SOFTWIRE

AN INTERACTIVE, COMPUTER-BASED SYSTEM

FOR ENHANCING LEARNING IN ELECTRICAL ENGINEERING,

USING SIMULATION

Prepared by: Mr C.D.Geerdts, MSc Student in the Department of Electrical and Electronic Engineering

Prepared for: The Department of Electrical and Electronic Engineering at the University of Cape Town

23 September 1987

Thesis prepared in partial fulfilment of the requirements for the degree of MSc in Electrical and Electronic Engineering.

For David and Patty

and

Anne-Margaret

ACKNOWLEDGEMENTS

My sincerest thanks and appreciation to Mr J.R. Greene for supervising this dissertation. His ideas and knowledge proved an inspiration throughout.

Thanks to Mark Levin for his help with the electronic aspects of the hardware design, and to my brother, Philip, for his considerable assistance in the final stages.

C.S.I.R. F.R.D. funding was used towards the early parts of this project.

SYNOPSIS

The aim of the dissertation is to explore a method of enhancing learning in the Electrical Engineering curriculum, which effectively exploits the computer.

The different modes used in computer-based learning are discussed and compared, and the conclusion reached that simulation provides the best basis for a learning enhacement system. It has the ability to facilitate demonstration of basic concepts, learning of estimation, modular system design, and the use of models in engineering, and group work. It is a useful complement to laboratory work. It also enhances motivation and interest.

A system is motivated loosely based on the architecture of the analogue computer, but tailored for an educational environment by being interactive, simple-to-use, adaptable and extendable by the tutor, and carrying a wide variety of educationally valuable in-built functions.

The system proposed, **SOFTWIRE**, consists of a software package, a hardware laboratory interface as well as a broad approach to strategy, based on basic Learning Theory. Design issues relating to the hardware, software, and implementation of pedagogic strategy are discussed. Some examples of the use of the system are given. Thus both the broad and specific approach are covered.

Some of the modes of useage discussed are demonstrations, design problems involving the construction of simulations in **SOFTWIRE**'s language, interacting with simulations to experience them and to see the effects of parameter and structure changes, and learning about modelling and its relation to the real world.

Evaluation is discussed, especially in relation to **SOFTWIRE**. Conclusions are drawn, and suggestions made about future development of the **SOFTWIRE** system, as well as relevant trends in computer education.

TABLE OF CONTENTS

GLOSSARY

Some terms and abbreviations used in the text:

SOFTWIRE: the software simulation program developed and described in this dissertation.

CBE: Computer-based-education.
CAL: Computer-assisted-learning.
CBT: Computer-based-training.
CML: Computer-managed-learning.
CAI: Computer-assisted-instruction.
CBL: Computer-based-learning.

ADC: Analogue-to-digital-converter
DAC: Digital-to-analogue-converter

NDPCAL: National Development Programme in Computer Assisted Learning. This is described in Chapter 2.

CHAPTER 1

INTRODUCTION

The **SOFTWIRE** system, which is the focus of this dissertation, arose from a frustration with the ineffectiveness of the present training of Electrical Engineers, in both the theoretical and practical dimensions. Some of the problem areas in the theoretical side regard the imparting of crucial concepts, and the instilling of an effective engineering approach and attitude to the discipline. The problems on the practical side are in providing an adequate level of hands-on experience, and in ensuring this experience does actually prepare students for the reality of their later occupation.

1.1. PROBLEMS IN EDUCATION

1.1.1. A SHORTAGE OF RESOURCES

The largest problem is the limited (and diminishing) availablity of teaching resources. Faculty in the United States, the United Kingdom, South Africa and other countries are on average experiencing a dwindling of funding towards student training. This reduces the lecturer-to-student ratio, as well as the number of competent people willing to lecture. It also reduces money available for equipment.

1.1.2. PROBLEMS IN THE TEACHING OF THEORY AND PRACTICE

In the teaching of theory, lecturers often lecture as a sideline to their research, thus lacking the necessary commitment. There is also very little scope for one-to-one interaction in large classes.

In the laboratory there are problems with the logistics of large classes and equipment. Laboratories are often time-consuming to those setting them up **and** those doing them. This, together with the cost of equipment and demonstrators, results in only a small part of coursework being supplemented by laboratories, and the teaching of theory and practice are often unsynchronised.

During a practical there are countless distractions from the main point of the experiment, coming from wrongly connected circuits, faulty wires and equipment etc. Although this is part of the learning process, it often obscures the more important concepts the experiments were intended to demonstrate. It also wastes the time of the demonstrator supervising the practical, whose primary function of giving skilled assistance is obscured by the need to administer and check equipment.

Modern equipment is also becoming very specialised, and correspondingly expensive, and it is more costly to give classes of students meaningful exposure to state-of-the-art technology.

1.2. THE POTENTIAL OF THE COMPUTER

The computer -and more especially the microcomputer- was considered as the basis of a partial solution to the problems. Its potential merits in education are the source of substantial literature. Further, it has already been integrated into the Electrical Engineering curriculum at many levels, including its use in direct education.

What was sought in teaching (very briefly) was an implementation which could (from a lecturer's point of view):

a) be resource-effective enabling more learning to be achieved at present budgets.
b) have a low initial lecturer training and capital outlay, to prevent an initial barrier to adoption of the scheme.
c) be easily progressively implementable enabling lecturers to pick-and-choose the areas where they want to use the system, with options to phase the system in or out with minimal adjustment, as well as with minimal disruption of their present curriculum.

There are many areas which have potential for exploitation by the microcomputer, including content delivery, student testing and progress monitoring, and the provision of enhanced learning environments and intellectual tools. many modes were examined in which computers could enhance learning, and a number of previous attempts were analysed, regarding their implementation and results. It became apparent that not all of them fulfill the above aims, and nearly all of them have yielded either no or marginal improvement over conventional teaching methods, in spite of the optimism shown by the literature.

This investigation led to the conclusion that many areas should be avoided until significant breakthroughs had been made, but at the same time it led to the forming of a confident (but sober) belief that much could be gained in the use of computers specifically in **simulation mode**.

1.3. THE AIM OF THIS DISSERTATION

To this end a specification for a system (a software package, hardware and guidelines for use) was drawn up. In addition to the previous aims listed, the system was to be designed to fulfill the following objectives.

1. To encourage a modular, functional orientation toward the design of electronic systems.

2. To allow the student to experiment with systems, with maximum transparency. This meant minimising distraction due to the interface with the computer, and simplifying syntactic requirements of the description language, as well as making required interaction as intuitively obvious as possible.

3. To tie together simulation based on idealised component implementation (e.g. traditional analogue computation) and that based on an extended set of functional primitives, in which real-world limitations and complications can be progressively introduced, thus firmly rooting simulation in a real-world context.

4. To provide students with an early acquaintance with a linguistic (net-list) alternative (of growing importance) to graphic-based descriptions such as the block diagram.

5. To provide a vehicle for the rapid creation of demonstrations of basic signal-processing operations and techniques, and the dynamic performance of systems.

6. To provide a simulation language optimised specifically for Electrical Engineering, and an associated system with a hardware laboratory interface, so that simulations could be closely associated with (and progressively substituted by) actual laboratory experiments.

7. To provide a system for laboratory prototyping, in which part of the system could be constructed in hardware, and part simulated by **SOFTWIRE** modules, in an integrated system.

8. To provide an extendable simulation language and system, in which new functional primitives can be incorporated by the user.

Such a system as has been described would then be used to fulfill a number of pedagogical aims. Broadly, these aims would be to contribute in giving students a means of exploring and assimilating some of the mathematical rules which govern electronic systems, and developing an "intuitive" familiarity with these rules. Further, students would be guided in such a way as to develop good design habits, such as a systematic and modular "top-down" approach to design and problem solving, together with the confidence associated with a sense of mastery of the fundamentals, as well as a development of judgment and estimation powers. The student would also be familiar with the use of simulation and modelling in engineering, and the proper relationship that these techniques have to the "real world".

1.4. THE SCOPE OF THIS TEXT

The dissertation will deal with an educational system, rather than a specific product. In this system, the hardware, the software package, the curriculum and the role of the tutor are all important to the effectiveness.

Part 1 will give a backdrop to the main underlying themes of computer learning, simulation and the analogue computer.

Part 2 will discuss hardware and software design in detail.

Part 3 will discuss the pedagogy and application of **SOFTWIRE**.

Part 4 will review what has been achieved and point to the way forward.

A SIMULATION INVOLVING GAUSSIAN NOISE IN PROGRESS ON A **SPERRY PC**, WITH THE USER INTERFACE CONNECTED ON THE LEFT

PART 1

A HISTORICAL BACKDROP

The context, the setting and any other discussion regarding the **SOFTWIRE** system is more meaningful when the setting is fully fleshed out. This setting has an historical aspect (**SOFTWIRE** is seen as a step on a path), and a relational aspect (strategies and choices are seen as related to alternatives).

CHAPTER 2

A BACKGROUND TO COMPUTER-BASED LEARNING

Computer-Based Learning is the term used here to describe the process where a student learns, using a computer. Other names used are computer assisted / aided / based learning / education / instruction / training. CBL, or Computer-Based Learning is preferred here, as it emphasises that the process is BASED around the computer, and sees LEARNING as being more important than TEACHING

2.1. TYPES OF COMPUTER-BASED LEARNING

The use of **SOFTWIRE** as an educational tool can better be understood by placing it within the framework of different modes of useage of the computer in education.

The different modes correspond to different pedagogical aims, and different circumstances. They are:-

Learning About The Computer: The aim is to learn about what the computer can and cannot do and the social implications of computers.

Computer Literacy: Here one learns how to use the computer to perform functions such as word-processing, spreadsheet computaton and using databases. Programming may also be included -especially for students intending to go into computing or science fields.

Using The Computer As An Intellectual Tool: The computer becomes a tool for enabling the learner to extend his intellectual abilities.

Computer Instruction: The computer is used to deliver information and/or aid learning.

Learning Management: Although the computer may not be directly involved in the learning process, it is used to store information on the progress of the student. This may be used to record progress, isolate areas of weakness and strength, and determine the future direction of learning. The computer often does part of the analysis and control itself.

Two of these areas may be further examined, since the aims of those areas correlate with those of **SOFTWIRE**.

The use of the computer as an intellectual tool is an interesting concept. Someone who played a significant role in developing this concept was Samuel Papert. His programming language, **LOGO**, provides a means of using the computer as an intellectual tool.

Intellectual tools are described by Bork as "extenders of the intellect", which "expand the power of our minds" [1]. Just as mechanical tools allow a person to perform physical tasks which would otherwise have been far more difficult or impossible, so an intellectual tool greatly assists one in performing intellectual tasks. This is achieved by the computer in various ways:

Firstly, the computer can keep track of information as one progresses through a task. It thus removes some of the tedious and time-consuming housekeeping chores from a task. Secondly, the computer can do complex mathematical computation quickly and accurately, enabling one to get results rapidly. Thirdly, the computer can provide an environment conducive to learning. Just as France provides a good environment for learning French, so

LOGO, for instance, can provide a good environment for learning about geometry -what Papert calls "Mathland" [2].

Bork lists a number of software genres which can be considered intellectual tools. These include word processors, spreadsheets, paint and graphics programs, simulation languages, and computer conferencing, and **LOGO** [3]. It would probably be appropriate to include arithmetic symbol manipulating programs and T. Dwer's Solo-Mode computing programs [4].

In a similar way, it is hoped that **SOFTWIRE** will provide an environment conducive to learning about concepts related to electronics and systems design. It is also hoped to improve the rate of learning by combining powerful graphics features, with a flexible simulation language, in order to perform the necessary computation-intensive systems simulations, and provide easy input and meaningful output.

The use of the computer for instruction will now be considered. Traditionally this is divided into three areas. The simplest, and the oldest established method is known as "drill-and -practice" and its development is attributed to Suppes [5][6]. Here the computer is used to present a problem to the student, test for the correct answer, then present another problem. It has its major application in school mathematics, for activities such as learning multiplication tables.

The next major area is that of tutorials. This is the area in which most research has been conducted. The computer is used to deliver information to the learner. Complexity ranges from the computer being a "glorified textbook" to a sophisticated system where the student is taught through questions and the computer uses artificial intelligence techniques to make the lesson in as

natural English as possible, and to attempt to use answers to the questions it sets in order to identify problem areas, and to direct the lesson accordingly. This may also be linked to a database and management system. In some ways, the aim is to follow the model of university "tutors" in giving specialised attention to students [7].

Finally, there is the use of simulation. Computer simulation was originally used by scientists to model systems in which the mathematical models involved were computation-intensive. The computer enabled systems or phenomena to be modelled which were previously too complex for hand-calculation analysis. It also enabled less simplistic and more complex models to be used, allowing more confidence to be placed in results. Many simulation languages have been developed. Some techniques are analytical and require mathematical modelling using systems of linear or differential equations, while others model by attempting to resemble as closely as possible the process they are simulating. Some use specialised techniques (such as the finite-element analysis used by **Nostrum**). Some cover a range of applications which allow common techniques. For instance **GPSS** is suited to applications based on queueing theory. Some are very application specific, such as **SPICE** for analysing electronic circuits, **STRESS** for analysing structural strengths and **COGO** for assisting in surveying.

Although simulation languages are used, they are often too slow, and other high-level languages are used when specific applications are required, with occasional use of assembler code too.

The use of simulations in education is the main area of focus in this dissertation. Simulations are used in mainly scientific applications to illustrate concepts, develop problem solving

techniques and explore complex interactions. The simulations used are not normally as complex as those used in research. They are used in a wide variety of applications. Some examples are found in the fields of history, psychology, business and science. In science they are often used to demonstrate phenomena in physics -such as magnetic fields- and chemistry -such as equilibrium in reactions. They often take the form of laboratory experiments, with the computer workstation taking the place of the laboratory, on the basis that "simulations provide opportunities for learning in much the same way as laboratory work while being quicker and more convenient" [8].

This is closely related to the place that **SOFTWIRE** has, as will be shown later. It is thus primarily used in simulation teaching mode, often in conjunction with a tutorial. It is also used in teaching situations sometimes to teach via a specific simulation, and other times to teach by virtue of its being a simulation language. As mentioned earlier, it can also be thought of as a teaching tool.

A final distinction needs to be made between using the computer as a stand-alone (primary) mode of learning, or using it (as an adjunct) to supplement or enhance other learning processes.[9]. The emphasis in **SOFTWIRE** will be very much as an adjunct.

These classifications are made very arbitrarily, and have much overlap, and alternative conceptual schemes are possible; for instance, R. Taylor classifies all computer education in terms of the three main roles played by the computer. These have the computer as tutor (where the computer is programmed by "experts", and then tutors the child, and thereby imparts information) as tool (where the computer is used to provide a useful capability and preserve intellectual energy) and as tutee, (where the

student learns by "teaching" the computer through programming) [10]

L. Hatfield goes to another extreme and offers eight categories under the instruction mode alone [11]. These are:

Practising -of certain elements of thinking and behaving.
Tutoring -introduction, explanations, and then questions involving different teaching branches, feedback, and reinforcement.
Simulating -demonstrating phenomena, exploring functional aspects or applying ideas which are being studied.
Gaming -simulations with a competitive component.
Demonstrating -examples, instances, summaries, counterexamples, illustrations, questions.
Testing -present test items, accept a response, then record and report scores.
Informing -students access and query computer for information.
Communicating -for instance providing word-processing power, with help in spelling, vocabulary and grammar. Also in the use of bulletin boards and data communication.

It is clear that there has been a wide divergence of modes and techniques, and not all have been mentioned here. It will become apparent in PART 3 that **SOFTWIRE** can be used in many different ways, some of which are not easily categorised. However, the above is included to provide an overall, albeit brief, perspective of the computer in education, to facilitate an understanding of the contribution **SOFTWIRE** has to make.

2.2.4. DRILL-AND-PRACTICE

In 1963, at Stanford university, Suppes, Atkinson and Morningstar started pioneering work in the technique of "drill and practice". They were involved in a massive project amongst disadvantaged children. They have claimed to have achieved success at schools and universities [14][15][16]. This mode "is probably the oldest, most widely used, most widely sold, most widely criticised, most widely studied and tested, and most widely imitated" [17]

There is still considerable use of this today, though mainly at the elementary school level.

The next two projects mentioned resulted because "by the end of the 1960's, many of the computer-assisted learning projects which had blossomed forth were beginning to wilt" [18], and there was felt a need to prove that CAL **should** be effective.

2.2.5. PLATO

Another landmark was the **PLATO** system, developed at the university of Illinois in the mid-1970's, and registered by Control Data Corporation (CDC). This system included an authoring language and advanced hardware, such as improved graphics and a touch-screen. The system was evaluated in a project involving five colleges, in five major subjects. The project was heavily funded. It achieved success in results regarding positive achievement in mathematics. Student attitudes towards **PLATO** itself also improved, in general. No other significant improvements were found, and the results were not conclusive regarding achievement or drop-out rates. Some students felt favourable towards the system, while others com-

plained about the constant down-time and impersonal nature of the system.

Regarding **TUTOR**, the authoring language used, more than half the teachers did not use it at all.

This system was purchased in South Africa at Rhodes University, and in a combined Stellenbosch University and the University of the Western Cape venture, after which other South African universities did not follow suit.

This system is still operating today, although many of the applications based on tutorial presentation have been replaced by simpler testing and drill-and-practice useage.

2.2.6. TICCIT

In another project, called **TICCIT**, new approaches to hardware and the use of learning theory were used. This project was aimed at teaching English and Mathematics to undergraduate students. The results of the project also did not indicate any significant changes. It did have a positive impact on student achievement amongst students who completed the course, but the effect was negative regarding completion rates. The faculty response was not very positive, and they were not convinced it had been successful. The evaluation stated not that the project had been successful, but that the field had "potential". The material was not adopted by universities other than those involved in the testing, and little is heard of it today [19], although it is used in isolated projects.

Between the this project and PLATO, $60 000 000 was spent by the United States Government. The overall feeling after this time is expressed by Chambers and Sprecher:

"By the mid to late seventies, disenchantment with CAI had begun to settle over many educators as well as the key funding agencies. PLATO and TICCIT, as indicated, had been found wanting in a number of key respects." [20].

There is no convincing demonstration of its effectiveness in spite of the fact that "CDC have invested $600 million over a period of 20 years in this product, encouraged by the almost missionary zeal of their chairman" [21].

2.2.7. NDPCAL

In Britain, a large computer educational development programme (NDPCAL) was funded by the government between 1973 and 1977 [22]. It cost about 2.5 million Pounds Sterling. There were over 30 projects in many different disciplines, and in many colleges and universities. It used both the tutor and laboratory modes. The laboratory modes were concentrated in higher cognitive skills -problem solving and decision making. Particularly noteworthy were the **CUSC** (Computers in the Undergraduate Science Curriculum) simulations, which allowed experiments to be done in ways and situations which would have been impossible without simulation. Some examples are **Satellite Motion**, where the motion of a launched satellite is simulated for different orbits, **HABER**, which simulates the expensive and complicated process of producing ammonia, **DYE**, where students can inject indicator dyes into simulated bodies, and **OPERON**, where radio-active decay of substances with long half-lives may be time-scaled by the computer to take a few minutes.

The project was directed by R. Hooper, whose final evaluation report stresses the importance laid on the simulation-based

projects (which he refers to as projects within the "revelatory paradigm" [23]), and states that the success of the project was greatest in these projects.

2.3. SOME MAJOR CONTRIBUTORS TO COMPUTER-BASED EDUCATION THEORY

BORK

Alfred Bork is well known for his work. He emphasized graphics and learner-centred lessons. His early work, starting from 1969, was on simulations in physics, and some of these simulations became part of the physics curriculum at the University of California at Irvine. These are regarded as being enriching and allowing experiences students would not normally have. He has now broadened his approach to favour the tutorial approach, developing very costly programmes in a number of subjects, although he firmly believes in a combination of modes being used where appropriate. He sees computer delivery as being the major solution to present education resource-shortage problems.

PAPERT

Seymour Papert is well known for his introduction of **LOGO** as a tool for learning. The interest that this language still inspires suggests that it has been successful as a language for the learning of programming itself (which was not exactly what it is intended for). Studies have shown that **LOGO** has had success in the area it was intended for, which is as an intellectual tool, used by children to aid their cognitive development. It has been shown to be useful in learning procedural thinking.

Papert claims to base his theory on the work of J. Piaget, a psychologist who has studied child learning patterns. Papert's

idea behind the use of the "turtlegraphics" in **LOGO** is to provide the learner with an object (in this case a turtle), which obeys simple (geometric) laws. The student may "play" with the turtle, and so become intuitively familiar with its laws of behaviour, and also have an environment whereby his intellect may be extended through experimentation and problem-solving.

2.4. THE EFFECTIVENESS OF COMPUTERS IN EDUCATION

There has been extensive research done on the use of computers in education, and vast sums of money have been invested in it. Some of the major areas and people concerend have been mentioned -there have been many more.

There was, however, very little evidence to show that significant improvements have justified the money spent. They have not shown a significant improvement in learning, although they have shown a reduction of learning time. There is at most tentative evidence that an improved attitude develops towards computers and towards the subject-matter, and only in some cases.

The general feeling that resulted from the large projects was disenchantment, and although microcomputers lifted that a little, the disenchantment still persists after a decade.

There does seem to be an indication that human interaction **is** important -that one can significantly improve results by the student's having contact with both computers and with tutors. This leads to one key reason for failure in computer education -the immense complexity and subtlety of tutor-student relations, and the role of this in learning processes. The success of the large projects is harder to evaluate, because teachers and lecturers were involved in differing degrees during the testing,

and the extent to which their involvement subtly affected learning is yet another variable in the evaluation. It may have indicated that computer may be effective as aids, but certainly not as primary deliverers of knowledge, as was originally hoped.

One can conclude that the computer still shows potential to be used effectively for learning, but it requires certain stringent conditions to be successful. These conditions are high-quality software, dedication on the part of the tutors, and low cost. Even then, success is not guaranteed. There appear to be further conditions which at present are little understood.

The historical process of bringing computers into education has both created and highlighted more problems than it has solved.

2.5. PROBLEMS IN COMPUTER EDUCATION

If anything, the history of computers in education has culminated in a deep sense of failure. There is a deep sense of failure amongst people who are acquainted with some of the early projects. The failure is evident in the number of large projects which have resulted in material which has failed to "catch on". Evidence of this failure is also shown in the reluctance of large funders to sink more money into projects. One of the few modes which is still used extensively is simulation, and this is mainly used to support what has already been taught.

Many writers have attempted to identify and address problem areas. Some of these problems are discussed by Bork [24], Chambers and Sprecher [25], and O'Shea and Self [26]. These points, together with personal observations, will now be discussed:

2.5.1. DEMOCRACY

Although the previous sections of the chapter give the impression that the price of computers is now within reach of many more people, there still remains the problem that for many educational institutions the computer remains prohibitively expensive. Exploiting the advantages of the computer thus widens the gap between privileged students and others. Attempts to use the computer should alleviate this problem of increased inequality by developing material which allows more than one person to use the computer at once, and also gives the computer an adjunct rather than primary place in education, so that more students can benefit from each computer. The emphasis should also be placed on making any educational hardware and software as cheap as possible. Moreover, there needs to be research into how to introduce the computer into technophobic societies in less-developed regions.

2.5.2. COST

Hardware costs at the moment are on the border of becoming acceptable, and in a few years may be. However, courseware costs are still prohibitive (for good software). These costs will only decrease with large government funding, and a significant increase in the user base to amortize these software costs to an acceptable level. The large budgets of previous projects (which then had little lasting impact) give an idea on the type of financing that would be required to make significant progress.

2.5.3. FEAR OF TECHNOLOGY

A fear of new technology causes many people to irrationally reject the computer. Even in an engineering environment one often finds a resistance to new technology. This arises both from a

fear of what is not known, and a resistance to change -a conservatism which disparages anything new. This is especially significant as the field becomes more learner centred, and hence less scientifically quantifiable than the hardware involved, resulting in more skepticism from the more "empirically-minded" scientific community.

Teachers also fear that they will have their jobs replaced, or be forced into a role they do not want, both in the classroom and in administration (becoming computer-minders).

2.5.4. EXAGGERATED CLAIMS MADE

People involved in computer education research and marketing are under pressure to show positive results. As a result, and also as a result of premature optimism, exaggerated claims have been made about the potential of computers. This has led many to regard new claims with skepticism.

Even with the more objective researchers, there has been optimism based on the seemingly infinite potential of the computer. For instance, as Programmed Instruction was beginning to flounder two decades ago, it was resurrected by the belief that the newly emerging computer could solve the problems of proliferation of choice that seemed to be the main cause of failure. When the limitations of Computer-Assisted Instruction (which had salvaged the Programmed Instruction cause) were likewise pointed out by skeptics, Artificial Intelligence came to the rescue, with claims that (given enough memory and speed) the computer would (with the aid of a knowledge base and adequate rules) so closely model a teacher in the delivery mode, as to be able to replace one (and actually be better, since the computer would emulate a **good** teacher). Over twenty years later, the limits to these well-intentioned predictions have been realised.

2.5.5. INADEQUATE COGNITIVE THEORY

As the previous paragraph states, Cognitive Theory has recently reached an impasse, and Artificial Intelligence research has followed suite, precisely because the latter is based upon the former. As R. Shank says, "The AI people recognise that how people use and represent knowledge is the key issue in the field" [27]. This significant lack of knowledge has resulted in a number of schools with very different approaches, and hence a very shaky basis on which to build any educational theory.

Previous opinion, that people may be modelled by expert systems, is now being challenged by the concept of an intuitive component of the brain, which remembers whole images rather than merely sets of rules. Almost nothing is known about the actual mechanism the brain uses, and, even if it was better known, the computer, and especially one with a Von Neumann (traditional) arhitecture, would be hard pressed to imitate it.

2.5.6. NOT ENOUGH TRAINING

It requires experienced tutors to make optimal use of educational software. Unfortunately, there has not been enough time and money spent on training people to use it, and that is assuming the trainers are training in soundly based concepts. The best example to use here would be in using **LOGO**, since the problems in teaching with this could easily be problem areas in teaching **SOFTWIRE**:

The problem is that teachers have to learn more than a simple technique. They also have to do more than simply become familiar with the software. They actually have to believe in and have confidence in a certain pedagogical philosophy. For instance, in

teaching **LOGO**, students are supposed to learn by directed-discovery, and the use of Turtlegraphics is aimed at this approach. A teacher would ideally show students how to move the turtle forwards and sideways, and then, say, ask them to produce a square on the screen. But a teacher who is not familiar with the concept of self-discovery will simply **show** the pupils beforehand how to do a square, and their role will be to attempt to faithfully copy the code they are given. This then kills the main object of the lesson, of the students discovering anything for themselves. Cases of this happening have actually been reported. The teacher needs more than a lesson on software!

In South Africa, where a shortfall of 50 000 teachers has been predicted by the minister of education by the end of the decade, what hope is there for adequate teacher training in the subtleties of **LOGO**!

2.5.7. QUALITY OF THE CURRICULUM

If good educational software is not backed by a good curriculum which exploits it, it will fail. Up until now it has been common for such software to be tagged-on to other courses, almost as an afterthought. It is important that future devisors of curricula take possible computer use into account with the other major considerations. Eventually it is desirable that "the computer is seen as part of a total learning environment, which also includes the teacher, other students and other educational aids" [28]

2.5.8. FAILURE TO FULLY EXPLOIT THE POTENTIAL

People involved in computer education do not approach it with an open mind, and ask what assets a computer has, and what unique contribution to education each of those assets can make. For

example, many regard a computer as a glorifed textbook or page-turner, and ignore its ability to animate or to interact with the student. Thus, even though they may have already decided in principle to use the computer, they then use it in such a way that they may as well have used alternative resources.

2.5.9. RIVAL INTERESTS

The introduction of computers to education is influenced by a number of constituencies with rival interests. In schools these include pupils, teachers, school policy committees, parents, the communities in which schools are placed and governments. At universities there are students, lecturers, heads of departments, university funding committees, industries and governments. There are also researchers and funding bodies, and then producers of software and publishers outside the educational sector.

Amongst these groups, there may be personal, financial, political, cultural or ideological interests, as well as non-rationally motivated ones. Decisions may also be made for technological, rather than pedagogical reasons.

2.6. PRESENT TRENDS IN COMPUTER-BASED LEARNING

Improvements are being made in the understanding of computer-based learning at many levels. A few of these are mentioned below.

2.6.1. USER CENTRED

Far more attempts have been made to make the focus of the learning the student rather than the technology. In the early days the technology was very restrictive and so required more

attention. Now that the technology is adequate, far more attention has been paid to the study of learning, simultaneously from a cognitive (how the learning process works) and behavioural (how to motivate students) viewpoint.

The implications of this are that more serious developers are producing interactive, individualised and student-centred material with a relatively sound psychological basis and pedagogical strategy (although many psychologists are doubtful as to whether present understanding is helpful or harmful). How this pertains to **SOFTWIRE** is discussed in PART 3.

2.6.2. THE PERSONAL COMPUTER

The establishment of personal computers since the late 1970's has had a significant effect on the trends, as focus has shifted to these devices. The cost of computing is now far more within the reach of many educational institutions for which systems such as **Plato** were not realistically priced. At first, personal computers were disadvantaged by their slowness, their inadequate graphics and their minimal storage. However that is no longer the case, and fairly sophisticated education programs are now implementable. Features such as a student management database are now available.

2.6.3. TECHNOLOGY

The improvement in technology will continue to allow bigger programs, more data storage, faster execution, better graphics, and better reliability, all at increasingly lower cost. However, more significant are the strides that are being made in improving the computer-human interface. These improvements may eventually allow the user to communicate with the computer in intuitively

obvious ways such as natural spoken language and the use of physical gestures such as looking and pointing [29]. Icons and data-management will also be increasingly utilised. In this way a novice will soon be capably communicating with the computer, and will not have the problem that exists at the moment with computer education. That is that each lesson is actually two lessons -the user has to learn how to master the computer before being able to master the subject-matter of the lesson.

2.6.4. ACCESSIBILITY

For reasons of cost, as well as a change in priorities by educational authorities, more students each year have access to computers. This in turn provides for more and better educational materials, as a larger potential market develops. The trend is likely to be self-reinforcing. Some universities in the United States already require every student to have their own microcomputer [30].

2.6.5. INTEGRATION

Not only will people gain experience in producing computer-learning resources, but it is likely that progress will be made in integrating these materials into the rest of the curriculum, and finally actually devising curricula with the computer in mind. Also, at least in developed regions, people will become more familiar with computers in their lives, and using them in the classrooms will become a more natural process.

2.7. POTENTIAL FOR THE COMPUTER IN EDUCATION

There is considerable literature on this topic -but then that has been the case from the start. A few areas will be considered which point the way forward to greater exploitation of hitherto unexploited strengths of the computer in education.

2.7.1. VIDEODISC

The compact disc used in video combines storage of software, video sequences (at any speed), slide images, sound, and a knowledge base. With the computer controlling this, the potential for realistic interactive simulations is increased, with an increase in speed both in setting up and run-time.

2.7.2. ARTIFICIAL INTELLIGENCE

The use of artificial intelligence, even at the present state-of-the-art, has considerable potential in enhancing the quality of interaction of the computer with the student. The storage and speed requirements which previously limited this facility to mainframes have now been met by personal computers.

2.7.3. USER INTERFACE

The conventional use of the keyboard may now be replaced by analogue input devices. Conventional text output may now be substituted by graphic output. More sophisticated menus, with more creative formats, windows and icons also make input more intuitively obvious. Data representation and manipulation techniques make the student more aware of how the work he is doing relates to the whole.

2.8. THE FUTURE FOR SOFTWIRE IN THE LIGHT OF EXPERIENCE GAINED

The first lesson learnt was that of the modes possible for computer-based learning, **simulation** appeared to be the best possible mode for starting a programme in learning about electronics and systems.

The simulation mode allows a later choice of whether the computer will be used in primary or adjunct mode. At the one extreme a simulation may be used by the lecturer to demonstrate a simple phenomenon. This requires a few minutes of preparation in typing in the description of the simulation, and a few minutes during a lecture to demonstrate what has been taught. At the other extreme, the tutor can prepare a complex tutorial, which requires hours of preparation and revision and takes a few hours for the student to work through. This would be the only instruction given in that area. The lecturer can decide at any time what investment to place in the use of it.

Versatility in simulation allows **SOFTWIRE** to compete favourably with other methods of demonstrating many phenomena. In a few key-strokes one has a measuring device, function generator, analyser or control device. The student can thus have a number of experi-ences without the limitation imposed by the cost of the equip-ment. Furthermore, if the lecturer is demonstrating things himself, there is no time wasted in setting up each device -a file is simply loaded from disk.

If tutorials are produced, there are inherent features in **SOFTWIRE** which result in a lecturer producing a better lesson with less work. These are related to the program's being inter-active and user-centred and allowing self-exploration. In a pure tutorial program, the developer has to account for every single possible keypress of the user, and each minute of the tutorial requires extensive preparation. In **SOFTWIRE**, the program checks for many typing errors itself, and more serious errors often result in an obviously wrong simulation result. There is also scope for open-ended exploration, which means that a few lines of tutorial can keep the student busy for a long time.

A program such as **SOFTWIRE** may be distributed cheaply, and with relatively little expenditure in developing it. It may then serve as a motivation for developing a library of application possibilities as different users prepare material and find it to be effective enough to share with other users.

A more thorough comparison of modes, and motivation for simulation, will be conducted in Chapter 3.

Present improvements that would be useful in simulation would be better graphics, faster computers, more memory and improved analogue interfacing. These would help to make the experiential component of simulations more realistic. At present, simulations do play an important role, but the experience wears thin because the simulation is not transparent enough. The best example of a successful simulation in training, resulting from improved hardware, is the flight simulation used to train airline pilots.

2.9. CONCLUSION

This chapter has sought to introduce the reader briefly to the present state of computer-based education, looking at the past and near-future, and indicating where possibilities lie and what issues dominate the field at present. It is hoped that the pessimism with regard to traditional computer delivery methods has been conveyed. These basic understandings are important to understand the reasoning given later in opting for an approach based on simulation.

REFERENCES

1. A. M. Bork, Personal Computers For Education, Harper and Row: New York, 1985, p. 43.

2. S. Papert, MINDSTORMS, The Harvester Press: Great Britain, 1980, p. 6.

3. A. M. Bork, 1985, Chapter 4.

4. T. Dwyer, "The Significance of Solo-mode Computing for Curriculum Design", R. P. Taylor (ed), The Computer In The School: Tutor, Tool, Tutee, Teacher's College Press, 1980, Chapter 6.

5. P. Suppes and M. Morningstar, Computer-Assisted Instruction at Stanford, 1966-68: Data, Models and Evaluation of the Arithmetic Programs, Academic Press, 1972.

6. R. P. Taylor (ed), "Patrick Suppes", The Computer In The School: Tutor, Tool, Tutee, Teacher's College Press, 1980, p. 213.

7. A. Bork, 1985, p. 62.

8. K. Tait, "The Study Station Concept In Computer-Based Learning", R. F. Smith (ed), University Computing, March 1987, p. 25.

9. J. A. Chambers and W. Sprecher, Computer-Assisted Instruction. Its use in the classroom., Prentice-Hall: New Jersey, 1983, p.

10. R. Taylor (ed), The Computer In The School: Tutor, Tool, Tutee, Teacher's College Press, 1980, Chapter 1.

11. L. L. Hatfield, "Toward Comprehensive Instructional Computing in Mathematics", V.P. Hansen (1984 yearbook editor) and M.J. Zweng (General Editor), Computers in Mathematics Education 1984 yearbook, National Council Of Teachers Of Mathematics (USA), p. 3,4.

12. Jack A. Chambers and Jerry W. Sprecher, p. 6.

13. Jack A. Chambers and Jerry W. Sprecher, p. 7.

14. Jack A. Chambers and Jerry W. Sprecher, p. 7.

15. P. Suppes and M. Morningstar, Computer-Assisted Instruction at Stanford, 1966-68: Data, Models and Evaluation of the Arithmetic Programs, Academic Press, 1972.

16. R. P. Taylor (ed), "Patrick Suppes", The Computer In The School: Tutor, Tool, Tutee, Teacher's College Press, 1980, p. 213.

17. A. Bork, 1985, p. 63.

18. T. O'Shea and J. Self, Learning and Teaching With Computers, The Harvester Press: Great Britain, 1983, p. 86.

19. W. Chambers and J. Sprecher, 1983, p. 13.

20. W. Chambers and J. Sprecher, 1983, p. 17

21. G. Beech, Computer Based Learning, Sigma Technical Press,

22. H. I. Ellington, E. Addinall and F. Percival, Games and Simulations In Science Education, Kogan Page Limited: London, 1981, p. 77.

23. R. H. Hooper, National Development Programme in Computer Assisted Learning, Final Report of the Director, Council For Education Technology (NDPCAL), 1977, p. 38-39.

24. A. Bork, 1985, p. 4.

25. J. Chambers and J. Sprecher, 1983, p. 17.

26. T. O'Shea and J. Self, 1983, p.217-222.

27. R. C. Schank et al., "Panel on Natural Language Processing", International Journal on Computation and Artificial Intelligence, Proceedings, 1977, p. 107-108.

28. J. Fox, "Approaches to computer-assisted reading", University Computing, March 1987, p. 37.

29. R. A. Bolt, The Human Interface, Lifetime Publications, 1984.

30. D. Osgood, "A Computer On Every Desk", Byte, June 1984.

CHAPTER 3

THE HISTORY, NATURE AND ADVANTAGES OF SIMULATION

3.1. WHAT IS SIMULATION

Many people have attempted to define simulation. They have come from an engineering, industrial, mathematical, military or business background, as well as from economics, social sciences and other perspectives. There have also been attempts to define it from an educational point of view. Some of these educationally related viewpoints will be considered, although even within this framework there are variations.

H. Guetzkow describes a simulation as "an operating representation of central features of reality" [1]. J. Bloomer has endorsed this description [2].

This is reinforced by H. I. Ellington et al. [3] who again stress it must be a **real** situation, and it must be **operational**, therefore it is an ongoing process -not static. They also distinguish it (in education) from games, which are similar, but are characterised by the fact that they have both overt competition and rules. Simulations are also distinguished from case studies, which represent an examination of a real-life or simulated situation to illustrate special and/or general characteristics of a situation or phenomenon. The relationship between simulation, games and case-studies is represented by a three-way Venn diagram: each has overlapping and non-overlapping areas with the other two. This gives seven possible different modes -three pure and four hybrid. For instance, it is possible to have a

case-study taken from a simulation, or a game based on a simulation. They are discussing this only from an educational point of view.

In M. D. Garvey's words, "a simulation is the all-inclusive term which contains those activities which produce artificial environments or which provide artificial experiences for the participants in the activity." [4]. Again, the concept of environments and experiences is important in education.

J. L. Taylor and K. R. Carter describe simulation as something which "delineates a range of dynamic representations which employ substitute elements to replace real-world components" [5]. The significance for **SOFTWIRE**, which often models devices actually called "components" is clear.

P.A. Twelker and K. Layden are very succinct in describing simulations as 'simplified reality' [6]. K. Jones is even more succinct, saying that "simulations are reality". It is important (he argues) to leave out the 'simplified', because this implies that simulations are not complex. He says that for educational purposes it is better to err on the side of simplicity, and to extract what it is about the simulation that is educationally useful. He goes on to say "It is concepts that matter: the concepts of reality and power are two key issues in understanding simulations." [7]. The person operating the simulation is actually learning something real, and also wields power in controlling the simulation.

Other key aspects Jones describes are that in a simulation "the participants are on the inside, with the powers, duties and responsibilities for shaping events... it is not a static event." [8]. This gives students far more responsibility and involvement than they might otherwise get.

R. A. Sparkes describes the use of simulations in education as being a mathematical process [9], and stresses the importance of being aware that students are discovering a model of real-world phenomena, not the actual phenomena themselves. The types of simulations he describes are very similar to **SOFTWIRE**, with graphical or numerical display of results being a feature. This distinction between a model and the real-world is something key to the use of **SOFTWIRE**.

To provide an overview of more general terms: H. D. Naetscher categorizes all simulations into **physiological simulations**, in which animals may imitate other animals, or man imitates an activity without technical aid. This may be acting, for training, or it may be to assist in choosing a vocation. **Technical simulations** are those done using technical hardware -usually computers. They form the domain of which **SOFTWIRE** is a part. Then there are **conglomerate simulations** which combine the first two types [10]. He isolates "imitative simulation" as the use of simulation in demonstrations [11]. Again, this is an area where the application for **SOFTWIRE** will be picked up on later.

R. J. Ord-Smith and J. Stephensen state that "simulation is the technique by which understanding of the behaviour of a physical system is obtained by making measurements or observations of the behaviour of a model representing that system." They note that "the word model is closely related in meaning to the word analogue" [12]. This has implications later when the analogue computer is discussed as a basis for **SOFTWIRE**.

Continuing on the topic of computers, T. Naylor sees a computer simulation as being "a numerical technique for conducting experiments with certain types of mathematical models which

describe the behaviour of a complex system on a digital computer over extended periods of time." [13].

Each definition or description highlights only a fraction of the total picture of what simulation is. The best definition of simulation is governed by the purpose for which the simulation is required. However, this discussion will not try to add to the list, but will rather take key points applicable to **SOFTWIRE**.

a) The simulation is based on a mathematical model of real-life processes.

b) The model may be simplified in cases, to make the simulation possible, and to aid learning. The extent of its complexity may determine how faithfully it simulates the mother process.

c) The simulation may take on many forms, it may be a board game or card game, a large hardware emulation device, a computer program, a small analogue device, or many other implementations.

d) There are many purposes for simulation. It may be for calculation (a slide rule) or observation, for training, or for demonstration. It may also be used for "para-feedback"[1] . It may be used in a "what-if?" situation to assess alternatives.

d) The simulation may be continuous or discrete, in real time or not.

[1] A word (coined purely for this discussion) to describe simulation for control, where continuous feedback is obtained not from the output of the system being controlled, but by a simulation of it -for instance a thermal device simulates an induction motor in a protection relay

3.2.A BRIEF HISTORY

3.2.1. GENERAL USE IN EDUCATION

This account of the history of simulation must be supplemented by the discussion on the history of the analogue computer in the next chapter.

Ellington et al. [14] give a good summary of some of the more important features of simulation:-

They claim that the application of simulation (in its broadest sense) has been around for 1000's of years, but that the application to education is recent. Its first serious use was in military simulations in the late 18th century. Apart from chess, which came from Persia or India in the 6th Century as a supposed game for training war strategy, Prussia used more realistic simulations in both games (for training officers) and field situations (for training troops). This started in 1798. About 150 years later, in the mid-1950's, simulation spread to business training applications. In 1956 the first business game was introduced to help bridge the gap between theoretical business training and the real-world situation.

In the 1960's training by simulation spread to tertiary level education. An early development was in 1962: training teachers in a simulated school. In the late 1960's, simulations were common in Europe and the USA in the social sciences. During the 1970's it spread to many fields, and teaching in science has progressed from 1970 onwards.

3.2.2. HISTORY OF SIMULATION IN SCIENCE EDUCATION

It is interesting to follow Ellington's account of simulation more specifically in science teaching [15]:-

A series of board and card games were introduced between 1970 and 1975. **Circuitron** was a game for up to lower tertiary level and was a board game designed as a supplement to conventional experimental work on electrical circuits. The idea was to join up circuits in different configurations. It was published by **Griffin and George** in 1972, and received a thorough evaluation. The results were encouraging regarding gain in knowledge and student attitude. By 1972 there was a variety of work being done on computer simulations.

3.2.3. HISTORY OF SIMULATION IN ELECTRICAL ENGINEERING

Although there is extensive literature on the use of simulation in chemistry and physics, and it has been used in teaching physics-based electronics such as semiconductor theory, capacitor discharge and vacuum tube-conduction, there has been a gap in its use in electronics at a tertiary level. There have been a number of simulations based on power electrical work and alternative energy sources, with more emphasis on the issues involved than the technical aspects. Circuit simulation programs exist, such as **SPICE**, which operate using nodal analysis, but these are computation-intensive, and are thus disadvantaged in educational simulations, which are better performed with interactive terminals and thus a level of activity on the screen. There is also the program **SOPHIE**, which is used in teaching circuit troubleshooting [16]. This uses an artificial intelligence strategy to guide the student in approaching the problems it sets. It has a disadvantage, though, in having no graphic output.

3.3.A MOTIVATION FOR THE USE OF SIMULATION IN EDUCATION

H. D. Naetscher cites J. Nowak as stating in 1973 that computer simulations can be compared in importance with the steam-engine, the aeroplane and the atomic bomb. Naetscher himself thinks this is an over-estimation [17], and the present writer would similarly not be as optimistic, but the point is clear -that simulations will continue to make an increasing impact on our lives, and education will be an area where it is particularly useful.

3.3.1.THEIR DE FACTO IMPORTANCE

One of the biggest motivations for the use of simulation in education is that it has already gained its own momentum. People of different persuasions and for different reasons have decided it works for them. Educational institutions are often slow to adopt new ideas, but simulations are entrenched in spite of this. Industry and commerce is quick to adopt a technology that they are assured yields the results they want, and they have welcomed simulation. In these fields, business simulators are used by business schools and banks alike, and training is provided in companies such as airlines and those which require skilled controllers -oil refineries and chemical and power plants. The military and space programs are known for their position at the technical edge, and they adopt simulation extensively.

Bork endorses the willingness of faculty to adopt simulation packages in tertiary institutions, and ascribes this to their familiarity with the concept of simulation [18]. This may be the case, and is itself a good reason to use simulation.

Computers have been a major factor in the proliferation of simulations, since without them, most complex modern simulations would be impossible. Microcomputers have enhanced the role of computers. Simple mathematical models may now be implemented easily. Spreadsheet programs such as **VISICALC** produced by Visicorp, and implemented on the **Apple II** (now superceded by programs such as **Lotus 123**), made simulations a way of life to many businesses and tertiary education institutions teaching business. Spreadsheets have also been used succesfully in engineering training.

3.3.2.SIMULATION FROM A "LEARNING THEORY" PERSPECTIVE

In Part 3, the types of learning will be covered in full, and the advantages will become more clear then. For the moment, it is only possible to mention briefly these areas:

a) discovery learning. Simulation lends itself to what R. Sparkes calls "discovery learning" [19]. This is open-ended investigation, where the student is left to discover the principles for himself or herself. It is not "teaching", but pointing along a specific path. There is no guarantee that learning will take place -but experience (according to Sparkes) shows that it does. Learning is taken to involve not only facts, but insight too.

b) enhancing the curriculum, and integrating with other methods and tools. For instance, using the computer for demonstrations, or for reinforcing concepts that were covered in previous lectures.

c) developing an understanding of the relationship between "real world" experiences and abstract concepts. The student often uses a model to aid in understanding or calculation, but must then know the weaknesses of that model (such as the simplifications that have been made) and also the proper relationship that the model has to the real situation.

d) allowing the student to control the lesson. The aspect of student control and responsibility is something easily accomplished in an interactive simulation on a microcomputer.

e) teaching estimation and judgment. These are far more easily taught in a dynamic interactive environment, where estimation and judgment are immediately and directly related to their consequences in the results.

f) allowing students to develop confidence. The marginal cost of each simulation is negligible, and the student is free to make mistakes[2]. This removes the sense of inhibition that students have in trying something new, because the consequences are minimised.

g) allowing self-pacing. Students proceed at their own learning pace, and can cope with this speed. Since all the input is not from one lecturer, but also from the computer, speed differentiation is possible.

h) providing immediate feedback (reinforcement). The result of the simulation can be strongly related to the input.

i) providing a basis for group work. This is not so much inherent in the simulation, as in the problem-solving mode that simulation allows. **SOFTWIRE** is even more conducive to this, as it allows a modular approach to problem-solving, which is a crucial part of team design.

j) developing intuitive skills. Simulations have the potential for giving dynamic feedback to a student, rather than the static feedback of tutorials. For instance, the student can be continuously varying an input, and watching a resultant output. They may also hold latent information, which is only revealed as the student probes for it. In this way a student may develop intuitive skills, just as he would in a real-life situation.

