in a number of stages. At each stage the overall problem is broken into a number of sub-problems each dealing with only a portion of the overall function. The process is then repeated taking each sub-problem as a problem in its own right. As the process proceeds the designer is faced with smaller and smaller problems expressed in an ever increasing level of detail. The fact that the increase in level of detail is balanced by a corresponding decrease in the scope of the problem in hand prevents the designer from getting bogged down by excessive attention to detail early in the project. The philosophy of top-down design can, in the case of software be extended to the implementation phase of a project with impressive results. This point will be followed up later in this paper.

One of the key tradeoffs to be examined in the system design phase of a project is that between hardware and, software. The popularity of the microcomputer derives from the fact that despite being a mass produced component with the attendant low cost the detailed function of the device is not determined by the manufacturing process but by the software written to perform a particular function. In the process of system design the designer will often be faced by sub-problems which can either be solved by special-purpose hardware or by software. The aim should always be to use software wherever possible. The main need for purpose build hardware arises from speed requirements in excess of those possible with a software only solution. But the loss of flexibility attendant on this approach should be seriously look at since it may impose severe limitations in future design modifications. Once the system design phase is complete the project divides into two parallel tasks, those of software and hardware design and construction.

Hardware Design

The role of the hardware in a microprocessor system is to provide resources which can be manipulated by software to perform the functions of the system. Because the detailed operation of the hardware is not precisely defined by its physical design and construction it is not unusual to find very similar hardware cropping up in a wide variety of applications. For example all microcomputer applications require memory and a processor. Thus, for a given microprocessor, all applications will exhibit similarities in these two areas. This argument can be extended to a wide range of input/output devices and even specialised devices such as high speed arithmetic processors. Most newcomers to microprocessors are somewhat taken aback when they realise that the much vaunted 'computer on a chip' actually needs to be surrounded by twenty or thirty other integrated circuits before it can perform a useful task. Most true single chip computers are used in applications that require extremely low cost coupled with high volume (e.g. washing machines, pocket calculators etc.) Such processors have in effect lost all trace of flexibility since the software for such devices is 'manufactured in'.

The fact that the so called 'multi chip' microprocessors require substantial supporting hardware has been capitalised on by many firms in the electronics industry who now produce 'board level' microcomputer systems. This approach is of major importance to research workers in areas such as acoustics since it enables hardware design to be accomplished with the minimum of electronic knowledge. Many researchers reject this approach by claiming that these printed circuit boards are more expensive than home made assemblies of integrated circuits.

Such claims are usually based on the user accepting a poor level of performance, doing without 'optional extras' such as documentation and circuit diagrams and holding up the project for three to six months while a board is designed. Even accepting these shortcomings the build it yourself approach is still not cheaper unless you consider your time to be worthless in cash terms. Manufacture of board level microcomputer components has now become established to the point where standards are emerging which enable a user to mix sub-systems from a range of manufacturers. Most standards originated with one manufacturer but, by agreement were generalised and licensed for others. Some have reached the point where they are covered by an international standard (e.g. IEE S100) others are well documented by groups of manufacturers (e.g. INTEL multibus, Pro Log/Mostek STD bus). The vast majority of research users would benefit greatly by the use of such standards. Even where it is essential to design purpose-built hardware this should be compatible with a standard so that only the minimum hardware need be built with the remainder, including processor and memory, bought as commercial items.

Software Design

When designing software for a microprocessor it is important to be aware of the similarities between a microcomputer and a mini or mainframe computer. Of course the key similarity is that they are both computers sharing the characteristics of stored-program control. Thus techniques for programming a large computer, such as the use of high level languages will be useable on the small machines. Early in the 1950's it became recognised that the full potential of computers could not be realised if all programming had to be performed at the machine language level and great efforts were made to design problem oriented languages, starting with FORTRAN which could be automatically translated to machine code. Unfortunately the development of the computer programs needed to perform the translation task was, and still is, a time consuming labour intensive process.