[2]Although some argue that this gives a wrong impression of the real world, it is more likely that a student will benefit from realising his abilities, and will be able to adapt later accordingly when the stakes are higher.

k) practising system and modular design. This may not apply to all simulations, but is possible with **SOFTWIRE**, which forces a modular approach by the way data is input.

This is only one example of the areas where simulation is useful. A strong motivation for the use of computer simulation in teaching science is given by H. I. Ellington, E. Addinall and F. Percival [20]. They list nine areas in which simulation aids learning:

1. Simulations are seen as a highly versatile and flexible medium for achieving a wide range of educational aims and objectives in both cognitive, affective and psychomotor domains. "It has been found that gaming and simulation techniques are ... particularly useful for achieving high-level cognitive objectives relating to such things as analysis, synthesis and evaluation and also for achieving affective objectives of all types". They are recommended as a complement to front-line teaching, for reinforcement and for demonstrating applications of relevance.

2. Compared to a real-life situation, a simulation is more amenable to tailoring -to meet the needs of the exercise. Single real-life situations rarely include all the features that an educator wishes to include in an exercise, whereas in a simulation they can be built in. Real-life situations are often too complex, and simplifications can be simulated which reduce complexity to manageable proportions.

3 "Research indicates that well-designed games and simulations can achieve positive transfer of learning -the ability of participants to apply skills acquired during the exercise in other situations."

4. "In many cases, games and simulations constitute a vehicle whereby participants can use and develop their initiative and powers of creative thought". This is more useful if the course includes divergent thought processes.

5. "Apart from their purely cognitive, content-related outcomes, many games and simulations help foster a wide range of non-cognitive skills (such as decision making, communication and inter-personal skills) and desirable attitudinal traits (such as willingness to listen to other people's points of view or appreciate that most problems can be viewed in a number of different ways). Indeed, some workers believe that this is the area in which games and simulations are capable of making their most valuable contribution to education. Exercises that involve interaction between the participants are especially effective in this regard." (This becomes especially relevant as engineering work becomes more team-orientated).

6. Student involvement is normally very high (this is specially helpful for less able students) and thus extremely enjoyable.

7. The competitive element increases motivation, whether overt or latent. (There is an exercise in **SOFTWIRE** given later, called "the module game" which neatly exploits this aspect)

8. If simulations have a basis in more than one discipline, they help integrate such disciplines. (With **SOFTWIRE** this can be applied to sub-disciplines. For instance, experiments can combine measurement and control, or measurement and signal processing, or analogue and digital).

9. If multi-disciplinary, they require experts in different fields to work together efficiently and harmoniously. Interpersonal skills will be useful later in life. (This is an area where engineers often need to develop skills, as their job begins to contain less technical component, and more interpersonal).

3.3.3. SIMULATED LABORATORY AND EXPERIMENTS

There is also real practical significance in simulations of experiments. Various authors discuss the situations which motivate the use of simulation [21][22][23][24]. It is most useful when real situations (usually experiments) are:

a) impossible (e.g.-ve gravity).
b) very difficult to do satisfactorily.
c) not accessible.
d) time consuming (to prepare or complete).
e) dangerous (e.g. radioactive or toxic materials).
f) too expensive.
g) too slow (e.g. plant growth).
h) too rapid (e.g. an explosion).

There are many advantages of using simulations to replace or enhance experiments. S. Engh and K. Ratzlaff give a detailed account of the advantages. They say that laboratory simulation automates some of the tedious and less instructive tasks, produces directly printed tables or graphs of results (which means the output is easier to interpret and faithfully recorded) and can aid in interpreting or displaying results (which means there is more reinforcement, and that spurious and suspicious

values can be re-checked). Without this, recorded values may be in error, and it may only be found that the experiment requires re-doing after the data has been processed elsewhere.

The computer can also be used to replace much of the electronics portion of an instrument. The cost of an entire computer can be comparable to the cost of just the signal processor of some instruments. Such replacements are now common in the current generation of commercial instrumentation. Microprocessor-controlled equipment is inferior in education, however, because it isolates the student from the data entering at the transducer. In other words, transparency is required in most applications, in which measuring is the goal, but may be a hindrance in education, where learning **about** measurement is the goal. With computer manipulation at the control of the student, the student has the flexibility to program all the parameters and not be isolated. i.e calibration can be done. Also software enhancements can be added on later at will [25.].

The use of instrument simulation is strongly advocated by H. Wohltjen and R. Dessy:-

"The advent of inexpensive microprocessors and microcomputers is permitting many instructional laboratories to add interfacing experiments to their regular curriculum, either through new courses, or as additions to exisiting laboratories."

Students must write programs which "emphasize sampling frequency (Nyquist Frequency), digital smoothing techniques, and data correction as well as the routines necessary to present the corrected and/or condensed data in suitable form to the user"

"The output signal level can be altered, as well as the amount of white-noise superimposed on the signal. Typical experiments that can be conducted using the instrument simulator involve Nyquist Frequency experiments, or digital filtering techniques such as box-car averaging, weighted digital filtering, and ensemble averaging. Peak-picking algorithms are easily explored. Auto-correlation and cross-correlation experiments are also possible." [26].

K. J. Johnson motivates simulation specifically in the chemistry curriculum:-

"The least expensive way to enrich the undergraduate chemistry curriculum is with simulation and data reduction programs. These programs are not necessarily interactive and do not have the same objectives and software requirements of CAI programs." [27].

Johnson also lists specific advantages of simulations [28]:

1. Less time is required for simulations than for CAI programs.
2. They can enrich labs by allowing the simulation to be done first.
3. They allow "stimulation and challenging of highly motivated and well-prepared students", who would otherwise be bored because the lecture is usually pitched to the middle of the class.
4. They can allow self-grading. Students can carry on working to get better results. Having optional work with extra marks often motivates students to put more work into the course than they normally would.

3.3.4. MICROWORLDS

S. Papert has an interesting concept, which builds on the advantages of computers in learning. He develops the idea of "microworlds" [29], which are "incubators for knowledge". One can create microworlds, where features are removed which detract from what is being taught. An example may be taken from Newtonian physics: a body with a certain velocity should continue at that velocity unless a force acts on it. However, to an observer in nature, bodies do not appear to continue at their original velocity, because of friction. On the computer, the effect of friction can be removed at first -to demonstrate Newton's First Law of motion, and then later included, when the law is understood.

Simulations are very closely associated with these "microworlds". As A. Bork comments, "an extension of laboratory-like simulations, controllable-world simulations provide a richer collection of phenomena, beyond what is practical in ordinary laboratories." [30].

3.3.5. COMPUTER AS TUTEE

In some of the applications, the lecturer uses **SOFTWIRE** to create simulations, but in many of the possible ways that **SOFTWIRE** may be used, the student uses it as a simulation language to create their own simulations. In this way the system is being used in a way that S. Papert made famous -computer as "tutee". This is of significant pedagogical value, as will also be discussed later, in Part 3. However, the ability to use this mode is now mentioned as being an important motivation for the use of simulation in conjunction with a good description language.

3.3.6. GAMING

Simulations allow the implementation of competitive games. Competition increases student motivation considerably. A game called "the module game", uses this feature in **SOFTWIRE**. This will be discussed in the chapter on **USES OF SOFTWIRE**.

3.3.7. EVALUATIONS

A problem with education using computers is the difficulties involved in doing evaluations. Hence, there is more conjecture and hypothesis than good evidence. However, there are some results to point to:-

Percival, Ellington and Addinall recount how Norman Reid did systematic evaluation of games, case-studies and simulations at Glasgow university in 1978, and recorded both cognitive and non-cognitive results, with cognitive development and long-term retention [31].

They also discuss the 'Science in Society' evaluation on the 'Energy' section. This was run by the Association for Science Education. It was tested in the United Kingdom in 50 High Schools, in 1978-1979, using both experimental and control groups. The results were highly favourable for both teachers and students, regarding attitude changes, motivation and what was learnt [32].

In the next section, Richard Hooper's account of the results of the extensive testing of the **NDPCAL** programme in the United Kingdom will be related.

3.3.8. COMPARISON WITH OTHER METHODS

The previous chapter hopefully made clear the sense of failure that was experienced in many spheres of computer-based education. However there has been evidence that the mode of simulation may have been more successful, and a motivation has been put forward as to why this is so. It is important to compare the different methods, and see why simulation should fare better.

Many authors firmly believe that simulations are the learning activity with the most potential. For instance, G. Orwig states "that simulation is potentially the most powerful teaching technique available on the microcomputer. It is also the most complex, requiring detailed subject knowledge and mathematical abilities." [33].

Ellington, Addinall and Percival are firm believers in the use of computer simulation in teaching science, compared with other methods. They state:

"It seems doubtful that computer-assisted learning will ever flourish if it is seen merely as an expensive alternative to the classroom teacher. If it is to become an established weapon in our educational armoury, it is vital that we identify and investigate uses to which the computer is uniquely suited. The computerised laboratory certainly seems to be one such area." [34].

G. Hubbard compares CAL in the light of the proliferation of choice:

"Earlier I referred to the proliferation of choice when one tries to give a realistic simulation of the complex world. Exponents of

gaming and simulation were quick to see the advantages of the computer as a way of deploying this range of choices and, suddenly, the computer -particularly the microcomputer- has become a passport to respectability... but more significantly, those who have, for whatever reason, started to explore the applications of the microcomputer in the classroom have found that it is at its most effective not as a way of presenting rigidly sequenced material -which makes it no more than an electronic page-turner -but in its role which Richard Hooper, the Director of the National Development Programme in Computer Assisted Learning (NDPCAL) described as 'computer as laboratory'." [35].

Since R. Hooper has been quoted, it would be interesting to delve further into his impressions. His report is the final evaluation of the massive NDPCAL project, which was described in the previous chapter.

"Many CAL applications involve the computer performing complex tasks, for example simulation, that cannot be done by other media (including the live teacher), and concern the qualitative, rather than quantitative, improvement of teaching." [36].

Hooper himself cites various authoritative sources as backing up this view that pure instruction by computers is no more than equal in effectiveness to other methods, and that the only area which has convincingly positive results is that of 'enrichment' modes (such as graphic demonstrations and simulations) [37]. However there is a warning not to oversimplify the categorization of different pedagogical modes. The CALCHEM project, for instance, is said by P. Ayscough to be "enhanced tutorial prog rams, providing a number of alternative dialogues, the routing through which may be determined by the student's current and

previous responses, and which may make use of facilities for simulation, calculation, graph plotting, etc, within the tutorial sequence" [38].

Hooper's final recommendations to the British government regarding simulation were that "In the years ahead, the curriculum in many subjects will reduce the traditional emphasis on content and factual transmission for which CAL is not a cost-effective medium anyway, in favour of problem-solving skills for which CAL seems well suited." [39].

Hooper also endorses the views of J. F. Rockhart and S. Morton, who state that "CAI has been overemphasized. In the areas of learning where it is applicable, other technologies dominate along the relevant dimensions... The primary clear target of opportunity for the computer in Higher Education is in 'enrichment activities'. For almost all kinds of material, problem-solving, games and simulation can provide the learner with better ways of integrating and testing the knowledge he has acquired than other available technologies... It follows that the impact of CAL will be for the most part adding to rather than replacing current learning mechanisms." [40].

3.3.9. THE CONTINUED USE OF OTHER MODES

If simulation is such a powerful learning device, and there has been such difficulty in the drill-and-practice and tutorial modes, then one must ask why there is still such persistance in them.

It seems that with drill-and-practice, the techniques do work, but the computer can only teach in a limited domain. It can teach a few rules that a beginner needs to start off in a specific

area. After that it is no longer appropriate. S. Papert provides a possible reason why the area has attracted so much interest:

"Drill and practice applications are predictable, simple to describe, efficient in use of the machine's resources. So the best engineering talent goes into the development of computer systems that are biased to favor this kind of application" [41].

If a teacher has decided that the class will do rote learning of multiplication tables (the usefulness of this is also questionable, but assume for the discussion that it has been decided), then it may be that drill-and-practice is superior to the teacher in providing motivation for this, and having patience. Students will concentrate for longer and be given individual answers. However, if a wrong answer is given, and the computer attempts to analyse what is wrong, the programming task becomes far more complex. In simple addition alone, there may be about thirty different common errors that the child could make, as well as a host of uncommon errors. The wrong answer given may not give a simple clue as to what wrong-thinking resulted in that answer. If the subject material is more complex, the possibilities proliferate. Thus, there may be a place for this way of teaching in the acquisition of simple rules by practice, but in acquiring complex or less specified skills, there is doubt as to its effectiveness.

H. L. Dreyfus and S. E. Dreyfus comment on drill-and-practice, in the light of their belief that there is far more to thinking than simply "rule-based" analysis:

"Until the traditional philosopher or epistemologist comes up with some argument or evidence that skills are implicit theories, educators would do well to side with the phenomena. If one does

stick to the phenomena, one can understand why computer-assisted instruction (CAI) has worked well for drill but that when the tutor needs to understand the domain and the tutee, work so far has been disappointing...there is nothing wrong with drill-and-practice ...[they are] excellent ways to begin to acquire skills in appropriate domains -but only to start with" [42].

Tutorials are also still firmly entrenched regarding development of new programs. This is partly because of the tradition of "programmed instruction" from which these derived, partly because of the enormous effort that Alfred Bork has put into the field, as well as the funding he has acquired through propagation of his programs, and partly because of the optimism that exists because it is linked to artificial intelligence, where, until recently, people have seen all problems as being surmountable purely by providing computer tutorials with more rules and larger knowledge-bases. However, this is changing. As part of his criticism of the present threat of technology to cultural education, R. Sardello writes:

"What is called 'computer-assisted instruction' also does not pose a real threat to culture. Teaching machines or programmed instruction have already shown themselves to be dismal failures, precisely because they turn the learner into a mechanism, who duly responds with frustration and boredom" [43].

Teachers are often attracted initially by the prospect of material delivery that is far more attention-capturing than a textbook, and more personalised than a teacher with a large class. However, it seems that once the initial enthusiasm wears off, and certain initial hopes are not realised, the computer material has been packed away, and the class resorts to traditional methods.

In this, CAI is not unique. Early personal computer users had electronic notepads, diaries, recipe-books, and phone directories all at the push of a few keystrokes, and home banking and shopping were exciting developments. These things still exist, and do effectively what they are meant to do, but there is still an enormous affinity to the pen-and-paper approach, and an attraction for its concreteness, immediateness and human-ness, and many programs which were purchased now also share back-room storage space with education programs.

It is worth noting here what will be taken up later, that **SOFTWIRE** does not dispense with the use of pen and paper, but actually encourages it as part of the learner's involvement, using a medium which he is already accustomed to thinking with. This is done by requiring reports written during simulations to be handed in, mainly containing diagrams and responses to ensure the student has applied his mind during the lesson.

This discussion should not give the impression that all modes except simulation are spurned. It is firmly believed that the computer will ultimately be used in many different ways in learning. The aim of the evaluation and motivation was more to decide what type of system would be the most cost-effective and useful system to introduce to an electronics curriculum **now**, and to develop this system given the limited time and money available.

3.3.10. ADAPTING FOR EDUCATION

Naetscher provides a transition from this motivation for the use of simulation, to a provisor:

"In the pedagogical field technical simulation still stands on new ground. The lack of suitable models is partly due to the fact that the precise formulation of technical simulation in its pure form does not seem to be appropriate for pedagogical application" [44].

The need to adapt simulation for educational use in the specific area of Electrical Engineering is a major concern of the remainder of this dissertation!

REFERENCES

1. H. Guetzkow, Simulation In International Relations, Prentice-Hall, 1963, Chapter 1.

2. J. Bloomer, "What have simulations and games to do with programmed learning and educational technology?", Programmed Learning and Educational Technology, Vol 10 n.4, 1973, p. 224-234.

3. H. I. Ellington, E. Addinall and F. Percival, Games and Simulations In Science Education, Kogan Page Limited: London, 1981, Chapter 1.

4. M. D. Garvey, "Simulation: a catalogue of judgments, findings and hunches", in Tansey, P. J. (ed) Educational Aspects of Simulations, McGraw-Hill: London, 1971.

5. J. L. Taylor and K. R. Carter, "A decade of instructional simulation in urban and regional studies", in Armstrong, R. H. R. and Taylor, J. L. (eds), Instructional Simulating Systems In Higher Education, Cambridge Institute of Education: Cambridge, 1970.

6. P. A. Twelker and K. Layden, "A basic reference shelf on simulations and gaming", in Zuckerman, D. W. and Horn, R. E. (eds), The Guide to Simulations/Games for Education and Training, Information Resources Inc: Lexington, 1973.

7. K. Jones, SIMULATIONS: A Handbook For Teachers, Kogan Page/Nichols Publishing Company, 1980, p.20 (for both quotes).

8. K. Jones, 1980, p. 10.

9. R. A. Sparkes, Microcomputers in Science Teaching, Hutchinson, 1981, p. 105.

10 H. D. Naetscher, "The incorporation of computer simulation into the system of simulation", in B. Hollinshead and M. Yorke (eds) Perspectives on Academic Gaming and Simulation, Simulation and Games: The Real and Ideal, 1980, p. 93

11 H. D. Naetscher, "The incorporation of computer simulation into the system of simulation", 1980, p. 97

12. R. J. Ord-Smith and J. Stephensen, Computer Simulation of Continuous Systems, Cambridge University Press, 1975, p. 3.

13. T. Naylor, Computer Simulation Experiments with Models of Economic Systems, Wiley: New York, 1971, p. 2.

14. H. I. Ellington, E. Addinall and F. Percival, 1981, p. 20-24.

15. H. I. Ellington, E. Addinall and F. Percival, 1981, p. 30.

16. A. M. Bork, Personal Computers For Education, Harper and Row: New York, 1985, p. 68.

17. H. D. Naetscher, "The incorporation of computer simulation into the system of simulation", 1980, p. 93.

18. A. Bork, Learning with Computers, Digital Press, 1981, p. 106.

19. R. A. Sparkes, Microcomputers in Science Teaching, Hutchinson, 1981, p. 105.

20. H. I. Ellington, E. Addinall and F. Percival, 1981, p. 20.

21. R. A. Sparkes, Microcomputers in Science Teaching, Hutchinson, 1981, p. 106.

22. E.M. Glass "Computers: Challenge and Opportunity", in V.P. Hansen (1984 yearbook editor) and M. J. Zweng (General Editor), Computers in Mathematics Education 1984 yearbook, National Council Of Teachers Of Mathematics (USA), p. 11.

23. G.W.Orwig, Creative Computer Programs For Learning, Prentice-Hall, 1983, p. 11.

24. H. I. Ellington, E. Addinall and F. Percival, 1981, p. 78.

25. S. Engh and K. L. Ratzlaff, "The Use of Computer-Assisted Instruction in the Teaching Laboratory", in J. W. Moore, ed, ITERATIONS: Computing in The Journal of Chemical Education, 1981, p. 54-55.

26. H. Wohltjen and R. Dessy, "An Instrument Simulator for Use in Computer Interfacing Laboratories" in J. W. Moore, ed, ITERATIONS: Computing in The Journal of Chemical Education, 1981, p. 57-58.

27. K. J. Johnson, "Simulation and Data Reduction Programs" in J. W. Moore, ed, ITERATIONS: Computing in The Journal of Chemical Education, 1981, p 66.

28. K. J. Johnson, "Simulation and Data Reduction Programs" in J.

29. Seymour Papert, Mindstorms: Children, Computers and Powerful Ideas, Basic Books: New York (or The Harvester Press Limited: Great Britain), 1980, Chapter 5.

30. A. Bork, Learning With Computers, p 111.

31. F. Percival, H. I. Ellington and E. Addinall, "A large-scale evaluation of gaming and simulation materials: some problems and pitfalls", B. Hollinshead and M. Yorke, eds, Perspectives on Academic Gaming and Simulation, Simulation and Games: The Real and Ideal, 1980, p. 203.

32. F. Percival, H. I. Ellington and E. Addinall, "A large-scale evaluation of gaming and simulation materials: some problems and pitfalls", 1980, p. 204.

33. G. W. Orwig, 1983, p. 12.

34. H. I. Ellington, E. Addinall and F. Percival, 1981, p. 78.

35. G, Hubbard, "The Prospect of Respectability", B. Hollinshead and M. Yorke, eds, Perspectives on Academic Gaming and Simulation, Simulation and Games: The Real and Ideal, 1980, p. 17.

36. R. H. Hooper, National Development Programme in Computer Assisted Learning, Final Report of the Director, Council For Education Technology (NDPCAL), 1977, p. 10.

37. R. H. Hooper, 1977, p. 36.

38. P. Ayscough, CALCHEM: Final Review Report, Department of Physical Chemistry, The University of Leeds, 1977.

39. R. H. Hooper, 1977, p. 144.

40. J. F. Rockhart and S. Morton, Computers and the Learning Process in Higher Education (a report prepared for the Carnegie Commission on Higher Education), McGraw-Hill: New York, 1975.

41. Seymour Papert, Mindstorms: Children, Computers and Powerful Ideas, Basic Books: New York (or The Harvester Press Limited: Great Britain), 1980, p. 36.

42. H. L. Dreyfus and S. E. Dreyfus, "Putting Computers in Their Proper Place: Analysis versus Intuition in the Classroom", p. 40-62.

43. R. J. Sardello, "The Technological Threat to Education", p. 93-101.

44. H. D. Naetscher, 1980, p. 95.

CHAPTER 4

A BACKGROUND TO ANALOGUE COMPUTING AND ITS DIGITAL SIMULATION

4.1. THE BEGINNINGS OF ANALOGUE COMPUTING

The history of analogue computing is traced by M. Moyle [1] and also by R. Ord-Smith and J. Stephensen [2].

The analogue computer dates back to the 17th century, when Galileo developed a device for scaling drawings, and later Oughtied constructed a slide rule. In 1876 the idea of using systems of mechanical integrators to solve differential equations was devised, and in 1931 a mechanical analyzer was constructed which could add, multiply and integrate using mechanical parts.

According to Moyle, the idea of the electronic differential analyzer was introduced in 1947, and paved the way for the modern electronic analogue computer. Ord-Smith believes the analogue computer was developed by 1943 at M.I.T, and had mainly military applications.

Either way, the device had both static and dynamic inaccuracy sources, and errors are commonly of the order of 1%. Accuracy and precision are directly related to the cost of the computer.

One of the important uses of the analogue computer was in control.

4.2. A BRIEF DESCRIPTION OF THE ANALOGUE COMPUTER

The electronic analogue computer (which will simply be referred to as an analogue computer) usually consists of a patch board of active and passive devices which may be connected together to configure different simulations. The active devices are based on operational amplifiers, and include input-voltage sources, potentiometers (for scaling), integrators, summers, multipliers and often many other function-generators and functions.

A process is described mathematically and then modelled, usually in one of three ways. Models may consist of sets of **differential equations**, which represent the process being simulated. One then configures the patch board to represent the differential equations by equivalent analogue modules. The "solution" then appears as a voltage at the output of one of the modules, depending on what value is required.

A model may also consist of a **block diagram**, where each block represents a sub-process of the process being simulated, and the interconnections represent the transfer of information between blocks. This may also be directly simulated by having an electrical equivalent of each block, and interconnecting these according to the block diagram.

An alternative is to model using **state-variables**. These may then be patched in a similar way to the differential equation implementation.

4.3. THE BEGINNINGS OF DIGITAL SIMULATION

The history of digital simulation of analogue computers is traced by R. Stephensen [3].

The work was pioneered in 1955, when R. G. Selfridge implemented a system in machine-code and with fixed-point arithmetic, which was based on concepts which are still used in implementations today. In 1958 H. F. Lesh used a fourth-order Runge-Kutta scheme and expanded the number of functional blocks to make a system for solving differential equations. Also in 1958, a language called **ASTRAL** was developed, which directly modelled an analogue computer and produced code in **Fortran**. It also introduced the idea of sorting the module order. Between 1958 and 1963 many implementations appeared. At the end of this time, **MIDAS** was developed, with improved sorting and integration. It also supported implicit integration. It was followed by **Pactolus**, which allowed limited interaction from a terminal.

MIMIC then followed, which was a true modern-day implementation. **DSL-90** allowed input in free format and in either block notation or as differential equations. **CSSL** was an important successor, as it was an attempt at standardisation.

4.4. THE HYBRID COMPUTER

In 1958 the first hybrid computer was made. It became necessary in simulation related to the space program, where the high speed of the analogue computer was necessary for real-time simulation, and the storage and precision of the digital computer was required because of the magnitude of the task [4].

The analogue computer works in parallel, and this is what gives it its main advantage -computation time is not a function of complexity. It is also inherently interactive. Implementation of non-linear functions was also more simple on analogues. The situation is different today, where digital computers ars so much faster and more precise, and easier to program.

The digital computer has far more advantages. It is far more readily available, and much more work is now being done on digital simulation than analogue. It can be programmed to an arbitrary level of precision. Data representation, storage and documentation is also far easier. Digital computers are becoming so fast that the advantage held by the analogue computer is reduced.

Hybrid computers are used when the advantages of both types are required. This may include applications where widely differing parameters in equations produce both high and low frequency components in the simulation. They are used when simulating systems which are themselves discrete-time (sampled data) systems, when complex functions are being modelled and when considerable post-simulation numerical analysis and processing of data is required. In the near future, the solution will probably be in the use of parallel processors, which are at the same time digital and work in parallel [5].

4.5. SOFTWIRE AS A SIMULATION PROGRAM

The task in devising **SOFTWIRE** is to use the procedures and techniques gained from previous experience in simulation to write a program which is sound, reliable and powerful enough to be used for simulation in its own right, and to adapt this simulator to fulfil the pedagogical goals which have been set out for it. To this end, **SOFTWIRE** does not claim to supercede the previous implementations which have been discussed, except possibly in the role assigned to it.

An important goal was to make the program simple. To this end, some powerful features of other implementations have been left out. Another decision was that the program should not be run in batch mode, but done "on-the -fly". This means that error-control which is linked to step-size (with iterations being done more than once if the error is too large), and some integration and corrector algorithms, were not applicable. Implicit integration was also not possible, as it requires many iterations for each time-increment. Speed was also of the essence, and so some features which were time-consuming were not included.

One of the features which has been omitted to simplify the program, is the use of free format. The **Fortran**-like format which **MIMIC** uses, was also used here, unlike the more modern simulation languages. The fixed format is in no way detrimental to **SOFTWIRE**, since its rigid and simple syntax is designed to map directly from module to net-list, and the idea of one module represented per line ties in well with this desgn choice.

The facility of automatic module-order sorting has been implemented to remove the distraction of sorting completely from the student.

Regarding hybrid computers, **SOFTWIRE** is a hybrid in the sense that it has an analogue and digital section. However the reasons for this are more from an educational viewpoint than a simulation one. These will be discussed later in PART 3.

An important addition to previous digital simulations is that this one is more interactive. Editing is all done at the keyboard, and data can be input, and is output, during the run.

In most simulations the operator inputs data, runs the simulation, and then analyses the output. In an educational context this may also happen. However there is more importance placed on assisting in the **association** of the input and output. It is more crucial that the user obtains a firm understanding that a particular input to a given system produces a particular output. The accent is more on leaving the user with a concrete picture of this causal relationship, than with the amount of information obtained from the output.

Analogue computers and simulations tend to be designed around the solution of differential equations -hence the importance of the integration routine. **SOFTWIRE** has a module, or block-diagram approach, and has been designed with this in mind (although it can easily be used to solve differential equations). This is a subtle, but important difference.

In the educational environment, one is concerned not only with the final output of a system, but also with how each part of the system has contributed towards that. There must be flexibility in switching between examining the whole, and the individual parts.

Many features could be added to make **SOFTWIRE** more powerful, but one must be careful not to fall into the temptation of adding

features, and then removing the simplicity and "obviousness" whick are key to educational use. There are also features which are transparent to the beginner, but may be used later. These features include the ability to:

1. Give symbolic names to nodes, as an alternative to numbers. This can make output more meaningful.
2. Replace values with expressions.
3. End the simulation on conditions other than elapsed time.
4. Change the independent variable to one other than time. This can be used to plot transfer functions.
5. Have more parameters and/or inputs per module.
6. Allow symbolic constants i.e. π and e.
7. Allow iteration and branching.
8. Add code directly in high level language (in **SOFTWIRE** this can be done in a limited sense).
9. Solve implicit statements.
10. Allow **hold** and **reset** of the integrator module.
11. Have a set of functions for performing Inverse Laplace.
12. Store comments. This makes the net-list more readable.

At present, the system does not have these features, and it is felt that they are "extras" rather than pre-requisites for a good teaching system, and that some of them may complicate the program. Some would also increase ambiguity, which reduces the ability of the program to differentiate between intended and accidental input. For instance, if nodes could be specified by symbolic identifiers, rather than numbers, the computer could not then identify a typing error by checking the numerical-syntax, as it would think the user intended to type non-numerical digits. If this problem was overcome by requiring declarations beforehand, the intrinsic simplicity of the program would be lost.

4.6. CONCLUSION

In this chapter a brief history of analogue computation and its digital simulation was given, and the place of **SOFTWIRE** in this was indicated. It was shown that **SOFTWIRE** is rooted in this development path, but also differs significantly because it has been designed for specific educational purposes.

REFERENCES

1. M. P. Moyle, Introduction To Computers For Engineering, John Wiley & Sons, 1967, p.131-134.
2. R. J. Ord-Smith and J. Stephensen, Computer Simulation of Continuous Systems, Cambridge University Press, 1975, p. 1-6.
3. R. E. Stephensen, Computer Simulation For Engineers, Harcourt Brace Javonovich, 1971, p. 163-165.
4. M. P. Moyle, 1967, p. 196.
5. R. J. Ord-Smith and J. Stephensen, 1975, p. 6.

CHAPTER 5

DESIGN CONSIDERATIONS IN DEVELOPING THE SOFTWIRE SYSTEM

PART 1 has given a very brief outline of some of the historical events which are of importance to the project. It has also highlighted some of the more important issues that have been involved in the fields of simulation and computer-based education. With this basis, some important design considerations will be mentioned, which led to the final **SOFTWIRE** system development.

5.1. SHOULD WE HAVE SOFTWIRE AT ALL

Given the problems in present Electrical Engineering instruction of theory, as outlined in the introduction, it seemed that the design aims (that were also listed in the INTRODUCTION) would make a useful impact in providing potential in solving those problems. The set of problems listed included practical problems, and it was felt that a system could be implemented using the computer, which would contribute towards alleviating these frustrations.

On examining the knowledge gained in the use of computers in education, it was felt that simulation had potential in employing computers usefully in the field of interest.

A system specified would be seen as a tool, and as such would be developed to facilitate a particular process. In this case the process is education in a specific field. As a useful tool, it had to be shown to have a purpose, and preferably to surpass other tools in carrying out that purpose. However, a tool does not itself perform a function, it merely assists the user in

performing a function. As Descartes said, "tools suggested things to one's reason, but never replaced reason". To be useful, the tool would not only have to be developed, but have to be used wisely.

5.2. WHAT SHOULD IT CONSIST OF

The initial plan was to have no more than a basic system. Any extra complexity provides a further barrier to learning how to use **SOFTWIRE**. It then had to be decided what constituted a basic system.

What was initially required for the student was a program which would enable the inputting of simple net-lists, and a minimum of operating parameters, and then be able to run a simulation of the system defined by the net-list, and give output in a useful form. The user could then make changes and re-run the simulation as required.

From the point of view of the tutor, the same functions were required. In addition it was important to have a means of altering or adding module functions as desired. It was also useful to be able to save working net-lists to give to the class or to use in demonstrations.

The input was to be in an intuitively useful format. Input format and syntax checking had to be comprehensive and detailed enough to make the program "bombproof" and to allow the student to easily understand the nature of an error made -if possible to help correct it. Error checking was also to extend to whether or not the net-list was adequately defined.

Output had to be at least a graphic representation on the screen, with at least three outputs shown at once. The usefulness was greatly increased if one could plot one output against another.

One should then be able to modify a data input, including adding to or deleting from the net-list, changing operating conditions, and starting a new simulation. It was also considered useful to be able to examine the net-list while editing it.

The whole development of the system would be along iterative lines. Software would be developed and tested, mistakes found or improvements suggested, and modifications made. This prototyping approach was modelled on the work of the philosopher, Karl Popper. His method of iteration states that

"

$$P_1 \text{--> TS --> FE --> } P_2$$

where P_1 is the initial problem, TS the trial solution proposed, EE the process of error elimination tried and P_2 the resulting situation with new problems. It is essentially a feedback process. It is not cyclic, for P_2 is always different from P_1." [1].

The implication is that the system development is never fully complete, but always under evaluation.

5.2.1. RESOURCE LIMITATIONS

In order to make the program as widely available as possible, it was considered important to limit the hardware requirements of the system as much as possible. Since an **IBM PC** had been chosen as the machine on which **SOFTWIRE** was to be run (see the chapter on CHOICE OF SYSTEM), it was felt that a useful program would be able to run on the minimum configuration of a monochrome monitor, a graphics adapter, 256K memory and one disk-drive. The adapter was the only item which might have to be purchased.

This minimum configuration would still allow the system to be used effectively. Since it is presumed that a large number of institutions which could benefit from **SOFTWIRE** already had one or more **IBM PC**'s, the marginal cost of the system would then only be that of disks for the program. It was hoped also that many potential users already had computer laboratories.

One step up would be the inclusion of a printer for hard copies of the outputs. This is useful for enabling the students to include results in reports they wrote, or to show that they had done the required work.

Another step up would be the use of a colour monitor. This enables outputs to be more clearly distinguished. Increasing the resident memory from 256K would also enable more memory to be allocated to functions which required large arrays for their computations. Both these improvements are small, and would not alone justify the purchase of these extras.

An important purchase for enhancing the system is the **User Interface**. This greatly increases the utility of the system, by adding a significant new set of uses. It allows real-time simulation to be performed, and allows analogue devices to be coupled to the simulation. It also opens up application for teaching in the field of measurements, and related signal modification.

5.2.2. SCOPE AND AIMS OF THIS STUDY

Since the preparation of this dissertation was restricted in terms of funding, it was obviously important to define the scope and the aims of the study very specifically.

The overall aims of the study were originally to investigate the use of computers in teaching electronic engineering, and to provide a program and material that would be both cheap and cost-effective, and be useful to a lecturer in assisting his education task without his having to spend many hours at the keyboard preparing lessons. It was apparent from the absence of meaningful input, that not much research had been done on the potential of computer assistance in this specific field of education.

The goal then became to create a software and hardware system that would facilitate learning in the ways discussed. This system was to be fully implemented -working, debugged and reliable. The software had to be well written enough to allow expansion and modification. This was to be created early on in the project, so that it could be tested in suitable ways and used to create and test lessons. As stated, it was assumed that it would undergo iterations. The hardware also had to be designed with possibilities for expansion and modification in mind. (The results of this form the basis for PART 2).

The next goal was to develop an informed approach to learning, and how it is best effected. It was then important to define more precisely what was being taught, and apply learning theory to this. From this a strategy could be outlined as to how best to use the **SOFTWIRE** package to accomplish the learning goals.

Out of this strategy, sample lessons and demonstrations could be created and tried out by students and peers. Informal evaluations could then be done, and the process outlined in this and the previous two paragraphs be repeated until a satisfactory completed system would evolve. (This entire process is discussed in PART 3)

5.2.3. WHAT PROBLEM IS IT SOLVING

SOFTWIRE has been proposed as a solution to a particular problem. The question is exactly what problem it is that needs to be solved.

The decision that there was a need for **SOFTWIRE** arose from the observation that the teaching of theory in the undergraduate electrical engineering curriculum was hampered by the inability of students to develop an intuitive feel for the devices and systems they were learning about. It was also felt that students would benefit in gaining this intuition, and also in other ways, by more practical exposure. However it was in gaining this exposure that difficulties presented themselves.

The first problem was one of capital cost. To allow large numbers of students to gain experience in the use of a variety of equipment was costly. It meant that a number of devices and instruments had to be purchased, maintained and upgraded or updated continually. As the extent of the electronic engineering field increases, equipment becomes more diverse and often more costly.

At practical sessions there is also the need to employ a number of people to maintain equipment and to assist the students in setting up equipment and in understanding what is going on. This is costly. The demonstrators who are employed have to be qualified to assist and are paid accordingly. However they often spend a lot of the time they could be spending on instructing,

attempting to diagnose why people's instruments or circuits are not behaving as indicated. Their time is not optimally used.

From the students' point of view, it is important that they do gain experience in soldering and breadboarding and coping with faulty circuits. However there is a limit to what experience of that type is useful. It is the author's experience that many students doing laboratories have left the session disillusioned and frustrated and having learnt little of what was intended, all because of an intermittent problem with a breadboard or instrument transducer or lead.

One also experiences difficulties with demonstrations during lectures. Many demonstrations which would be useful are not done because they are guaranteed to take up too much of the lecture time. The time is often spent in setting up demonstrations and diagnosing why they do not work!

5.2.4. THE TARGET ENVIRONMENT

A major problem with computer-based education programmes is that they do not give sufficient consideration to the situation in which the learning will take place. They often assume laboratory conditions of adequate equipment, and surroundings free from distractions. The ratio of (highly trained) tutors to students is high enough to guarantee almost personalized attention to the small classes. The students are depicted as motivated and sensitive learners. In reality this is not always the case.

There is often a shortage of equipment, and the available equipment has not been placed in a distraction-free environment. The tutors may be technically trained, but they do not always know how to use the new material. There is usually a staff

shortage, and classes are larger than the optimal size. Students come with a wide range of aptitudes and attitudes.

Another problem is that the people who teach are more in command of the material than the students, and as a result often do not have insight into how the learner is viewing the material. For instance, a lecturer might view a proof as "elegant" or a phenomenon as "curious", and attempt to convey this to a class who cannot see the significance of these. This causes problems in the use of the computer, because material is often developed away from students, and the developers may find both the modes of teaching and the material fascinating and intriguing, and be tempted to include these interesting aspects into the curriculum. The students who finally use the material are thoroughly bored because they fail to see the significance of the additions. The problem would be obviated if the tutor could explain the significance to individual students, but this is often not logistically possible.

5.2.5. THE HUMAN INTERFACE

One of the problems associated with computers that has attracted the most attention and research is the interface between the computer and the human using that computer. This is because the two entities use different natural languages. The solution so far has been a compromise, where both entities speak a common language that neither speaks naturally.

Problems also arise because the human is prone to error and does not always specify his intent precisely.

The problem is not only with communication symbols, but with the physical means of communication. Devices such as the keyboard

cannot be used intuitively, but at present their use must be learnt. This takes time and presents a barrier to the novice, where the potential energy of the initial enthusiasm is not used so much to compute, as to learn how to communicate with the computer.

The question of the human interface presented itself in **SOFTWIRE**, and was dealt with in ways described below:.

5.2.6. PROBLEMS IN SOFTWIRE'S DOMAIN

Interface problems present themselves in the use of **SOFTWIRE**. To begin with, the student may not have used a keyboard before. This draws away enthusiasm and concentration from the actual material being learnt. Even when more familiarity is gained, the keyboard and console are not conceptually the same as switches, knobs, meters and lights -devices which are far more intuitively obvious sources of input and output. To plug away at keys to adjust a voltage level is limiting from both an intuitive and practical point of view. Also, in a more complex system with analogue and switch inputs all represented by different keys, the keyboard becomes too confusing.

An interface problem at a more abstract level with **SOFTWIRE** is that when data is to be input, the student may not know what information is being expected. He may either not know what it is that is being asked for -information about what?- or may not yet be equipped to give the information. Many otherwise good programs give people a negative first impression when they ask for screeds of data to be typed in by the first-time user, who never sees what the program actually does because he does not know how to enter sufficient meaningful data to activate it, even though he knows clearly in his own mind what he wants to do.

5.2.7. AIMS RELATED TO THE INTERFACE

The aim is to have a scenario where new students can begin to use **SOFTWIRE** soon after first encountering it. They must also be able to begin simulations as soon as possible after a session has begun, and "see" the results of the simulations in an intuitively obvious way -not be distracted or confused by the computer itself. Also, the user should not be discouraged from wanting to experiment and make changes. Thus editing should be as straightforward as possible.

It is also an aim to allow simultaneous progression of available power and complexity. The user starts requiring very little knowledge to run a simulation, but having correspondingly little decision making power as to using alternative available options. He then progresses until he acquires a more thorough knowledge of the program and obtains the ability to command the full power of the program.

5.2.8. SOLUTIONS RELATED TO THE INTERFACE

Steps were taken to implement the above aims.

High-technology input devices are beyond the scope of this project as they are still expensive and their development still in early stages, and their advantages would only slightly benefit the system being described, as will be shown later.

The main immediate concern is to minimise the use of the keyboard. More especially, the concern is to minimise the use of the keyboard for the novice, and extend the use of the keyboard as the student gains experience. To this end, the novices was

allowed to start with a **useful** subset of the program, and advance to full features. Thus at different points in the program where options exist, these are given useful default values. This applies to different modes relating to time and output, as well as file names.

The tutor also has the means to set up complex simulations and save them to a file for novices to use. The student can then load an entire net-list of arbitrary complexity and operating parameters, and run the simulation, with only four single keystrokes.

There is also inherent progression in the use of function modules. The beginner is given only a useful subset of modules to work with. As he progresses, more modules are introduced. Not only this, but there are modules which perform the same basic tasks, but with varying complexity. An example is the integrator module. The beginner starts with a simple implementation, and progresses in complexity, and also in the number of parameters.

One of the major solutions in improving the human interface is the provision of the **User Interface**. This is hardware which provides the user with a means of communicating in ways more appropriate to the material being learnt. The user inputs analogue signals via potential dividers ("volume controls") and binary signals via switches. The computer outputs analogue signals via panel meters, and binary values via light-emitting diodes.

The "mouse" has become very important in many applications. The use of this for an input is described briefly in the chapter on CHOICE AND IMPLEMENTATION OF MODULES.

5.2.9. HARDWARE, SOFTWARE, TUTORS? -SCOPE OF THE PACKAGE

The word **SOFTWIRE** nominally refers to a program, but should really refer to a system. The system that is distributed consists of the program and the **User Interface**. However this does not constitute education. The system is not complete without materials which have been compiled using strategies outlined in the chapter on SUGGESTIONS FOR USE. The system is also not effective without the involvement of a human tutor. The tutor must at times play low-profile role (but nevertheless still be present to assist and encourage), and at times dominate the delivery. Appropriate style is essential.

Only when all the ingredients are present will **SOFTWIRE** be a workable system.

5.3. CONCLUSION

To sum up, then, a system is to be devised:-

1. Based on lessons learnt from literature in the areas concerned.
2. Incorporating software, hardware, the lecturer, the students and materials.
3. Fulfilling the AIMS laid out in the INTRODUCTION.

In this system, the hardware must be cheap enough to be easily available, and require minimum configuration, with upward expandability.

The software must be written for students with no previous computer experience, but whose course requires them to become computer literate. It must also be easy to use and make sense intuitively, so that the user's mind is not distracted by details

of the computer. Help must be sensible, relevant, and unambiguous. The user must not be confused by the "mode" the computer is in, but must always know where he is and what is required. A novice should be able to use the basic functions straight away, with room for options later.

The simulation should be fast, so that it is not boring, but is interactive (forcing a user response), and provides a tailored environment according to the user. It must depict enough information on the screen and in such a way as to enhance understanding.

PART 2 will discuss the design implementation which adequately meets these requirements.

REFERENCE

1. B. Magee, Popper, Fontana Modern Masters, 1973, p. 65.

PART 2

TECHNICAL DESIGN ASPECTS

Part 2 will describe the software and hardware design, and discuss the choices that had to be made in an educational environment. Other related technical choices will also be discussed.

CHAPTER 6

CHOICE OF SYSTEM

6.1. CHOICE OF COMPUTER

In South Africa, 1985, when the commitment was made to do this project on a particular computer system, it would have been foolish not to use an **IBM PC**[1]. Since the cost restriction was made a very important one (by decision), only microcomputers were feasible. At the time, the University of Cape Town had made a policy decision to use only **IBM's** in future. Also, most computer shows, magazines and institutions were dominated by **IBM**'s. A sample of five large universities in the U.S.A.[2] indicates that the **IBM** and the **Apple MacIntosh** are the most popular for tertiary education [1]. The withdrawal of **Apple** from South Africa makes the **Apple MacIntosh** less attractive. An **IBM** was thus chosen as the hardware for the system. The system was then designed to run with only one disk drive and a monochrome graphics adapter, and 256 Kb of memory (the practical minimum configuration). This was due to a limited budget for the project, but fitted in well with the design philosophy of a low capital-outlay educational tool.

6.2. CHOICE OF PROGRAMMING LANGUAGE

There are many criteria used in determining which computer language and programming environment to use. The program was written in the **Turbo Pascal** language and environment. The

[1] Any reference to an **IBM PC** here refers equally to all genuine clones. The original work was done on a **Sperry PC** with a 320x200 colour card with four colours.

[2] Reed College, Stanford, Carnegie Mellon, Drexel and Brown Universities.

decision was made using an elimination process similar to that suggested by Gary Elfring [2]:-

6.2.1. APPLICATION SPECIFICATION

The application was largely scientific/mathematical and required real-time processing in a small section. Hardware interfacing and good graphics capabilities were required. At the time of deciding, there were no proven real-time languages readily available on the **IBM PC**. **Turbo Pascal** had a proven track record for scientific applications. This was enhanced by the ability to compile into higher precision binary-coded decimal, or to support the 8087 co-processor, if required.

Speed in computation was important. A pilot program was originally written in interpreted **BASIC** to ascertain the feasibility of the idea, and it was found to be unreasonably slow. The choice was thus limited to compiled languages.

The program was expected to start off with a small (less than 2500 lines) working program, but with the option of increasing the size and complexity if it was succesful (to include hardware customizing and add more features). Its size was never expected to extend beyond 5000 lines. This is small enough not to require a team of independent programmers, and thus the need for a highly modular language, with independently compiled modules, is reduced. In **Turbo Pascal**, 5000 lines takes only four minutes to compile, which it was decided was acceptable. Independent modules may have been preferable, but the experience of the writer on **UCSD-Pascal** on the **Apple II+**, is that modules were not helpful if many parts of the program were continually being revised, as more time and energy was spent re-compiling and re-installing the modules, and then linking them, than was saved. A library of

modules is far more useful when the modules in the library are complete, and no longer need to be modified.