When microprocessors appeared on the scene in the early 1970's the same factors still prevailed and very soon the limitations of machine level programming showed themselves in the form of project delays, cost overruns and severe project management difficulties. In contrast to hardware developments, however, the improvements on the software side have been relatively slow. It still takes several man years to produce an automatic translation or compiler which can convert a high-level problem oriented language into a low-level machine language program. This process has been further complicated by the lack of standardisation in the high level languages developed. The situation is now beginning to clarify and one or two high level languages are emerging as widely supported standards. Perhaps the most widely used and best known of these is PASCAL. One reason for the ever growing popularity of PASCAL is that programs produced using it are comparable in efficiency measured in terms of size and execution time, with programs produced in low level languages and the production of a PASCAL program is at least ten times faster than that obtained using a low level language. Another less obvious benefit of PASCAL is that it directly encourages a top down approach to software development. Although it is inappropriate to go into details here, language facilities such as block structuring, data structures, meaningful variable names and the passing of data to low level functions by value all assist top down design.

Many critics of PASCAL will rightly argue that despite its undenied ability to accelerate the program production schedule there are certain tasks, particularly those associated with high speed operation and interrupts, which cannot be catered for by PASCAL alone. This does not mean that problems involving such tasks cannot be tackled in PASCAL but merely that those portions of the problem specifically associated with such tasks will have to be coded in a low level language. An outstanding example of such approach is given by the UNIX operating system developed at Bell Laboratories. This quite large operating system consists of 10,000 lines of a PASCAL like language called 'c' and 1,000 lines of low level coding. This example has been an outstanding success in terms of maintainability and the time it took to produce. The clear message is to use a high-level langauge wherever possible and don't be put off because a small part of your project cannot be implemented in such a language.

Integration and Testing

This is the point when all of the problems 'designed in' to the system raise their ugly leads. In order to survive the phase of the project it is essential to adopt an orderly approach to the task of modular testing and gradual integration. Some simple statistics illustrate the problem well. If your system comprises six modules, three hardware, three software and the probability of a hardware module working correctly is 70% and that of a software module 60% then the probability of the system functioning correctly first time is $0.7^3 x 0.6^3 x 100\%$ or 7.4%! The message is quite clear to have any chance of success the probability of each module functioning correctly must be substantially 100% before integration. The only way to achieve this is to exhaustively test modules individually before integrating them. This argument provides yet more weight to the case for buying hardware wherever possible. Bought-in hardware is typically 99% reliable due to the manufacturers test procedures. Thus the above example using bought-in hardware would have improved its chance of success to 21% not good but three times better than before. The integration process becomes particularly difficult when problems cannot easily be attributed to either hardware or software. The only method which can be usefully adopted to determine the source of such errors is to use expensive test equipment such as in-circuit emulators which are often supplied as a part of microprocessor development systems. If you do not have access to such equipment you would be well advised to steer clear of systems in which the hardware and software are both 'home made'. At least with commercial board level systems, software becomes the prime suspect.

Summary and Conclusions

Once you have completed the above stages you should be left with a working system At that point you may not wish to know that according to industrial estimates you have expended about 30% of the effort which the project will ultimately require. At least 70% of the total project cost in terms of time and money will be incurred in the maintenance phase after you thought you'd finished the job.

In summary you should recognise that good engineering practice particularly at the start of a project are essential to the ensuing success of that project and that the use of ready made sub-systems and high level languages can ease the design and integration tasks substantially.

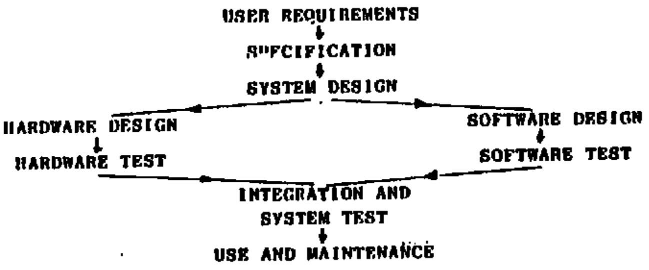

Figure 1
The Design Process