6.2.2. FEATURES

A language was required which suited the above application, and was intended as a serious, high-level, structured language with concise and unambiguous syntax, which was preferably originally developed for the purpose of solving a similar type of problem to the one here.

BASIC was thrown out early as an option. Interpreted versions had already been rejected. The built-in graphics in many compiled versions were better than **Turbo Pascal**, but could be done satisfactorily with the routines provided in the **Turbo Graphix Toolbox**. **BASIC** simply did not have the level of data and coding structure required[3]. Although very popular among users, it does not enjoy the support of many computer scientists [3][4][5].

The functional block forms the heart of **SOFTWIRE**, and it was preferable to use a language whose data types could be used to mirror the form of the functional block i.e. to logically group data for each module, consisting of inputs, outputs, functions, parameters and initial and intermediate values. A record in **Pascal** (and indeed in many similar languages) is ideally suited, especially since it can be user-defined to mirror the **SOFTWIRE** modules exactly.

The ability to use sets was also extremely useful, since it allows modules to be logically grouped according to their characteristics, and makes it easier for them to be modified later, as groups of sets represent lists.

[3] **TRUE BASIC** has more to offer, but had not yet established itself in South Africa

Portability is often a consideration in programming, in order to increase the number of tools available to the programmer, to reduce dependence on hardware or environment and to increase the target domain of users of the program, by reducing the effort required to transport the program. In this case, portability was deliberately excluded as a consideration, as the aim was not so much to produce code, as to evaluate **what the program did**. By sacrificing portability, time could be freed to write the program more rapidly, since it was anticipated that much work would be done in areas of graphic output and hardware interfacing, which are tradtionally difficult areas to produce portable code for.

The hardware interface required access to machine code (in order to provide critically timed input and output control via the ports. Real-time control was a desirable feature of the language. **Turbo Pascal** allows the inclusion of machine code procedures into the program. It also allows **Pascal** identifiers as labels in the machine code itself. This facilitates the transfer of data to and from **Pascal** variables (which is necessary for interfacing).

A language would preferably have a compiler which compiled stand-alone code. This meant that the program could be run from **MS-DOS**, rather than the programming environment. This obviates the need for a site-licence for the final product, and it is also preferable for it to run directly from boot-up, so as not to confuse the novice user.

Features such as networking, restricted file access and multi-tasking were not useful to the application.

6.2.3. PRACTICAL CONSIDERATIONS

Although a particular language and system may be the best option theoretically, it may not be optimal for a number of practical reasons, which ultimately turn out to have the most sway.

The most important consideration is what compilers exist for the micro-processor, and how availabile they are. If one has a tight budget, then cost is an important factor, and **Turbo Pascal**'s low price ($70.00) was very attractive.

Also, given a few to choose from, it is far safer to select a language with an established track record, and one which is likely to be around for some time to come. **Turbo Pascal** was a very popular language, which had established itself before the **IBM PC**, on the **APPLE II**. With such a large base of users, it was unlikely to simply die off. In fact its popularity has meant that its users are well looked after. For instance programs are available to compile larger programs in **Turbo Pascal** than the **Borland** version allows. There is also a program to convert **Turbo Pascal** to **Modula 2**.

Having a large user base for **Turbo** was important for **SOFTWIRE**, because it contains provisions for customizing the code. This is only a useful feature if the facilities for doing the customizing are widespread, and if many people are familiar with **Turbo Pascal**.

One always tends to find quirks in a language or compiler, where things don't happen the way they should, or one requires an unusual application which does not exactly match the data types or syntax of the language. In these cases one requires accurate and thorough documentation, and it helps to be able to contact other people who have had experience with that particular language. Many hours can be saved in this way, and it is thus an important practical consideration.

The reasons given above meant that **Modula 2** was decided against. Although it supercedes **Pascal** in many ways, compilers at the time of deciding were more expensive, less easily obtainable and had not yet established a base of users in South Africa. Indeed, they have still not caught on, as potential users have tended to rather opt for **Prolog**, **C**, and **TRUE BASIC**, depending on tha application.

Programming is enhanced considerably if one is aided by tools for developing the hardware and software, such as emulators and debuggers. A library of routines is also helpful, especially for input and for graphics. Thus one must choose an environment which is compatible with such tools. In this application, no hardware emulators were used, as hardware debugging was done with a logic analyser. For the software, no debugging tools were used on the bulk of the program, and the **IBM** supplied program **DEBUG** was used for bugs in the hardware interface driving code. This was possible because **Turbo Pascal** produces native code files which are directly executable from **MS-Dos** and hence from **DEBUG** too.

The compiler iteself may be very attractive, but one needs to take into account one's memory limitations, for the compiler, the program text editor and all the other programs associated with the language. The size of the code generated must also be considered. The compiler should also preferably be fast, to reduce frustration during development. The computer being used had a memory of 256K, and two disk drives. This fitted the complete **Turbo** environment, with editor, compiler and file management -as well as the object code. One disk was sufficient with the source code files, and data files when running **SOFTWIRE**.

6.2.4. THE FINAL CHOICE

Although there were many factors requiring consideration, and one attempts to be as objective as possible, the final choice made is also determined by personal choice and bias, especially if one has spent much time programming satisfactorily in a particular language.

REFERENCES

1. D. Osgood, "The Difference In Higher Education", Byte, February 1987, p. 165.
2. Elfring, "Choosing a Programming Language", Byte, June 1985, p. 235.
3. A. M. Bork, Personal Computers For Education, Harper and Row: New York, 1985, p. 28.
4. E. W. Dijkstra, "How Do We Tell Truths That Might Hurt?", Sigplan Notices, May 1982, p. 14
5. S. Papert, MINDSTORMS: Children, Computers and Powerful Ideas, The Harvester Press: Great Britain, 1980, p. 34-36.

CHAPTER 7

THE SOFTWARE IN DETAIL

Once the specifications for the program had been completed, and the computer, language and environment decided, the actual program was designed and coded.

7.1. CONCEPTUAL OVERVIEW AND TECHNIQUES USED

7.1.1. DIFFICULTIES OF ITERATIVE TESTING

One of the difficulties in writing optimal code for **SOFTWIRE** comes from the fact that most of the screen design and features of **SOFTWIRE** as it is now, were originally very different. Many significant changes were made to the original, and this affects the way the program is laid out. Obviously a program which evolved will not be the same as one written from scratch. It is easier to spend a long time beforehand specifying the program, but this was not possible in this instance because experience in what was required from the program could only be gained by actually using and testing the program. This approach resulted from the observations of the philosopher, Karl Popper:

"complex processes are only to be created and changed by stages, through a critical feedback process of successive adjustments. The notion that they can be created, or made over, at a stroke, as if from a blueprint, is an illusion which can never be actualised." [1].

The most significant example is the code used to read the data input by the user. Initially the syntax for the input data was

very limiting, allowing simpler coding for the sake of testing the effectiveness of **SOFTWIRE** for teaching. It was also written in **BASIC**. This was translated to **Turbo Pascal**, with minimal structural changes to reduce the possibility of bugs creeping in. The input syntax was then adjusted a few times until the most intuitively meaningful input format was found. It was only when a syntax was finally decided on that small changes were made to allow some simplification of the program code and minimisation of redundancy. However, if there had been time to start from scratch one would be more likely to use techniques found in writing modern compilers, such as a powerful routine to fetch tokens and pass them to a set of recursive-descent procedures to check for syntax and store the data. This would produce code which is more easily read and maintained. The syntax of **SOFTWIRE** is very simple, and it would be trivial to represent it in a format such as the **Backaus-Naur Form**, suitable for LL1 recursive-descent coding.

7.1.2. IMPORTANT DATA STRUCTURES

Most of the program is centred around the main data structure, which is based on a circularly-linked-list of records. Each record represents one module in the net-list. In addition, there is a base module, which keeps track of the time. This module is the "start" of the circle (when one is required). Although it is intrinsic to the data structure, it is treated by the program the same way as the user-input modules are. In other words, to the user it is simply "module 0" -the module to output the time, but to the program it also indicates the start of each iteration, and performs other control operations which will be discussed later. The order of execution of the modules is determined by its position in this circular list, with the timing module first.

The other important data structure is the vector of values which defines the status of the modules during simulation. The "status" refers to the instantaneous output values of all the modules at a given time.

The program can be understood in terms of procedures which act on these two structures. The two main programming activities are editing (which includes data inputting) and simulation running.

In the editing phase, adjustments are made either to the net-list, or to a couple of other simple parameters. The parameters are merely type real variables, but the editing of the net-list involves adjusting the structure of the linked-list. Modules are either added to or deleted from the list, or their contents are changed (usually the field of the record containing the parameters). After the list has been edited, the order is re-determined (according to a strategy which is discussed later), by actually changing the pointers.

During simulation, the modules' linked-list is scanned circularly, using the circular data structure of the module list to process each module in turn. As each module is processed, data is retrieved from the status vector (to provide input data for the modules) and this data is then processed (according to the nature of the module function) to provide fresh data which is then used to update the status of the vector.

During all data outputting, the vector is accessed.

7.1.3. USE OF STRUCTURED DATA TYPES, AND SYNTAX

The **Pascal record** is well suited to the circuit description. The records defined in **Pascal** to represent modules are closely

analogous to the way one would represent these modules on paper with block diagrams.

Pointers have been useful in that they allow the dynamic allocaton of memory, which reduces the restrictions on the size of the net-list only to the amount of memory available. It also allows faster execution time. This is because the modules are arranged sequentially via a linked-list, and to access the next module only requires an indirect memory access using the pointer (which is extremely fast) whereas to access a module via an array requires an address calculation containing a multiplication each time. A net-list of 15 modules, with 1000 iterations would require 15000 such calculations.

Another possible approach to the data structure described in the previous section, would be to extend each module record to include its own intermediate output value during simulation, and to have pointers to the output values of other records that it requires for its inputs. This would logically simplify the data structures, by obviating the need for a status vector array, and would speed up execution significantly, for the same reasons mentioned in the previous paragraph. However the code would be more complex in this case, since the pointers would all have to be set up after the module order had been determined.

Scalars are useful in **Pascal** for defining one's own appropriate data types. In **SOFTWIRE**, the function names describing each module are suitably represented as scalars. This allows for self-documentation in the declaration section. It also allows the use of sets, and the large **CASE** statement used during the simulation to process each module in turn according to its function. Unfortunately, **Pascal** does not allow scalars to be used in input or output. This means that the module names have to be declared

twice -first as three-character strings (for input and output), then as the actual scalars. This is a messy and error-prone business, but worth doing for the extra power scalars provide regarding data manipulation.

The **CASE** statement mentioned in the previous chapter is pivotal to the simulation. Each time a module is processed, the course of action is decided by this statement. This is far more powerful then the **THEN..IF..ELSE** approach in coding complexity, self-documentation through layout, and ease of changing the possible modules used (which is helpful, since a stated design aim is that the code defining the modules is easily adapted and extended by the user).

7.1.4 ORDER OF COMPUTATION

The order in which the module outputs are updated during simulation is important to the calculation. This is discussed in the chapter on MATHEMATICAL CONSIDERATIONS of simulation. In order to control the order, one may either have an order table which is referred to during the simulation, or one may change the natural order of occurence of the modules to indicate the computation order. The second method was preferred, since it allows faster simulation. This does change the order in which the user inputs the modules. However, this was not important as most applications in education use relatively short net-lists. If more time had been available to develop the software, one could have two sets of pointers, one for the listing order (or numerical sequencing might be preferred), and one for the computation order.

7.1.5 GOOD PROGRAMMING -BENDING THE RULES

Since the actual simulation loop requires extensive number-crunching, there has been a temptation to use global parameters and fewer, larger procedures -to marginally improve simulation speed. This is especially the case with output to the screen, where code has been timed running first with procedures for the plotting, with local variables, then with no procedures and global variables. The faster version was nearly 6% faster. A compromise was reached regarding the mix of procedures and global variables used. The global variables are values frequently accessed during the actual simulation. A better solution would be available if program modules were used, where all the printing routines could be placed together in a module, and a limited scope be given to the variables. Another useful feature here would be level zero local variables, where the value remains unchanged from the time a procedure terminates until it is called again (a feature which has been abandoned by modern compilers because of the cumbersome means of providing an initial value).

7.1.6 WINDOWS

Windows definitely aid in the representation of data, and the **Turbo Graphix Toolbox** supports windowing. However it was decided that windows were not necessary in developing and testing the fundamental aspects of **SOFTWIRE**. Their advantage did not justify the extra time spent in developing the code, and the extra size of the code -since using the **Toolbox** would have doubled the source code length.

7.1.7 GENERALISING THE CODE

A very challenging aspect of the program was generalising the code so that the modules could easily be changed, depending on the application required. This is difficult because modules differ in many ways, regarding topology just as much as function. They differ in the number of inputs, outputs and parameters required, and in what intermediate data storage they require during simulation. Some require no initialisation, others require once-off initialisation, and others require initialisation before each re-run of the simulation (such as modules which have non-zero starting outputs). The same applies for termination.

The method of generalisation was to produce a module definition which was capable of everything that a module was likely to be required to do i.e it had two inputs, two parameters, an array for intermediate values etc. This uses up more memory, but is feasible because the amount is small, and the advantages great. Lists of **Pascal sets** are then used to tailor the individual modules. This allows a systematic approach to tailoring -easier to understand and less likely to make mistakes. The sets play a functional role similar to lists of flags, which direct the flow of the program depending on their status.

Pascal does allow variant records, and these would have been suitable in catering for generalisation/tailoring to an extent. The data declaration would then have resembled the functional modules more closely. However, a restriction is that only one variant is allowed at any level. Thus, one could have one variant each for the choice of zero, one or two inputs to each module, but each one of those would then require another variant each for the choice of zero, one or two parameters. This restriction

imposed by **Pascal** thus begins to make the implementation required awkward.

Instead, the method of using sets was preferred. This also gives the programmer insight into the modules. By glancing through the list of sets which "define" the individual structure of each type of module, the programmer is given information in a structured way, as though the data were in a table.

SCRATCHPADS

Something which can consume memory quickly, is the need for modules to have large amounts of intermediate storage space. Since most common modules are explicit, or need at most one or two values stored, it was deemed wasteful to supply a large array for each module created. Instead, each module record has a pointer allocated. If a module needs a "scratchpad", an array is allocated in memory dynamically, and the pointer records its position.

7.2 OVERALL PLAN OF THE PROGRAM

For a simple hierarchical overview and detailed module dependency and data flow information chart, refer to the table at the end of the chapter.

7.3 THE PROCEDURES IN DETAIL

The program will be described using pseudo-code and data-flow diagrams where appropriate.

7.3.1 THE MAIN PROGRAM

This is a simple program represented as:-

```
DisplaySignature
ActivateSoftwire
Change back to text mode
ResetUserInterface
```

DisplaySignature and **ResetUserInterface** are very simple. The former gives a brief text introduction to the user. The latter turns off the LED's and zeroes the meters of the **User InterFace**. The bulk of the program is contained in **ActivateSoftwire**. This is itself a "main program", but was made a seperate procedure so that the user can re-start it when a completely new simulation is to be entered.

ActivateSoftwire

This procedure is represented as follows:

```
     ResetUserInterface
FetchUserInputDevice(FileFlag);
     if a file was used then close it
EnterAllUserData;
     SetUpModuleOrder
     repeat
        GetCommandViaMenu
     until user wants to exit
```

The modules used may now be discussed:-

FetchUserInputDevice

This gets information on how the user intends to input data. It may be from the keyboard, or from a file, whose name must then be specified. The filename is then validated, and the file assigned.

GetCommandViaMenu

When **SOFTWIRE** is not active, it sits in this procedure. A menu line at the bottom of the screen indicates the options available to the user. When a key is pressed. Appropriate action is taken. Each keypress results in a small local procedure being called to take action. The actions are well documented in the **Help** procedure, which is found in the declaration section of **GetCommandViaMenu**.

It should be noted that this procedure allows the user to start with a new system description via a recursive call to **ActivateSoftwire**.

7.3.2 USER DATA INPUTTING PROCEDURES

This group of procedures is responsible for inputting all the data the user wishes to enter about the circuit structure, timing and format of the output.

FetchAllUserData

This calls up the **user data inputting procedures** sequentially, when a new system is being entered.

EditStructure

To understand this, it is best to revise the input syntax of the structure (as found in the **SOFTWIRE MANUAL** in Appendix A). An example is:-

```
4 INT(3) 2
```

Which means that the value of node **2** (the input node) is **INT**egrated (the action) to give the output at node **4** (the output node), but with an initial value of **3** (the parameter).

It is also important to consider three modes of **EditStructure**, which are invoked by parameter. The editing is slightly different if the structure is being entered for the first time, or altered, or if the user simply requires to change one or more parameters.

The approach was thus to write routines which would save repetition and highlight the similarities so that the code was easier to understand and modify. These procedures are discussed below:-

FetchLine

Returns one input line, as well as the length of this input string. Resets the position of the text pointer which informs the user of the position of an error. The input line may be fetched from the keyboard or from an ASCII text file.

EatString

Call this routine, with two parameters -an input string, and a target character. The characters at the beginning of the input string are transferred to a new string until the target character is found in the input string or the end of the input string is reached. The remainder of the input string (which may be a null string), and the target string are passed back to the calling routine. The main purpose of this routine is to remove delimiting characters -usually spaces. An example is given below:-

To start with:
input string = 'the cat sat'
target character = ' '

At the end:
input string = 'cat sat'
target string = 'the'

ListStructure

Clears the screen and lists the present structure on the right-hand side of the screen. Modules are listed in their order of computation.

GiveErrorMessage

Beeps, and outputs an error message, together with an indicator of where in the input line the user made a syntax error. The message is indexed by an integer parameter, and a further parameter specifies the position.

ListModules

This is invoked by **EditStructure** when the user has supplied a mis-spelt or non-existant module name. It prints the list of all the module names, taken from the **CONST** array **OpName**, so that the user can select the required one.

ModNo

Returns a single module line from the input data string. Sets a flag if the data is invalid. This has an extra parameter which specifies the minimum number expected (the maximum number

expected is declared as a global constant). If the number is not within range an error is signalled. The minimum number is usually a '1' if the node number being read is an output node (since node 0 is the timing module output) and a '0' if the node number being read is an input node (since the timing module output may be used as input).

FetchParams

Three of the sets in **SOFTWIRE** are **zerparm**, **oneparm** and **twoparm**. The module function is passed as a parameter to **FetchParams**, which then consults the three sets to find out how many parameters to expect from the data line. It then retrieves these parameters and checks for correct syntax (i.e. correct parentheses, number format and delimiters), returning with the module's parameters, and a syntax-correct flag.

FetchNodes

Three of the sets in **SOFTWIRE** are **zeroinp**, **oneinp** and **twoinp**. The module function is passed as a parameter to **FetchNodes**, which then consults the three sets to find out how many input nodes to expect from the data line. It then retrieves these parameters, using the function **ModNo**, and checks for correct syntax, returning with the module's parameters, and a syntax-correct flag.

DeleteModule

Removes a module from the linked-list and frees the memory that was being used by this module.

ChangeStructure

This procedure is fundamental to the process whereby a line of data is converted to a module in the linked-list. Its functions can be summarized:-

(a) Input data lines repeatedly until a blank string is entered, using **FetchLine**.
(b) Check the syntax of the line, and give appropriate error messages, using **ModNo**,**FetchParams**,**FetchNodes** and **Give-ErrorMessage**.
(c) Add, delete and modify modules.

StartCircuit

Puts up a heading, inserts the timing module, then enters data using **ChangeStructure**

ChangeCircuit

Sets up a heading, lists the structure with **ListStructure** then enters data for changing the net-list, using **ChangeStructure**.

ChangeParameter

Sets up a heading then enters data for changing parameters in the net-list, in a similar way to **ChangeStructure**.

EnterPlotModules

This prompts for the node numbers of the modules whose outputs are to be plotted. It then sets up an array, **PlotMod**, of pointers

to the modules to be plotted. The array is arranged so that the modules are always plotted in numerical order.

PromptAndFetch

The next two data inputting procedures make use of an auxiliary procedure called **PromptAndFetch**, which is given a string parameter containing the prompt message. It outputs this as a prompt, and checks if the input is a valid real number. It repeats the prompt until a valid real is supplied. It is also possible to type a "?", and receive help as to what input is required (although this feature is only partially implemented as yet).

EnterPlotRange

Using **PromptAndFetch**, it inputs the user's minimum and maximum y-axis values, testing that the values input are realisable i.e. the maximum value exceeds the minimum value. It then calculates the offset, vertical scale and sets a flag if the plot has a zero.

EnterTiming

Using **PromptAndFetch**, it inputs the user's values for the time increment and total time required in the simulation, testing that the values input are realisable. If the parameters input will result in more than 2000 iterations, it asks the user for confirmation before continuing.

7.3.3. SIMULATION PROCEDURES

There is one procedure which handles the entire simulation computation. This is succinctly named **Calc**.

Calc

Calc performs more than one function, and so strictly speaking should be more than one procedure. However this is where speed is crucial, and so procedure calls and parameter passing are minimised.

(a) Parameters Passed

The parameters passed all refer to the options available during the simulation. These refer to either the timing or the data output representation.

(b) Initialisation Section

All modules whose outputs must be set to zero, or to an initial value are initialised. All counters used in the simulation are reset. The hardware timer is set (if necessary).

(c) Main Computation Loop

The main computation is controlled by a loop which exits when the maximum time is exceeded, or when the user presses a key.

Each module is processed in turn. The function represented by the module determines which statement in the **CASE** statement will be

executed. During each iteration of the loop a different module is made current.

Although the module **TIM**, which determines the time of the simulation, is structurally identical to the others, it performs a very different function. It updates the time variable **T**, during the run, and sets the loop-exit-flag when the time is up. It also provides synchronization during a real-time run. The data output processing is also controlled from here. This module was chosen to perform many different functions because it is the only module which it is guaranteed to be in every system.

(d) Output Representation

As mentioned in the previous paragraph, this is done from within the code for the function **TIM**. The correct function for data representation is called, depending on which output options the user has selected.

(e) Termination

This is done when the flag **TimeLeft** is reset from within the module **TIM**. It is reset when:

(i) The time duration specified for the simulation is complete.
(ii) The simulation is in continuous-run mode, and the user presses any key.
(iii) The simulation is in real-time mode and one iteration of the simulation takes longer than the increment time specified.
(iv) An error occurs in a calculation in a module, which is trapped by the module's error-checking code before it reaches the system.

7.3.4. UTILITY PROCEDURES

Many small procedures and functions were written to do tasks not specific to the application. They are stand-alone, needing no external declarations except the string type. These are covered below:-

Beep

Makes a noise. Used to draw the user's attention to a warning or error.

UpperCase

Inputs a string and outputs that string in upper case. Used to free the user from having to worry about the case of his/her input. This is a machine-code routine taken from the **Turbo Pascal** Reference Manual [2]

Timer

Returns a string with the date and time in the format (for example) 1980/01/01 12h30 for 12:30 pm on January 1, 1980. This was adapted from the example given in the **Turbo Pascal** Reference Manual [3].

Exist

Given an MS-Dos file name, it returns true if the file exists on the disk in the drive given.

7.3.5. USER INTERFACE PROCEDURES

These are procedures which are used to access the **User Interface**. They are also utility procedures and are all called from procedure **Calc** during the actual simulation. They are written in machine code. They are divided into two classes -input and output. The input is of the form:-

```
set up port address
input data byte from this address to 8088 accumulator
transfer this data to the specified Pascal variable
```

The output is of the form:-

```
set up port address
transfer data from the specified Pascal variable to the
        accumulator
output data byte to the port address from the 8088 accumulator
```

In each case, the port address is specified as a local constant, and a local variable of type **byte** is used in the transfer. Since the decoding is done from $310 to $31F (the range specified by **IBM** for the prototype board), 32 addresses may be accessed in the decoded range, corresponding to the five least significant bits of the address. Since there are only five types of device being accessed (cf the chapter on **THE HARDWARE ON DETAIL**), the two least significant bits are used to address within the type, and the next three are used to specify which type.

The procedures are:-

LatchOut

This is supplied with the bit position and value of the bit whose output is to be changed (true=high, false=low), and it outputs

the new value to the latch. Since a **byte** value is output every time a **bit** is updated, a variable must store the previous values of all the other bits, and the new value is OR-masked to this if it is true, and AND-masked to it if false. The variable cannot be local, as it must be remembered between calls, so it has to be global. Fortunately **Turbo** allows a global declaration anywhere in the declaration section, so the variable **OutLatchVal** is declared just before **LatchOut** -the next best thing to the Level-Zero local variable of Modula 2. There is only one latch, so the least significant two bytes of the address are "don't care". Only the least significant five bits of the latch are used.

LatchIn

This is called together with the bit position to be read, and returns **true** or **false**. There is only one latch, so the least significant two bytes of the address are "don't care". Only the least significant five bits of the latch are used.

ADC

A call to this, together with the channel number, returns a byte value. The range 0..255 corresponds to 0..5V in channels 1 and 2, and -5..+5V in channels 3 and 4.

The actual chip, the **ADC 0809**, is activated by first sending a channel address, enabling the address latch -via the **ALE** (**A**ddress **L**atch **E**nable) pin- and activating the conversion start, then (after a pause), enabling the output latch -via the **OPE** (**O**utput **E**nable) pin- and reading the byte value. This is done by an output statement in the same format as before, where the lowest two bits of the address specify the channel, followed by a pause, followed by an input statement. To simplify decoding hardware,

the **ADC 0809** is treated as two devices, one being the **ALE** the other the **OPE**. Thus the least significant two bytes of the **OPE** address are "don't care".

DAC

When called, this outputs a voltage corresponding to the byte parameter passed to it, to the channel specified by a parameter. As for ADC, the range 0..255 corresponds to 0..5V in channels 1 and 2, and -5..+5V in channels 3 and 4.

ROUTINES FOR THE TIMER:

These two routines are similar to the above ones for the **User Interface**, in that they use the same input and output formats. They are slightly more complex. The **Intel 8253** chip used has three timers, and the least significant two bytes of the address are used to specify which timer is being used, where addresses 00_2=>timer 0, 01_2=>timer 1 and 10_2=>timer 2. Also address 11_2 is used when the mode of the timer is being changed.

SetMode

This is the only **User InterFace** utility procedure which is not called from the actual simulation loop. It is called by **Calc** before the loop to initialise the counters. The input clock to the counters runs at 894.886 Khz. **Setmode** sets the divide rate of the first counter to 477, which means a continuous clock output from this counter of period 0.5 ms.

The second counter is then set with twice the value the user specifies in milliseconds (with a very small correction factor to bring the error to less than 0.1%). The value it is set to, is

given as a parameter when **SetMode** is called. This counter receives its clock input from Counter 0, and its output in turn goes to Counter 2. Thus if a user requires say 10ms intervals, the second counter is programmed with 20 (2*10). If 2s intervals are required, the counter is initialised to 2001 (2*2000+the small correction factor)

The third counter is set with the maximum count for the lowest divide rate possible. Its function is to count the number of clock outputs from Counter 2. This count value is later used by **Calc** for determining time increments.

Count2Val

When invoked, this function returns the count value from Counter 2, using the same input format as used before.

7.3.6. OUTPUT REPRESENTATION PROCEDURES

A number of procedures are written to process the outputs. These handle different aspects and methods of outputs. The categories are:-

OUTPUT DEVICE:		Screen, file, printer.
OUTPUT MODE	:	Table of values, variables against time, one variable against another.
TIME MODE	:	Stops after a fixed time, continues indefinitely.
ACTION TYPE	:	Set up (initialise) output device and set up headings etc, output during simulation, terminate device and output footings etc.

The procedures dealing with the output representation, initialisation and termination are **SetUp** and **Terminate** respectively. These are called from **GetCommandViaMenu**, with flag parameters to indic-

ate the categories as mentioned above. **SetUp** in turn calls other procedures. The procedures which output during the simulation are called from **Calc** itself. The individual procedures themselves are not important, since they could easily be changed for different outputting requirements. One thing they nearly all have in common, is code such as:-

```
for I = 1 to maxplots do
        get(ModuleValues[PlotMod[I]^.OpNode)
```

PlotMod is the array which records the modules which are to be plotted. It does so by pointing to them. **MaxPlots** says how many of them there are. **ModuleValues** is the array which records the actual output values of the modules during each simulation, however it is indexed by the actual node numbers, which are supplied by the module records that are being pointed to, by the variable **OpNode**.

Each procedure will be briefly mentioned:-

SET UP OUTPUT PROCEDURES

ScreenGrafInit

Plots y-axes, according to the range given by the user. Also displays user-supplied parameters underneath the plot area.

ScreenOsclInit

Plots x-axes and y-axes, according to the range given by the user. Also displays user-supplied parameters underneath the plot area.

ScreenTablInit

Sets up headings for the module numbers required by the user, as well as a time and date stamp. Also displays user-supplied parameters. A line such as the one given here would be drawn:-

TIME op 1 op 2 op 3 op 4 op 5

PrinterInit

The user is prompted for a heading (for his own archival purposes). This is printed, together with user-supplied parameters and the time and date stamp, as a header. The procedure then prints a heading for the data which is either a line (for a table) or an axis for a graph), depending on the parameter supplied. (If a graph is drawn, it is done along the length of the paper, so that the graph can carry on indefinitely on the time-axis.)

FileInit

A file is opened on the disk, after a filename has been prompted for. Into the file is then written the same information as for the previous procedure **PrinterInit.** (The file is written in ASCII, to make data retrieval less complex. In fact, the file can be used directly by any word-processor which uses ASCII, and be included in text as a table of data.)

SIMULATION-TIME OUTPUT PROCEDURES

ScreenTablLine

Outputs the simulation time, followed by values of required output nodes.

PrintTablLine

As for **ScreenTablLine**, but it outputs to the printer.

ScreenGrafLine

Plots appropriately scaled dots on the screen. The vertical scaling is determined by the y-axis range, the horizontal by the total time, so that each simulation fills the screen exactly horizontally.

ScreenGrafLineInf

As for **ScreenGrafLine**, except that there is fixed horizontal scaling, and every time the plot gets to the right hand side of the screen, the screen is shifted half-a-screen to the right, to allow for continuous plotting.

PrintGrafLine

As for **ScreenGrafLine**, except that there is fixed horizontal scaling. (If a graph is drawn, it is done along the length of the paper, so that the graph can carry on indefinitely on the time-axis.)

ScreenOsclPoint

Plots appropriately-scaled dots on the screen. The scaling is determined by the y-axis range specified by the user.

FileLine

Outputs the simulation time to a file, followed by values of required output nodes, all in ASCII, with a carriage return after

each line.

TERMINATION PROCEDURES

Terminate

If the printer was used, it draws a closing line. If a file, it simply closes it. Other outputs do not require termination. The procedure is not complex, but is necessary to generalise the program so that different output modes may be added later if required.

There is one other method of outputting, and that is doing a screen dump to the printer after the simulation has finished. Since **Softwire** was written, resident programs such as **HardCopy** have become more easily available, which allow one to do this with the [Shift][PrtSc] keys, for many different printers. If available, these are recommended. However, a procedure was written for the **Epson** printers (based on the procedure **HardCopy** found in the **Turbo Graphix Toolbox** [4]) which gives the screen dump as an option. It leaves the previous screen plot intact, and replaces the menu line with a far more useful stamp of the date and time printed.

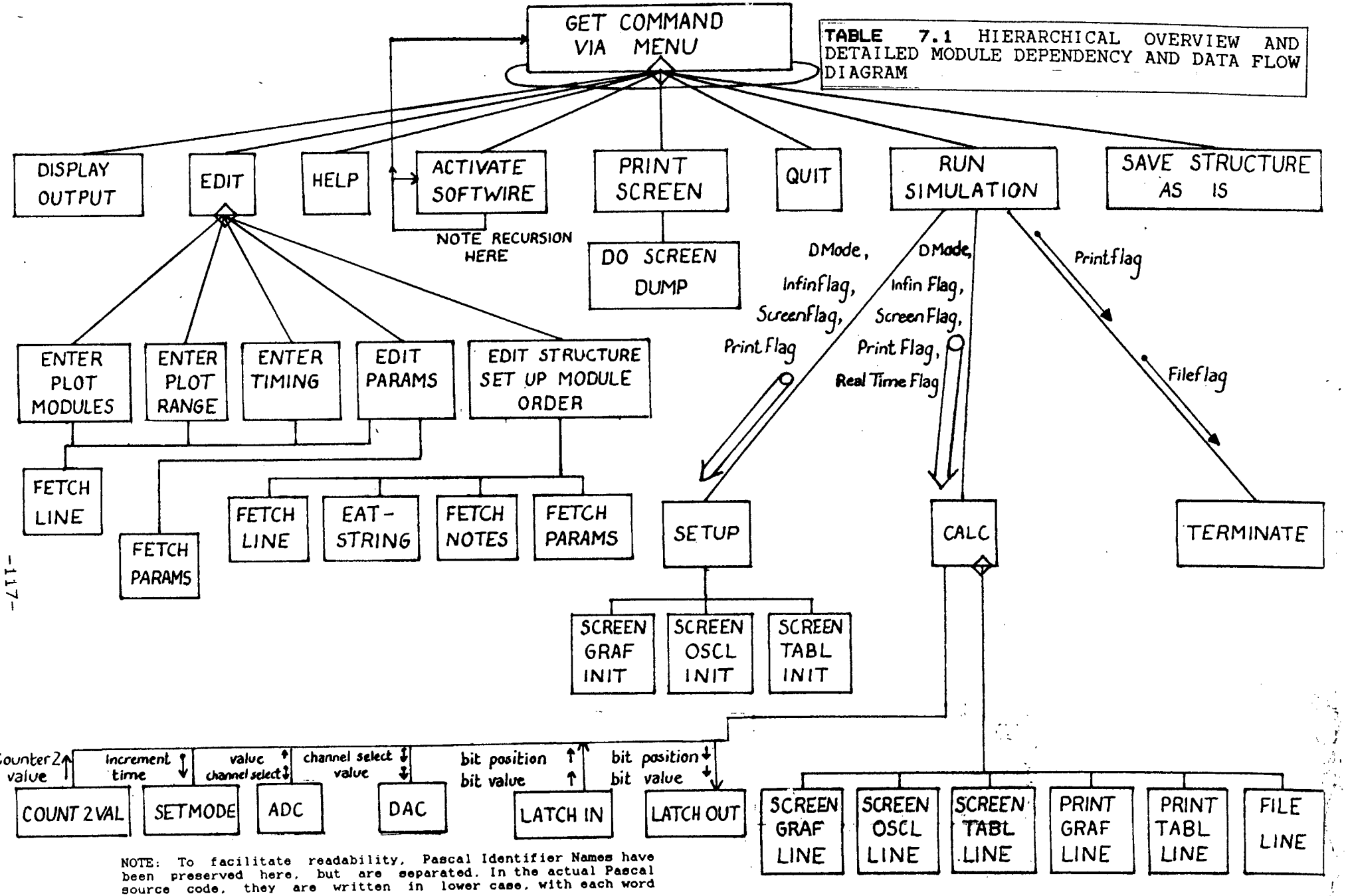


TABLE 7.1 HIERARCHICAL OVERVIEW AND DETAILED MODULE DEPENDENCY AND DATA FLOW DIAGRAM

NOTE: To facilitate readability, Pascal Identifier Names have been preserved here, but are separated. In the actual Pascal source code, they are written in lower case, with each word starting in upper case.

REFERENCES

1. B. Magee, Popper, Fontana Modern Masters, 1973, Chapter 5, p. 67.

2. BORLAND International, Turbo Pascal Reference Manual, p. 244.

3. BORLAND International, p. 209.

4. BORLAND International, Turbo Graphix Toolbox, 1985, p. 170.

CHAPTER 8

IMPROVEMENTS TO SOFTWIRE

It is possible to give a substantial list of improvements, which would say nothing useful, and probably never be implemented, and if they were, it would not require any exceptional programming skills. The purpose here is to show only some of the more important potential improvements (which were not made because of time constraints, which would more closely further the aims of **SOFTWIRE**, which were discussed earlier, in the INTRODUCTION. There is also discussion on how present hardware trends would result in improvements.

It is important to aim for extensions which are always useful, rather than obscure options. Options tend to take up memory space and increase the length of the user manual. To justify this they should be used often if they are to be included, For instance, windowing is immediately useful to any user, whereas the option of outputting to a plotter would only be useful in a minority of cases.

8.1. POSSIBLE EXTENSIONS IN SOFTWIRE

1. The extensions which would directly further the main aims of the project, are discussed in the CONCLUSION.

2. An alteration to improve accuracy, and allow implementation of the entire boolean logic family by having two value arrays for intermediate storage, instead of one, is mentioned in the chapter on MATHEMATICAL CONSIDERATIONS.

3. Improvements based on other analogue simulators are discussed in the chapter on A HISTORY OF ANALOGUE COMPUTING AND ITS DIGITAL SIMULATION.

4. The simulation would run faster if the module function implementation was divided into two sections. The first would be the initialisation, executed for T=0, and the second for subsequent iterations. It is obvious that many modules behave differently at T=0, as there is often initialisation within individual modules. The saving in time is illustrated by CON, the module which outputs a constant value. At T=0, this would update the output array, but in subsequent iterations no code is required for it whatsoever. The programming change required for this extension to the software would be minimal.

5. The use of a background routine for doing outputting to the printer during simulations would remove the frustration of the simulation speed being restricted to the extent where each iteration has to wait for the completion of a printer line. This could be done simply by running a spooling program beforehand. Such programs are available in "shareware".

6. A file directory should be given if the user-supplied filename is not found on the disk.

7. A major increase in power could be tapped by adding the **Turbo Graphix Toolbox** facilities to the program. This allows windowing, paging of screens, provision of axes, scaling and other features. However, this is at the expense of simulation speed, program size and compilation time. This extension would not be considered helpful to most users of the system, as it would add unnecessary complexity, however, it provides more opportunities for the enthusiast.

8.2. SOFTWIRE'S FUTURE WITH RESPECT TO NEW TECHNOLOGIES

Hardware advances are being made at such a rate that significant changes are being made every year. In the near future these may alter the importance of some of the design criteria used in **SOFTWIRE**. Some of the expected changes are discussed. In some cases there have already been appreciable changes since work first began on **SOFTWIRE** in 1985.

8.2.1 Improved Speed

The speed of the processor becomes critical only when the actual simulation is being run. At other times the response of **SOFTWIRE** to user input is adequate. If the simulation is not in real-time mode, faster processing merely reduces the time the user has to wait for the simulation to end. It is thus mainly a reduction in inconvenience, but it does encourage the user to use smaller (and hence usually more accurate) time increments. It makes it more practical to use a higher order integration routine. It also makes it more practical to include other more complex modules, and run simulations with larger net-lists. One may also include more error checks during the simulation, as these have to be done once for every module, repeated on each time increment.

If the simulation is being done in real time, the constraint is that the computer has to complete all its calculations and output for each time increment, within the time increment. This severely limits the minimum time increment possible. A typical minimum value for a simulation with an 8 MHz clock is 40 ms. This allows a real-time sine function of a maximum of 12 Hz, which excludes many useful applications.

8.2.2 More Memory

Memory has not been a constraint with the present **SOFTWIRE**. It runs on a 256 K machine, with enough memory over for larger net-lists than are envisaged in an educational environment. The only module which uses appreciable memory, is the DELAY function, which uses an array to remember previous values. For instance with a 1 ms time increment, a 1 second delay requires 5 K. Even now, much larger memory capacities are becoming commonplace.

If the output is to a printer, the simulation is severely restricted by the printer speed, which is considerably slower than a screen output. This limitation is serious in real-time applications. If there is enough extra memory, a buffer can be set up before **SOFTWIRE** is run. The buffer must be large enough that it does not full up before the simulation is finished. If the printer is printing a graphic output, it will be in bit-image mode, where it is slower, and usually uses (for an 80 column printer), about 60 bytes per time increment.

8.2.3 BETTER GRAPHICS

The screen is the most important means of imparting information to the user. The screen depicts the net-list, the operating parameters and the output. The ideal is to have all the information on the screen at once, so that the user does not have the distraction of having to recall information from a previous screen in order to interpret the output. Not only does the information need to be present, but it needs to be displayed in such a way as to appear uncluttered -which requires the information be grouped by position, size and colour. To do this, one requires a higher resolution screen, and one with more colours.

8.2.4 REDUCED COST

Since one of the prime aims was that the **SOFTWIRE** system be both cost-effective and extremely cheap, reduced cost will motivate more widespread use of it. It will also mean a shift towards the use of better hardware, which allows all the optional features of **SOFTWIRE** to be exploited. This includes the use of a colour monitor, a printer, a hard-disk drive and the **User Interface** card.

8.3. CONCLUSION

The enhancements which have been listed would make the system more useable. In addition, the use of the program over time should highlight other weak areas to work on, strong areas to consolidate, and features to include or leave out. In the meantime, however, the impression should not be given that the present implementation is not fully adequate in meeting the demands made upon it.

CHAPTER 9

MOTIVATION FOR THE INTERFACE UNIT

For reasons which will be given later in the chapter, it was decided to construct a hardware interface unit, which would extend the capabilities of the system, and would make it simpler to use. The details of this unit are covered in the next chapter. It will be referred to as the **User Interface.**

9.1. A BRIEF DESCRIPTION OF THE UNIT

The hardware consists of an instrument panel connected by ribbon cable to a hardware card, which interfaces to the computer by plugging into one of the slots provided on the motherboard. On the panel are potential dividers, meters, switches, lights, and extra analogue and binary inputs and outputs, which allow the user to construct actual hardware implementations of modules, instead of defining them in software. Provision is made for each device to be labelled, depending on the application. **SOFTWIRE** uses these inputs and outputs for its simulation in exactly the same way that it uses the modules defined in software. The ability to change the labels and modules related to the panel, make it a **virtual instrument**. The unit also provides circuitry for real-time simulation.

9.2. ADVANTAGES OF THE USER INTERFACE

9.2.1. ANALOGUE I/O DEVICES

The basic micro-computer configuration of a keyboard and console are useful for trained computer users, but are often an obstacle

to beginners. The intuitively natural input and output mechanisms of the panel, and the I/O transparency, help to overcome this problem.

9.2.2. REAL TIME

INTUITIVE VALUE

In some cases all that is required from a simulation is an output plot. In many other cases, information is contained in the actual speed at which things have happened. This information provides a more intuitive understanding of a process. It is often the most important component of understanding, and is close to the central philosophy of **SOFTWIRE**, which is to develop a more intuition-based approach to teaching.

Consider the principle of feedback and stability in closed-loop circuits. The simulation could be one on the **User Interface** unit where the user has to use the potential divider as a control. One of the panel meters represents a required target value, and the other meter represents the present value. Within **SOFTWIRE** there would be a transfer function relating the control to the output voltage, including a delay function. The user would be required to track the target voltage. A short attempt at this could easily bring out the difficulties involved, and could also illustrate the steps that need to be taken to overcome them. An experiment such as this done in real time would emphasis the need for speedy response, and give a more lasting impression of the results of instability.

NECESSITY

In cases where the user interface module is being used, real-time may be more practical. For instance, when the user is inputting a

voltage via a potential divider, it is often easier to have a slower simulation, and to have a time correlation between the user and the simulation.

If the user interface is being used, and the user has included an external circuit module which is time dependent (such as a function generator, integrator or energy-storing device), then without real-time, the results are meaningless.

CHART RECORDER

Although this application, which was covered in Chapter 3, is not primary to the philosophy of **SOFTWIRE**, it is a spin-off requiring no extra work to implement. In this application, the real-time facility is used to govern the intervals at which recordings are made. There is application for this in teaching instrumentation, and in observing transients or long-term behaviour of devices. In an experiment done in PART 3, this facility is used to turn the virtual instrument (the **User Interface**) into a capacitance-measuring device, using the real-time facility to measure the rise time of an RC circuit containing the capacitance being measured.

9.3. CONCLUSION

There is considerable potential in the addition of the interface unit, although it is not a pre-requisite for the effective use of **SOFTWIRE**.

The design of the interface is the subject of the next chapter.

CHAPTER 10
THE HARDWARE IN DETAIL

The two main components of the hardware are the card, which fits into a slot in the motherboard of the IBM PC, and the cabinet with instrument panel and input / output connecters. The two components are connected via a ribbon cable, and D-type 25-pin plugs, with connections as shown in Figure 10.10.

10.1. THE MOTHERBOARD CARD

This contains circuitry for the functioning of an analogue-to-digital converter, a digital-to-analogue converter, input and output latches -all used for I/O- and a programmable timer, used for real-time synchronization. The logical layout of the board is given in Figure 10.1.The buffering and decoding to the range $310-$31F are based on the **IBM Technical Reference Manual** [1], with a few amendments.

THE USER INTERFACE CABINET AND CARD SLOTS WITH CARDS

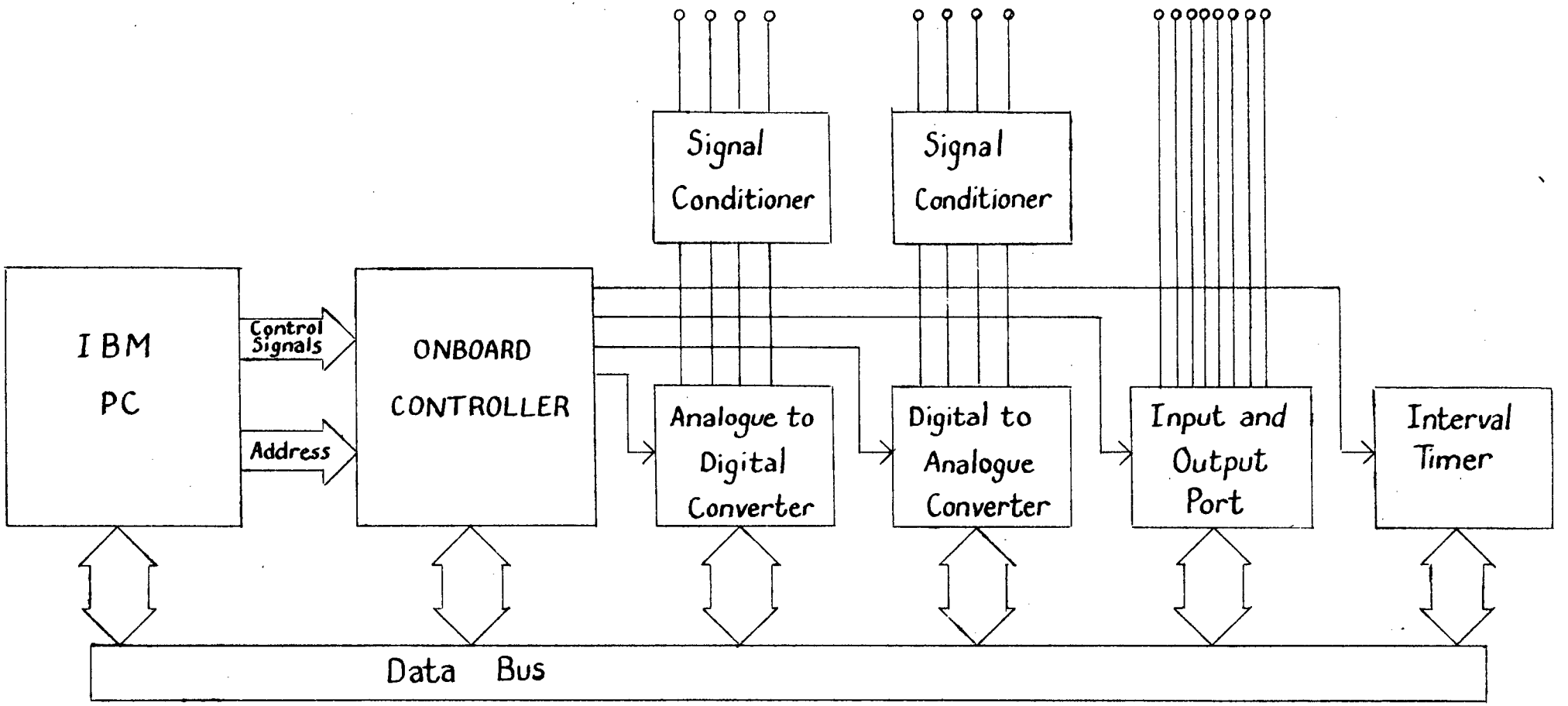


FIGURE 10.1

THE LOGICAL LAYOUT OF THE MOTHERBOARD

10.1.1. PHYSICAL DESCRIPTION

This is a printed circuit board, with a double-sided edge-connector which is 31 pins long. This plugs into the motherboard, and has direct access to the main CPU bus. From this, address, data, clock and control signals are obtained.

10.1.2. SIGNAL BUFFERING

Schematic 1. shows how the circuitry on board buffers all input and output lines. The input (signal and address) lines are buffered by two octal tri-state drivers. The data lines are buffered by an octal bus transceiver. This provides protection to the motherboard, and increases the fanout of lines such as the data lines and the lowest two significant bits, which drive big loads.

10.1.3. ADDRESS DECODING

Schematic 1 shows how the Decoding Circuitry enables the required chips. First, the hexadecimal range $310-$31F of port addresses is decoded. This is the range which **IBM** has allocated to the prototype board. The output from this is used to drive the data buffer, and the next stage of the decoding.

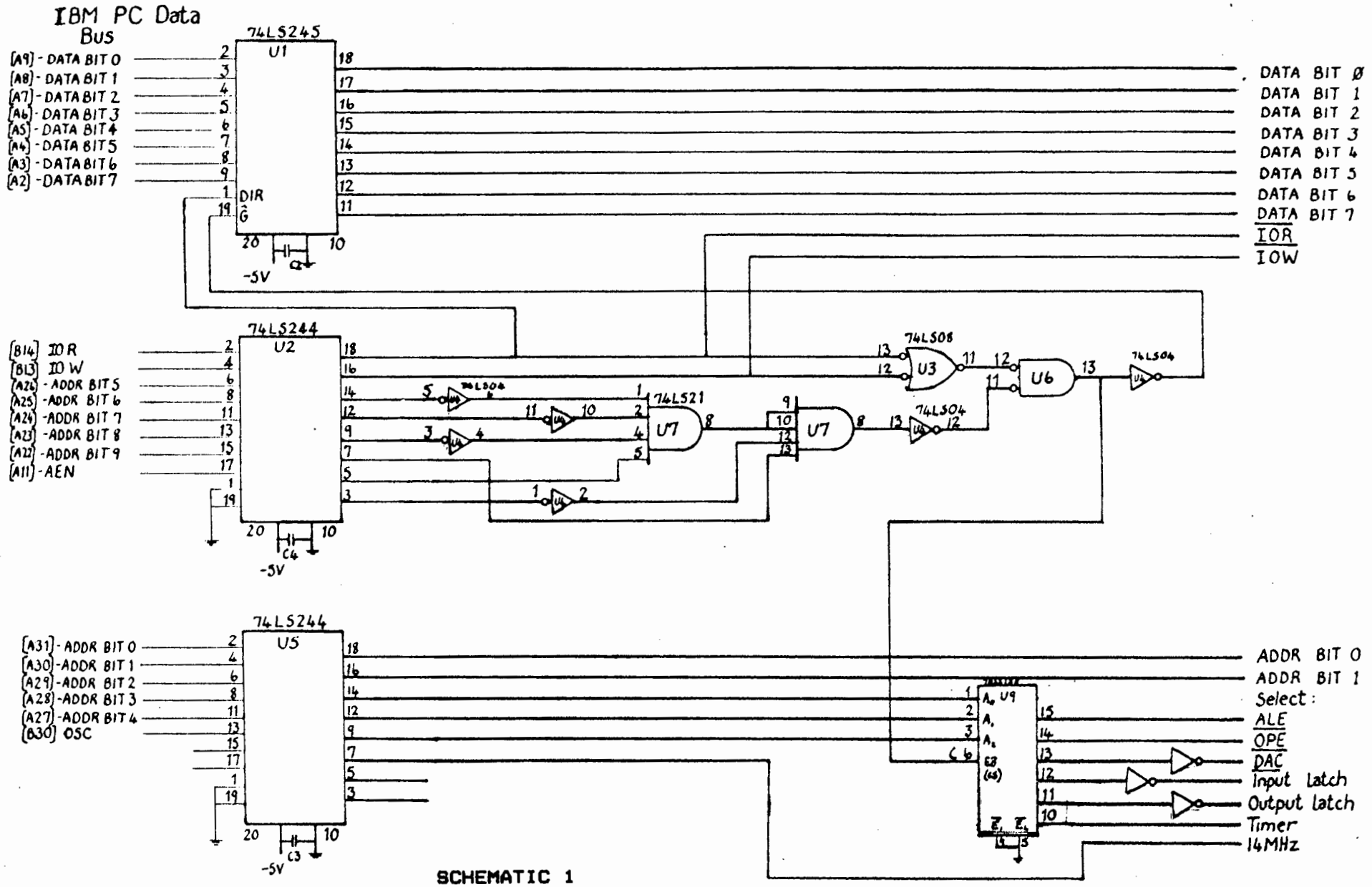
This next stage uses a 1-of-8 decoder to enable whichever of the set of devices has been accessed. A hex inverter is available for chips which require positive triggering. There are five address lines available within the decoded space. The most significant three of these are used for the 1-of-8 decoder. The two least significant are made available to each set of devices. Further decoding is thus specific to the different sets of devices, and will be covered individually.

Table 10.1 shows the addresses which activate each device.

Address bit 9 8 7 6 5 4 3 2 1 0
Chip Select: CS= 1 1 0 0 0 X X X X X_2 i.e. $300-$31F

Device	Address (Hex)	Address (Binary)
ADC (ALE)	300-303	11000 000XX
ADC (OPE)	304-307	11000 001XX
DAC	308-30B	11000 010XX
Latch In	30C-30F	11000 011XX
Latch Out	310-313	11000 100XX
Timer	314-317	11000 101XX
(unused)	318-31F	11000 110XX

TABLE 10.1 DEVICE ENABLE ADDRESSES



SCHEMATIC 1

BUFFERING, DECODING AND EDGE-CONNECTOR PINOUTS ON THE MOTHERBOARD

The different sets of devices mentioned are discussed in full in the sections following:

10.1.4. ANALOGUE-TO-DIGITAL CONVERTER

The converter and associated circuitry is shown in Schematic 2.

(a) **Features**

A **National Semiconductor ADC 0809** was used to do the conversion. This is an eight-bit converter with a maximum of 0.5 LSB un-adjusted error and no missing codes. It features eight input channels with on-chip multiplexing. The output is tri-state, and can thus be connected directly to the data bus.

(b) **Supply Voltage**

The chip requires only a single supply voltage and operates ratiometrically. This means no zero or full-scale adjustment is required, and the input potentiometers used can be driven from the supply voltage and guarantee readings covering the full digital range.

(c) **Clock**

A Schmitt Trigger, together with an RC circuit, is used to provide stable clocking for the converter. The frequency is chosen at slightly less than the maximum rated value of 640 Khz. The clock output is passed through another Schmitt Trigger to clean the signal. This is shown in figure 10.2.

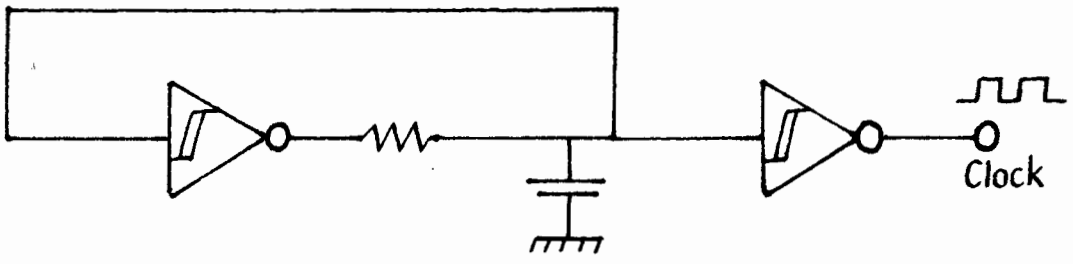


FIGURE 10.2 CLOCK INPUT

(d) **Input Channels**

There is provision on the converter chip for eight input channels, with on-chip multiplexing. This allows for address lines to be connected directly to the chip. In this case only four inputs are used. Two come from potential dividers on the **Interface Cabinet** and so are used ratiometrically, with a 0 to 5V range. The other two also come from the cabinet but from user-supplied circuitry. They have a -5 to +5V range.

(e) **Protection**

Protection for all the channels comes from opamps as voltage-followers at the board inputs, and from diodes and resistors at the **ADC** inputs, configured to limit the current for over- and under-voltages. The use of voltage-followers at the inputs allows larger value protection resistors to be used, without affecting the accuracy. The value of the resistors as a proportion of the input impedance must not exceed 1/(255*2), or there will be error in the reading of the **ADC**. (The value 1/(255*2) is specified by the 8-bit word used.) It is preferable for there to be a safety margin of about 10x as well.

(f) **Signal Modification**

The channels with the bipolar range are further level-shifted and scaled with op-amps, to give the required 0 to 5V inputs to the **ADC**. This results in a voltage inversion. The simplest solution to this was to wire the potential dividers so that they were naturally inverted too, and then correct the inversion of all four channels with a single software inversion function. Figure 10.3 shows one of the bipolar channels.

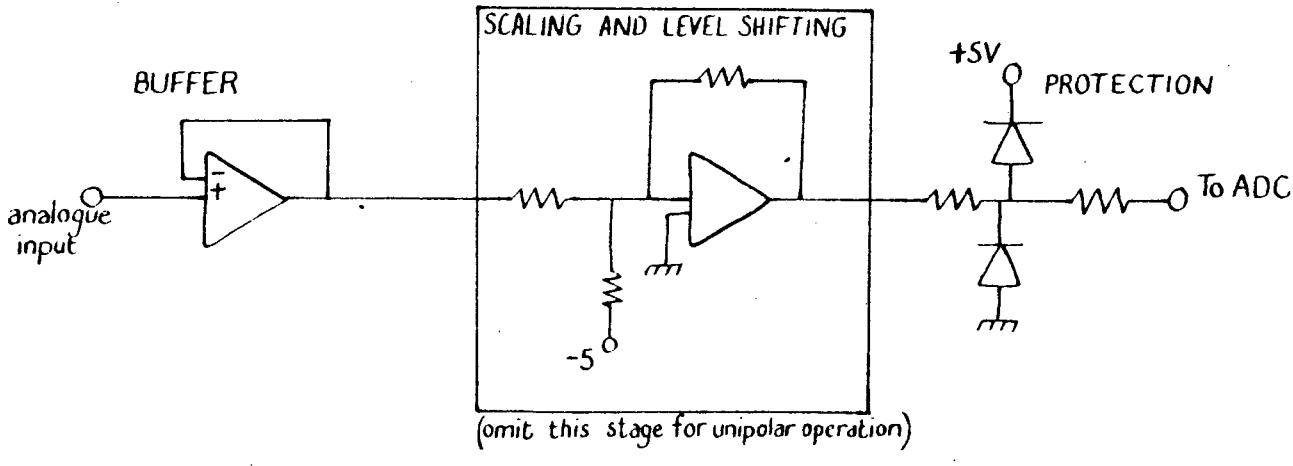


FIGURE 10.3 INPUT PROTECTION AND MODIFICATION

(g) **Decoding**

As mentioned in the previous section on decoding, there are eight enable lines available to the different sets of devices. The **ADC** uses two of these. The first line is enabled to start the conversion. At the same time, it latches the channel address on via the two least significant address bits. This line is referred to as **ALE** (**A**ddress **L**atch **E**nable). The second enable line, **OPE**

(Output Enable) is used at the end of conversion to signal the chip to place its data on the bus. Since this second line does not require the two least significant address bits, three of the four addresses in the address space are wasted. However, this is not a problem as there is not a shortage. The conversion time is determined in software. The process can be shown as follows in Figure 10.4.

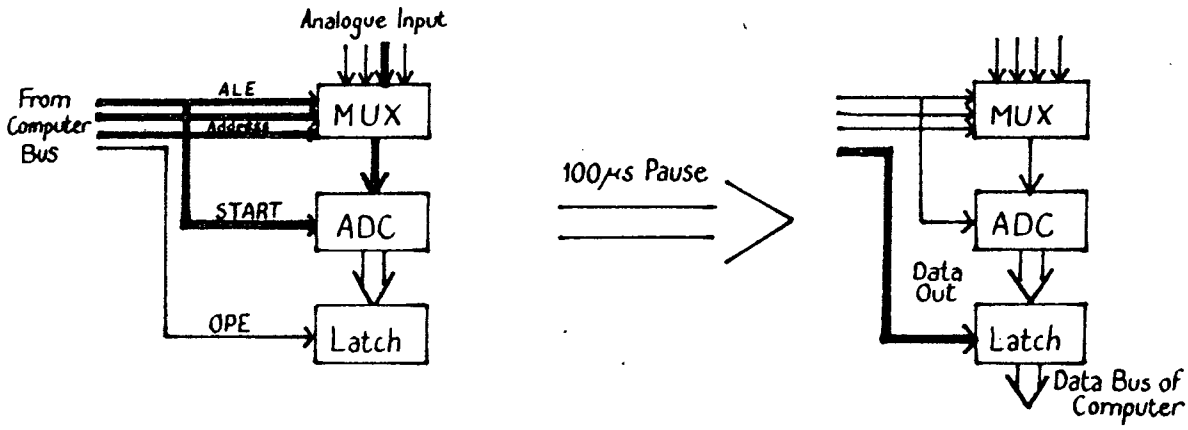


FIGURE 10.4 THE BASIC CONVERSION PROCESS

(h) **Calibration**

This is not necessary on the two ratiometric inputs. On the other two inputs it is not necessary if the resistors used in the scaling and level shifting are exactly equal, so if precision resistors are available, these may be used. It is preferable to use resistors all of the same value and, for the scaling resistor, to use two in parallel, as this reduces deviations. The precision must be better than 0.39%, as this is the resolution of the 8-bit converter (1/256*100%).

Usually one does not have such resistors, and must either select like values manually, or use the preferred method of calibration, with variable resistors. Since there are three resistors involved

-one for level shifting, and two for scaling- two calibrations are needed. The interdependence of the two can lead to endless complications, unless this inter-dependence can be eliminated. This is done by making the feedback and shifting resistors variable. The input is first set with +5V. The op-amp should then be calibrated with the level-shift resistor, so that the output is at 0V. This is when the two input resistors to the op-amp cancel completely. Note that there is no interaction through the feedback resistor, since there is no current flowing through it. Once the one resistor is calibrated, the other one is easy. The current nulling is shown in Figure 10.5.

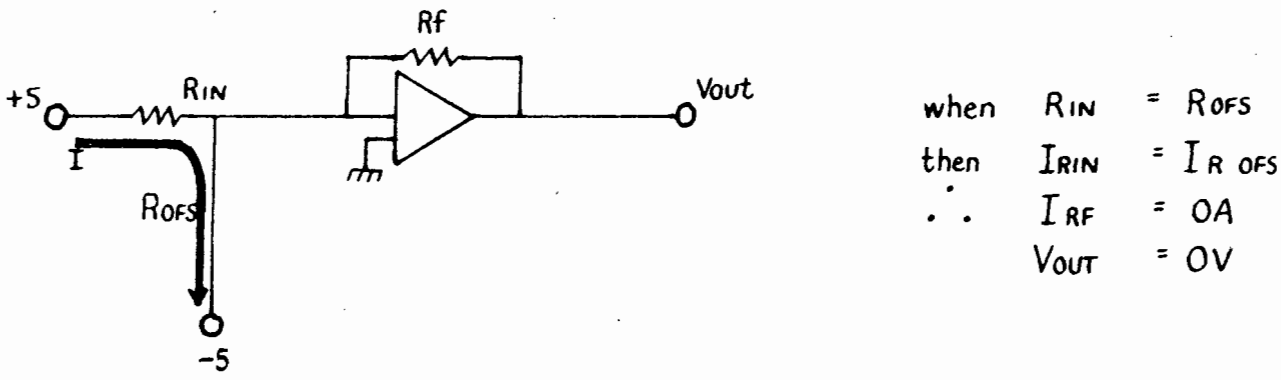


FIGURE 10.5 CURRENT NULLING TO SIMPLIFY CALIBRATION

(i) **The Question Of Range**

On the bi-polar voltage range chosen, there are 40mV increments, and because there are 256 levels (an even number), one cannot have steps which simultaneously are symmetrical around the zero point, and pass through the zero. A symmetrical configuration will step from -20mV to 20mV. A configuration which passes through zero will range from -5.00V to +4.96.V or from -4.96V to +5.00V, depending on whether an extra bit is used in the positive or negative pole. The three options are as follows in figure 10.6.

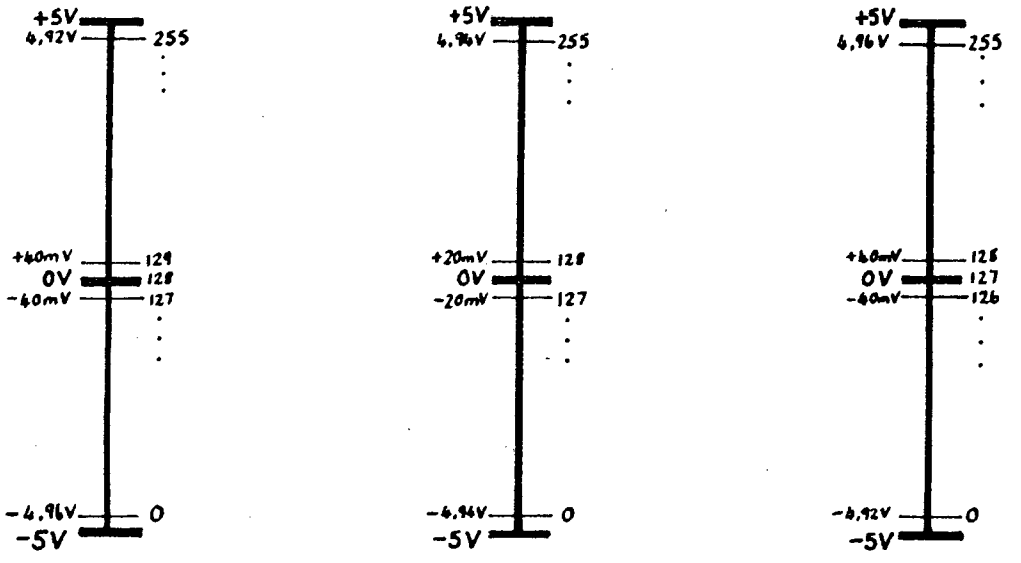
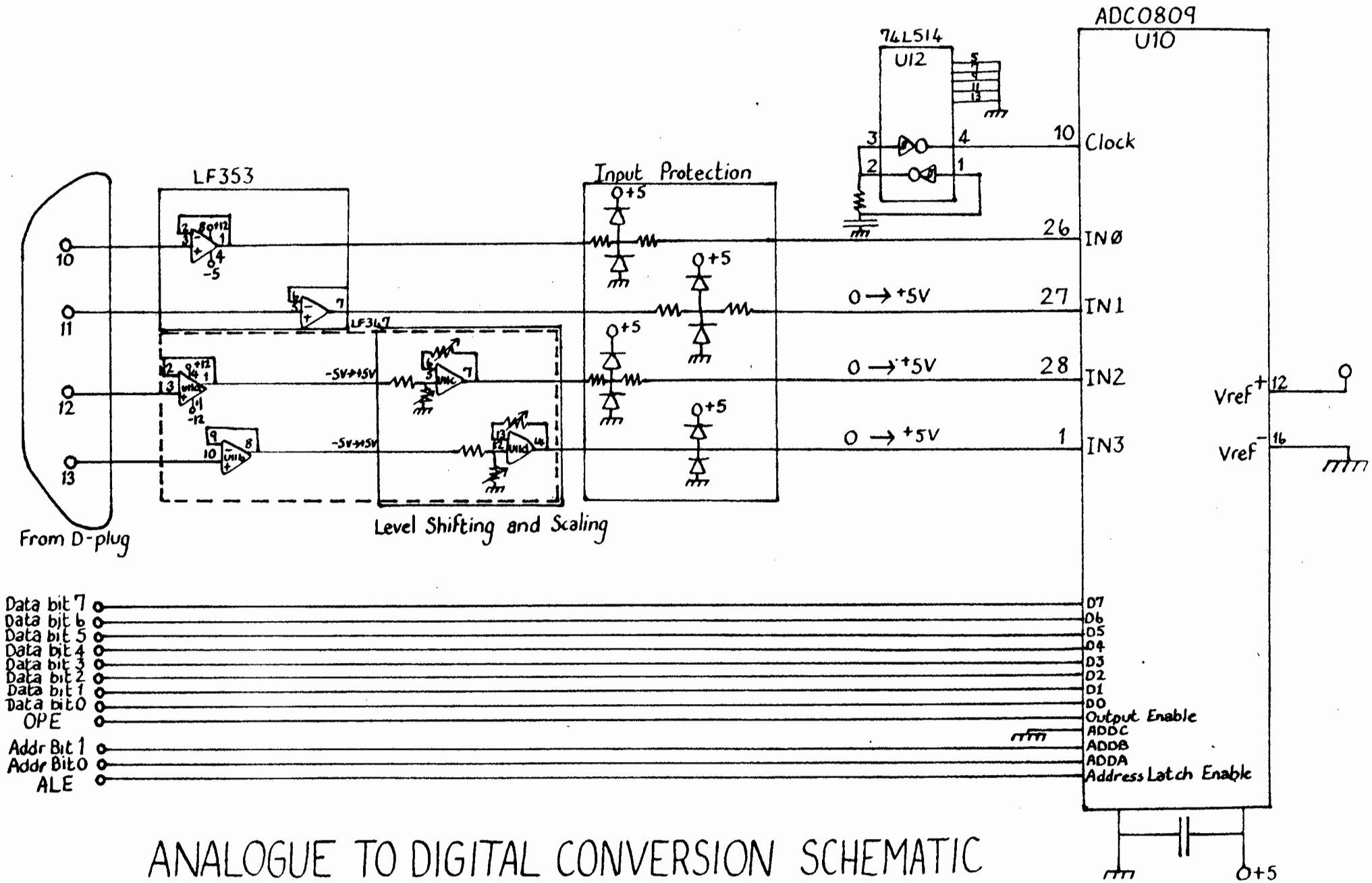


FIGURE 10.6 THE POSSIBLE RANGES USED

In this case, it was decided that the zero-point was important. Although it is obviously preferable to have symmetry in the absence of any trade-offs, the trade-off here of the zero crossing is very important. This is because many electronic devices (such as the comparator and the idealised rectifier) behave very differently depending on the voltage sign, and the point of change of sign is crucial. For educational purposes then, one must be able to detect this zero crossing. This means that the least significant bit now represents -4.92V, and the most significant bit represents +4.96V. There is still symmetry in the sense that the transitions of the device are at the same absolute values for both signs, except for zero and +4.96V. The only difference is that the positive range extends one bit more. This bit only represents a 0.78% difference between the maximum positive and negative value (1/127*100%), which from an educational point of view has very little significance.

The choice of resistor magnitude should be made so that the effect of bias current is negligible, but with the **LF 347** chosen, this current is extremely small, at 50pA. This has virtually no effect on the calibration.



ANALOGUE TO DIGITAL CONVERSION SCHEMATIC

10.1.5. DIGITAL-TO-ANALOGUE CONVERTER

There are two approaches to the problem of multi-channel digital-to-analogue conversion. The first is the analogue-switching option which involves one converter, an analogue multiplexer, and a sample-and-hold device for every channel. The second, digital-switching option has a digital multiplexer, and a latch and converter for every channel. The two options are shown in Figure 10.7.

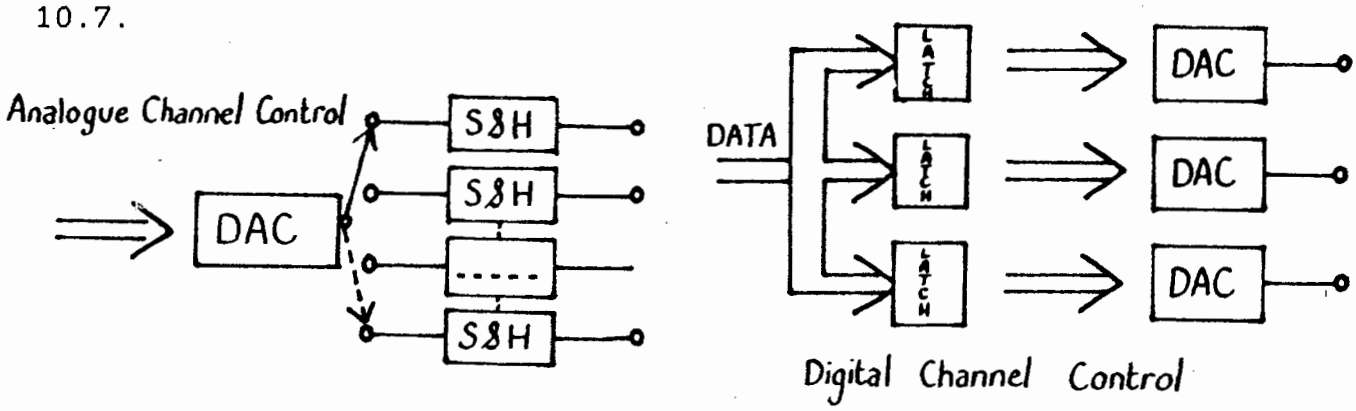


FIGURE 10.7 ANALOG AND DIGITAL MULTIPLEXING OPTIONS

The first approach seems to be the more obvious, as one considers the converter chip to be the expensive resource which must be optimally utilised. This approach was considered and tried. However there were numerous problems:

(a) Sample-and-hold chips were found to be difficult to obtain in small quantities, as they are normally ordered in bulk. They are also expensive[1].

(b) Sample-and-hold circuits have many design criteria, such as aperture (and rise time), droop, overshoot and distortion. This requires precision parts and careful design.

(c) If a single chip is not used, the result is bulky and has a high component count.

[1] At the time, the popular sample-and-hold chip, the **LF347**, was R15.00, whereas the convertor chip, the **DAC 0800** was only about R10.00.

The second approach turned out to be both cheaper and far easier to design, and allowed substantially greater design tolerance. It features:

(a) Zero "droop". This is the voltage drop at the output over time. This is unavoidable in the sample-and-hold circuit from capacitor leakage and op-amp bias current. This allows great flexibilty in the time increments used in real-time simulation.

(b) Negligible overshoot. This is important, since overshoot may cause the circuit a user is designing and testing with **SOFT-WIRE** to behave unexpectedly i.e. it may cause the output to trigger a threshold detector.

(c) A significantly smaller aperture time than is necessary. This is a maximum of 900nS (100nS for the **DAC** to settle, and 800nS for the worst-case rise time of the op-amp), which is fast since **SOFTWIRE** only updates each **DAC** every few milliseconds, and the **ADC** requires 0.1mS per conversion.

(d) Reduced cost. The latch and **DAC** combined are cheaper than the sample-and-hold chip.

The digital approach is now discussed:

(a) **Decoding**

The section has a 1-of-8 decoder (with only two inputs used) using the two least significant address bits, which activates one of four octal latches. Thus all four latches receive the data on the bus, but only one is enabled to transfer it to its **DAC.**

(b) **The Digital-To-Analogue Chip Used**

Each latch drives a **National Semiconductor DAC0800** (or any **DAC-08** equivalent). This is a single channel, 8-bit device. It is popular, in good supply and cheap. Although it has two-quadrant multiplying capabilities, in this application the input voltage reference is fixed positive and it outputs a current in a single quadrant (i.e. output is uni-polar).

(c) **Output Modification**

The current output from the **DAC** could be converted to a voltage by a resistor. However it is better to use an op-amp configuration since it provides a buffer. This makes the output voltage far less susceptible to load variation, and provides protection. It also boosts the power output capability. In the "virtual earth" configuration, level shifting and scaling is also easy to achieve, and the virtual earth ensures that the **DAC** remains within its region of compliance. In this application, two 0 to 10V output ranges were supplied, which drive panel meters. There are also two -5 to +5V outputs provided for the use of the user. These required level shifting.

(d) **Calibration**

This is necessary on all four outputs. Two of the outputs are unipolar, and since they have a fixed point -the zero- they need only one variable resistor for calibration.

The other two outputs are bipolar, and since there are two resistors involved -one for level shifting, and one for scaling- two calibrations are needed. As with the **ADC**, the interdependence

of the two can lead to endless complications, unless this inter-dependence can be eliminated. To do this, the input of the **DAC** is set with $FF, corresponding to maximum output current. The op-amp should then be calibrated with the level-shift resistor, so that the output is at 0V. This is when the current from the **DAC** and the level-shift resistor to the op-amp cancels completely. Note that there is no interaction through the feedback resistor, since there is no current flowing through it. Once the level-shift resistor is calibrated, the other one is easy.

There is another variable value in a **DAC** -the reference current- which is determined by a reference voltage and resistor. Calibration of this is not important, since the calibration procedure already mentioned supercedes this. There is a consideration though, that a high reference voltage produces a high output current. As long as this is within acceptable distortion limits of the **DAC**, this is preferable, as it reduces output noise and minimises the effect of the bias current.

(e) **The Question Of Range**

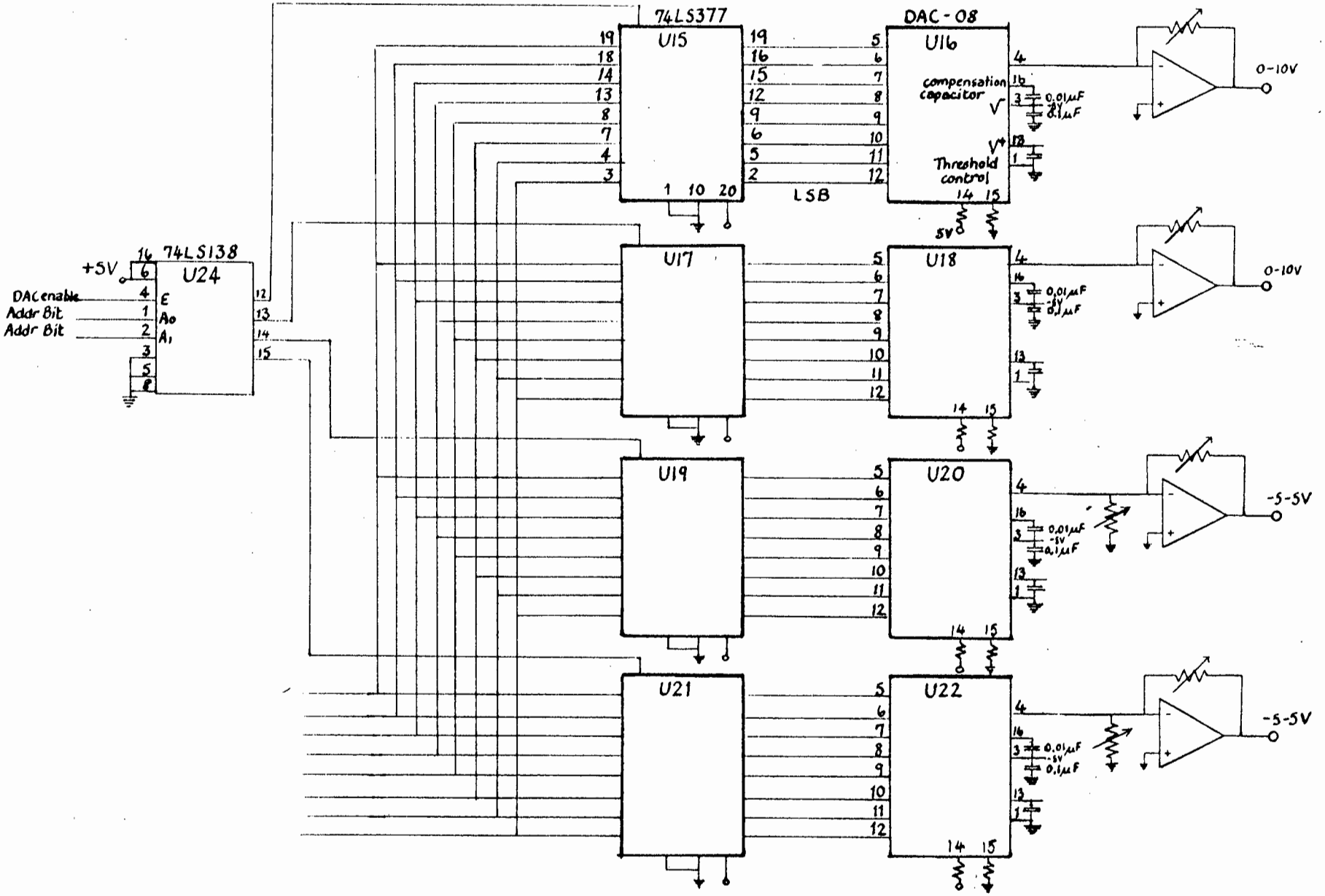
The same question arises as for the analogue-to-digital convertor, and the considerations were thoroughly dealt with in that section. The same conclusion was reached for the **DAC**, and for the same reasons -especially the need for a true zero. It was also seen as being important that the output levels of the **DAC** corresponded with the input levels of the **ADC**. If this were not so, one would loop a voltage out of the **DAC** output and back into the **ADC** input, and get a different reading from the one put out. This would make no sense to a student!

(f) **Dithering**

Each **DAC** channel is updated in discrete time and level intervals. The time intervals are of the order of milliseconds, and the step level-changes may be up to 10V. The causes a very discontinuous output, which is not desirable for driving certain circuits. The concept of dithering was examined as a means of reducing the impact of the step changes. However it was decided that in the common uses dithering would not be useful. It does not help with time-independent modules, and integration modules do not benefit. On the other hand a device such as a threshold detector may cause glitches at the dithering frequency.

SCHEMATIC 3

CIRCUITRY RELATED TO THE DIGITAL-TO-ANALOGUE CONVERTER



10.1.6. INPUT AND OUTPUT LATCHES

There were two options here. The first is to use an **Intel 8255** chip (or equivalent, such as the **MC 6821**), with different ports programmed for input and output. This is a neat and easy-to-implement solution. Allowance is made for 24 I/O bits.

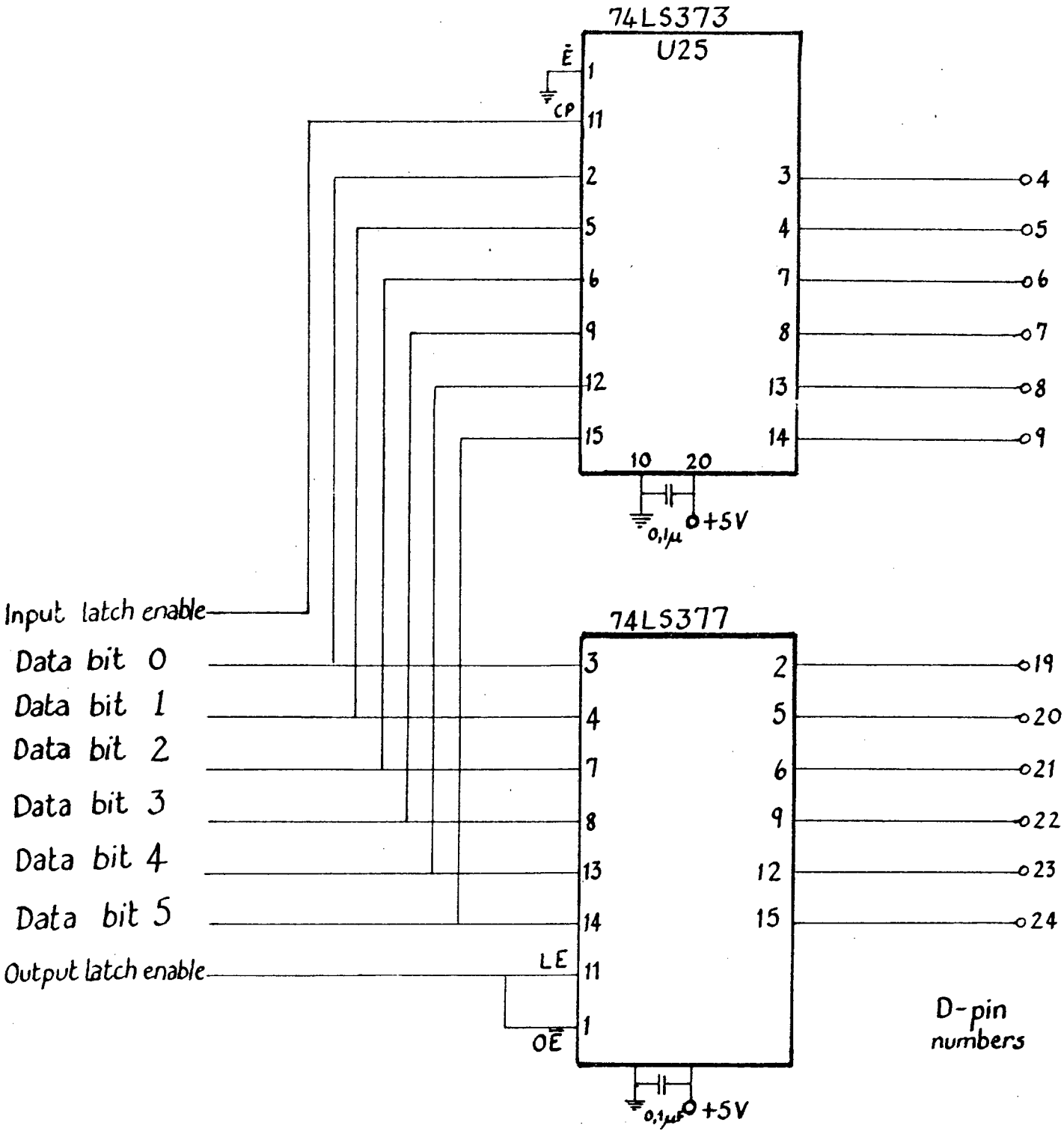
The second option is to use two octal latches. This is cheaper, and allows easier and cheaper replacement in case of an error on the part of the student using the latches.

The latter approach was adopted, largely because of the relative abundance of the chips at the time of development and because four latches were already being used for the **DAC**'s. The input latch requires a tri-state input to the data bus.

The address space allows for four input and four output octal latches, but only one of each was deemed necessary. Further, only six of the bits on each latch were used, due to a limited number of pins on the output plug. Bit values are eight times less efficient than analogue switches in terms of their information capacity per pin used.

SCHEMATIC 4

CIRCUITRY RELATED TO THE INPUT/OUTPUT LATCHES



10.1.7. INTERVAL TIMER

This set is slightly different from the others since it is not part of the **User Interface** and is not available to the user. It was necessary to put it on the same board. Its logical link to the board is that most real-time applications are likely to involve the **User Interface**.

The timer consists almost entirely of a single, powerful chip -the **Intel 8253**. Since this chip allows a maximum clock input of 2 MHz, and the **IBM** system clock is 14 MHz, a divide-by-eight chip has been used to step-down the clock rate. An **8253A** chip could have been used, allowing higher input clock frequencies, but this is more expensive and not as readily obtainable.

The chip is part of the **8088** family, and so easily interfaces directly to the control lines of the bus. It has three counters, which are distinguishable by a two-bit address (the fourth address specifies the mode register of the chip).

The first counter, called Counter 0, accepts the incoming clock impulse. It is used in the repeated-divide-rate mode. Once set, its output goes low for one clock cycle every full count. This output has been used in this case to clock twice per millisecond. The output then goes to Counter 1, which is configured in exactly the same way as Counter 0, except that its divide rate is not fixed, but is determined by the user. The two counters are thus connected in a cascade. This gives an extraordinarily large range of interval periods from 0.5 ms to 32 second. The range could be extended by making both counters variable, but this is both un-necessary, and not as neat an implementation.

The number of events output at Counter 1 is counted by Counter 2. The output to this counter is not used, as the count is read directly by **SOFTWIRE**.

The logical connection of the timers is shown in Figure 10.8.

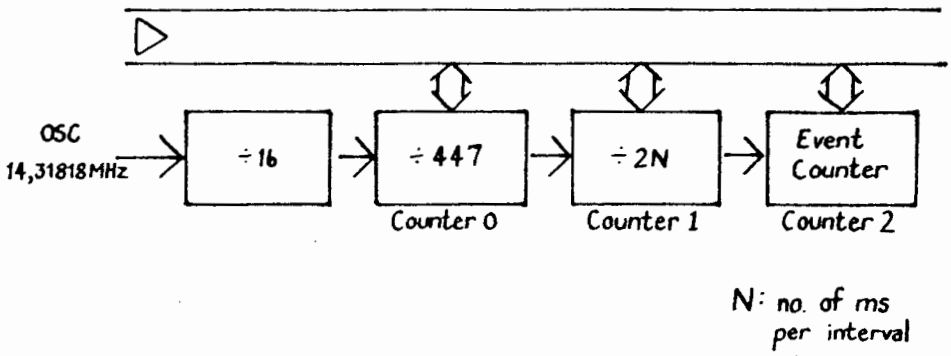
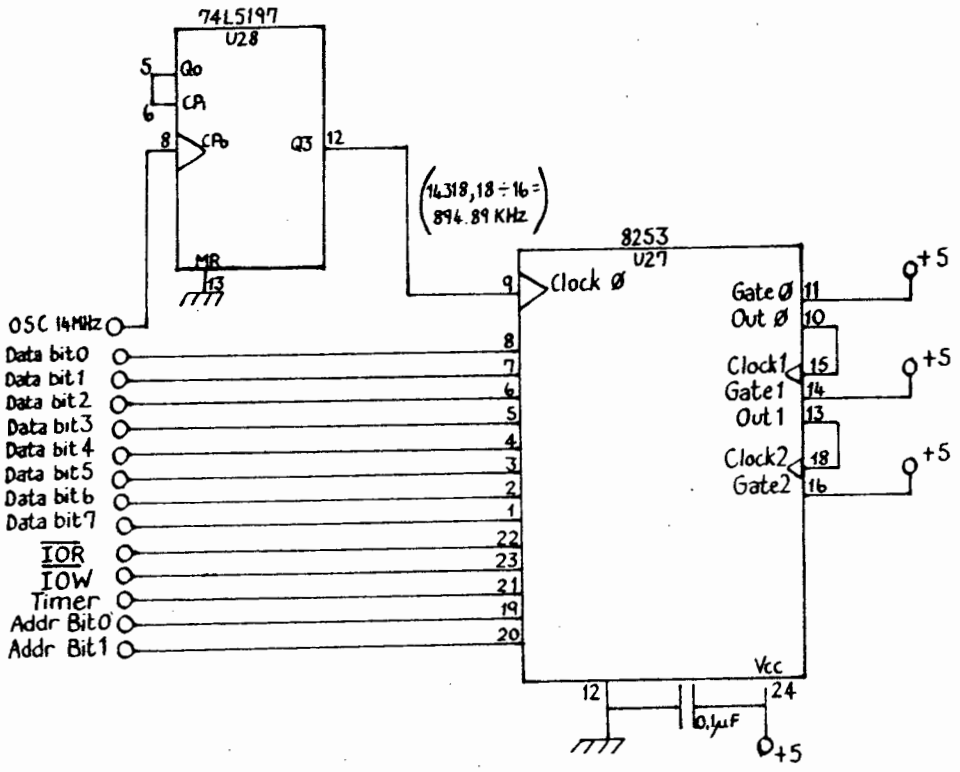


FIGURE 10.8 TIMER CONNECTIONS

SCHEMATIC 5

CIRCUITRY RELATED TO THE INTERVAL TIMER



PROGRAMMABLE INTERVAL TIMER CIRCUITRY

10.1.8. OUTPUT PLUG

This is a 25-pin D-type plug. This was chosen because of its availability arising from its popularity with **Centronics** and **RS-232-C** interfaces. The number of pins is restrictive, and placed slight limitations on the number of I/O channels. However the limitations were worth the practicality, and at least more use is made of the 25 pins than in most other applications.

Figure 10.9 gives the pin-out of the D-type plug.

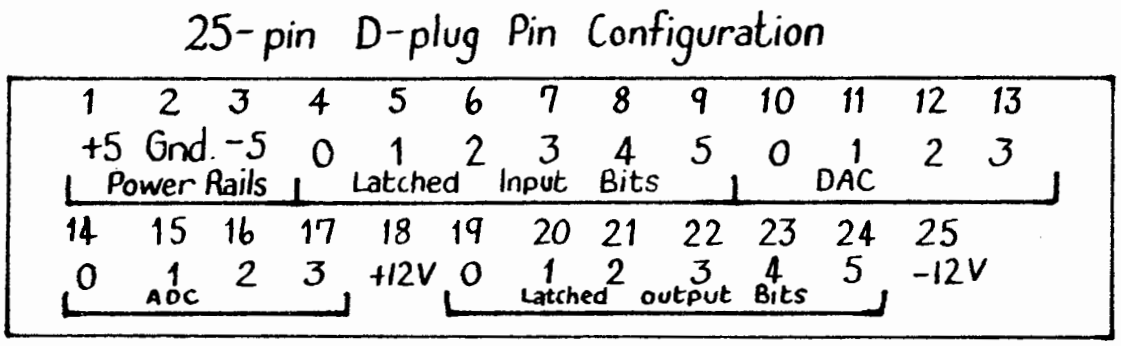


FIGURE 10.9 PIN-OUT OF THE MOTHERBOARD EXTERNAL CONNECTION

10.2. THE INTERFACE CABINET

10.2.1. THE STRUCTURE

The structure of the interface cabinet -especially the front panel- was governed largely by personal taste[2]. Criteria used were also the logical grouping of similar inputs and outputs, and the convenient positioning of devices for ease of use. The prototype one has two panel meters and two potential dividers, three switches and four LED's. The switches are one pushbutton type, one single-pole, and one double-pole. The LED's are red, green and yellow.

Since cost and simplicity are important criteria, the cabinet has been made with no active components. Only plugs, slots, LED's, switches and meters are included. These are all off-the-shelf and easily obtainable.

10.2.2. LABELLING

The devices are not labelled, but provision is made beside each one to provide a label appropiate to the simulation in question. The labels specify what the device is representing, and may include markers or calibrations. This facility greatly enhances the versatility of the panel display, and allows the user to utilise symbols which are personally meaningful. However, it is easy to fall into the trap of neglecting to use adequate labels, and becoming distracted by having to remember them, to the point of forfeiting the intuitive value of the device.

The cabinet is described in Figure 10.10.

[2] For instance the prototype cabinet made had the inputs on the left, and the outputs on the right. This is designed to be positioned on the left side of the computer and used by left-handed people such as the author.

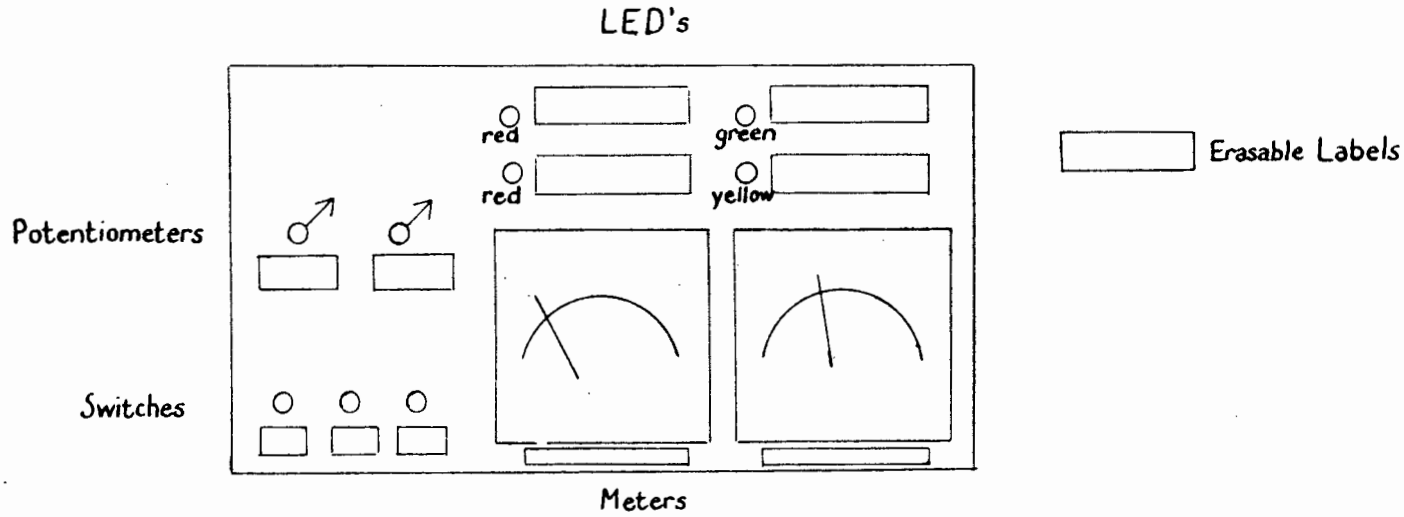


FIGURE 10.10 a) CABINET LAYOUT -FRONT VIEW

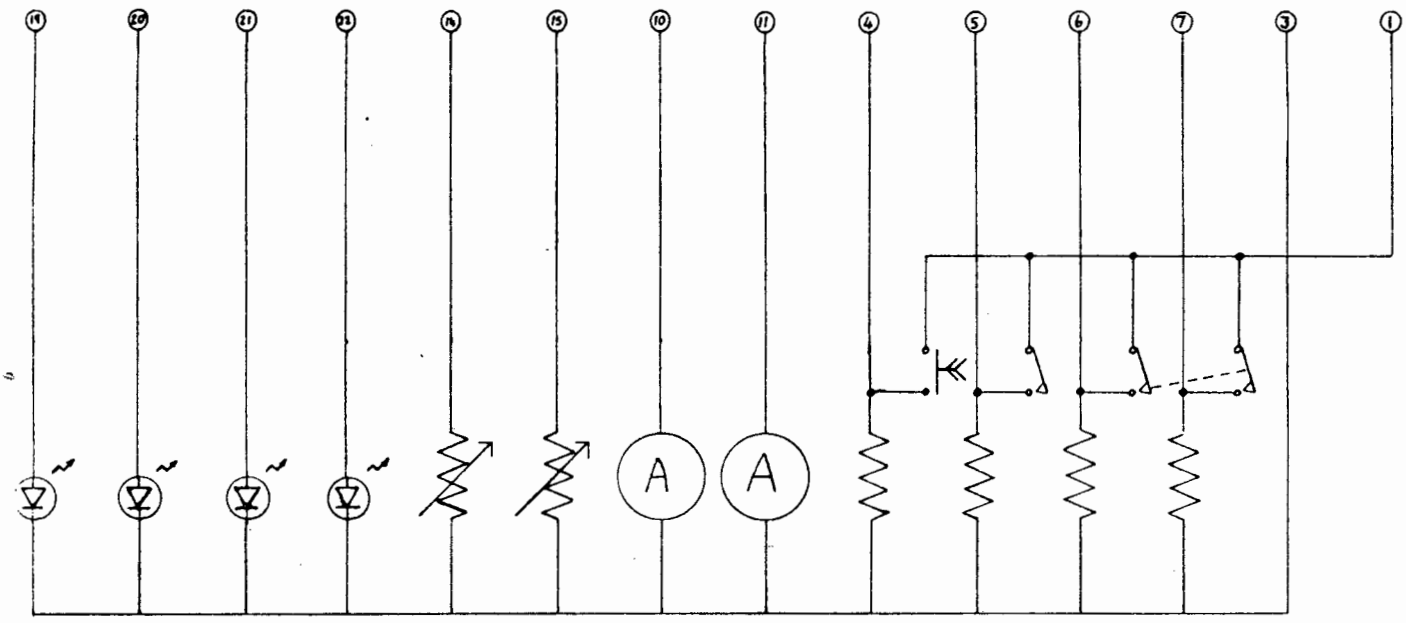


FIGURE 10.10 b) CABINET -CIRCUIT DIAGRAM

10.2.3. SLOTS

The cabinet is provided with two slots into which **Veroboard** "cards" may be inserted. These cards contain user-made circuitry. The slots provide power to the board, bipolar analog I/O and a TTL bit each for input and output. The cards are simply the standard **Veroboard** with copper rails, with no amendments except that they must be 31 pins wide.

Having two slots means that different design options exist. The first decision was whether to make all I/O channels interact with both slots. The restrictions on the number of channels made this necessary, so that more complex circuitry could be included in a single slot.

The next decision was whether or not to make the slots identical. It was decided not to, but to reverse the order of the bits and channels in the second slot. To understand why, consider the design of an integrator card with circuitry on which uses analogue input and output channel 1. If two such cards are used simultaneously, they will both be competing for channel 1. If the slots were identical, a different integrator, hard-wired to use channel 2 would be required for slot 2. This would increase the difficulties associated with producing boards used in learning, as twice as many different types of boards would have to be designed. Instead, the board plugged into slot 2 is automatically connected to channel 2. If one of the boards requires two channels, then the problem does not exist because one could then not use two boards anyway.

The slot pin configuration is shown in Table 10.2

SLOT PIN NUMBER	DESCRIPTION	D-PLUG PIN NUMBER
1	(unused)	
2	-12 Volts	25
3	-5 Volts	3
4	Positioning Key	
5	Ground	2
6	Output bit 1	23
7	Output bit 2	24
8	Output bit 3	(19)
9	Output bit 4	(20)
10	Output bit 5	(21)
11	Output bit 6	(22)
12	Input bit 1	8
13	Input bit 2	9
14	Input bit 3	(4)
15	Input bit 4	(5)
16	Input bit 5	(6)
17	Input bit 6	(7)
18	Ground	2
19	(unused)	
20	+5 Volts	1
21	Analogue output 1	12
22	Analogue output 2	13
23	Analogue output 3	(10)
24	Analogue output 4	(11)
25	Analogue input 1	(16)
26	Analogue input 2	(17)
27	Analogue input 3	14
28	Analogue input 4	15
29	(unused)	
30	+5 Volts	1
31	+12 Volts	18

1. D-plug pin numbers in brackets indicate overlap with panel instruments (connection is optional)
2. Slot position no. 4 contains a positioning key to ensure the card is correctly orientated.

TABLE 10.2 CABINET -USER SLOT PINOUT

10.3. EXPANSION CONSIDERATIONS

The design discussed in the chapter so far refers to a cheap and functional hardware interface. However, it was also constructed with the potential for expansion in mind.

10.3.1. LIMITATIONS

THE D-TYPE PLUG

All 25 pins of the D-plug are utilised. In order to increase the number of channels one must use a connector with more pins. The simplest upgrade would be to use the same type of plug, but the 37-pin version. This allows 12 extra pins, which could be used as four extra analogue outputs and four analogue inputs, and two extra bit-inputs and two bit-outputs.

DECODING

The present system uses the three most significant bits to identify a set of devices, and the least two to access entities within the set. For instance, the **ADC** may be considered a set, and each channel an entity. This limits each set to four entities. Expansion would require either more sets or more entities. To have more sets, expansion is already possible as not all addresses are being used at the moment. To have more entities, one would need to use two bits to address the devices, and three to address the entities. This is possible because the present design is based on logical simplicity rather than maximising the decoding range.

The devices could be grouped as three sets: the **ADC**, the **DAC**, and the **Timer** and the **Latches** together. The changes then made would

be minimal. The main 1-of-8 decoder would lose its least significant bit, and be used as a 1-of-4 decoder. This extra bit would be used to select entities within sets.

10.3.2. POSSIBILITIES

Expansion is possible in the following areas.

a) ADC : This would use the extra bit mentioned in the previous paragraph to provide up to eight channels. The ADC 0809 chip used has an 8-channel multiplexer on-board already. Four extra bi-polar or uni-polar inputs could be added in the same way as the previous one's. Changes to **SOFTWIRE** would be minimal, and would be confined to the procedure called ADC().

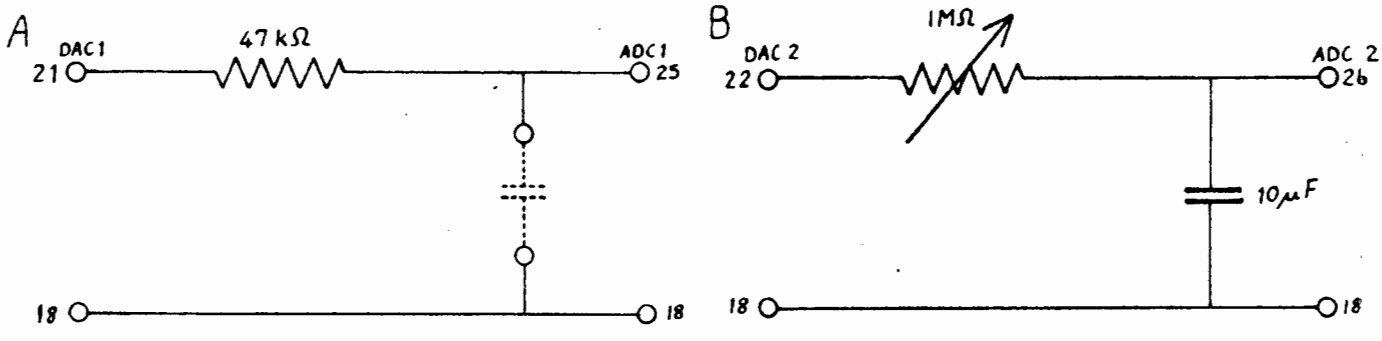
b) DAC : This would use the extra bit mentioned in the previous section on **DECODING** to provide up to eight channels. The extra bit would be input to the 1-of-8 digital multiplexer already provided. Four extra bi-polar or uni-polar inputs could be added in the same way as the previous one's, each with a latch, **DAC** and op-amp circuitry. Changes to **SOFTWIRE** would be minimal, and would be confined to the procedure called DAC().

c) Input And Output Latches : There is provision for two extra bits on both as the circuitry stands now. One could also add three more octal latches to the circuitry. The address space exists, but the decoding would have to be arranged. The best arrangement would probably be to have a 1-of-8 decoder. Four of the eight outputs of this would drive the **Timer** chip (using a NOR gate), and the other four would each enable a separate latch.

d) Any User-Supplied Circuitry : This could be driven by the unused output of the main 1-of-8 decoder provided. It would have an address space of eight coming from the three least significant bits.

10.3.3. DEMONSTRATION CARDS

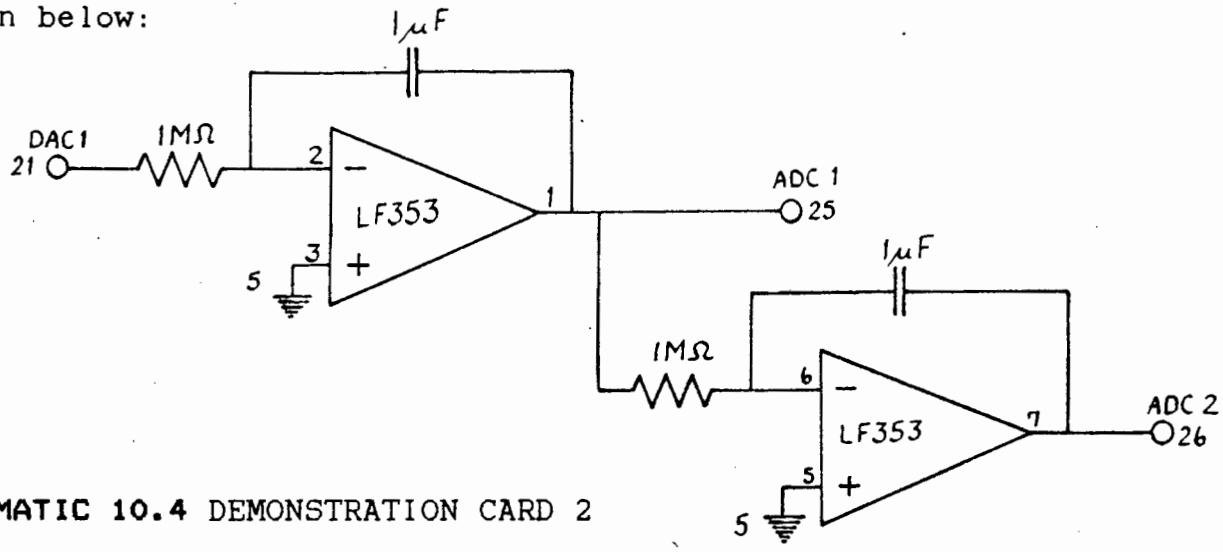
Three cards have been included in the package, to demonstrate their construction and use to the user. The first has two Gaussian (RC) low-pass filters on it. The first is built on Channel 1. It has a spring-loaded receptacle for east replacement of the capacitor. This may be used as a capacitance measuring device, since the resistor is fixed (the method is described in PART 3). Channel 2 has a fixed capacitor and variable resistor, for easy provision of adjustable time constant. An LED indicates power-up (this is mainly for demonstration). The schematic is given below:



SCHEMATIC 10.3 DEMONSTRATION CARD 1

The second board contains two cascaded integrators, each with a gain (RC product) of 1. Channel one outputs to the first intergrator, and input Channel 1 reads the output from this integrator, which also goes to the second integrator. Input channel 2 reads the final output.

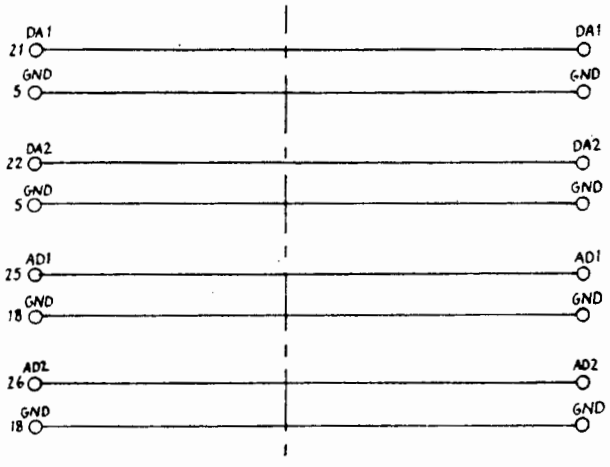
This circuit can be used as part of a state-variable oscillator, or one may simply use the first integrator. The schematic is given below:



SCHEMATIC 10.4 DEMONSTRATION CARD 2

The third card contains colour coded and labelled leads for connecting to signal generators, XY-Plotters and oscilloscopes. It has banana-plug and optional crocodile connections.

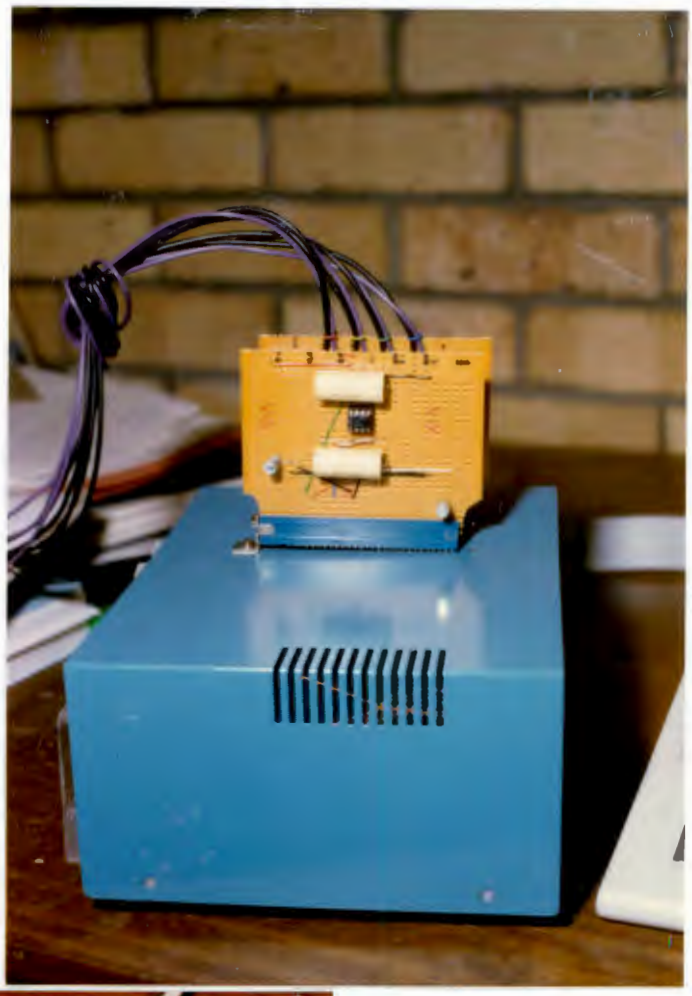
The boards are all made from commercially available **Veroboard**. Each one contains colour coding and thumb-nail sketches where appropriate, to indicate its function. The schematic is given below:



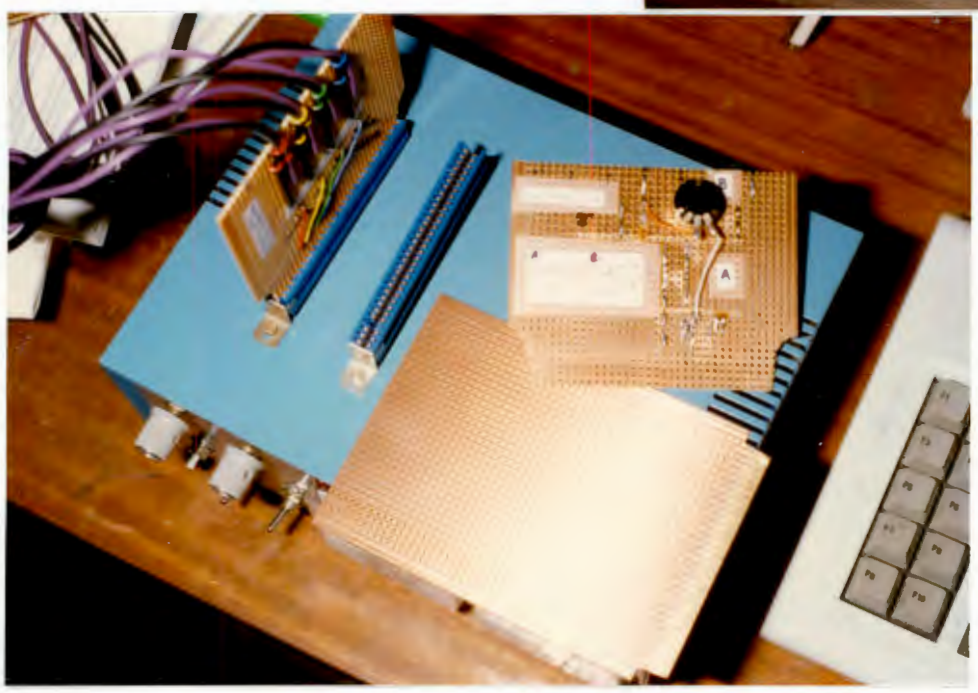
SCHEMATIC 10.5 DEMONSTRATION CARD 3

REFERENCE

1. IBM Technical Reference Manual (for the IBM Personal Computer).



Photographs showing the demonstration card plugged into the **User Interface** (right), and (below, from left to right), the multi-purpose connecter card, a commercially available veroboard card with edge connecter, the RC-filter card.



CHAPTER 11

ACCURACY AND OTHER MATHEMATICAL CONSIDERATIONS

11.1. IDENTIFYING IMPORTANT MATHEMATICAL CONSIDERATIONS

There are three important questions in any simulation. One must ask if the simulation is valid, reliable and accurate.

For the simulation to be valid, it must simulate what it purports to simulate. This requirement refers largely to the conceptual design involved in the simulation. To be reliable it must be above all **repeatable**. To be accurate it must be valid and be mathematically sound at the implementation level of the simulation.

The simulation is based on fundamental and well-known concepts, and the validity will thus not be examined further. The repeatability will likewise not be examined. In almost all the testing done on **SOFTWIRE**, simulations were done on systems with which the author was relatively familiar and the answer was already known.

Accuracy is an important question which needs to be addressed more thoroughly.

inaccuracy leads the student to believe that the modules of a system they have entered are responsible for a particular phenomenon, when in fact it is caused by intrinsic inaccuracies.

This problem is compounded when non-linear devices are modelled. The effect of error in non-linearities is not systematically documented, as the rules involved are significantly more complex than in the linear case. This is vividly demonstrated in the case of a chaotic system, or strange attractor (an example of the "Hennon Attractor" is given at the end of PART 3). The use of feedback also compounds the complexity of the issue.

In attaching a specific value to the accuracy requirements, it must again be emphasized that each situation imposes different requirements. The tutor must use discretion in each case, using guidelines provided in this chapter.

SUGGESTION FOR TESTING A SYSTEM SIMULATION

The first step would probably be to examine the net-list, and see if any problematic areas are apparent (by inspection). It is then suggested that the simulation is run, and the level of error judged for each case. For instance, if one was demonstrating instability in a system, inaccuracy would possibly not be such a problem, because the two important issues to convey would be firstly that the system no longer "behaves" after the point of instability, and secondly that the exact point of instability was not easy to find, but that a certain margin had to be included.

On the other hand, if one was showing that an (ideal) integrator with a sinusoidal input produces a cosine output, it would be important that the output remained true for long enough to demonstrate that the relationship was an analyically exact one.

It must be stressed that **SOFTWIRE** has been used extensively and has been found to be more than adequate in the applications tested. All the examples given in the PART 3 for use have been tested and give no problems when used in the way described.

11.3. TYPES OF ERROR

The types of error are well categorised by Thomas [1].

11.3.1 GLOBAL ERROR

The simulation makes extensive use of numerical approximation, with the consequence that there is inevitable error in the solution. This is **global error** -the difference between the true result and the calculated one.

11.3.2 LOCAL ERROR

There are two major components of global error which arise. When the output of each module is calculated, a local error arises. This results from error from simplifications used in modelling the module, truncation (or round-off) error from the mathematical limitations of the actual computer, and approximation error from using discrete (rather than continuous) time intervals.

11.3.3 PROPAGATION ERROR

When an output from one step is output to the next step, the local error associated with that step is propagated. This gives rise to **propagation** error, which is obviously an accumulative value.

11.4. CONTROLLING THE ERROR

11.4.1 ARITHMETIC IMPLEMENTATION

The error sources which can be controlled are limited. The truncation or rounding error could be reduced by working at a higher precision. In **Turbo Pascal** this involves using the binary-coded decimal compiler for precision at the cost of speed (this is not recommended), or if an **8087** chip is available, to use the **Turbo-87** compiler.

The magnitude of real numbers in **Turbo Pascal** ranges from 1E-38 through 1E+38 with a mantissa of 11 significant digits [2]. It is unlikely that many electronic education systems simulations will utilise the full range either in module outputs or in time increments. Further, even a simulation with 10 000 iterations should not accumulate error, from truncation alone, greater than 0.001%, which is far smaller than error from other sources. **TURBO-BCD** pushes the range to 1E-63 through 1E+63 with 18 significant digits [3]. **TURBO-87** has a range of approximately 4E-307 through 1E+308, and it has 16 digit accuracy [4].

To change from one to another, the procedure remains exactly the same for the user, except that a different compiler is used. Even files saved by **SOFTWIRE** on one system work on another since they are saved with an ASCII format compatible to all three [5].

It is definitely not necessary to use the specialised compilers for controlling error in normal use of the package. The loss of speed in the former case, and the extra cost in the latter are outside the design criteria of this package.

11.4.2 THE TIME INCREMENTS

The discrete-time approximation error can be reduced by the user, by his reducing the time interval -the trade-off being the time taken to complete the simulation. This is a run-time choice which is simple to select.

11.4.3 INDIVIDUAL MODULE IMPLEMENTATION

The programmer has control only of the code of each module, and therefore the choice of the type of mathematical implementation. In many modules, such as the simple summation modules, there is an obvious best option. In many, such as a voltage-controlled oscillator or integrator, this is a more complex issue. The next chapter will deal with some of the module implementation issues.

A CAUTION WITH NON-LINEAR MODULES

As has been mentioned earlier, the error arising from the linear blocks is well behaved, and generally can be reduced by judicious module implementation. This is not the case with non-linear blocks, and the user must be cautious in their use. There are many reasons for this:

In modules which output any form of ideal step-functions, the output undergoes a transition from one level at one time interval, to another at the next time-interval. The slope of this transition is ideally infinite. However in discrete-time simulations it appears to subsequent modules to be inversely proportional to the time interval, and thus varies rapidly as a function of the time interval. The problem is not so much that there is always a finite slope, since this models more closely

the charcteristics encountered in nature. Rather the problem lies in the fact that the value of the slope is entirely dependent on the time interval, and is thus falsely made to be reliant on the unsuspecting user and not the programmer of the original module.[1]

Non-linear devices (such as diodes and multipliers) tend to cross-modulate the input harmonics. A small error is thus not propagated as one error but as a series of errors across the frequency spectrum. One consequence of this, is that if a signal is passed through a non-linear function module, and subsequently accumulates an error, if the signal then passes through an inverse of the first function again, it no longer resembles the original signal.

[1] Obviously the programmer can include code to emulate a finite slope in an implementation.

11.5. ERROR IN THE REAL-TIME CONTROL

If data is being logged, or if a real-time simulation is being done and there is a module plugged into the **User Interface** which includes a time-dependent component, then the accuracy of the real-time clock is critical.

11.5.1 IN-DEPTH ANALYSIS

The clock pulses used by the **User Interface** are obtained from the **IBM PC** system clock, and thus are guaranteed to have the same accuracy and stability. The stability is typically of the order of a few parts per million [6]. The pulses are divided down by the first programmable counter to provide a clock of period 0.5 mS. However, an error arises because the clock pulses into the programmable counter are not an integral multiple of 2 KHz. The timer would need to divide by 447.443, and it can only divide by 447. This gives a worst-case deviation of 0.1% maximum.

The worst case arises when time-increments of less than 253 ms are used in the simulation. At all intervals less than this, a second counter is programmed with a value of twice the number of milliseconds required, and the output clock is consistantly 0.1% fast in these cases. However a correction factor of 1.00099 is actually applied to the count value, and the result is rounded. In other words, the count value is actually given by:

count value = round(no_of_milliseconds x 2 x 1.00099)

This means that at less than a count of 506 (253 ms), the correction is less than 0.5 and is truncated. At 506 the correction exceeds 0.5 and the counter is programmed with a value

one greater (in this case 507). The error for the two borderline cases is then:

505: error = ((447.443*505)-(447*505))/(447.443*505)=0.00099
i.e. the clock is 0.1 % fast up until here.

506: error = ((447.443*506)-(447*507))/(447.443*506)=-0.00098
i.e. the clock is 0.1 % slow at this point.

As the intervals increase, the error decreases, until at a count of 1012, which corresponds to a 506s interval, the error is a negligible 0.002/1012 or 0.0002 %. It then increases and decreases cyclically, but with an overall reduction in amplitude on each cycle.

11.5.2 CONCLUSION

One can thus conclude that a maximum deviation of 0.1% is guaranteed in any real-time increments. This is unlikely to be the critical determinant of error, since it is usually used with external modules which are far more error-prone. Furthermore there are no other errors resulting from the **User Interface** timing.

11.6. CHOICE OF INTEGRATION TECHNIQUE AND ACCURACY

There is a lot of literature on digital integration simulation, as it has been a major application of computers. However a few methods are now popularly used. The ones shown here are all based on recursive design, and all use present input values and past values at the time intervals specified by the time increments.

11.6.1 DIFFERENT METHODS

The Euler method is the best known simple integration method. It can be thought of as the first two terms of the Taylor Series [7]. Unfortunately it is not directly useful in cases such as this where the derivative is not a given.

A modified rule is used, in which each value, **y**, in time is obtained from the last **y**, from the formula:

$$y_{n+1}=y_n+x_{n+1}*delta\_t$$

In a simulation, one takes the result from the previous time interval, and adds the product of the time interval being used, and the new input to the integrator.

This formula is an approximation which is justified by a triangular representation.

This method has the advantage of being simple and fast. There are no special complications at the start and end values. The only exception to the calculation is at t=0, where an initial value must be supplied.

As this method uses a triangular (zero order) approximation it is entirely accurate when integrating a constant, but its accuracy diminishes rapidly as the order of the input function increases. This large and erratic error is not desirable.

Better methods use the Trapezoidal Rule (also called the "Modified Euler Method"), which is given by

$$y_{n+1}=y_n+1/2(x_{n+1}+x_n)*delta\_t$$

or Simpson's rule, given by:

$$y_{n+1}=y_n+1/6(x_{n+1}+4x_n+x_{n-1})*delta\_t$$

These methods all work by approximating the actual solution curve with straight line segments, whereas one can also approximate instead with a section of a parabola. This results in a more accurate method -the Adams Method, given by

$$y_{n+1}=y_n+1/2(3x_{n+1}-x_n)*delta\_t$$

One may also obtain higher-order versions of this such as the fourth-order:

$$y_{n+1}=y_n+1/24(9x_{n+1}+19x_n-5x_{n-1}+x_{n-2})*delta\_t$$

In each example, the accuracy with higher order input functions increases, but so does the number of previous values required.

Gibson [8] claims that the Milne method is the most elaborate of the algorithms mentioned, (but does not substantiate). A formula here would then be:

$$y_{n+1}=y_{n-2}+4/3(2x_{n-1}-x_n+2x_{n+1})*delta\_t+remainder$$

This method will not be pursued, and the remainder will not be discussed here (in normal calculations it is ignored).

11.6.2 THE FAST OPTION

Regarding **SOFTWIRE**, a choice of two methods is suggested for implementation. The Euler method is very fast, and in most simple demonstration simulations it is adequate. In fact it was used while testing, and was found to be adequate in almost all the tests done, and the benefit gained from the extra speed was worthwhile. If one does the examples suggested in PART 3, one will find that the Euler integrator suffices for these options.

11.6.3 THE ACCURATE OPTION

Adam's Method should be used when accuracy is critical. As this is an extremely accurate integration, it is unlikely that any user will be dissatisfied with the accuracy provided by it. The error [9] is guaranteed to be below:

$$-19/720(delta\_t)^5 f(z)^{(4)}\text{; where z is an element of } (t_n, t_{n+1})$$

This is more computation-intensive, and so may slow the simulation of a system down appreciably. It also has the disadvantage that it is not defined until the first two time intervals have elapsed. An alternative algorithm has to be provided for these intervals. One has to use the Trapezoidal Rule for the first iteration, and Simpson's for the second, before Adam's Method takes over. This introduces more error from the start, which is propagated through the entire simulation. There is also an inherent lag introduced in higher-order approximations, resulting from the influence of previous values.

11.6.4 ADAM AND EULER COMPARED

As a simple way of comparing the two methods, a speed and an accuracy test were conducted.

In the speed test a system net-list was input to **SOFTWIRE** consisting simply of an integrator with a constant input. The net-list input is given below.[2]

```
1 CON(4)
2 INT(0) 1
```

The time intervals used were 0.1ms, and the total time was 1s, giving 10 000 iterations.

The Euler integrator took 60s, and the Adams integrator took 93s. Adam's thus took more than 50% longer. However one must bear in mind that not all the time is spent calculating the integration. Thus the actual increase in time spent was more than 50%, but the overall effect on the speed will seldom exceed 50%, and in larger systems may be less.

In the accuracy test, a more complex system was input. The net-list was:

```
1 CSW(0.1592)
2 INT(0.0000) 1
3 NT1(0.0000) 1
7 SNW(0.1592)
4 NEG 7
5 SUM 2 4
6 SUM 3 4
8 GAI(1000.00) 5
9 GAI(1000.00) 6
```

[2] See the Appendix chapter SOFTWIRE MANUAL for interpretation, but suffice to understand this is just a constant input of 4.0 from module 1 into an integrator, module 2, with initial value zero.

The block diagram of this system is given in Figure 11.1

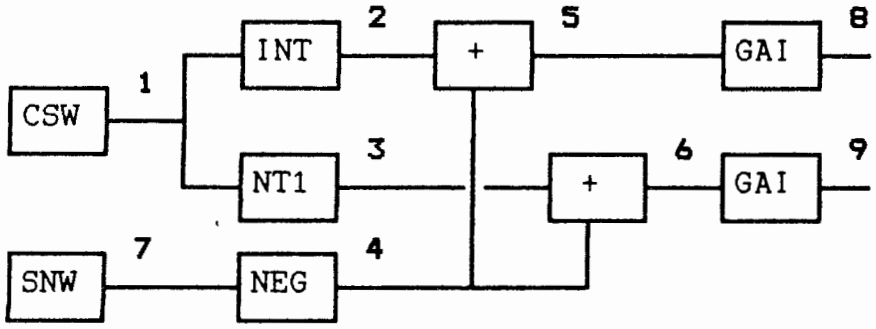


FIGURE 11.1 BLOCK DIAGRAM OF SYSTEM USED TO COMPARE INTEGRATORS.

A cosinusoidal function was chosen because it is easy to check by comparing with a sinusoid. It is also an easy-to-generate curve with a very-high order (polynomial) slope (actually infinite), since it is the change in slope that presents the challenge to the integration. A cosine (as opposed to a sine) is preferable as it allows the integrator output to start at zero. The frequency of the waveform is necessarily set to 1 rad/s ($1/2\pi$ Hz) if the integrator's output is to be of the same magnitude.

A time increment of 0.15s was used, and the simulation was run from t=0s to t=3.00s. The increment was chosen deliberately large to introduce error. The simulation was stopped when the maximum error was reached.

The outputs from modules 2, 3, 4, 8 and 9 were given.

The output was sent directly to an ASCII file. The results are given in Table 11.1.

TIME	EULER	ADAMS	TRUE ANSWER	EULER ERROR	ADAMS ERROR
0.00	0.000	0.000	0.000	0.000	0.000
0.15	0.148	0.148	-0.149	-1.161	-1.161
0.30	0.292	0.294	-0.296	-3.981	-1.472
0.45	0.427	0.437	-0.435	-8.397	1.813
0.60	0.550	0.567	-0.565	-14.31	1.782
0.75	0.660	0.684	-0.682	-21.59	1.754
0.90	0.753	0.785	-0.783	-30.06	1.730
1.05	0.828	0.869	-0.868	-39.55	1.710
1.20	0.882	0.934	-0.932	-49.83	1.694
1.35	0.915	0.977	-0.976	-60.68	1.684
1.50	0.926	0.999	-0.998	-71.85	1.679
1.80	0.880	0.975	-0.974	-94.15	1.685
2.10	0.748	0.865	-0.863	-114.7	1.712
2.40	0.543	0.677	-0.675	-131.8	1.758
2.70	0.283	0.429	-0.427	-143.7	1.818
3.00	-0.009	0.142	-0.140	-149.6	1.887

TABLE 11.1 TABLE SHOWING THE ERROR FOR EULER AND ADAM INTEGRATORS

Notes

1. The error has been shown multiplied by 1000.
2. Some steps have been removed from the table as they contained no useful information.
3. The actual simulation runs to more significant figures than have been shown (there are actually 11).

The results clearly show the superiority of Adam's method. The error is only one-thousandth that of the Euler integrator, and in simulations an integration error of such small magnitude would usually be overshadowed by other error sources. Note that the error after the first iteration is the same for both, since the ADAM integrator does not have enough previous values, and so has to resort to the Euler Algorithm.

11.6.5 RUNGE-KUTTA

It would be valid at this point to ask why the well-known Runge-Kutta formula is not used, or an appropriate modification of it. This method is actually preferable where it can be used, and it is used in other analogue simulations, such as **MIMIC** and **CSSL** [10]. However such programs run simulations in "batch mode", and not "on the fly". This means they are able to compute values into the future, and at positions between the time intervals. **SOFTWIRE** is used as a demonstration program for education, and so its sequential execution and simultaneous depiction on the screen are regarded as being very important. As a result, the extensive Runge-Kutta family of methods cannot be used as no future values or intermediate values are available. This restriction on intermediate values also precludes the Midpoint Formula Integrator.

11.6.6 STABILITY

The user must be aware that for a given simulation there does exist a value beyond which the solution is unstable. This is especially serious when the slope of the input to the integrator changes rapidly. The solution is to use smaller step-sizes, but in some cases the steps become unreasonably small. In such cases, the higher-order integration method is preferable, since it is stable for larger time increments.

As an example of instability, consider the solution of a second-order differential equation modelling a swinging pendulum. An example with the graphic solution is taken from Gibson [11]. This

is shown in Figure 11.2.

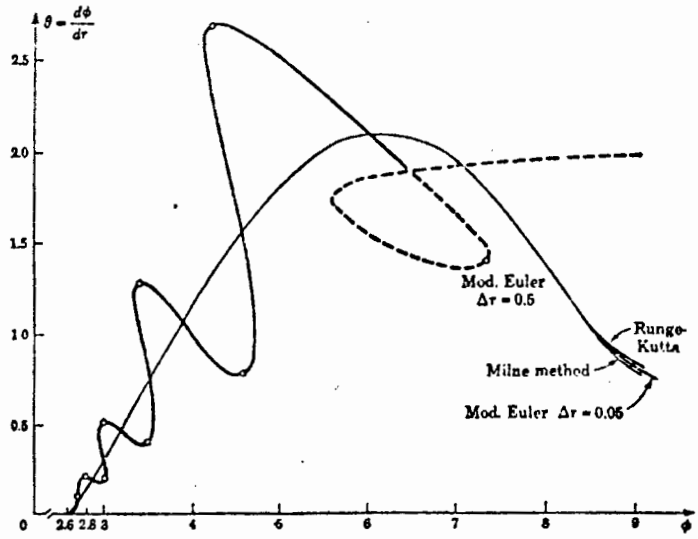


FIGURE 11.2 PHASE-PLANE PORTRAIT OF PENDULUM EXAMPLE, SHOWING INSTABILITY.

From this plot it can be seen that the lower order Modified-Euler method (which is another name for the Trapezoidal Rule) is unstable at increments of 0.5s, whereas the higher-order methods are not. To achieve the same accuracy for Euler, one would have to use a 0.05s step size.

11.6.7 FREQUENCY RESPONSE

Frequency response is one aspect of numerical integration which is seldom discussed, and this would seem to imply it is not important. (Again, for the normal use of **SOFTWIRE**, the recommendation is to choose Euler or Adam and to simply go ahead with the required application.) For the serious simulator however, and especially if one is concerned with frequemcies approaching the Nyquist limit, one further consideration may be useful -the frequency response of the integrator. More specifically, one is interested over the range from DC to a frequency of half the sampling rate (given by the time increments).

To get an idea of what is meant, consider the plot shown in figure 11.3, taken from Hamming [12].

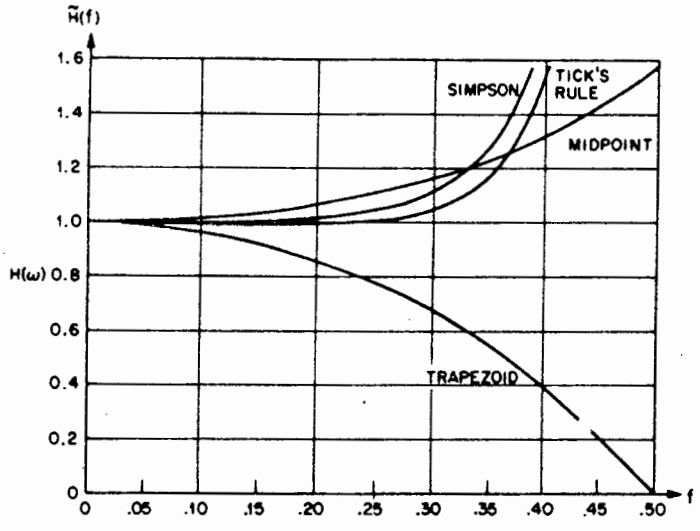


FIGURE 11.3 FREQUENCY RESPONSE OF INTEGRATION FORMULAE

As a result of this plot, one can see that Tick's Formula, which was not mentioned before, is a slightly better approximation than Simpson's, to a maximally flat response, which is useful in the setting mentioned. It was in fact designed specifically for this purpose, using an approach based on Chebyshev Criteria. The three-point formula to use [13] would be

$$y_{n+1}=y_{n-1}+(0.3584x_{n+1}+1.2832x_n+0.3584x_{n-1})*delta\_t$$

This design allows one to expect the same error at the maximum error for any frequency over the first half of its working spectrum. The price to be paid is less accuracy at lower frequencies. It is also less useful if used with the **User Interface**, since it allows through the higher frequency noise, which the Trapezoidal Integrator clearly rejects.

11.6.8 AN "IMPLICIT" BONUS

Most of the discussion so far has indicated the difficulty of coping with errors which arise after each section of a calculation and seem to work to the disadvantage of the user at every turn. This need not be the case with implicit systems. Although there are still limitations from discrete-time sampling, the presence of a negative feedback loop may actually serve to reduce the overall error not only in the system being simulated (if that is what it is being used for) but also in the simulation itself. This is particularly helpful if a device such as an integrator is present in the forward loop.

11.7. ERROR IN OTHER MODULES

Most of the other modules are well behaved. Since most of them are non-recursive, they are not as problematic as the integrator. (Reference to other modules from a mathematical viewpoint will be made in the next chapter.)

11.7.1 FILTERS

The exception to the last statement is the family of recursive filter modules. They are recursive and so do compound error. However, it is safe to assume that the digital representation will normally be more accurate in **SOFTWIRE** than in a real-life implementation, since an analogue filter of less than 1% accuracy is not common in everyday applications[3] , and a digital implementation often uses a word length similar to that of **SOFTWIRE**. Moreover one does not have to use a recursive algorithm, since the facility to store past values exists, and a non-recursive alogorithm may be used.

[3] This obviously does not include precision filters for specific applications, such as precision anti-aliasing filters.

11.7.2 DIFFERENTIATION

This should be used with caution. Differentiators are inherently noisy, whatever the application. In linear systems they may be used with more confidence, but in systems involving "hard" non-linearities, discontinuities and step changes, results may be unacceptable. bear in mind, too, that the differentiator has the following frequency response [14].

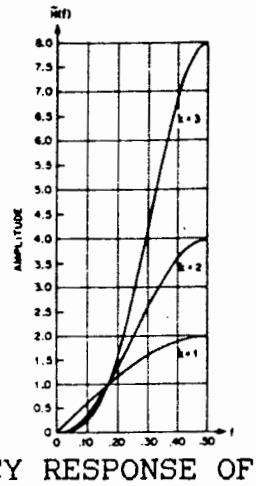


FIGURE 11.4 FREQUENCY RESPONSE OF A SIMPLE DIFFERENTIATOR.

11.7.3 HARD NON-LINEARITIES

These produce difficulties when significant changes occur between discrete sampling instants. For instance, a square pulse might have a step-change half-way between instants. In many cases this is unimportant, such as when observing the response of an LRC circuit to a step input. The user is looking for a trend, not a specific set of values.

11.8. IS THE SYSTEM DEFINED AT T=0- OR T=0+ ?

Although it is usually not of interest to the student, and does not significantly affect the result, it is important that the

issue be addressed of whether simulation starts at t=0- or t=0+ and whether a module is updated at the beginning or end of a time interval.

11.8.1 THE START

The approach decided was that the first iteration would represent t=0 for those modules with which this is theoretically possible, and t=0- for the others. The different categories of module are assumed to be at different stages, determined by their class of function:

1. All modules which have user-supplied or implied initial conditions start with these values as outputs.

2. All modules which require inputs and are time-dependent (such as integrators, filters etc) are supplied with initial conditions. If not specified, these are set at zero.

3. All modules which require no inputs and are functions of time only, or are time-independent, are evaluated.

4. All other time-independent modules are evaluated.

Before the simulation begins, the order of modules is adjusted, and a check is made that all required inputs to modules are defined. It is thus guaranteed that this group of modules will now be calculable (i.e. all outputs will be defined when their values are accessed).

11.8.2 SUBSEQUENT ITERATIONS

The user must in some cases be aware that only time-dependent modules require simulations with time-increments. This seems to

be a tautological statement, but this fact could easily be overlooked, and deceptive results could occur, particularly if feedback is modelled. For example, if one was modelling an amplifier system with feedback, and the amplifier was an idealised one in which there was no phase lag, in this case, there could be certain situations (especially particular signals input to the amplifier) where the phenomena reflected at the output bore no resemblance to what would actually happen. In fact, the effect of the discrete intervals would be the same as applying a zero-order hold to the system in front of each module. This could result in oscillations which were not characteristic of the idealised system being modelled, which is in reality perfectly stable. The sampling rate will be discussed in general later in the chapter. In this case, the point is that the sampling rate (the inverse of the time-increment) for such a system should be infinite, since the idealised system responds infinitely fast. Thus an error is inevitable in the simulation. This is an error related to "stiffness" in a system.

Ideally, all modules which are time-dependent functions of other module outputs are assumed to be evaluated at the end of an interval, and the outputs are calculated as though this was the case. This does hold if there is only one time-dependent module in a system, and it is evaluated first. In the other cases problems occur with serial, rather than parallel, computation. These problems are discussed further in the next section.

11.8.3 SERIAL COMPUTATION

The need to evaluate the output from modules serially is a necessary part of any single-processor digital simulation of a system. The constraints involved therein will be discussed.

In systems which have some form of feedback, the serial computation is artificial, since modules linked in a loop are affecting each other simultaneously. In most case, this is not any more of a problem than the use of discrete time intervals in the first place, and the results of the two constraints may be treated together. In the non-linear case, the problem is more serious, and may require individual consideration in specific instances.

BOOLEAN SYSTEMS

A non-linear case where the problem cannot be handled at all by the present implementation of **SOFTWIRE**, is with synchronous boolean devices with storage, such as latches and flip-flops. Because they are boolean, any "error" in an output state cannot be tolerated at all. As a result, a simple serial approximation fails.

The reason will be given using edge-triggered (non-transparent) latches as am example. A latch takes an input at one clock edge, and presents it at the output at the next clock edge. Meanwhile, at this same clock edge, it has stored the next input in order to output that at its next clocking. In other words, many latches in a system work synchronously, and process data in parallel. For this to be implemented in a simulation, two arrays would be needed to store intermediate results. At each iteration, the modules would all take their inputs from the first array, and output their results to the second array. For the latch, this would purely require a transfer of data, but other devices, such as a JK Flip-flop, would require more processing. In the next cycle, the second array would become the input array, and the first would be the output. In this way, parallel processing is simulated. On the other hand, if there is only one array for

intermediate results, as **SOFTWIRE** has, the modules which are processed first will adjust this single array, and modules which subsequently process inputs from the array may be using values which have been updated during the same clock cycle, instead of the previous one.

If each module uses its own intermediate storage, the problem presents itself in a different form, but the consequence is still the same. Further, if non-delay elements are used in the same systems, they are not synchronous, and would thus have to take values only from the second array. The results could also differ depending upon which modules were calculated first.

NON-BOOLEAN SYSTEMS

All systems involving storage elements have the inherent problem just discussed, but in linear cases -and most common non-linear cases- the error that results is as acceptable as, say, truncation or discrete-time error, and of the same order.

TIME-DEPENDENT MODULES

Many simulations in engineering are first or second order. In the case of first-order systems, where only one time-dependent module is used, then serial processing of modules causes no additional error. If two are used, the error from an Euler Integration approximation cancels much of the serial error. For instance, in the simple case where the two integrators are cascaded, the first integrator uses as its input the value which was presented to its input at the end of the previous iteration. This integrator then in a sense has a "lag" which is approximately half of the time increment value. The output from the first integrator is then calculated. The calculations for the second integrator are now

done, but this integrator is using as its input a value which has just been calculated in the **same** iteration. This integrator then in a sense has a "lead" of approximately half of the time increment value. The net effect is that in a linear system the errors would be more likely to cancel than to compound. By this logic, and also by actually observing the error in systems which were likely to arise in engineering courses (such as a state-variable integrator), it was decided that serial computation, with its inherent simplicity, would suffice in an educational context.

A SUGGESTED IMPROVEMENT

Ideally -as mentioned in the section on boolean simulation- one uses two arrays for intermediate values. Suppose they are called arrays **A** and **B**. Start by assuming the intermediate values at the end of one time interval (and hence the beginning of the next) are in **A**. The order of calculation is arranged so that all time-dependent modules with no inputs are calculated first, and they output their values (which are functions of the present time only) to **B**. Next, all other time-dependent modules input their required values from **A**, and process them according to what the module function is, and output their values to **B**. Finally, all remaining modules take their input values from **B** (not **A**), and they also output values to **B**. The values in **B** are transferred to **A**, and the process repeated.[4]

11.8.4. DISCRETE-TIME IMPLICATIONS

All digital simulations such as **SOFTWIRE** are based on discrete-time intervals with iterations -in fact a large part of numerical

[4] In an actual implementation, one would not actually swap the **entire** array, but rather save time by merely switching the names or pointers (references). However that is not important to the concept in discussion here.

method in calculations involving time is based on this principle. The output remains constant over the time-interval, and is thus only guaranteed to be the correct value at the initial sampling instant. There is thus **necessarily** error involved, and the only solution is to try to understand and minimise this error. The only instance there is no error, is when one is modelling a system which itself utilises discrete sampling. Also, as the time increments are made smaller and smaller, the simulated discrete-time system begins more and more to resemble a continuous system.

Discrete-time simulation is closely analogous to zero-order-hold sampling. In fact it is, because each output is sampled at the beginning of an iteration, and held constant for the rest of the time-period. Analysis of errors and problems will thus draw on what has been learnt in these areas. The error takes different forms which will now be discussed. However, it must be noted that some modules use computational methods which approximate higher-order holds, and their response may be slightly different regarding phase response and gain.

GAIN

Gibson [15] points out that it is advantageous to hold each value for the complete sampling interval. This is precisely what happens in numerical processing, and also what has been implemented at the analogue output of the **User Interface**.

PHASE-LAG

The zero-order-hold process can be intuitively seen to produce a lag in the system.

This may be a problem for instance in a system consisting of a controller with feedback. Here the phase lag would be greater in the simulation than in the actual system being modelled.

In Figure 11.5, which was obtained from Gibson [16], one can see how the attenuation and phase shift respond with frequency. The gain response is of a (sin x)/x form, and the phase lag is linear.

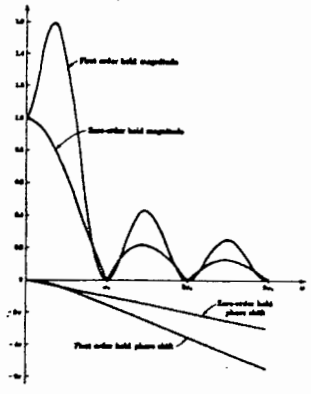


FIGURE 11.5 ATTENUATION AND PHASE SHIFT OF ZERO-ORDER HOLD VERSUS FREQUENCY

NYQUIST SAMPLE RATE

If the sampling rate is below the Nyquist sampling rate of frequencies which are of interest, information will be lost, according to Shannon's law. In many cases this lost information may be in the higher harmonics or frequencies outside the range of interest. In many applications one may use parameters which place the output well within the sample rate. For instance, many demonstrations can be done effectively with driving signals chosen at the normalised frequency of one hertz. Then with a time increment of one millisecond, the output will appear continuous. The one hertz frequency chosen may be unrealistic in real-world applications of audio and radio, but it is often satisfactory in demonstrating a phenomenon, and it enables the student to mathematically understand what is going on by simplifying mental calculations.

The same problem with Nyquist sampling occurs in a slightly different form in digital filter realisation. Often the problem will be obviated when it occurs at frequencies outside the range of interest. For instance, if one is looking at the step response of a first-order low-pass filter, there is absolutely no obvious error in the time-domain plot that results. In this case the low-pass filter is conceptually being regarded in the time domain as a weighted-averager (or a smoother), and this is a use which will be found in many of the applications.

However there are cases when the conceptual model is in the frequency-domain. In this case, if the input to a digital filter has not been stripped of all harmonics greater than half the sampling rate, then these high frequencies are **aliased** to lower frequencies -causing distortion. This aliasing is shown in Figure 11.6 (from Hamming [17]).

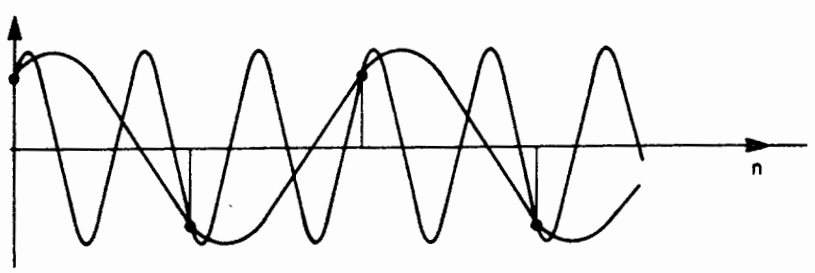


FIGURE 11.7 ALIASING

The **User Interface** analogue/digital converters do not escape this problem. Ideally a filter should be placed before the **ADC** to filter out frequencies above the Nyquist limit. A filter should also be placed after the **DAC** to smooth out high-frequency distortion from the step-changing outputs. In critical cases, dithering at the output also improves the smoothing. However,

this has been left to the user to take care of. The reason is that a filter is difficult to implement which simultaneously caters for periods from milliseconds to minutes **and** has satisfactory specifications (an anti-aliasing filter should ideally have a very steep frequency fall-off). There are also many applications which have no need for filters.

VARIABLE SAMPLING-RATE REQUIREMENTS

There are many systems which include different parts which require different time increments. For instance, a small feedback section in a larger system, or a highly non-linear element, may require more iterations to maintain the overall accuracy. Automatic increment adjustment is implemented in simulation languages such as **MIMIC** and **CSSL**, but it is only feasible in batch, where the error after each step is calculated, and a certain step may be repeated with a different time interval until the required accuracy is achieved. When simulating on the fly, this is unrealistic.

In analogue computer literature, a special case of this problem exists with the solution of systems of differential equations. If the solutions change so rapidly that exceedingly small step sizes are required, the system is said to be "stiff" [18]. In the context of **SOFTWIRE** (and indeed with any real-time simulator) there is very little that can be done about this problem, other than to use the smallest possible time increments.

STABILITY

In Control Theory it is recognized that instability can occur spontaneously in a linear sampled-data system with feedback. This is very unlikely if the sampling rate is greater than five times

the system bandwidth [19]. What happens is that the system appears to be behaved if a plot is made at the sampling instants, but the system is in fact oscillating (often wildly) between sampling instants. Since one of the applications of **SOFTWIRE** is in teaching Control Theory, this must be borne in mind.

USER CONTROL OF ERROR

The error caused by discrete-time sampling can be reduced by the user, by choosing a different time interval. This can be an advantage, but has the disadvantage that the error control is now left in the hands of the student and not the programmer.

THE LOWER LIMIT

Although it can be safely maintained that the scope of simulations that **SOFTWIRE** is written for excludes this problem, it must be stated that there is a lower limit to time increments. The smallest value that could be used is 1E-38. However this value is not recommended as a practical minimum as it may lead to large errors in the calculations. For instance, the Adam Integration routine divides the increment by 24, and the low-pass filter divides it by the user-specified time-constant. It would be safer to remain above say 1E-34, to allow a few digits of accuracy.

Of course, if the **User Interface** is used, the lower limit is in the order of milliseconds. The program only allows a minimum, in real-time mode, of one millisecond in any event. However if a time increment is specified by the user, and initially allowed by **SOFTWIRE**, the program may still reject it at the time of running a simulation. This is because after each iteration of the simulation it checks to see if the computer has managed to com-

plete the iteration in the specified time. If not, it stops with an error message. This is because it is obviously meaningless to continue when both the time increments and the outputs from time-dependent modules would be subject to intolerable error.

APPROXIMATING A CONTINUOUS SYSTEM

According to Gibson, if the ratio of sampling rate to the maximum frequency or bandwidth of the system under simulation is 10 to 1 or greater, one may use the approximation of a continuous system [20].

11.8.5. THE USER INTERFACE

As has been discussed, there is inevitable error in numerical processing. The various types of local error at each stage of a simulation have been discussed. The analogue section of the **USER INTERFACE** introduces three further possible sources of error. The first is quantization error, occurring at the analog-digital interface. The second is the distortion introduced in the analogue processing. Finally, there is a finite conversion time at the analog-digital interface.

ANALOGUE DISTORTION

The analogue distortion arising from signal processing, multiplexing, and noise in the signal and power rails has been designed to be significantly less than one quantization step and thus adds no appreciable error to the discussion. The voltage range to be quantized is either 0V to +5V, or -5V to +5V. The magnitude of the ranges are thus 5V or 10V respectively. Since linear 8-bit converters are used, there are 256 quantization levels. The step is therefore either 5V/256 or 10V/256, which is about 20 mV or 40 mV respectively.

QUANTIZATION ERROR

As the calculation in the previous paragraph shows, the quantization step can be as much as 40mV. The maximum quantization error (sometimes called "quantizing distortion") is thus 20mV. This represents (0.5*100)/256% or 0.2%.

In most applications using the analogue I/O, the external circuitry connected will have a resolution which far exceeds this value of 0.2%. For instance, if (for a practical) a class had to connect up a single-BJT inverting amplifier to observe its transfer function, they would probably use preferred values to bias the transistor. The range of possible output values would then be closer to 20% than 0.2%. The quantization error is thus more than adequate in such a case and would still be, even if 5% resistors were used.

CONVERSION TIME

The finite conversion time results in error because of the delay caused by the conversion. With a digital-to-analog converter, the combined conversion and settling time is guaranteed less than 125nS. The operational amplifier which processes this has a rise time of 13 V/μs. The output thus always settles within a microsecond -usually less. Since the speed of the computer limits real-time increments in a simulation to five milliseconds, this error never exceeds 0.01%, and then only in rare cases. In most cases it is negligible.

The analogue-to-digital converter requires a constant 0.1mS for its conversion, making a larger contribution to the error, of up to 2%. It will not be often that such small time increments are

used in real-time simulation. If the simulation is running in continuous mode, the output on the screen is continuously being shifted to the right, and this action alone takes the program 20mS, so a simulation with 25mS increments is more likely, implying a maximum error of 0.5%. If the output is not significantly time varying, or the simulation is not in real-time, the conversion time ceases to become a concern.

Far more important a consideration is that the conversions are not all done at the instant of the beginning of the time interval, but are done sequentially, as a result of serial computation. The error thus increases with the number of modules in a net-list. Thus, if the values being used vary rapidly with time, it is advisable to limit the length of net-lists.

OTHER ASPECTS OF THE USER INTERFACE

Only one aspect of the **User Interface** has been covered so far -namely the analog I/O. These channels are available to connect to the user's equipment. However there are also analogue inputs and outputs already connected to devices. Two inputs are connected to potential dividers and two outputs are connected to panel meters. There are also binary (TTL) inputs and outputs, where error is very unlikely to occur.

The potential divider inputs are provided to allow the user to input values during the simulation. Any inaccuracy here is likely to be more from the user than the device. The user may also plot the output from the pott. on the screen for visual feedback. The intention, though, is not for high accuracy rates. The user is not supposed to develop a "video games" proficiency before mastering the input, but rather to have a simple way of replacing the keyboard with a more intuitively natural input mechanism.

Also, the range of the potential divider can be adjusted, and hence the resolution and sensitivity. Also, more accurate inputs can be provided via the keyboard.

The meter used depends on the user. With the prototype a 0 to 10V meter with 50 divisions was used. This gives at most a 1% resolution. More expensive meters could be used, but the primary purpose of having the meter is again to provide a more intuitive format for the output. More accurate output can always be obtained using the tabular output mode on the screen or printer.

11.9. CONCLUDING REMARKS

One's overall impression after this analysis may be that the simulation is fraught with dangers. However, numerous demonstrations have been done on this system during testing, and have been found to be satisfactory. The main use of this chapter should be to make the lecturer aware of some of the potential problems when setting up learning programs. That is not to say they are at all likely to manifest themselves in each application. The tutor would presumably try out the systems where possible before giving them to the student. Further, there are many cases in which it is obvious that there is a problem. For instance, if a sine wave is being generated, and it does not actually look sinusoidal, a natural inclination is to vary the sampling rate until a respectable output appears.

The reassurance given that error problems are not likely to be troublesome comes from the educational context, where it is assumed that the systems simulated are usually very small by comparison with systems which are modelled or simulated for research or design testing purposes. As a consequence one assumes that the tutor already has an idea of what the output will be.

THE TUTOR'S RESPONSIBILITY

The tutor is thus expected to be able to institute error control where necessary. This is different from sophisticated simulation packages,which have their own error-control algorithms -where the maximum possible error is estimated at each step, and must be kept within a certain tolerance -often specified by the user. This type of error control is omitted here for a number of reasons:

It is simpler and faster not to have it. It also usually necessitates automatically re-calculating a section which exceeds the error, using progressively smaller time-increments -something which has already been discussed in this chapter as being infeasible while on the fly. Further, the error-control algorithm normally centres around the integration algorithm. **SOFTWIRE** will be used in many applications which do not even involve integration.

Another very important reason is that the system handles non-linear modules, and in particular possible discontinuities. These cause an error-control algorithm to drastically reduce step size to overcome the error that results. **SOFTWIRE** has been designed to ignore this error to avoid possible unnecessary distraction to the user.

SPURIOUS RESULTS

Even in sophisticated simulation packages, spurious results occasionally occur [21]. It is always advisable to check that the output appears to be reasonable. One may test suspicious results by using a far smaller time increment, and comparing results.

OVER-ACCURACY

Often the problem is that the simulation is <u>too</u> accurate. It is usually more accurate than analogue equivalent systems. There is, for instance, absolutely no drift. The tutor may have to actually build in noise, drift and distortion to make it more realistic. As an example, a state-variable oscillator with two integrators in feedback can be made to maintain a very nearly constant output amplitude. This is not the case in the real world, where the

amplitude either dies or the integrators go into saturation. To this end, one of the modules on the disk has been programmed to have slight drift, noise and saturation. Error is thus deliberately introduced.

COMPARISON WITH AN ANALOGUE COMPUTER

Even the slightest misgivings one may have about the digital simulation of analogue processes may be dispelled if one considers the use of an actual analogue computer. Consider the implementation of multiplication, variable delay (which is either mechanical or uses a discrete-time process anyway), or gaussian noise of variable mean and deviation. This is especially difficult if the process is to be slowed down to a speed which the student may observe.

REFERENCES

1. Benku Thomas, "The Runge-Kutta Methods", BYTE, April 1986, pp. 191-192

2. BORLAND International, Turbo Pascal Reference Manual, p. 42.

3. BORLAND International, p. 293.

4. BORLAND International, p. 302.

5. BORLAND International, p. 302.

6. Paul Horowitz and Winfield Hill, The Art Of Electronics, Cambridge University Press, 1980, p. 616.

7. John Gibson, NonLinear Automatic Control, International Student Edition, McGraw-Hill, 1966, p. 132.

8. Gibson, 1966, p. 137.

9. B.G. Sherlock, "A General-Purpose Analog Simulation Package For The PDP-11, Written In RTL/2", Research Review, University of Cape Town Department of Electrical and Electronic Engineering, March 1983, p. 68-69.

10. Robert E. Stephenson, Computer Simulation For Engineers, Harcourt Bruce Jovanovich Inc, 1971,p. 172.

11. Gibson, 1966, p. 135.

13. Hamming, 1977, p. 40.

14. Hamming, 1977, p. 43

15. Gibson, 1966, p. 103.

16. Gibson, 1966, p. 106.

17. Hamming, 1977, p. 18.

18. Thomas, 1986, p. 203.

19. Gibson, 1966, p. 125

20. Gibson, 1966, pp. 117-126

21. Thomas, 1986, p. 210.

CHAPTER 12

CHOICE AND INPLEMENTATION OF MODULES

12.1. PROVIDING FOR AN EDUCATIONAL CONTEXT

It is evident that **SOFTWIRE** is similar to an analogue computer in many ways and one major similarity regards the choice of function modules. However there are two significant ways in which the traditional analogue computer's modules have been adapted for an educational context. First, there exists a far wider choice of modules. Secondly, the user of the program may add specific customised modules to cover particular applications that arise.

12.1.1. CHOICE OF MODULES

The choice of modules used was based on three criteria. The first is to have a set of modules which will combine to form a family of modules with an application in one specific area. An example is the logic modules, which can be used together to create a variety of logic systems.

The second criteria is to have a wide range of modules covering diverse functions, so that a user who has a creative and unorthodox idea is less likely to be stunted by the lack of modules to serve his purpose.

Thirdly, specific one-off lessons were intended when creating some of the modules. For instance, there are three Euler-integrator modules, which represent different levels of departure from the ideal model. These may be used specifically to teach

concepts relating to idealisation, implementation, and modelling ideal modules to make them resemble real-world implementations.

An overall design problem is that of having too small a choice of modules causes an unnecesary increase in the size of net-list, and may restrict the power of **SOFTWIRE**. On the other hand, having too many presents a barrier to learning how to use **SOFTWIRE**, and also slows down the simulation. The tactic adopted was to ignore the problem of too many modules, and to include a wide range of modules, and allow individual lecturers to delete the one's they don't want in the copy they supply to students, or simply to not mention them, and to allow more inspired students to explore the full range on their own initiative. The limitations on computer memory are not considered to be problematic. Each module requires marginal extra RAM. Over time, it will become evident which modules are not used enough to justify inclusion into the standard set.

12.1.2. ABILITY TO CREATE NEW MODULES

The teacher may create his own modules with relatively little effort, using a procedure specified in the **SOFTWIRE MANUAL** (in Appendix A). This provides him with the freedom to explore any area of System Design he chooses.

12.2. CHOICE, APPLICATION AND IMPLEMENTATION OF MODULES

12.2.1. BASIC MODULES

There are a few modules which have been included as a basic minimum to the system. They are modules which are not as straightforward to implement by the user. There is a degree of arbitrariness in deciding whether a module is basic to the system

or not. Most of the modules included here would not be easily implementable by a user simply by following the guidelines in the **USER'S MANUAL**. They do not fit as simply into a generalised strategy for implementation.

These modules are:-

TIM: This module increments the time. It also controls the length and mode of the simulation, and what is output.

TAB: A set of values found in the file "TABLE.INS" defines a function numerically, giving 20 pairs of values between 0 and 1.

WFM: A set of values found in the file "WFM.INS" defines a waveform numerically. A parameter gives the frequency.

VIN: Allows the user to input values continuously during a simulation, via the keyboard.

INT: Integrates the input to it with respect to time.

The **TIM** module is discussed in the chapter on **THE SOFTWARE IN DETAIL**. Its function is not only to output the time, but also to perform certain program controls. This module may not be included in a net-list, as it is always included automatically by the computer, and is transparent to the user except that its output is available as module 0.

TAB and **WFM** are discussed in a subsequent section on **NUMERICALLY DEFINED MODULES.**

VIN is the only module which accepts data from the keyboard during execution. This means that a check must be made on the keyboard on each iteration.

The **INT** module has been discussed in detail in the previous chapter.

12.2.2. USER INTERFACE MODULES

The 'x' in the module name refers to a single digit, which specifies which output channel is required.

MTx: (Meter). Outputs are scaled and sent to one of the analogue channels. The range represented is given by two parameters.

PTx: (Potential Divider). Scaled inputs are entered from an analogue channel. The range represented is given by two parameters.

LTx: (Light). A light goes on if a threshold is exceeded. The threshold is given as a parameter.

SWx: (Switch). A switch on the interface outputs a true or false value. (**1** or **0** resp.)

These implementations have also been provided as basic elements because they are entirely dependent on the hardware design. Each module makes use of a procedure written for it. This is discussed in the chapter on THE SOFTWARE IN DETAIL.

12.2.3. CLASSICAL MODULES

These are found on most analogue computers, and still have an important place in an educational context.

INT: (Integrator). Already discussed under **BASIC MODULES**.

GAI: (Gain). Scale the input by a constant specified by parameter

CON: (Constant). Provide a constant input specified by parameter.

SUM: Sum two inputs.

DIF: (Differentiate). Differentiates the input. This has also been discussed in detail in the previous chapter.

12.2.4. ARITHMETIC MODULES

SIN: Takes the sinusoid of the input (in radians).

COS: Takes the cosinusoid of the input (in radians).

EXP: (Exponent). Takes the natural exponent of the input.

LIN: Takes the natural logarithm of the input.

LOG: Takes the logarithm of the input (The base is provided as a parameter).

NEG: (Negative). Negates the input.

ABS: (Absolute). Gives the absolute value of the input.

SQR: (Square Root). Gives the square root of the input.

ATN: (Arctan). Gives the arctan of the input (in radians).

DSP: (Deadspace). Follows the input except for a deadband, where it outputs zero. The deadband is specified by two input parameters.

MOD: (Modulus). Gives the first input modulo the second input.

ARG: (Argument). Input two values, representing x and y rectangular coordinates, and this module gives the radian angle of the corresponding polar coordinate.

LEN: (Length). Input two values, representing x and y rectangular coordinates, and this module gives the magnitude of the corresponding polar coordinate.

HOR: (Horizontal component). Input two values, representing magnitude and argument in polar coordinates, and this module gives the x value of the corresponding rectangular coordinate.

VER: (Vertical component). Input two values, representing magnitude and argument in polar coordinates, and this module gives the y value of the corresponding rectangular coordinate.

MUL: (Multiply). Multiplies two inputs together.

QUO: (Quotient). Divides the first number input, by the second.

Most of these modules make direct use of the functions provided by **Turbo Pascal**. The others are very simple to implement.

12.2.5. FILTER MODULES

LPF: (Low Pass Filter). Behaves as a simple low-pass RC filter.
DLY: The output follows the input after a time delay specified by parameter.

Filters can be described as any devices which manipulate data in the time or frequency domain. Apart from the normal filters which allow or attenuate particular frequencies, integrators, averagers differentiators, delay modules and interpolators can conceptually be included here. The list is restricted here to causal filters, since **SOFTWIRE** can only operate on data at the present or past time intervals.

There are two strategies for implementing digital filters. The first produces transversal filters. It uses the **finite impulse response** approach, which performs weighted summation of previous values. The (transport) delay element is such a filter. This approach is conceptually simpler, and produces inherently stable filters (they cannot oscillate unless an input is applied). The **DLY** module uses this approach.

The second strategy uses the **infinite impulse response** approach. This is a recursive approach, where each output is a function of both the new input and the previous output. The low-pass filter (the LPF module) used in **SOFTWIRE** implements this approach. It is potentially more accurate, and uses less storage.

One may also use a combination of the two approaches. This is done in most of the integrator implementations discussed.

A difficulty with the transversal approach is that it requires an array of past values, and the utility is increased by increasing the length of the array. This extra utility must be traded off with the extra computation time and storage required. In the case of the integrator this is a simple choice, since the storage of only five previous values gives ample accuracy. With a delay module, however, the array length is more critical. Although the module is 100% accurate, the maximum delay time that can be emulated is directly related to the array length.

The array is provided as an optional "scratchpad", given to those modules which are flagged as requiring it. The array length is simple to change in **SOFTWIRE** -only one line at the beginning of the program. However the value, once chosen, is the same for all modules that require extra memory. If a long array length is chosen, all modules that require extra memory will have that long array allocated as they are added to the net-list. This is unlikely to be a problem, as even a 256K **IBM PC**, will be able to support a large number of even 1K arrays.

The delay module may have a long array and not take any extra time (in fact less time) to execute. This is achieved by implementing a circular queue, where the queue does not move, rather the pointers do. This time-saver applies to all transversal filters. However only in the delay element does one fully eliminate the extra overheads associated with a longer array. This is because in the delay module, all previous values are weighted with zero, except for a single value which has unity weighting -a situation which requires no arithmetic computation. All other filters require a multiplication with each member of the array -a time-consuming pursuit.

12.2.6. BOOLEAN MODULES

These modules have the logic convention that a value of **1** is regarded as **true** (or **high**), and all other values are false.

AND: The inputs are AND'ed.
NOT: Gives the logical inverse of the input.
NOR: The inputs are NOR'ed.
ORR: The inputs are OR'ed.
XOR: The inputs are XOR'ed.
NND: The inputs are NAND'ed.
MON: (Monostable). Output stays at **0** until there is a **1** at the input. The output then stays at **1** for a time specified by a parameter. It is non-retriggerable.

The difficulty with boolean modules is that they manipulate elements in a different symbolic domain from modules representing analogue processes, i.e. values represent only **true** and **false**. This may be confusing to the learner. There is no way the program can easily overcome this. If **SOFTWIRE** were to check net-lists for anomalies it would not be helpful, because the intermixing of modules does often have application. For instance, it is useful to use the switch inputs to provide a step response to an analogue system, or to have the LED outputs indicate when a threshold value is reached. An XOR gate also makes a simple phase-comparator with excellent linearity.

The tutor must be aware of this when using **SOFTWIRE**, and use discretion.

A limitation was also mentioned earlier, in the previous chapter, namely that any boolean modules which are clocked may not be used

without the limitation that they may not be included in systems with feedback, and if they are used in explicit systems, the order of computation has to be taken into account. For this reason such modules have not been implemented. That is not to say they may not be implemented. There is no problem if the caveats are correctly observed. The working of a device such as a ripple counter could easily be simulated by a net-list of flip-flops.

12.2.7. SPECIAL-PURPOSE MODULES

VCO: Outputs a sinusoid. Parameters specify the centre frequency and the sensitivity (in Hz/unit).

ZOH: Emulates a zero-order sample-and-hold at a frequency specified by a parameter.

The following are all integrators with modifications:-

NT1: Uses the fourth-order Adam's formula.

NT2: An Euler integrator which inverts and saturates outside the range of -1 through +1.

NT3: The same as **NT2** except that it displays drift and has noise added.

There are two ways to implement the **VCO**. This device is necessarily recursive, since its output is an integration of all previous instantaneous rates-of-change-of-phase. To see this, observe that a step-change at the input does not produce a step-change of the output **value**, but rather of the output **phase**.

The first method is output-recursive, and adds an increment to the previous output at each iteration. This increment is given by (time_interval x slope) where the slope is the derivative of the sinusoid at the required frequency, which is given by the cosinu-

soid of the function at a value given by the present_time x frequency. The entire expression is then divided by the frequency.

This is a complex calculation, and it can easily be shown that it is prone to error. At each step the computer must do a sinusoid computation -which is error-prone- and add the error to that from all the previous steps.

The second method is intermediate-recursive. It uses the input to calculate an intermediate tally, which it then uses for its sinusoid output. The problem of sinusoid error propagation is thus obviated, but an extra store is needed for the intermediate step.

This latter method was chosen. Two calculation steps are required. The first step is to calculate the intermediate value, which is an integral of the value (scaled by the sensitivity factor). The next step is to calculate the sinusoid of this value -a straightforward computation.

As this module is already very slow, the Euler-integrator was preferred, with its extra speed. Further accuracy was not considered a priority, as the prototype device was mainly used to demonstrate the concept of a voltage-controlled oscillator and also a phase-locked loop (where it does not have to be accurate as it is in a feedback loop). Anyway, because the device is so computation-intensive and slow, it cannot be used in many applications at realistic frequencies. To use a more accurate implementation would slow it down even more.

12.2.8. NUMERICALLY-DEFINED MODULES

WFM and **TAB:** These have both been described in the list of **BASIC MODULES**.

A simple linear (first-order) interpolation has been used, which only takes one point above and below the input value into consideration. This speeds up the program execution considerably, but is simple to implement, and avoids problems at the boundaries, where there is insufficient information for more complex interpolations. It is difficult to implement a more complex interpolation formula in general-purpose modules, where each application may have vastly different requirements.

A cubic-spline (second order) interpolation is obviously more accurate, but could only be implemented by using the original user-supplied array to generate a larger enhanced array of intermediate values, and this would all have to be done before the simulation began.

12.2.9. RANDOM MODULES

RND: This gives random values uniformly distributed within a range specified by two parameters. The output is described by figure 11.1

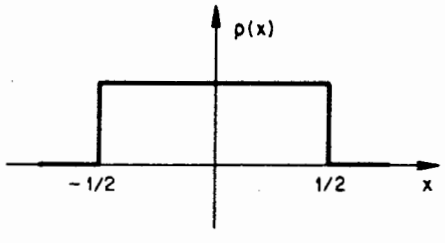


FIGURE 11.1 UNIFORM RANDOM DISTRIBUTION

GAU: This gives random values with a normal distribution. The first parameter specifies the mean, and the second the

standard deviation. This is useful in generating noise or for teaching queueing theory. The output is described by figure 11.2.

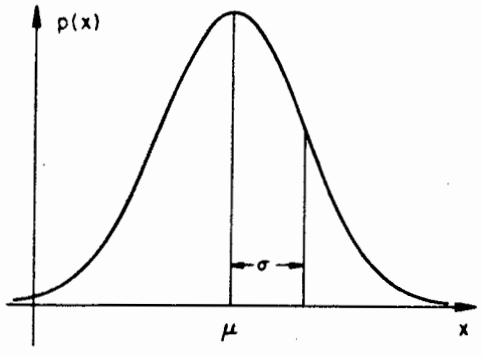


FIGURE 11.2 NORMAL RANDOM DISTRIBUTION

The unifom (rectangular) random generator was implemented using the **Turbo Pascal** implementation directly.

The normal distribution was chosen for implementation because it is very popular in the mathematics and statistics fields, and especially since it has application in simulating electrical noise and queueing in computer networks, as well as in vibration monitoring of large electric motors. The distribution was obtained using a formula which gives a reasonable approximation. The technique [1] involves taking the average of **N** random numbers between 0 and 1. This gives a gaussian approximation of mean 0.5. One then subtracts 0.5 to get a mean of zero, and divides by the square-root of (12***N**). This gives a standard deviation of unity. One then simply multiplies by the required standard deviation, and adds the required mean. Using N=4 should give sufficient accuracy.

The approximation is adequate for most applications (since the Normal distribution is itself only a useful approximation in many applications. A limitation, which will probably never be of

A GAUSSIAN NOISE RMS SIGNAL

consequence, is that with N=4 one never generates numbers more than 3.9 standard deviations from the mean [2]. However, the module is well documented, and one can make **N** a higher number, sacrificing only speed.

An alternative approach is to use a formula which does a single conversion from linear to gaussian.

12.2.10. FUNCTION GENERATOR MODULES

PLS: Outputs a rectangular pulse of amplitude **1** over the time period specified by two parameters.

CSW: Outputs a cosinusoidal waveform, with a peak-to-peak amplitude of **1**, at a frequency specified by a parameter.

SQW: Outputs a square waveform, with a peak-to-peak amplitude of **1**, at a frequency specified by a parameter.

SNW: Outputs a cosinusoidal waveform, with a peak-to-peak amplitude of **1**, at a frequency specified by a parameter.

SAW: Outputs a sawtooth waveform, with a peak-to-peak amplitude of **1**, at a frequency specified by a parameter.

TRW: Outputs a triangular waveform, with a peak-to-peak amplitude of **1**, at a frequency specified by a parameter.

MPW: Outputs a rectangular waveform, with a peak-to-peak amplitude of **1**, at a frequency specified by a parameter. The duty cycle is specified by a second parameter.

12.2.11. COMPARISON MODULES

THR: (Threshold Detector). This outputs a **1** if the output exceeds a threshold value, otherwise it outputs **0**.

THH: (Threshold Detector With Hysteresis And Inversion). If the previous output is a **+1**, it outputs a **-1** if the input exceeds the value specified by a parameter, otherwise it continues to output **+1**. If the output is a **-1**, it outputs a **+1** if the input is exceeded by the value specified by a parameter, otherwise it continues to output **-1**.

COM: This outputs a **1** if the input exceeds zero, otherwise it outputs **0**.

12.3. SOME SUGGESTIONS FOR MODULES

There is an inexhaustible list of possible modules which could be used. A few are included here as seeds for the tutor's thoughts on applications. Many more ideas may arise after PART 3 has been read.

A first-order hold.

An input-triggered Sample-and-hold.

A filter which was defined by a table of weighted values. The values would then be entered in the same way as with **WFM** or **TAB**.

A state-variable simulator which consists of a set of feedback loops and integrators, also defined by a table of weighted values

A random generator with poisson distribution.

A random (noise) generator to give optional pink, white or grey noise.

A frequency tester. Here the input updates an element of an array. The total in that array is then output. When used in the "y vs x output mode" the output becomes a frequency plot. This tester is useful, for example, as a random generator tester.

A limiter. Two parameters specify the limits within which the output follows the input.

A "mouse" interface. This module allows a mouse to be used to input data. This is useful on systems which already have a mouse. Implementation is simple because the **Turbo Toolbox** has a routine for interfacing a mouse.

REFERENCES

1. Arthur G. Hansen, "Simulating The Normal Distribution", **Byte**, October 1985, p.137-138.
2. Hansen, 1985, p. 138.

CHAPTER 13

THE EVALUATION STRATEGY

13.1 THE IMPORTANCE OF EVALUATION

"The approach to, and purpose of, the evaluation of gaming and simulations materials has been given fairly extensive coverage in the literature... With the recent increasing use of such materials in many sectors of education and training, there has been a corresponding increase in interest in the role played by evaluation" [1].

A design would not be complete without some form of evaluation. The critical importance of evaluation is constantly stressed in the literature.

13.2 PROBLEMS ENCOUNTERED

However, valid and systematic evaluation is extremely costly.[2]. There is also argument over how the evaluation is best done, especially since some characteristics are more tangible and easier to evaluate than others [3].

In fact, the basis of educational theory is generally lacking in scientific backing, and it is difficult to evaluate that which has not been adequately formulated in the first place, since there is not adequate articulation of precisely what experiments to do, how to obtain results and what, specific results would constitute success. This is emphasized by a pioneer in computer education, P. Suppes:

"The basic scientific data on these matters are pitifully small. Opinions can be found in every educational group, but these opinions are just that" [4].

For this reason, a meaningful evaluation of **SOFTWIRE** was beyond the resources available. However, some evaluation was necessary, since evaluation pinpoints strengths and weaknesses, and provides a firm basis for modification and improvement [5].

Evaluation had to occur at different levels. There are a number of things which may be evaluated. These are:

1. How faithfully the overall aims have been realised.
2. How effectively the system has implemented these aims.
3. **SOFTWIRE**'s programming style, and the **User Interface** hardware design techniques.
4. How true-to-life the simulation is.
5. Course material which had been prepared.
6. Overall effectiveness of the system.

Evaluation also had to be done in different stages. First the author evaluated the work and made adjustments until it was deemed satisfactory. At the same time others were consulted for advice. This included people involved in engineering and education. Thereafter, the program and tutorials were distributed to students and further evaluation was done. It was hoped to improve evaluation by encouraging criticism of the system.

13.3 THE APPROACH OPTIONS

There are two main approaches to choose from, each with their committed adherents. These are discussed by J. Bloomer [6]. The first he referred to as the "agricultural/botanical" approach.

This has the most "scientific" basis. Experiments are done with tight controls, and the resulting outcomes are carefully measured using objectives-oriented evaluation techniques. The second approach he calls the "social/anthropological" approach. This is more concerned with the ongoing process that takes place during an exercise, and is based on the argument that there are so many variables in computer education that they are hard to control, and that the inputs and outputs of such experiments are not easily measurable.

13.4 THE APPROACH CHOSEN

Since there was not enough time to become involved in a philosophical debate, and not enough money to carry out detailed testing, the evaluation that was done had to be carefully chosen to fit the circumstances and immediate feedback requirements.

Cognitive tests -assessing how much had been learnt- were done only on an informal basis, by asking students. The thrust of the evaluation was in non-cognitive techniques, more specifically observational techniques and self-reporting techniques.

The observational part of the evaluation was to watch students, and see how smoothly the lessons progressed. The informal section involved general discussion, and the self-reporting involved the use of evaluation forms, with open-ended questions and semantic-differential scales combined with a Likert-weighting of perceived importance of each area.

The evaluation forms were designed to be as comprehensive as possible, to ensure that all important aspects of the program and the material were considered. They also served as a check-list for the author, to ensure nothing was forgotten as the system

development went through repeated iterations. Not all questions had to be answered by each person, as different people were qualified to discuss different areas. For example, students were not expected to comment on the modularity of the **Pascal** code. A copy of the evaluation sheet is available as APPENDIX C. It is loosely based on the format used in the **TICCIT** project (this is referred to in PART 1, in Chapter 2.

Even the "formal" evaluation was intended to be informal, and to serve as a guideline rather than a proof of the success of the system. This is what Bork calls an evaluative, rather than a summative evaluation [7], or what Percival et al. describe as "developmental testing of prototype materials" [8] rather than "the large-scale evaluation of a series of exercises".

13.5 A SUMMATIVE EVALUATION

A summative evaluation is done after the iterations are complete, and is only done properly at enormous cost. Anything less than such a full scale evaluation will not be conclusive. The main reason for this is that a smaller investigation is unlikely to be representative of the situation in which the systems is finally used. In the absence of substantial funding, tutoring would have to involve the creators of the material, who are more motivated and understand it better than subsequent users, and so are not representative.

The students may not be representative either. If volunteers are used, they will be the students who are already motivated. A control experiment would also have to be done, in which some students were given an alternative program. A problem also exists with the "Hawthorne effect" -when people have an interest taken in them, they tend to perform better regardless of what the experiment is.

The lecturer/student ratio may not be representative either. A good test would have to try different ratios.

13.6. EVALUATING THE SIMULATION ITSELF

Some ideas are discussed relating to attributes of simulations. These were originally given by R. Sanderson [9]

a) Realism. One must be aware of which features of reality are stressed, and which are omitted, and the educational significance of that. This relates largely to the implementation of the modules. Since they are relatively non-complex, this evaluation is done simply by checking whether the result of a simulation was expected. This aspect is considered in the chapter on MATHEMATICAL CONSIDERATIONS.

b) Complexity. There must be consideration of the number of variables presented to the student, and whether this number increases as the simulation progresses. Also consider the number of decisions to be made, and range of alternative strategies to be chosen by the student. Since **SOFTWIRE** is actually a language, rather than simulation, it is up to the lecturer to be continually assessing these levels.

c) Flexibility. This is the ability to change the simulation for different situations. This aspect is crucial to **SOFTWIRE**, with a broad range of aims.

d) Information. The quantity, quality and format of what information is presented must be assessed. These aspects are crucial in determining the educational value of the system.

e) Interaction. Ask what dynamics existed and what interaction was there between students, between students and the tutor, and between students and the machine. These must be compared with those that existed when alternative methods were used. A

large part of this system's claim to success is due to improved interaction.

f) Tutor presentation. The tutor must do a self-evaluation of his role in the learning. When was a lecture given rather than advice, and was this appropriate? When was there intervention, and was this useful? Could the material have been presented in a better way?

g) Organization. How good is the hardware used? what is the timing of the different learning activities like? How well is space used? How accessibile is the system? These are questions also left to lecturers to reflect on.

13.7. CONCLUSION

Again it must be stressed that a thorough evaluation was not feasible. This is emphasized in the following opinion by Ellington et al.:

"There is a big deficiency in evaluation. This has resulted from disagreement on the methods to use, poor quality research so far, and intrinsic difficulties in doing evaluation. Most work has been short-term or had findings which could not be generalised."

"One factor common to most evaluations done is that they were not extensive enough for the results to be used with confidence" [10].

However, as Bork points out, this is much the same as with any other learning techniques [11].

A good test of effectiveness -arguably the best test- is analogous to the testing of a new product in the market place. If the

system ends up being regularly used, then it has been eminently more successful than if it ends up on the shelf like so much other educational software. So far the program is being used and this has not been the case.

REFERENCES

1. F. Percival, H. I. Ellington and E. Addinall, "A large-scale evaluation of gaming and simulation materials: some problems and pitfalls", B. Hollinshead and M. Yorke, eds, Perspectives on Academic Gaming and Simulation, Simulation and Games: The Real and Ideal, 1980, p. 201.

2. V.P. Hansen (1984 yearbook editor) and MJ Zweng (General Editor), Computers in Mathematics Education 1984 yearbook, National Council Of Teachers Of Mathematics (USA), preface.

3. H. I. Ellington, E. Addinall and F. Percival, Games and Simulations In Science Education, Kogan Page Limited: London, 1981, p. 114.

4. P. Suppes and M. Morningstar, Computer-Assisted Instruction at Stanford, 1966-68: Data, Models and Evaluation of the Arithmetic Programs, Academic Press, 1972, p. 6.

5. H. I. Ellington, 1981, p. 114.

6. J. Bloomer, "Paradigms of Evaluation", SAGSET Journal, 1975, V. 5, n. 1, p. 36

7. A. M. Bork, Personal Computers For Education, Harper and Row: New York, 1985, Chapter 8.

8. F. Percival, H. I. Ellington and E. Addinall, 1980, p. 202.

9. R. Sanderson, "Developing a model for appraising games", Simulation/Games for Learning, Vol 9, 1974, p. 20-28.

10. H. I. Ellington, 1981, Chapter 8.

11. A. Bork, 1985, Chapter 8.

PART 3

PEDAGOGIC DESIGN ASPECTS

Part 3 will briefly outline pedagogic aspects of the system, progressing through abstraction levels from broad learning theory, to pedagogic strategy, concluding with some practical examples.

CHAPTER 14

A GUIDE TO THE USE OF SOFTWIRE AS A TEACHING TOOL

14.1. PEDAGOGICAL GOALS

It is suggested that the **SOFTWIRE** system be used with clear-cut pedagogical goals. The use of an educational tool is every bit as important as the tool itself. In fact, the hardware, the software, the student, the curriculum and the tutor should be regarded as integral and inter-dependent parts of a system, so that neglect of any one area is detrimental to the effectiveness of the system.

"But in order to have a complete picture, we must also recognize a dialectical interaction between the content, the pedagogy and the technology" [1].

This viewpoint is not a popular one at present, as society has not sufficiently recognised the importance of inter-relating the two disciplines. This is probably because educationalists are usually not technically orientated, and scientists and engineers are more concerned with science than with the students learning it.

"Funding agencies as well as universities do not offer a place for any research too deeply involved with the ideas of science for it to fall under the heading of education and too deeply engaged in an educational perspective for it to fall under the heading of science. It seems to be nobody's business to think in a fundamental way about science in relation to the way people think and learn it" [2].

In this chapter, the curriculum and tutor will be examined together. They must work together to achieve certain pedagogical goals. Some of these goals are isolated and mooted in this and the next section. They are consistent with some of the present trends in good educational philosophies.

14.1.1. GENERAL GOALS

Although different schools of thought on education have different emphases or go to different lengths, the mainstream of modern thought is to produce education which emphasises the role of the student. Instead of being a passive receiver of information, the student is trained to become an active and independent learner. The qualities of such a learner are that he is critical and discerning in what information he retains. He knows what information he requires and how to acquire it.

The learning process then requires active participation by the learner. It gives the learner responsibility to determine what is being learnt, how it is learnt, and at what rate. The learning process involves an experience of what is being learnt (often the learner obtains this experience by being placed in an appropriate learning environment). It kindles enthusiasm in the normal learner, and a desire to learn more.

A few of the major concepts involved will be elaborated on. There is some overlap in the concepts, since they are usually different aspects of the same few key features of the modern learning paradigm.

DISCOVERY LEARNING

This refers to open-ended investigation, where it is left to the student to discover the principles for himself. It is not so much "teaching", as pointing along a specific path. With less teacher control, there is no guarantee that learning will take place -but experience shows that it does. It is concerned with learning not only facts, but insight too [3].

A more elaborate view of discovery learning is given by J. Piaget. The application of this in science is discussed by R. Batt.

"There is growing interest among science educators in the Piagetian learning cycle for teaching abstract scientific concepts and formal operational thought. Very briefly, a learning cycle organizes the student's learning experience into three phases: Exploration, Concept Invention or Development, and Discovery or Concept Application." [4].

Batt then elaborates on each phase. The details are not important here, but the main features that would apply in Engineering are best described by an example[1]. The steps are:-

1. The students are given a net-list, and they may experiment and look for trends etc with no guidance from the lecturer.
2. The lecturer then asks for a mathematical model or explanation of what was observed. The students realise they do not have the power to give one, based on what they have learnt before. At this point they are in a state of 'disequilibration' -where their previous knowledge cannot accomodate the new phenomena.

[1] A concept such as "feedback" would be one area where this could be used effectively.

3. During the lecture the class makes a mathematical model (guided by the lecturer) and applies it to the simulation that was run.
4. The model which is devised is now applied to slightly different situations.

The first phase is where the student discovers and experiences largely without guidance or experimental protocol. The last phase is similar, except the student now has more information on which to base further discovery.

LEARNING AN APPROPRIATE APPROACH TO PROBLEM-SOLVING

Learning facts is only a small fraction of the total learning process. Learning must also stimulate the development of a concept, a generalization, a procedure or a solution. The student must be prepared to accept a challenge, and know how to construct an understanding and how to reflect on the processes of thinking that might be used [5].

When faced with a problem, a good learner "solves, generalizes, abstracts, conjectures, searches, observes, simplifies, experiments, deduces or remembers". For development of this process there must be firsthand experience, with each learner engaged in his personal "activity and responsibility and individuality" [6].

INTUITIVE INSIGHT AND ANALYTICAL UNDERSTANDING

Theorists agree on the need to have an "intuitive" component to learning, but they differ as to what this means. This is because there is no sure knowledge on how the mind thinks and learns. Behavioural scientists concern themselves with motivating people

to learn, regardless of how, but others are hesitant to subject students to processes which may be destructive to their learning processes.

At one extreme, some believe that all thought can be reduced to rules and a knowledge base, and that learning is merely acquiring new rules and knowledge, or refining old rules. A slightly less simplistic view is held by S. Papert, who has the same basic viewpoint, but believes there is more complexity involved, and that the brain also stores perceptions and information on knowledge groupings. This understanding of the brain sees it as primarily **analytical**. At the other extreme (although the viewpoint is not really "extreme") H. Dreyfus and S. Dreyfus believe that the brain has an **intuitive** component, where thinking is done unconsciously, and is based on previous experiences, which are stored not only as facts, but as whole interrelated images, combining memory of personal actions, the consequences of those actions, and the emotional response to that (in learning this normally involves feelings of success or failure). The salient points of the experience are also highlighted.

The differences are shown in the perception of intuitive learning:

"Programming the computer in domains such as math and physics -and, in principle, in any domain in which intuition and knowledge are the goal- consists in learning to think procedurally, like a computer. One learns by acquiring and debugging a mental program. Seymour Papert is the best exponent of this computational model of human thinking, which he calls the epistemological view." [7].

Dreyfus and Dreyfus believe that learning occurs in five stages. At the first stage, all learning is rule-based, and success is

measured by how well these rules are acquired. These rules are not learnt in a context. As the learner progresses through the stages, the rules are applied to the context of what is being learnt. At this stage the brain begins to store images of experiences in learning, and to use them in future situations. Gradually the learner selects actions without being conscious of them, and ultimately he both assesses a situation and acts appropriately without being conscious of the process. Only when a new problem is encountered does he revert to analytical processing of the response. [8]

In another sense, Dreyfus describes Papert's student as starting with specific cases and gradually abstracting and interiorising more and more sophisticated rules, whereas Dreyfus' student moves from abstract rules to specific cases.

It seems that the differing theories would converge in advocating simulation as a useful learning process. By running a simple solution, the student can learn simple rules, and become familiar with them, thus developing analytical insight. By increasing the complexity, one can learn to contextualise these rules, until eventually this process becomes automatic. With even more complex simulations, one can develop the "intuitive feel" or "grasp" that is described in different ways by both sets of theorists as being important.

STUDENT RESPONSIBILITY

This is necessary if space is to be allowed for personality in learning, and if students are to become independent learners, able to acquire information themselves after they have left a learning institution.

"Moreover, it is not just a matter of making mistakes, but of taking decisions. So much of learning is constrained to predetermined routes, whereas gaming and simulation offer the student choices, problems to solve and decisions to take." [9]

The extreme responsibility of the student arises when the student actually programs. As Papert says:

"In many schools today the phrase "computer-aided instruction" means the computer teaches the child. One might say **the computer is being used to program the child**. In my vision, **the child programs the computer** and, in doing so, both acquires a sense of mastery over a piece of the most powerful technology and establishes an intimate contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model building" [10].

In **SOFTWIRE**, the student not only controls the simulations, but may also create his own simulations using the net-list language.

14.1.2. GOALS RELATED TO SIMULATION

R. Hooper, who directed the NDPCAL program discussed in Chapter 2, concluded that the pedagogical value of simulations (which he refers to as the "'revelatory' paradigm", when used in a particular way, was higher than with other known computer-based education modes. He describes the process:

"In terms of the underlying educational psychology, theorists such as Bruner (the spiral curriculum) and perhaps Asubel (subsumption theory) would be most supportive. Typically, the view of learning emphasises closing the gap between the structure of the student's knowledge and the structure of the discipline he

is trying to master. It could be labelled the 'conceptual' paradigm because of the importance attached to the key ideas of established knowledge fields. We call it 'revelatory' because these key ideas are more or less gradually 'revealed' to the learner." [11]

Hooper then mentions the key concept as being "discovery, intuition, getting a 'feel' for ideas in the field, etc.", and the curriculum emphasis: "the student as the subject of education". Simulation, he says, provides the educational means for the provision of opportunities for discovery and vicarious experience". He discusses the simulations used in NDPCAL: " Each simulation package is built around a mathematical model of a physical system; as the student manipulates it, he is expected to develop an intuitive understanding of the model. This understanding helps him to appreciate the theoretical formalisation of the model"

14.1.3. GOALS SPECIFIC TO SOFTWIRE

In using **SOFTWIRE**, three specific goals may be identified. These goals correspond to what should be the main foci of general electronics courses. Students must first be taught a sound, systematic and disciplined approach to electronics and related subjects. They must then be introduced to the problems experienced in implementation: the transition from modules on paper to electronics in the real world. Thirdly, students must develop an intuitive understanding of electronics, and be able to estimate and approximate confidently.

THE MODULAR, TOP-DOWN APPROACH

Two key concepts which are inherent in the way a system is described to **SOFTWIRE** when a net-list is input. These are modularisation and abstraction. Modularization is inherent because the net-list describes systems in terms of modules. Abstraction is inherent because the modules used abstract from their implementation in the real world.

These key concepts should form the basis of the systematic top-down approach, where the student learns to define modules at a high level, then recursively re-define modules in terms of sub-modules which are themselves defined, and which have their relationships between other modules at the same level strictly defined. This process continues until modules of a manageable complexity are attained. These modules are then realised.

Often electronics is not as "clean" as modular design would imply, with many issues related to interfacing modules and combining different modules in non-standard ways, which may suggest a solution which would not have been obtained using a modular approach. However it is still felt that the formation of good design habits and a sound modular approach is an essential basis for all work.

ESTIMATION AND APPROXIMATION

Students must also develop a confidence in themselves and a "feel" and comfortableness with electronics. They must be able to assess design projects rapidly, and have an idea of what it involves. They must learn how to make decisions and learn what criteria to accept or reject. They must also be taught how to

estimate and use approximations with confidence. They must learn to read a schematic quickly and intelligently and filter out what is or is not important. This requires a good overview and a good approach, rather than a preoccupation with accuracy and detail, which are necessary later on.

S. Damarin comments on teaching estimation, saying that this becomes incresingly important in proportion to society's use of computers and packages, but that the teacher has few resources for teaching this skill. He says that recent research reveals that students are generally poor estimators. Teaching estimation is not simple, however, since there are many issues, such as "how one decides how good an estimate is, how students can be forced to focus on estimation rather than computation, how one encourages the use of diverse strategies, and how to involve the context in the estimate." [12].

IMPLEMENTATION

A student who has a sound design strategy must be aware of the many ways in which the physical implementations of systems differ from their modular descriptions.

For instance, an integrator module on paper may be merely a block with an integral sign in it. The implementation may be inverted, work over a small voltage range and be subject to drift, as well as requiring temparature compensation.

SOFTWIRE has features built in specifically to teach implement-ation, although this is intended to remain at a simple level only. Two features in particular should be exploited.

a) The **User Interface** is designed so that actual hardware modules can be included in the system.

b) Modules may be designed in software and included in the **SOFTWIRE** library. These are of arbitrary complexity, and thus may have any limitations or quirks actually built into them. One could also model approximations of irregularities using ideal modules.

SIMULATION AND MODELLING CONCEPTS

Apart from learning concepts about implementation, the student needs to learn about simulation itself, and modelling, and its judicious use as a tool. Obviously **SOFTWIRE** has the intrinsic ability to enhance learning in this area.

For electrical engineers, the concept of continuous and discrete systems is an important one to become familiar with, as computers become involved in an increasing number of areas previously confined to the analogue domain, and the analogue-digital interface becomes more commonplace.

"Discrete mathematics is a very important part of mathematics, rich and useful. The student needs to be introduced to it, and helped to become master of the computer. But it is not the whole of mathematics; continuous magnitudes and their study form the other half of mathematics. The fact that a computer can approximate a continuous function with a high degree of accuracy must not distract us from the fact that it is continuous and gives discrete images of things" [13].

14.2. TEACHING THE USE OF SOFTWIRE

14.2.1. TEACHING SYNTAX

Various methods have been adopted for teaching students the sy syntax of computer languages. Their main aims are to make the student proficient in the language as fast as possible, and sometimes also to teach good programming style [bork pg20]. Since the distinction between programming and using a computer is not a clear one, the **SOFTWIRE** system description syntax may or may not be considered as a language. It is a language in the sense that it has an unambiguous syntax which represents branches, loops and iterations, but this syntax can also be seen as merely a way of representing data. However this is not important. The two main issues are how to teach the syntax, and whether or not style is important.

STYLE

Good style is important for readability of the system -quickly ascertaining what it is doing. It can also be important if one is using the language as an intellectual tool to train the user how to effectively convert the original problem into intellectually manageably sized and well-defined sub-problems with intellectually manageably sized and well-defined inter-relationships.

In the case of **SOFTWIRE**, the syntax is for the most part merely a direct mapping from a standard "functional modules" format to an input suitable for computers. The functional module concept is the intellectual tool. Good style then would be to convert problem solutions into their functional modules, in a way analogous to structured programming. As in the case of structured programming, one can then use the "top-down" apporach of sub-

gramming, one can then use the "top-down" approach of sub-dividing these modules into smaller units. With **SOFTWIRE**, unlike with most programming languages, the size of these smaller units is exactly fixed by the intrinsic module functions available.

In teaching style, one does not have much scope with **SOFTWIRE** itself, but the tutor can require that students draw clear block diagrams while they use **SOFTWIRE**. This is a crucial process, because it forces the learner to participate actively in the lesson, rather than to develop the "video stare" which is a common problem amongst passive learners using visual aids, where they discover after the lesson that they have learnt very little, but were rather mesmerised by the activities of the phosphor dots on the screen.

SYNTAX

There are many ways of teaching syntax. The most common is to teach progressively more complex grammar rules [15]. Although the most common, it is not necessarily the best system. Another method is to give a working program, then to run it and explain what each line does, as well as to give small changes to the program and re-run, to show what lines are responsible for what actions. It is important to note that these two methods are both forms of tutorial deliveries, and the issues that arise are covered more fully in a later section on tutorials. In that section, the first method discussed here is closely analogous to the "brick-by-brick" method, and the second is closely analogous to the "what-if" method.

Both options have merit, and could be used on **SOFTWIRE**. The first method has merit because **SOFTWIRE** has fairly powerful and simple features. This means that a first-time user can begin to see

results quickly, and his attention is secured. Also, the user can ensure that mastery is achieved at each step. This method is very similar to the "brick-by-brick" tutorial method discussed later in the chapter. In this context it has a major failing in that the person learning does not have an overview of what is happening, and cannot contextualise what is being learnt at the beginning into an overall understanding of the system.

The second method is easy because it is very simple for the tutor to set up a file before the lesson, and the user can then run this file with only a few keystrokes, and would be running a complex simulation already. This method provides a better overview, but can appear more daunting to the beginner, as it does not inspire confidence through a sense of mastery.

Either way, the syntax has not been found to be too difficult to teach, because it was intended to be as intuitively obvious as possible. There are relatively few grammar rules, and probably the only difficult concept to convey is the conversion of a particular module as represented on paper, into a line of the net-list. A feature of **SOFTWIRE** which makes it far more simple to learn, is that it has been especially designed to unfold in complexity only as the user requires more flexibility.

14.3. AREAS OF USEAGE OF SOFTWIRE

One can safely assume that **SOFTWIRE**, with its inherent flexibility, could be usefully employed in the education of anyone learning about electronics and related fields. It is felt that the standard of education could range from Secondary to Post-Graduate level. Indeed, it has been tried informally at both those levels and in-between. The fields where it could find application in teaching are in Measurements, Microwaves, Amplifiers, Oscillators, Rectifiers, Signal Processing, Telecommunications, Energy Utilisation (e.g solar cells) and Control as well as other fields of Electronics. At least one example of each area mentioned will be given at the end of the chapter.

14.4. MODES OF USEAGE OF SOFTWIRE

14.4.1. DELIVERY VIA A TUTORIAL

"Simulation is usually effective when combined with graphics and embedded in tutorial dialog that checks whether the student understands the objective of the simulation and can interpret the results correctly. [16]

This is the mode of useage in which the most exploratory work has been done in this project. The student is given a step-by-step worksheet (or tutorial), and proceeds through it while carrying out instructions in **SOFTWIRE**, adding modules and changing operating parameters as required, and seeing the result. It also involves the student's having to answer questions, make predictions and solve small design problems. Diagrams must also be drawn at each stage. Scope is given for the student to experiment and make extra observations.

At the end of the tutorial, the student may be given a small design problem which encourages the student to consolidate all the knowledge that has been acquired. This is similar to the "problem-solving" mode which will be discussed later.

The progress of the student is judged via a report that they hand in after the experiment, indicating in their own words what they did and what they learnt, as well as showing any answers and calculations required in the tutorial.

Advantages of this method are that it is self-paced, it allows open-ended exploration for more motivated students, and it requires constant active student involvement in the learning process. These ingredients have already been discussed at the

beginning of the chapter as being important for a good learning environment. It is a form of feedback control, where the learning process proceeds at a rate determined by how much the student perceives he has learnt. The self-pacing is desirable from the point of view of tailoring the speed of learning, and weaning the student away from more authoritarian learning environments. However the requirement that diagrams be drawn at each level ensures a disciplined approach.

The tutorial does not require close supervision. If it is done in a micro-laboratory, with many people at once, supervisors are provided purely to ensure the laboratory runs smoothly and that questions which arise are dealt with. It may also be done with fewer computers, on a basis where the student can go any time when the supervisor is close at hand, ready to be called if problems arise. Personal experience is that people regularly do the entire practical without any help at all.

Tutorials lend themselves to group work. There is a common assumption in computer education that each person must have a workstation. This is not logical, since in other educational contexts a high premium is attached to group work. Students working in a group often learn more rapidly through an exchange of ideas. This reduces the demand on computer resources while in no way diminishing the effectiveness of the learning process.

Tutorials are further useful in that they easily allow for consolidation of the work and also provide a mechanism for judging the performance of each student.

There are three different common formats in the tutorial mode:

a) Brick-by-brick

Here, the tutorial starts with a simple circuit, and slowly adds on one module at a time, to achieve a more complex circuit. At each stage the student observes the result and there is an exercise that the student must do, which shows that the student has understood the step.

This method has the advantage that it ensures that the student understands what is happening at each point of complexity, and it minimises the problem of the student building on something that is not understood. It is inherently self-paced.

This method is especially useful if it requires a lot of understanding to master particular single modules, and if a high level of familiarity with these modules is required before they can be used. A module which would not qualify would be the **summer**, where two inputs are summed at the output. At undergraduate level a student would already be familiar with this concept. An example of a module to be more carefully introduced, would be the integrator. Although undergraduate students in their second year might have encountered calculus many times in mathematics, they are often not at the point of familiarity that they can decide at a glance what output would result from even a simple input. For instance, a large percentage of students do not expect the output from an integrator to stay constant when they zero the input, although they "know" from their mathematical backgrounds that this is the case. As a result, it would be wise to dwell on the integrator for a while before introducing it into more complex systems.

b) Blow-by-Blow

This is similar to the Brick-by-Brick tutorial, except in this mode the student is first introduced to an idealised module, and then proceeds in steps to be shown how the physical implementation differs from the ideal, and what the consequences are.

The first step could include an absolutely simplified module, and the last step could include an actual hardware module, plugged into the **User Interface**.

An alternative to substituting simplified modules with more complex ones, is to model the complexities using other simple ideal modules. This could save time for the person setting the tutorial, as they would not have to create more modules for the **SOFTWIRE** library, if they did not already exist. More importantly, it is itself an important educational process to show how devices can be modelled by simpler devices. This fits in with the teaching goal of being able to rapidly estimate and approximate.

c) What-if

The user is supplied with an input file of a net-list which has already been created. This is loaded and run. The tutorial procedes in steps, where each step consists of making a simple alteration and observing the result. The two main alteration modes are firstly when alterations proceed in a specific direction (similar to the "brick-by-brick" method), or secondly when the user returns to the original system after each alteration. Various mode combinations can be tried, and if necessary, the user can always reload the file again, to

return to the original circuit. Questions can be asked of the student to test understanding at each point, and to encourage directed exploration.

This mode is especially useful when the concept being introduced is itself a system, rather than a single function (although there is no distinction). For example it would be the appropiate way to introduce a phase-locked loop system. One could then see how a change made to a component of that system affected the system as a whole.

This mode is a response to a problem that occurs in teaching via simulations which is lack of discipline. Simulations are seen as a powerful way of providing an intuitive feel for a system -as has been stressed. However caution must be exercised to minimise the problem that occurs when a student is allowed to "explore" a simulation. It is easy to twiddle knobs and make adjustments aimlessly, and waste time through not knowing what to look for. A good tutorial must thus increase the discipline of the student in approaching the simulation, while still being self-paced and providing the option of open-ended exploration. When the student is finished the tutorial, he must not only have a good grasp of the subject-matter of the tutorial, but also be a little more proficient at learning what type of aspects to consider in future simulations. The natural ideal conclusion would then be that students in later years would merely be given a system and a simple instruction "Investigate and comment on this system". The student would become an independent learner with the tutor there to facilitate rather than initiate learning.

PREPARING A TUTORIAL

A thorough description of this process may be obtained from G. Orwig's book [17]. Various points are highlighted.

Before starting, the teaching goals must be clearly stated. The instructional goals of students are more important than one's personal goals of instructor. Set a main goal for each lesson, (if there is more than one sub-goal, one can have more than one unit in each lesson).

The goal is described in terms of the learning objective, which describes the behaviour expected from the learner as a result of the lesson, the conditions under which the learner is expected to exhibit this behaviour and the minimum acceptable level of performance. This behaviour may be cognitive (recognition, recall, concept formation, problem solving, synthesis) or affective (change in attitude and approach etc). The behaviour is best described by actions e.g. "construct, repair, recite, recall, classify, describe, solve, create, predict". This makes the objectives more concrete, and easier to evaluate.

One must also specify the conditions of the lesson, such as restrictions (no books may be referenced, no calculator may be used) and aids (a demonstrator will be present, or a pen and paper are to be used). The author believes that a pen and paper should **always** be used in a simulation tutorial.

One then specifies the acceptable performance, in terms of the percentage of questions correct, and maximum time allowed. This will normally refer to a minimum standard in a report done on the tutorial, and the standard will be judged by appearance, clarity

of block-diagrams and how well the student appears to have understood what was being taught.

Orwig continues discussing the preparation, changing the emphasis from the lesson to the learner [18]. Before starting, he must have certain "tool skills", which are the minimal skills required at the start of the session. They refer to previous experience, and ability.

The next step is to describe the task, and the exact steps required. Decide whether the learning processes are to be active or cognitive. Most people find it easier to start with an active task. The tasks may be linear or there may be a choice of sequences. Usually **SOFTWIRE** tutorials are expected to be short and simple, so the linear mode is to be preferred.

A typical lesson would have an introduction, to give the student an overview of what is coming. Then there may be preliminary questions, to check if the student is too advanced or not advanced enough. This may often lead to the chance for advanced students to skip some of the early work. There then follows a cycle of teaching and testing, and opportunities for practice and familiarisation.

The teaching often follows the "ruleg-egrule" [19] (rule with example, or example followed by rule). Orwig suggests that egrule is normally better for new material, and ruleg for revision.

A response is normally expected to force student interaction. There are many possibilities here, but the most recommended is for the student to have to work out all block-diagrams, to have to construct net-lists, to have to specify module parameters which would make the simulation meet stated requirements, and to have to explain particular behaviour of modules and systems..

at the keyboard during the class, and let the class see what is being done. If not, he can go through the sequence before the lecture, trying out each step, and saving a new file at each step. In the lesson he need only load a sequence of files.

This method reduces the amount of equipment needed to do demonstrations, since they are all done on the same computer. It also reduces the distractions that occur when demonstrations need to be set up (often with a laboratory attendant walking in and out of the room), and often fail to work as planned, especially when a sequence of demonstrations is being done, and the wrong wire is pulled out, or a sensitive adjustment needs to be made, and the demonstrator cannot perform properly because of the pressure of the impatient, unbelieving or well-humoured class.

14.4.3. PROBLEM-SOLVING

After different concepts have been covered in a course, the tutor provides a problem for the students to solve, which requires different aspects of different sections of the course to be brought together.

This mode encourages consolidation, and does not allow the student to solve the problem without having a genuine understanding of the material.

SYNTHESIS

The phrase "problem-solving" would normally mean to provide the students with a list of modules to choose from, and ask them to create a net-list of a system which performs a particular function.

In this case, marking would be remarkably simple. The students hand in their disks with saved net-list files on them -possibly together with a short report. It will take very little time to run each person's file and see what it does. One could alternatively require them to provide a hard-copy of their graphic output and net-list. The ease in marking also allows the lecturer to set different problems for different members of the class, without having to solve each one. This reduces the temptation of copying.

The advantage of this mode is that it fits in well with a classical 'top-down' modular design strategy, and could replace the type of design problems which are traditionally given.

MATHEMATICAL ANALYSIS

An alternative method of "problem-solving" is to provide a net-list and ask what it does. the solution must be found first with pen-and-paper. This is preferably done in class, away from the temptation to run the simulation first. After people have examined the system in question and decided what it does, they may run the net-list and compare actual and predicted results. If they are surprised by the result, they may examine the output from any of the nodes and see at what stage their predictions went askew.

Many people graduate from electronic and related courses knowing design methods, but not how to analyse circuits and systems created by other people. They often a lack a disciplined approach as to where to start, in what direction to proceed, and how to group related parts of the system logically. They may know how to do 'top-down' design, but cannot then do 'bottom-up' analysis of someone else's design. The reality of the field is that job

descriptions do not involve only designing systems. Often the system exists and must be commissioned (it often does not work on installation), maintained and repaired, tested, modified (if it is required to perform a different task from what it was created for), upgraded, added-onto or have its efficiency or performance improved.

The method prescribed here is suggested for providing training in understanding systems when one is provided with the necessary documentation.

It is recommended that students be allowed to find out if their predictions were correct themselves. This is one mode where the exploratory nature of doing a simulation should be exploited to make students less afraid to make mistakes. A mistake is usually associated with a penalty -usually a mark taken off. This forces students to take careful steps where there is less certainty. In some cases this is a good thing as it prevents rash action. In cases such as exploration it stunts creativity.

SOFTWIRE creates an ideal environment for this type of analysis. It allows outputs from any part of the system to be examined, and so allows the student to see for himself where his errors are.

"BLACK BOX" ANALYSIS

Again the student is provided with a net-list and asked what the system represented does. In this case, however, the student may be present at a keyboard. Usually running the system will not give much insight. The student must decide what stimulus to give the system, and observe the results. From this, the function of the system must be decided.

Since in **SOFTWIRE** one can examine the net-list, the true "black box" situation cannot be ensured. The ways around this are to trust the student, to provide a circuit of the complexity which makes it less appealing to attempt to analyse manually, or to tailor **SOFTWIRE** by adding a complex module or modules or changing the names of exisiting modules so that the functions in the net-list are not meaningful to the student. It is suggested that the final option is not done, but that the net-list be made available, and the marking of the exercise be accomplished by the student handing in a short report of a sequence of inputs tried, observations made and conclusions drawn, as well as a justification for the choice and sequence of tests where applicable.

The inability to hide the net-list need not be a problem, but in some cases may be desirable. This is when the student is allowed to inject stimuli into any part of the system, and test the results. In this case the advantages are the same as for the mathematical analysis, where the ability to treat a large system as smaller sub-systems is required.

There is a "black box" approach which can be used, and that is to insert hardware into a **User Interface** slot, and present that to the students. By covering the circuitry with epoxy one could ensure secrecy of the contents. This mode could also be used to demonstrate the added complexities of testing a hardware black box as opposed to a well-behaved software black-box.

The advantages of using **SOFTWIRE** for this type of work are the simplicity and versatility in positioning and defining the type of the signals to be tried, and the ease with which the output may be obtained and displayed.

MODIFICATION

A more subtle method would be to provide a net-list, and ask for minor adjustments to be made in order for it to perform a specified function slightly differently or more stringently specified than its original function. A special-case of this is where a system is given which is faulty and troubleshooting is required.

The uses of this mode are similar to those of the previous one. It is obviously especially useful for people who will be in a job which requires a lot of repair work. It could be argued that this is best taught without the abstraction. For instance a TV-repair man would like to go straight into learning how to repair televisions. However, this may be short-sighted for three reasons. The first is that it has previously been less simple to give training at the higher level of abstraction that **SOFTWIRE** offers, and so it has not been considered. The second is that an experienced technician's repertoire of "short-cuts","tricks" and other boasts that have been acquired over the years look impressive enough to give the impression that that is all that is required in the job. For instance one often hears "The manual says wait for five seconds, but I never believe the manual...". This gives a sense that the "black art" approach is more valid. Sometimes it may be, but not always, especially as more and more fields of science are becoming rationalised and systematised and given a more theoretical backing. The final reason why abstraction is important is that it equips a person for promotion, by giving some insight into "why" and not just "how" something works.

"THE MODULE GAME"

Analogous to "the word game", one is given a number of modules and asked to create as many useful functions as possible, with a report on possible applications. More than anything else, this stimulates interest -especially amongst the more enthusiastic of the class- and should be interspersed amongst other modes. Again for the keen students, it is exceptionally open-ended, and may be educational to the lecturer too! This mode stimulates creativity and resourcefulness, which are two qualities said to be sought after in engineers, yet seldom targeted as goals for growth areas in a course.

14.4.4. HARDWARE DESIGN PRACTICAL

A particualrly neat application of **SOFTWIRE** if the **User Interface** is available, is to set practicals for students in which hardware must be designed and tested.

The **User Interface** Cabinet provides two slots into which small inexpensive cards can be plugged. As mentioned in the chapter describing this, the slots provide the cards with power as well as input and output digital and analogue channels. The use of these cards greatly facilitates electronic design, as it reduces the difficulty involved with providing a power supply, and input signal generators, and monitoring the output.

With this equipment a tutor could set the class a design project. The student would first find and describe the solution at a higher level of abstraction, which could then be tested directly using **SOFTWIRE** in very little time. The student would then design the hardware on a card and would write a short "test-bed" program

in **SOFTWIRE** to test it. This might even be the same net-list as was used when testing the software solution, with only a few minor adjustments to accomodate the hardware.

GROUP WORK

Group work in design projects is becomingly increasingly popular as an important component to technical education. One of the main reasons is that most large industrial design projects are done by teams of people, and a lack of group skills may seriously undermine a person's technical skills.

There is no reason why the design project on the **User Interface** should not be done as a group project. In fact it is ideal for this. That is because each person can devise a test-bed in which to test their circuitry, before the different sections are put together. In this way each person has individual responsibility, but the success of the group project depends on how well they have defined the task for each individual, and how well they have defined the interfaces between the designs.

Again the lecturer can easily test the work, by writing a test routine in **SOFTWIRE**, and simply plugging in the cards and seeing the result.

Some design work can be done successively. The entire solution is worked out using software modules at first, and then the software modules are weaned off one at a time and replaced with hardware, ensuring everything works at each step. Obviously this only works in situations where voltage values are passed between modules, not currents.

14.4.5. TRANSFER FUNCTION RECORDER

This option is only possible if the **User Interface** is available. It involves inserting a card into the slot, on which is the circuitry that one requires the transfer function of. One then plots Vout vs Vin, where Vout is the voltage from the module in the slot after a voltage, Vin, has been injected. Vin is typically spanned across a range by linking it simply to the time base. The total time specified by the user corresponds to the maximum X-axis value required, and the time increments represent the increments in Vin required. There is more flexibility in the input, of course, if it is processed by a few modules first. For instance a log "sweep" may be obtained by processing the output from the time module through the **log** module. The output may also be processed by a few modules (such as a filter) before it appears on the screen.

This is useful, for instance, in teaching about modules such as amplifiers where the voltage transfer function is important, and may not be well-behaved. It allows the student to get an overall picture of the transfer function, and then to change the input range to zoom in on various parts of the range which are more interesting.

This mode represents a distinct mode of useage of **SOFTWIRE**, but it is not necessarily a learning mode in itself, and can be used in many of the other learning modes.

14.4.6. CHART RECORDER

Here, the system is used to record data, which is input over time. The length of time can be arbitrarily long (up to a few

years!) or short (down to a few milliseconds), which includes many things likely to be investigated.

This makes available an investigative mode, in which the student can use **SOFTWIRE** indirectly to measure other electrical phenomena.

Since the output can be manipulated by a **SOFTWIRE** system, and directed to a file, in ASCII form, and time-stamped, there is a limitless number of ways in which the data can be used and processed.

A SYSTEM CONTROLLER

For instance, the computer could use the information it obtains to control a device. It may be part of a feedback loop, in which case it performs control with automatic logging. This is an ideal basis for experiments in control theory.

14.4.7. ANALYSING TEST-BED

This concept has already been alluded to in the HARDWARE DESIGN PRACTICAL mode. The student can inject voltages into circuitry and examine the results. This section is more concerned with the particular use of this mode where the system is used to process the incoming data (which may not even be from the **User Interface**, but may have been internally generated) in such a way as to extract useful information. In other words the net-list includes modules which actually make up an analysing instrument, to measure something such as the periodicity, phase-difference or true RMS of a signal.

In this way the class can do experiments with a variety of instruments, where it would be too expensive to actually stock

the entire range of instruments, and enough for a class to use. They also do not have to become familiar with each instrument.

An added advantage is that students can see exactly how the instrument is measuring, since the net-list is available. This advantage can be exploited even further by requiring the students to "write their own instruments" -devising their own net-lists, and in so doing obtaining and demonstrating an understanding of the instrument.

This mode of making one's own instruments could even be taken further to actually have, as part of an Instruments Course, assignments where the students are required to design instruments to measure various properties.

14.4.8. LABORATORY SIMULATION

Ellington et al. refer to this as "possibly the most important potential application" for simulation, where the computer is used as a "supplement or substitute for conventional experimental work" in teaching basic laboratory skills". [21].

Lower et al. describe the various ways the computer may be used to enhance normal laboratory work [22]

1. "Pre-lab CAI simulates a laboratory exercise in advance. Students do an entire simulated experiment and report before repeating the same experimental procedure in the real laboratory. This ensures that each student is adequately prepared. It also saves wear-and-tear on sensitive instruments.
2. Post-lab CAI allows a student to use the computer for manipulation of experimental results. (This is not an appropriate application of **SOFTWIRE**).

3. Lab-extension CAI simulates experiments that students already have performed in the laboratory. "Students can obtain an intuitive feeling for ... the effects of changing experimental conditiions, even though an experiment may be too time-consuming or require too much equipment to be repeated."
4. Lab-substitute CAI has been used to simulate experiments that cannot be done in the laboratory because they are too difficult ... or because the necessary equipment is not available ..."

CONSTRUCTING LESSONS

A final word of advice is that the lecturer should carefully consider what modes of useage are appropriate to his context. After an initial period of trying **SOFTWIRE** out in different ways on an informal basis, it is important to work out the curriculum and pedagogic strategy before considering exact use of the computer, so that **SOFTWIRE** is integrated into the curriculum in a pre-planned and consistant manner, rather than piecemeal and with uncertainty -don't let the computer be the tail that wags the dog! [23]

The next chapter suggests some practical uses of **SOFTWIRE**.

REFERENCES

1. Seymour Papert, Mindstorms: Children, Computers and Powerful Ideas, New York: Basic Books: New York (or The Harvester Press Limited: Great Britain), 1980, p. 185.
2. Seymour Papert, 1980, p. 188.
3. R. A. Sparkes, Microcomputers in Science Teaching, Hutchinson, 1981, p. 105.
4. R. H. Batt, "A Piagetian Learning Cycle for Introductory Chemical Kinetics" in ITERATIONS: Computing in The Journal of Chemical Education, J. W. Moore, ed, 1981, p. 79.
5. L. L. Hatfield, "Toward Comprehensive Instructional Computing in Mathematics", V.P. Hansen (1984 yearbook editor) and MJ Zweng (General Editor), Computers in Mathematics Education 1984 yearbook, National Council Of Teachers Of Mathematics (USA), p. 4.
6. L. L. Hatfield, 1984, p. 5.
7. H. L. Dreyfus and S. E. Dreyfus, "Putting Computers in Their Proper Place: Analysis versus Intuition in the Classroom", p. 43.
8. H. L. Dreyfus and S. E. Dreyfus, p 40-62.
9. G. Hubbard, "The Prospect of Respectability", B. Hollinshead and M. Yorke, eds, Perspectives on Academic Gaming and Simulation, Simulation and Games: The Real and Ideal, 1980.
10. Seymour Papert, 1980, p. 5.
12. S. K. Damarin, "TABS-Math: A Courseware Development Project", V.P. Hansen (1984 yearbook editor) and MJ Zweng (General Editor), Computers in Mathematics Education 1984 yearbook, National Council Of Teachers Of Mathematics (USA), p. 68.
13. D.C. Lukens "A compass-and-computer perspective on technology", V.P. Hansen (1984 yearbook editor) and MJ Zweng (General Editor), Computers in Mathematics Education 1984 yearbook, National Council Of Teachers Of Mathematics (USA), p. 18.

CHAPTER 15

SOME POSSIBILITIES AND EXAMPLES OF SOFTWIRE LESSONS

In this Chapter, some of the previous discussion will be concretised through some examples of the use of **SOFTWIRE** to enhance an Electrical Engineering Curriculum. The list is by no means exhaustive, as space is limited. It is intended to be a basis for lecturers to devise their own material.

Fully worked examples will be given, tapering to outlines of possible tutorial sequences, followed by more ideas of possible lessons. Some of the material is addressed to the lecturer **about** the lesson, and some to the student.

15.1 WORKED EXAMPLES

Two examples are given of tutorials loosely based on material which was actually given to students at the University of Cape Town in 1986. They were designed to teach the use of **SOFTWIRE** and engineering concepts at the same time (It may be argued thas this should be done seperately). They always assume that the student has a pen and paper handy, to take down notes and diagrams which will result in a short report on the work done.

A SOFTWIRE EXERCISE IN FOURIER SYNTHESIS

This is a simple exercise to familiarize you with **SOFTWIRE** commands. At the same time it aims to provide an interesting and useful insight into Fourier synthesis -building arbitrary functions of time by adding scaled sinusoids.

STARTUP

Place the program disk in drive A: (the top drive), close the lid and turn the computer on. After a few seconds the heading **SOFTWIRE** will appear on the screen:-

SOFTWIRE

Produced by J. R. Greene and C.D.Geerdts

SOFTWIRE is a teaching aid based on the

simulation of electronic modules

After another few seconds this will be replaced by the prompt message:

Input from keyboard or file (K/F)=>

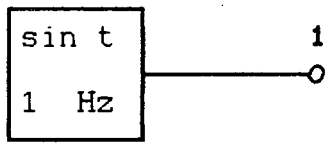
The user has to decide whether to input a system specification from the keyboard or from a file on disk. In this example, the modules are going to be input directly from the keyboard. The correct response in order to specify the keyboard is to type a '**k**' or a '**K**' (**SOFTWIRE** always accepts input as either upper or lower case). Do so now.

Throughout the instruction sections of this tutorial, bold writing will be used to distinguish the input you must type in, as opposed to normal print which will indicate the text already on the screen.

If the '**K**' is successfully entered, **SOFTWIRE** sets up a heading **STRUCTURE** on the screen. We now have to input the net-list representing the structure of the system. Our purpose here initially is to create a sinusoidal generator of 1 Hz. To do this, input this line exactly:-

1 SNW(1)

The '1' is a label for the output node of this module, so we now have:-



Note the single space after the first '1', which is used as a delimiter to separate the different parts of the line. This delimiter is used in all data entry to separate data. Note also that the [Back Space] key may be used to change mistakes made on a line.

SOFTWIRE does not respond until you press the grey button on the right marked "Retrn" (or with a reflexed arrow on some computers), which will subsequently be denoted [Retrn]. Press it now. If you have made a typing error, you will be required to re-type this line. **SOFTWIRE** is now awaiting another line, but since

our specification is complete, type yet another **[Retrn]** on the empty line to indicate termination of the structure entry mode. **SOFTWIRE** then lists your net-list on the right hand side, and your screen should now look like this:-

```
STRUCTURE                                  ¦ PRESENT STRUCTURE
                                           ¦
1 snw(1)                                   ¦   1 SNW(1.000)
                                           ¦
                                           ¦
   Want to change anything(Y/N)?=>         ¦
                                           ¦
```

It is now asking if you wish to make changes. If you have made a mistake (and your entry looks different from the screen shown above, type '**Y**' and enter the line again, otherwise type '**N**'.

SOFTWIRE has now written another heading, **PLOT MODULES**. This is the place to specify the modules whose output we wish to see plotted. Since we have only provided a single module, module '1', we enter (the bold print):-

PLOT MODULES

1[Retrn]

The heading **PLOT RANGE** now tells us to specify the range we want plotted, i.e. the y-axis minimum and maximum. Let's start with a range of [-1,+1]. We must therefore type:-

PLOT RANGE

Min y-axis value ("?" for help)=>**-1[Retrn]**
Max y-axis value ("?" for help)=>**1[Retrn]**

whereupon **SOFTWIRE** gives the heading **TIMING RANGE**. We will look at a 1 second time-span with time increments of 0.01 seconds. This means we must type:-

```
TIMING RANGE

Time Increment("?" for help)=>0.01[Retrn]
  Maximum time("?" for help)=>1[Retrn]
```

With the timing range we have just specified, when the simulation is run, the program will do its calculations at the times t=0.00, t=0.01, t=0.02 etc until t=1.00

This completes the data entry. The screen should now look thus:

```
STRUCTURE                                  | PRESENT STRUCTURE
                                           |
1 snw(1)                                   |   1 SNW(1.000)
                                           |
                                           |
   Want to change anything(Y/N)?=>         |
                                           |
PLOT MODULES                               |
                                           |
1                                          |
                                           |
PLOT RANGE                                 |
                                           |
Min y-axis value ("?" for help)=>-1        |
Max y-axis value ("?" for help)=>1         |
                                           |
TIMING RANGE                               |
                                           |
Time Increment("?" for help)=>0.01         |
  Maximum time("?" for help)=>1            |
```

RUNNING THE SIMULATION

If everything has been typed correctly, a menu will appear at the bottom of the screen. We need only type '**R**' or '**r**' to **R**un the simulation. A new menu appears, asking whether we will use **D**efault values or run **W**ith features. We do not want any extra

features, so we type '**D**'. The screen immediately springs to life with a plot of SNW(1) from t=0 to t=1s.

EDITING

The net-list will now be added to. The menu should have appeared on the bottom line of the screen again. To **E**dit, type '**E**'. A new menu appears, of the different editing options. In order to change the system description, one now types '**S**' (to **E**dit the **S**tructure). This results in the heading **STRUCTURAL CHANGE**. There is also a listing of the present net-list down the right hand column. We now type:-

STRUCTURAL CHANGE	PRESENT STRUCTURE
2 SNW(3) [Retrn]	1 SNW(1)
3 GAI(0.33) 2 [Retrn]	

A second sinusoidal generator, of frequency 3 Hz, has been added. Its output is scaled by a factor of 0.33, by module number three. Pressing another **[Retrn]** terminates the entry mode. If required (i.e. if you have typed in something incorrectly) you may then correct the net-list.

Now **E**dit the **O**utputs (as before):

PLOT MODULES

1 3 [Retrn]

(From now on, it will be assumed that you press the [Retrn] key when required. It will no longer be indicated.)

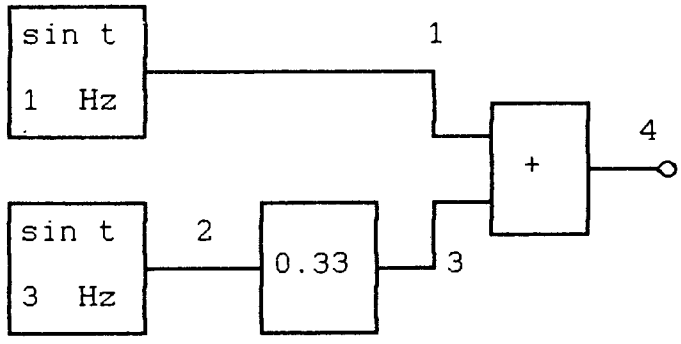
Then press '**R**' and then '**D**', etc to re-run the simulation. There should be a plot of the fundamental and a scaled down third

harmonic superimposed on the screen. By using the scale on the left hand side, one can confirm the amplitudes. The scale is divided into ten big divisions, and the zero mark is indicated. The range is written underneath.

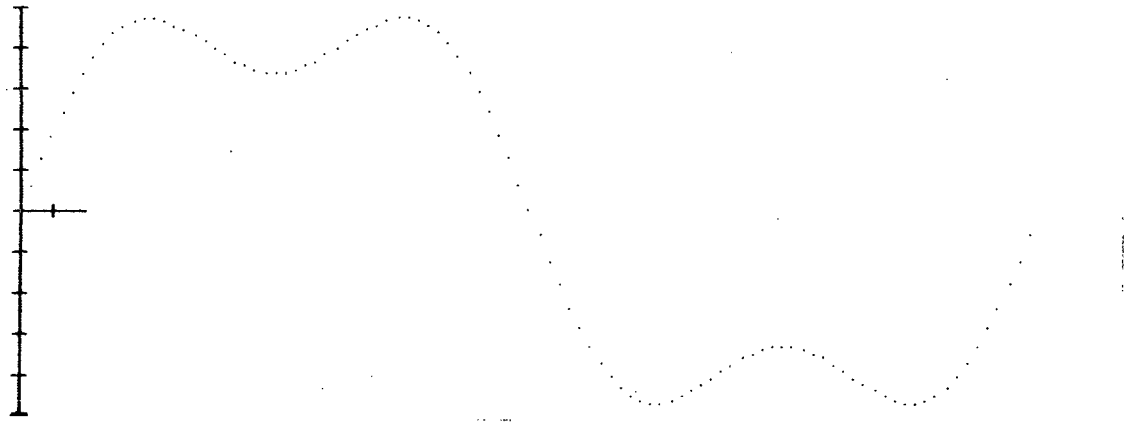
The two required signals are on the screen. Now let's add them! Assuming you are in the correct place in the program (and have the menu on the screen in front of you) press '**E**' again, then '**S**'. The heading **STRUCTURAL CHANGE** should be up, and one can enter the module:-

4 SUM 1 3

Module 4 adds the inputs from nodes 1 and 3. The system described by the present net-list may be represented thus:-



Again **E**dit the **O**utputs to plot node 4 only. Then **R**un the simulation again with the **D**efault option. The screen should look something like this.



This is the first-two-fourier-term approximation to a square-wave.

Now try adding the 5th harmonic (scaled for 1/5 or 0.2) and, if you feel energetic, the 7th (scaled for 1/7 or 0.14). You will, of course, have to use a new SUM module to add in each new term.

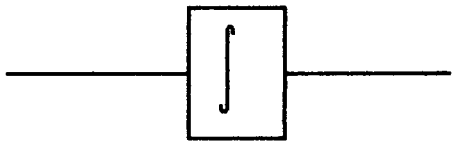
SOFTWIRE TUTORIAL ON INTEGRATORS

PART 1

Integration with respect to time is an operation often required in engineering. A very common example is an odometer for a car, which integrates the car's speedometer reading with respect to time, and gives the distance covered. Also, electrical power is integrated to give electrical consumption meter readings of Kilowatt-Hours. Analogue computers use integrators extensively in simulation and to solve differential equations.

This practical assumes you have been introduced to the program **SOFTWIRE**, and have performed at least very simple operations of editing the net-list, output, range and time parameters. You will now use **SOFTWIRE** to investigate some of the properties and behaviour of simple systems involving integration with respect to time.

The basic integrator module can be represented as follows:



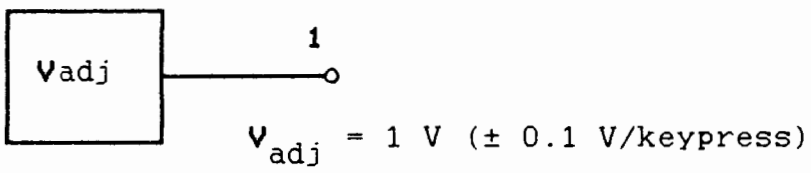
STARTUP

Remember to place the program disk in drive A:, and turn the computer on. As before, the signature heading of **SOFTWIRE** will appear on the screen followed by the prompt on the screen asking whether input is from the keyboard or disk. Initially this will be done via the keyboard, and this is specified by typing a '**k**' or '**K**'.

SOFTWIRE has set up a heading **STRUCTURE** on the screen. We now have to input the structure net-list. Input this line exactly:-

1 VIN(0 0.1)

VIN represents a signal generator, which spontaneously generates a voltage level which may be adjusted by the user during the simulation. (Its operation will become evident later). The values in brackets are parameters relating to the generator. The first parameter refers to the initial voltage, in this case 0V, and the second refers to the magnitude of voltage increments, in this case 0.1V. When the user changes the voltage during the simulation, by pressing the '+' or '-' keys on the extreme right of the keyboard, the step change will be 0.1V whenever one of the keys is pressed. The '1' is a label for the output node of this module, so we now have:-



Remember that **SOFTWIRE** does not respond until you press the [Retrn] key. Remember also, that **SOFTWIRE** is now awaiting another structure line, and you must type another [Retrn] on the empty line to indicate that the system specification is complete.

SOFTWIRE then writes the heading, **PLOT MODULES**. This is to specify the modules whose output we wish to see plotted. Since we have only provided module '1', we enter:-

1

The heading **PLOT RANGE** now tells us to specify the y-axis range. Start with [-1,+1], typing:-

Min y-axis value ("?" for help)=>**-1[Retrn]**
Max y-axis value ("?" for help)=>**1[Retrn]**

whereupon **SOFTWIRE** gives the heading **TIMING RANGE**. We will look at a 1 second time-span with time increments of 0.2 seconds. This means we must type:-

Time Increment("?" for help)=>**0.2[Retrn]**
Maximum time("?" for help)=>**1[Retrn]**

Which completes the data entry.

RUNNING THE SIMULATION

If everything has been typed correctly, a menu will appear at the bottom of the screen. We need only type '**R**' to run the simulation. A new menu appears, asking whether we will use **D**efault values or run **W**ith features. We **do** want a feature - namely continuous simulation, so we type '**W**'. The features are:

Continuous: the simulation continues indefinitely, until the user specifies the end by pressing the space-bar.
Real-time: the computer performs the iterations at the real time increments specified.
Output direction: this may be to the screen, a file or the printer (or any combination).

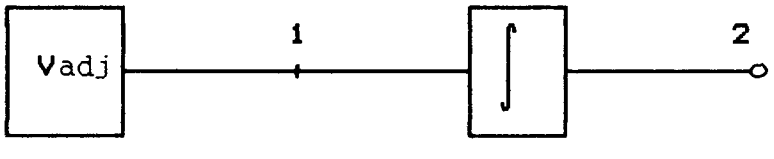
In this example we want to use the continuous mode, and output to the screen. Answer '**Y**' for these to options, when prompted, and

'**N**' for all the others. When this is complete, the screen springs to life with a continuous time plot of VIN. This is rather boring, since all we are seeing is a 0 V line. However, by pressing the '+' and '-' keys on the extreme right of the keyboard (to the right of the numeric keypad -the white keys on the main keyboard do not work here), one may alter the voltage by the increment of 0.1V we specified: (the '+' key is for incrementing, the '-' for decrementing). To end the simulation, just press the space bar.

It's high time the integrator was introduced, so an addition to the system description is necessary, done from the editor. The menu should have appeared at the bottom of the screen again. Type '**E**', then '**S**' to **E**dit the **S**tructure of the system. Under the heading **STRUCTURAL CHANGE**, you should now type:-

2 INT(0) 1

to indicate that an integrator is required which is identified as module 2, and gets its input voltage from the output of module 1, which is VIN. The zero in brackets is the initial value of the integrator -the integration constant. Press [Retrn] again to terminate the net-list entry. The net-list now describes this system:-



To see the output at module 2 as well as 1, **E**dit, the **O**utput, and under the heading **PLOT MODULES**, specify modules 1 and 2. Now run the simulation again to check whether the integrator is working.

THE USER INTERFACE

If you are using a computer which has a **User Interface** attached, then there is no need to use the VIN module any more, except when requiring great precision. Rather use one of the potential dividers on the interface's panel. To specify this, edit the structure by adding the line:

1 PT1(-1 1)

This replaces the VIN module with input from the potential divider. The parameters specify the range [-1 1] over which the potential dividers' outputs are scaled. Run the simulation as before, but turn the left potential-divider knob to vary the input to the integrator.

EXPERIMENTATION

Spend some time varying the input and examining the subsequent outputs. Express your observations in your write-up with at least three sentences similar in form to "When the input to the integrator ... the output then ...".

CONSTRUCTING AN OSCILLATOR

The integrator can be used to generate a triangular curve. Its output rises at a rate specified by VIN. When its output voltage exceeds a certain threshold value, we must somehow invert the voltage into the integrator, so that it is negative and the slope becomes negative. When the output then goes below a certain value, we must invert the input again, so that the slope is again positive. To do this, one uses a module called a "Threshold Detector". Its operation will become apparent when we use it,

and will be described after the system is set up. This will be done now, by making another structural change.

Assuming you are in the correct place in the program (and have the menu on the screen) press '**E**' again, then '**S**'. The heading **STRUCTURAL CHANGE** should be up, and one can enter the net-list:-

1 INT(0) 3[Retrn]
2 THH(0.9 1) 1[Retrn]
3 GAI(-1) 2[Retrn]
[Retrn]

Line 2 adds a threshold detector module (THH), which gets its input from the integrator, which is module number 1. The 0.9 in brackets is the threshold voltage, and the 1 in brackets is the initial output voltage, which may be 1 or -1V. Module 3 is a gain (multiplier) module, which multiplies its input by the parameter in brackets (in this case -1). Sketch what you think the system looks like.

Edit the **O**utputs to include all three modules on the screen.

Run the simulation again, this time using the default option i.e **R**un **D**efault. One can see the triangle and square outputs. From the screen, work out the frequency of the outputs. Before continuing, think of what could be done to alter the frequency.

In the example, the parameter used for the threshold was 0.9V. This can be changed by editing the parameters, pressing **E**dit, and **P**arameters. We want to change the threshold value to say 0.45, so we type:-

1 (0.45 1)[Retrn]
[Retrn]

Run the simulation, and check the frequency. It is possible to change the time scale as well. To do this **E**dit the **T**iming. Then type in a time increment and total time, when prompted. Choose various values yourself, and run the simulation. What values would you suggest to best depict the oscillator, but also to run reasonably fast and have adequate resolution. Draw out the output from these modules for yourself, and attempt to understand why the oscillator is working.

INTEGRATORS -PART II

Integrators obviously integrate any input curve. A reasonably complex input is a cosinusoid. First year maths enables one to predict what the output will be if this is integrated with respect to time. Type in '**N**' for a **N**ew Structure. Then specify **K**eyboard mode and enter:-

1 CSW(4)[Retrn]
2 INT(0) 1[Retrn]
[Retrn]

CSW is the mnemonic describing the module which will generate a cosinusoid the frequency which we specify in brackets. In this case we have specified 4 Hz. This is the parameter of the cosine function generator module. Sketch the system we now have.

We want to look at the Output from modules '1' and '2'. We want the plot range of ±1, and a the timing range which will enable us to look at a 1 second time-span with time increments of 0.002 seconds. Set this up.

Now **R**un the **S**imulation using **D**efault.

FROM THE SCREEN PLOT, YOU MAY CHECK IF THE FREQUENCY OF THE SIGNAL IS ACTUALLY 4 HZ. Remember that the time period we specified was 1 second.

In order to change the frequency of the cosine wave, one now **E**dits the **P**arameter of the CSW module, which in this case specifies the frequency of the wave. When the heading **PARAMETERS**

appears, to change the frequency to, say, 10 Hz, type:-

1 (10)[Retrn]

to indicate that the parameter of module 1 is now 10. One must now press [Retrn] again to indicate that no more parameters are to be entered. Now re-run the simulation and note the new frequency. Now **E**dit the **T**iming to give a time of 0.0005/0.5s, and run the simulation again.

Now spend a while trying runs with different frequencies and time ranges. In particular, give a qualitative comment in the write-up on how the gain of the integrator is related to input frequency. Even more desirable is to try to find an expression which relates gain to input frequency. [Hint: At least find the frequency which provides unity gain. Express the value in terms of radians].

Before you continue, set the frequency at 0.1592 Hz and the timing range at 0.03/15.

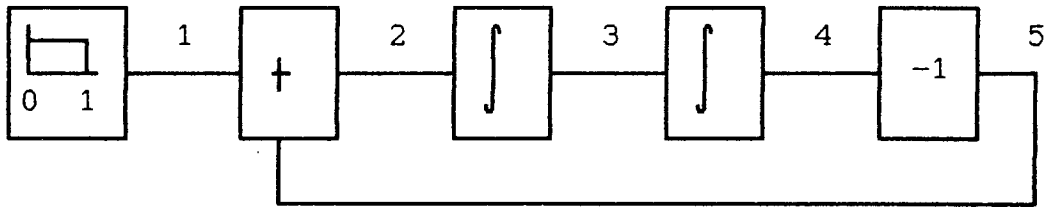
ANOTHER OSCILLATOR

An integrator can be thought of as a device which produces a 90° phase shift, (note that there is a 90° phase shift between a sine and cosine function). If one has two integrators in a row (cascaded), there is a 180° phase shift. If one then inverts the output (multiplies by -1), one produces a further 180° phase shift, since there is a 180° phase difference between a positive sine or cosine value and its corresponding negative value. Thus a circuit with two integrators and an invertor in series, produces a 360° phase shift. A sinusoidal input will thus produce a sinusoidal output in phase with the input. If this input is then fed back to the input, the system will continue to oscillate. In fact, any disturbance at the input will cause the system

to oscillate, since all pulses have a sinusoidal component (as will be discovered later in your course). To illustrate this, we set up the required system. Now **E**dit the **S**tructure by typing:-

```
1 PLS(0 1)
2 SUM 1 5
3 INT(0) 2
4 INT(0) 3
5 GAI(-1) 4
[Retrn]
```

The following circuit is now produced:-



Module 1 is a pulse generator. The output is zero at all times except in the interval specified in parentheses, when it is 1V. The time interval specified here is between 0s and 1s. Thus we start with an input pulse of 1V for 1s. The next module adds the pulse and the output from the system to the input of the first integrator. The two integrators then follow, and then a module which multiplies its input by the number given in parentheses. In this case it multiplies by -1 (GAI stands for 'gain'). We have thus set up the oscillator discussed earlier. The next step is to try it out.

Before we can do this, however, a new time range must be specified, as well as the outputs required. Set a time of 0.04/5s, and plot outputs to modules 2,3 and 4.

Now run the simulation under default mode. Also run it in continuous mode to see how long it continues to oscillate for. What frequency does it oscillate at? If you are curious enough, try to explain why? [Hint: The gain around the system must be unity for sustained oscillation.]

PART III

The first two parts of this tutorial have been given as an example, and the actual dialogue of the third part will be left to the tutor to complete. Since this lesson is just an example, it has squeezed in a number of different concepts. An actual lesson (as with the first two parts) would not be too full in new concepts for the student to grapple with at once.

This part will be about idealised and real-life integrators, and a few new programming features will be introduced.

The sequence will start with the description of the Euler Integrator, and an introduction to the Fourth-Order Adam Integrator. Their respective performances will be compared. In this way, the concept of numerical modelling will be introduced.

The deviation from the ideal will then be broached. An integrator will be introduced which inverts, and saturates. The next integrator is subject to drift, and a small amount of noise. Finally, a hardware integrator is used.

The new programming features introduced are the use of output options of numerical tables, and phase portraits.

The intended sequence is as follows:

1. A Short Introduction.

The text could be something such as the following:

"This part of the tutorial discusses idealised and real-life integrators. It also introduces you to the concept of tabular and

phase portrait output options. These concepts will become clearer as you progress.

The model we have been using for the integrator so far is both a simplified and an idealised version. The formula used to implement the integrator in software has been simplified to speed up the simulation. This is at the expense of accuracy. However, the accuracy can be improved by reducing the time intervals, and the accuracy also depends on the complexity of the input signal."

2. Running The First Test / Introducing The File Input Mode

The first test merely integrates a straight line, and compares the error of the two integrators. The error is easily found by comparison with a parabola. A sample text would be as follows.

"To illustrate this, we will introduce a more accurate integrator, and compare accuracy. This will be done first with a straight line as input to the integrator. From the main menu, type '**N**' for **N**ew. This clears the previous net-list, and returns the user to the beginning of the program. This time, input will be from a file, so when the prompt asks, type '**F**'. The prompt then asks for the name of the file. Type "LINETEST.DAT". If the correct program disk is in drive A:, the program will load, and the screen will behave as though the data had been typed in -only considerably faster! If the wrong disk is in, an error message will indicate that the file was not found, and the correct disk should then be inserted."

The file INTTEST would contain a net-list for a system to test the two integrators. An example of such a net-list, with a print-out of the simulation, can be found in Chapter 11, in the section which compares integrators. However, this example is of the Second Test. The First Test would use module 0 (the time module)

as linear input to the integrators, and then would square this value and divide by two as a reference.

i) Run the simulation in default mode, and observe the input function, output of the integrators, and error.

ii) Get the student to draw the block diagram to see what is happening. Provide a brief explanation.

iii) Introduce the tabular option, and get numerical results of the error.

Obviously there should be almost no error, since one is dealing with a first-order input function.

3. Running The Second Test

Do the same as for the previous step, but with the file "SINETEST", then discuss the error, regarding a sinusoid as an (infinitely) high order function. This net-list is the one found in Chapter 11, and would work without adaptation.

4. Introducing Less Idealised Implementations

Introduce module NT2, which has an inverted output, and saturates at ±1V. Use a net-list which demonstrates this. Then replace this with NT3, which is also subject to drift, and slight noise. A good system to use for this is the dual-integrator oscillator. The idealised (INT) version always behaves perfectly, whereas the NT3 model either saturates or dies.

14.2. SKELETON EXAMPLES

The following tutorials are not written out, but merely indicate the potential and possibilities with **SOFTWIRE**.

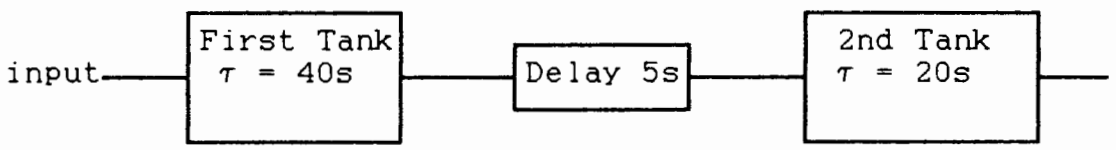
CONTROL THEORY

A TWO-TANK SYTEM AND CONTROLLER

This tutorial would teach concepts about PID control tuning. It also demonstrates instability. Other concepts are pointed out, such as response to a ramp excitation, and the effect of changing the parameters of the system being controlled. It is important too, because practising engineers often do simulations of control systems before implementing them. This tutorial thus prepares students to do what they actually will be doing later.

1. The Setup

The aim is to control the level of a tank of liquid in a chemical process, by varying the input flow. The system is represented as follows:



A PID controller is set up in **SOFTWIRE** to run this. The net-list

for this system is as follows:

```
1 CON(1)
2 CTL(0) 1 22
3 GAI(1) 2
4 INT(0) 2
5 GAI(0) 4
6 DIF 2
7 GAI(0) 6
8 SUM 5 7
9 SUM 3 8
20 LPF(40.0) 9
21 DLY(5.0) 20
22 LPF(20.0) 21
```

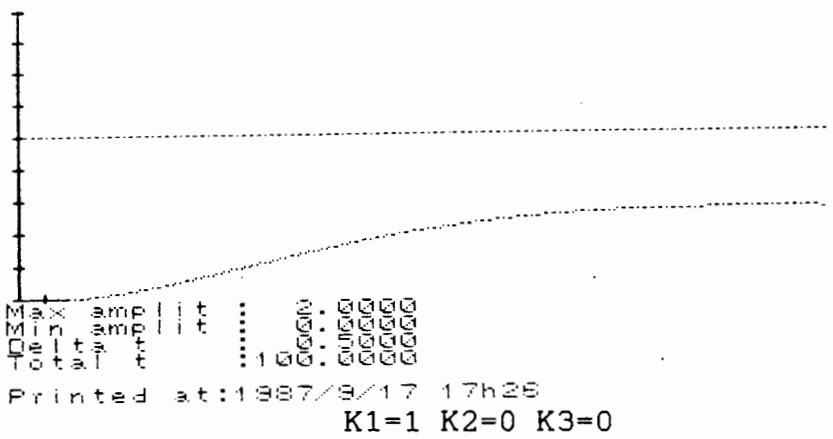
Module 1 gives the required output level. Modules 3,5 and 7 give the gains of the proportional, integral and differential controllers respectively (K1, K3 and K2).

The most useful outputs to examine are 1 and 22, giving the input and output levels respectively.

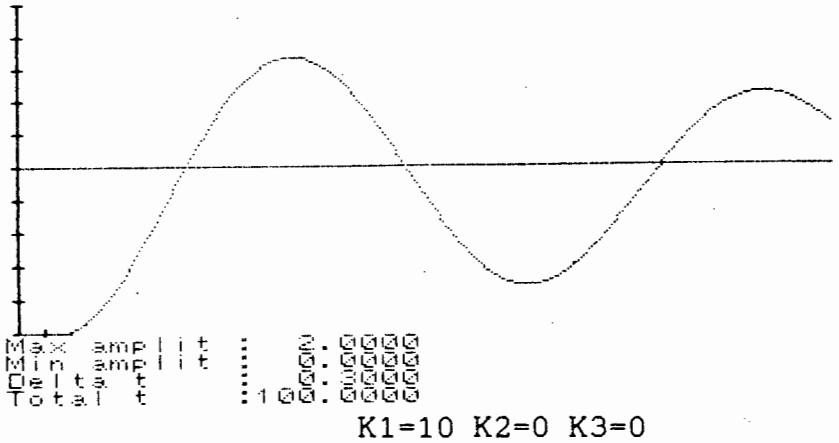
The output range for the y-axis is best set from 0.0 to 2.0, to allow for overshoot. Time increments of 0.3 are sufficient, and a maximum of 100.0.

2. Varying K1

Start with a low K1, say K1=1. Show how slow the controller response is, and the large final offset-error.

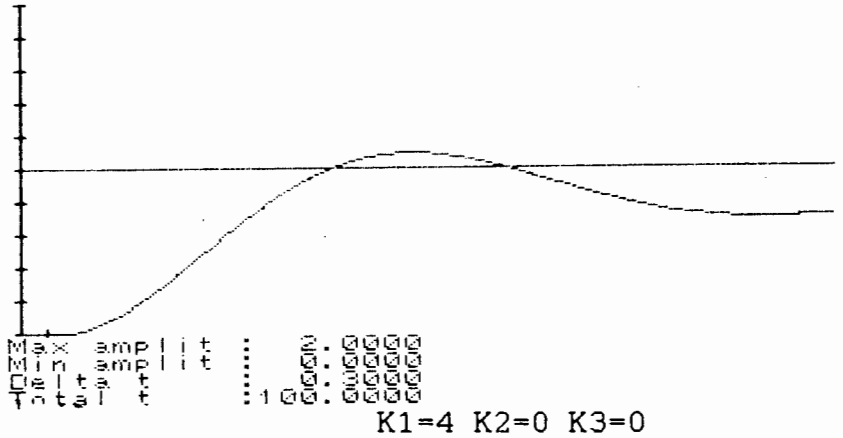


Try a large K1, say K1=10, and show the large overshoot.



K1=10 K2=0 K3=0

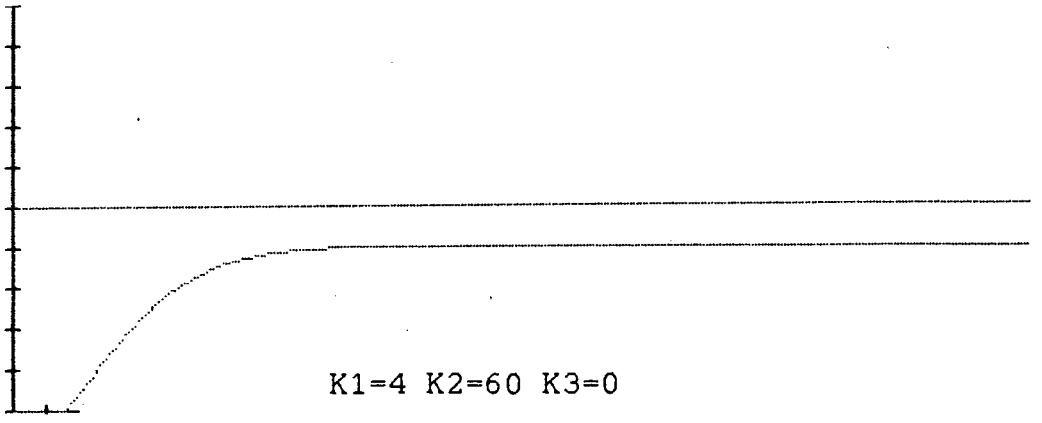
Let the student find an optimal value of K1 which will give a small overshoot (say 20%), but a reasonable rise-time and slight final offset.



K1=4 K2=0 K3=0

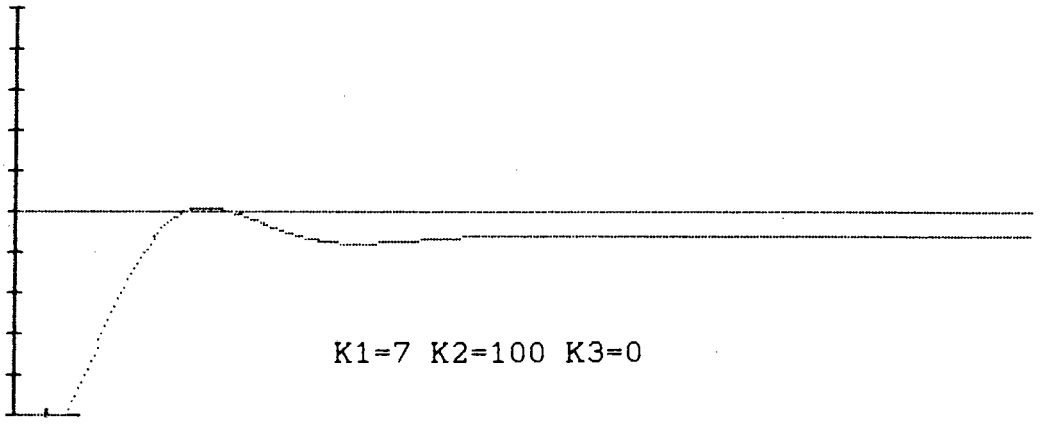
3. Varying K2

Use the gain from K2 to offset the overshoot. Comment on the effect this has on overshoot.



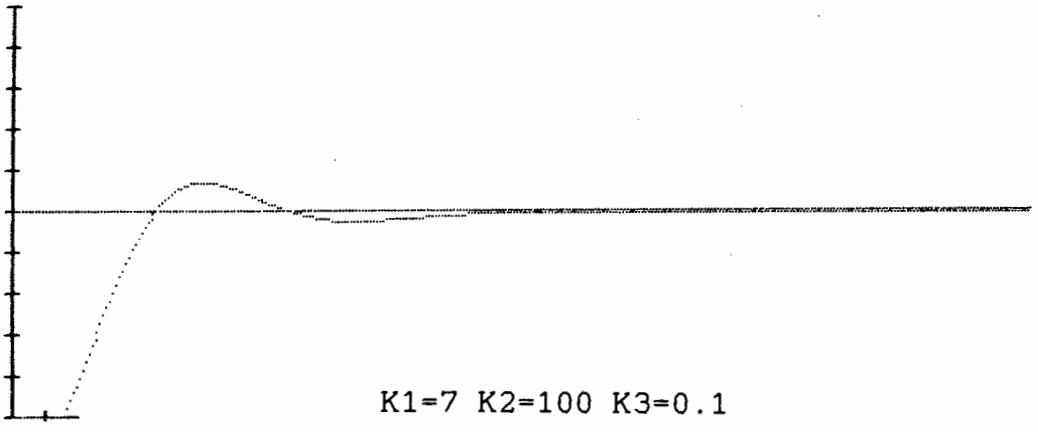
4. K1, K2 iterations

Increase K1 again, and again reduce overshoot. Do some iterations until you are satisfied that the values are optimal. Discuss the concept of an "optimal" integrator, regarding tradeoffs of overshoot, speed, stability, final off-set etc.



5. Varying K3

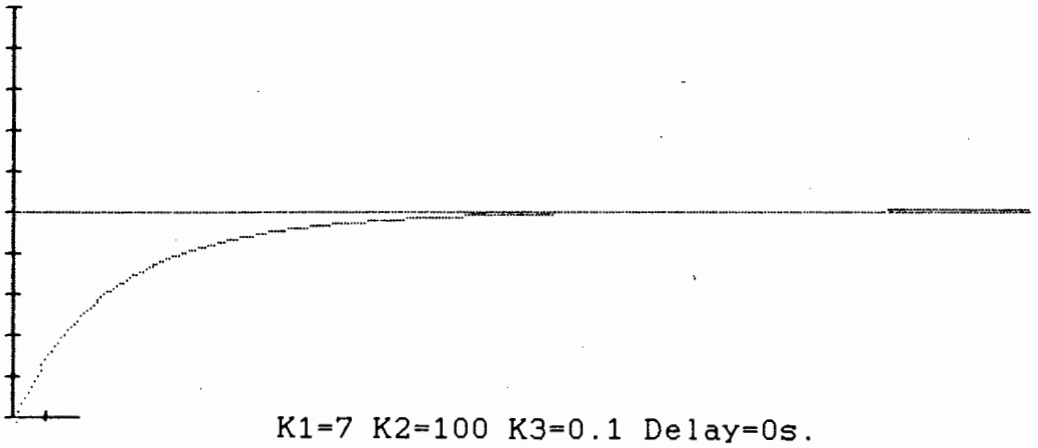
Add a small contribution from K3 to remove the final offset. If too large, there is oscillation. If too small, it takes too long. (The student can be left to discover that).



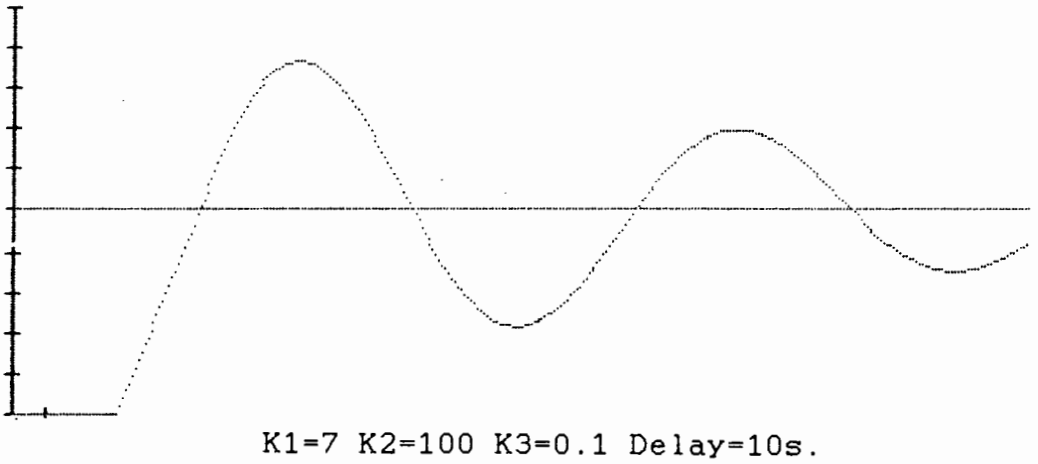
K1=7 K2=100 K3=0.1

6. Varying System Parameters

Vary the delay from 0 to 10s, examining the effect this has on the controller. Also vary the time constants of the two tanks.



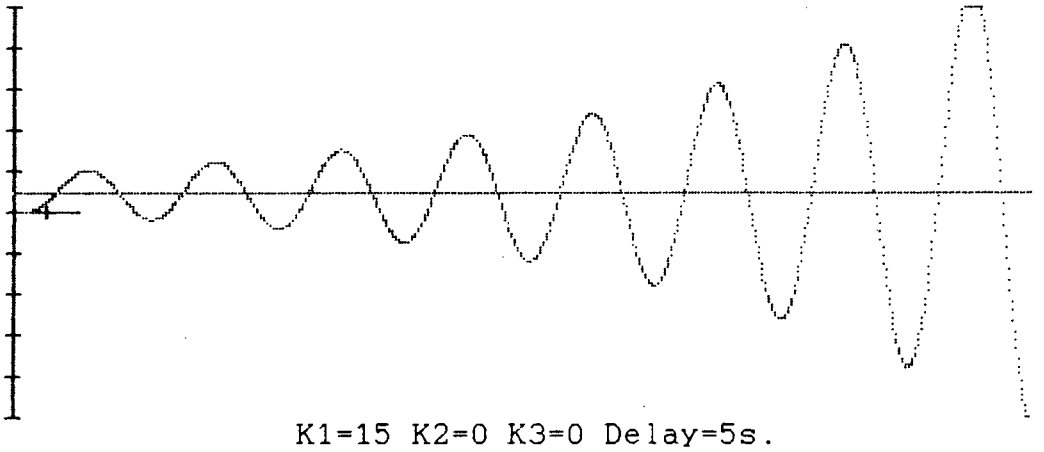
K1=7 K2=100 K3=0.1 Delay=0s.



K1=7 K2=100 K3=0.1 Delay=10s.

7. System Instability

Try values of K1=15, K2=0 and K3=0. Change the time to 0.4/400s, and the range to ±10. Note that the system can become unstable for a constant input.



K1=15 K2=0 K3=0 Delay=5s.

8. Higher Order Controller Requirements

Try a ramp input (replace "1 CON(1)" with "1 GAI(1) 0"). Comment on the controller's ability to follow this, and any offset error.

9. A Real System

A very satisfactory ending to this practical is to have a physical two-tank system, with its motors connected to the outputs of the MULTI-PURPOSE ANALOGUE INPUT / OUTPUT CONNECTOR supplied with the **User Interface**, and the inputs connected to level detectors. If the model used was accurate, the controller already devised should be adequate.

R.M.S. and POWER

Usually the subject of **RMS** is introduced by integrating $\sin^2 t$ symbolically and thus finding the average. The experience of the author is that students are often boggled by the complexity of the mathematics -especially of the integration- and so miss the main point of what **RMS** represents. If demonstrated, the computer does the mathematics, and the students merely observe.

This tutorial can have a problem-solving component: one can ask the students to devise their own net-lists to produce RMS.

This tutorial uses as the basis a net-list which squares and takes an average. This is in fact the mean-square, not the root-mean-square. There is no need to take the square-root, because we are only interested in power, which is the square of the R.M.S. Voltage.

The net-list is as follows.

```
2 MUL 1 1
3 INT(0) 2
4 QUO 3 0
```

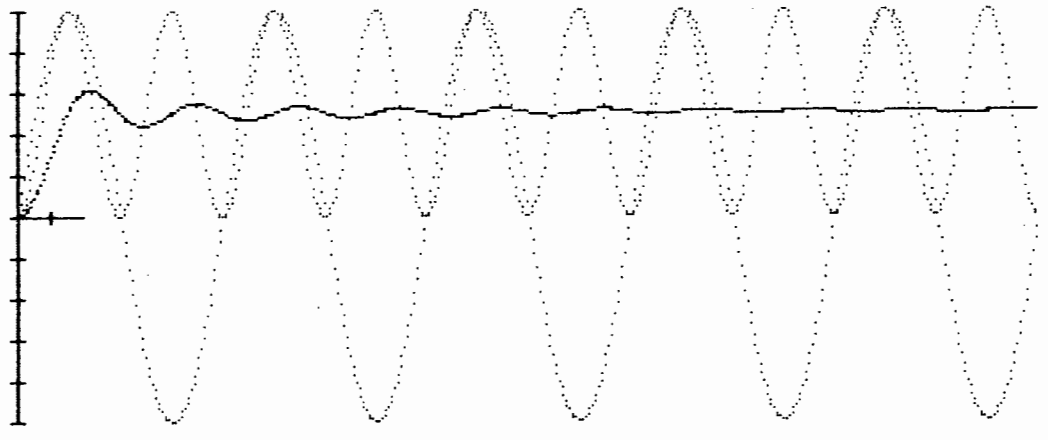
This circuit always gives an undefined output at module 4, because of the initial (divide-by-zero) error. However, **SOFTWIRE** takes care of that, so it is only necessary to point this out to enquiring students.

Module 2 does the squaring, and 3 and 4 do averaging. Module 1 is the module whose output is under test.

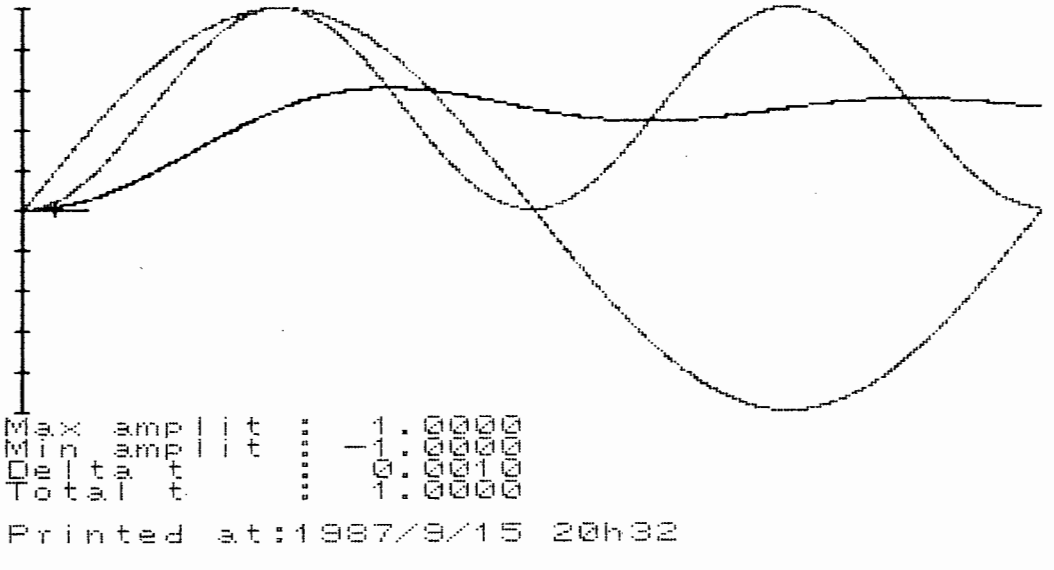
By inserting:

```
1 SNW(1)
```

and using time of 0.01/5s and a range of ±1, one has a graphic display of what is normally the introductory lecture material in this field. The RMS line can be seen to equal 0.5 after each complete cycle.



The RMS curve's oscillations become smaller and smaller as the accumulative total increases. Interestingly, this closely resembles the transient response of an RMS Meter. The display can also be expanded to show only two cycles.



For disbelieving students, a tabled output gives the exact figures.

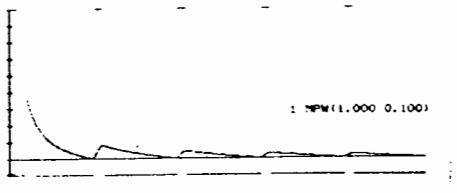
Calculation of RMS

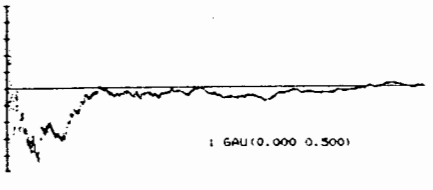
Printed at:1987/9/15 20h48
Max amplit : 1.0000
Min amplit : -1.0000
Delta t : 0.0500
Total time : 1.0000

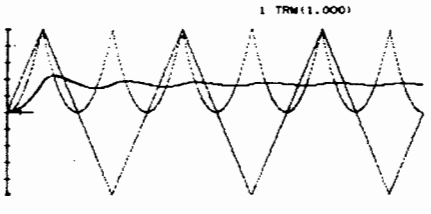
TIME	op 1	op 2	op 6
0.000	0.000	0.000	0.000
0.050	0.309	0.095	0.095
0.100	0.588	0.345	0.220
0.150	0.809	0.654	0.365
0.200	0.951	0.904	0.500
0.250	1.000	1.000	0.600
0.300	0.951	0.905	0.651
0.350	0.809	0.655	0.651
0.400	0.588	0.346	0.613
0.450	0.309	0.096	0.556
0.500	0.000	0.000	0.500
0.550	-0.309	0.095	0.463
0.600	-0.588	0.345	0.453
0.650	-0.809	0.654	0.469
0.700	-0.951	0.904	0.500
0.750	-1.000	1.000	0.533
0.800	-0.951	0.905	0.557
0.850	-0.809	0.655	0.562
0.900	-0.588	0.346	0.550
0.950	-0.309	0.096	0.526
1.000	-0.000	0.000	0.500

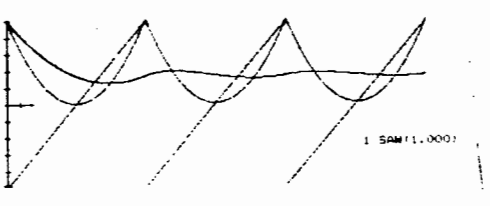
For interest, a number of different possibilities are given after the next paragraph. They show a sawtooth and triangular input, and a square function. This is replaced by square pulses of different duty cycles, where it is shown that the power for 1V input is given exactly by the duty cycle.

A very interesting example at the end is gaussian noise. Students in signal processing courses are always told that "noise power" is given by σ^2, but this is hard to accept intuitively. Here is given a graphic example. With σ=0.5, the RMS line always tends to 0.25.









SIMPLE LISSAJOUS

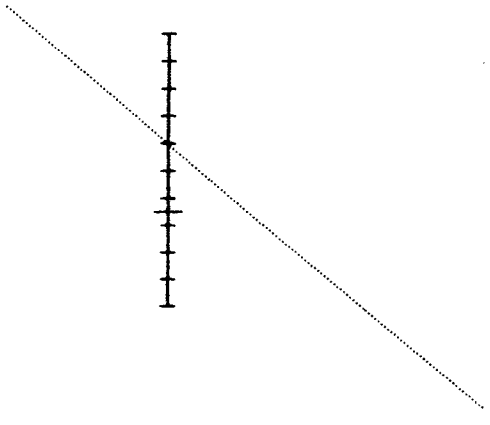
SOFTWIRE has a mode similar to an oscilloscope "y vs x" mode, but different in that points plotted on the screen do not disappear. This enables it to be used to complement a normal oscilloscope laboratory (it is not recommended that it replace such a laboratory, since this is an important area where 'hands on' is required).

1. Straight Line Lissajous

The net-list

```
1 SNW(1)
2 SNW(1)
```

produces a straight line on a graph in oscilloscope mode. Use a time of 0.002/1s and a range of ±1.



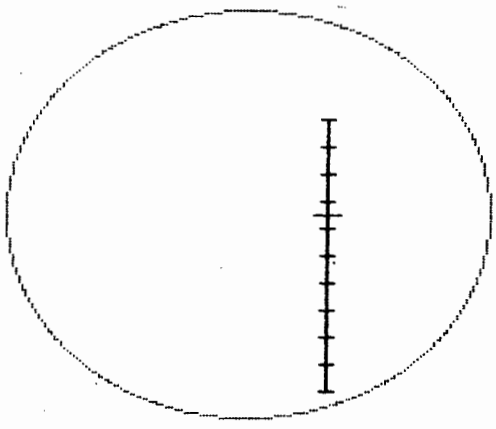
Get the students to predict this, and analyse it. One could say:

y = sin t ; x = sin t therefore y = x (straight line)

Next, replace module 2 with

```
2 CSW(1)
```

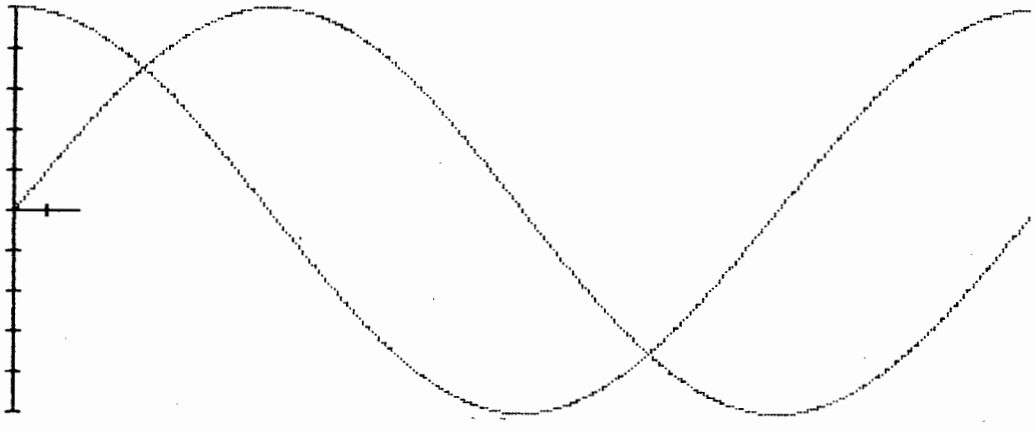
And do another plot. The following output should appear:



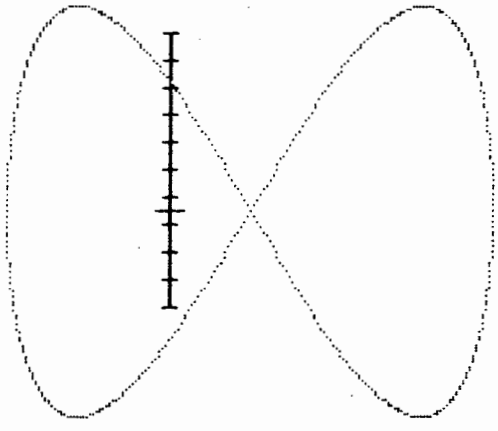
Again this should be analysed. One could say:

$y = \cos t$; $x = \sin t$ therefore $x^2+y^2 = \sin^2 t + \cos^2 t = 1$
which is the equation for a circle.

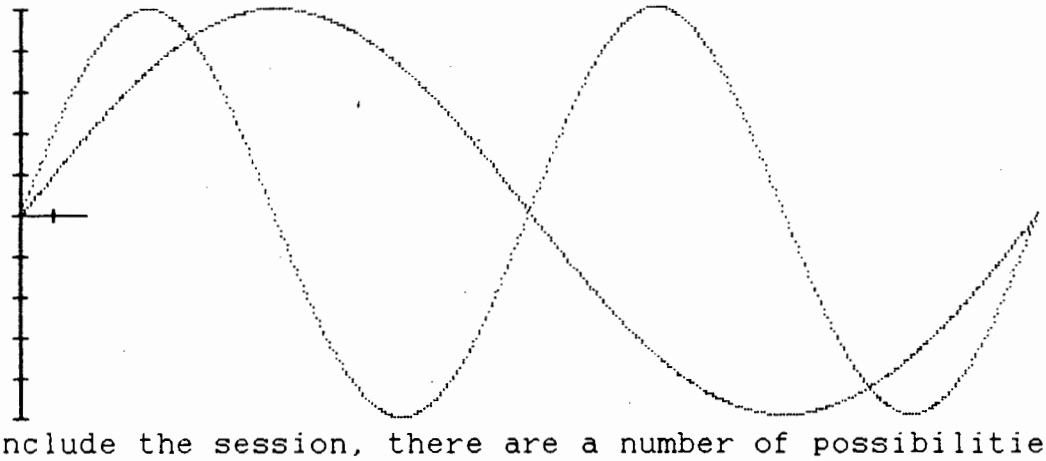
The constant phase difference could be highlighted by plotting the two outputs against time.



One could then produce the following output:



and ask the students to input the correct net-list to obtain it. They could also attempt to analyse what is potentially a very complex curve, by explaining it in terms of a straight line becoming a circle, then back to a straight line (this time y = -x), then back to a circle, as the phase differences move from 0° to 90°, then to 180° and so on. A plot of both curves against time is helpful here.

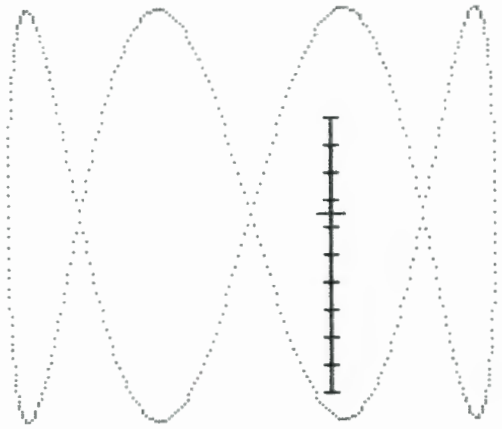


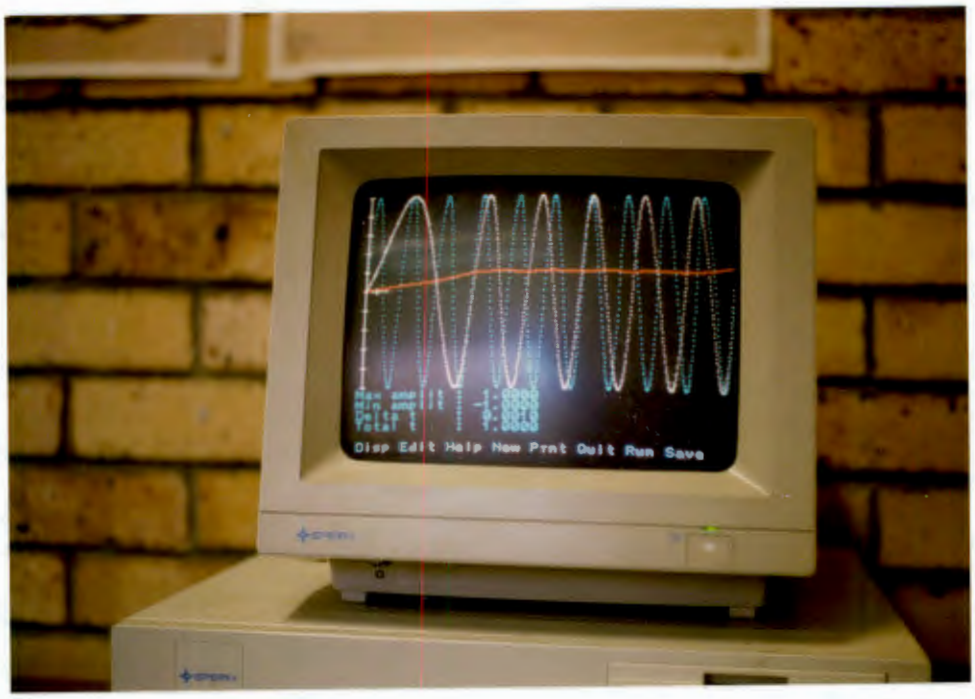
To conclude the session, there are a number of possibilities, but further work is best done on the actual oscilloscope. However, one could leave with one more net-list:

```
1 SNW(1)
2 SNW(4)
```

The student could examine this output and attempt to find a

relationship between the harmonic number and the lissajous plot.





A PHASE LOCK LOOP "LOCKING"

THE LOW-PASS FILTER

There are many aspects to the low-pass filter which can be explored with **SOFTWIRE**.

One could start by having the student build a low pass gaussian (RC) filter on verocard, to slot into the **User Interface** (although one has already been provided with the interface). Have a time constant of 10s. If its inputs and outputs were both connected to analogue channel 1, one could then input the net-list:

```
2 DA1(1) 1
3 AD1(1)
```

Module 1 would be whatever input was intended. One could try a step excitation (using the PLS module), a sinusoid excitation, a user controlled input from the **User Interface** pott. (PT1), and any other functions given on **SOFTWIRE**.

The above exercises are obviously real-time operations. However, real-time mode is not always necessary, unless measurements are to be taken.

In order to introduce digital filtering, one could compare the above results with the **SOFTWIRE** module "LPF", also given τ=10s (here real-time mode must be used if both filters' outputs are compared on the same plot). When comparisons have been made, one could discuss the concepts of **finite impulse response** and **infinite impulse response**.

Students could then implement their own software filters, using **SOFTWIRE** blocks. Some possibilities are (overleaf):

1. A recursive filter, using the same implementation as the present LPF module.

2. A filter using weighted delay elements. See how many delay elements are required to obtain 5% accuracy, then 1% accuracy.

3. A filter using state-variables (and integrators).

4. An implementation of a block-diagram (or signal flow diagram) realisation of a filter (using voltage and current relationships).

The different implementations are important, because they emphasize the number of different approaches to the same problem. They also verify that the theory taught does actually work!

As a final step, one can use the Veroboard device for a completely different application -a Capacitance measuring device. With the known resistor, various capacitors can be tried. They are either excited with a step voltage, and the rise time used to calculate τ, or an oscillator is made, and its frequency used. The Verocard supplied with the interface has a spring-loaded slot for changing capacitors.

THE "MODULE GAME"

This is suggested as a slight deviation from standard engineering academic activities. It also creates motivation through both class and personal competition.

Students are given a limited number of module functions to use, and they must create as many useful circuit configurations as possible from this sub-set. This is analogous to the "word game" found in many newspapers.

This was tried with a set of five functions, namely MUL, SUM, PT1 (or VIN), INT and GAI. With these, one may produce various oscillators, including a voltage-controlled one, a squaring module (in fact, any integral positive index may be realised), a subtractor, amplitude modulation, frequency and phase modulation, a differential equation solver, and many others.

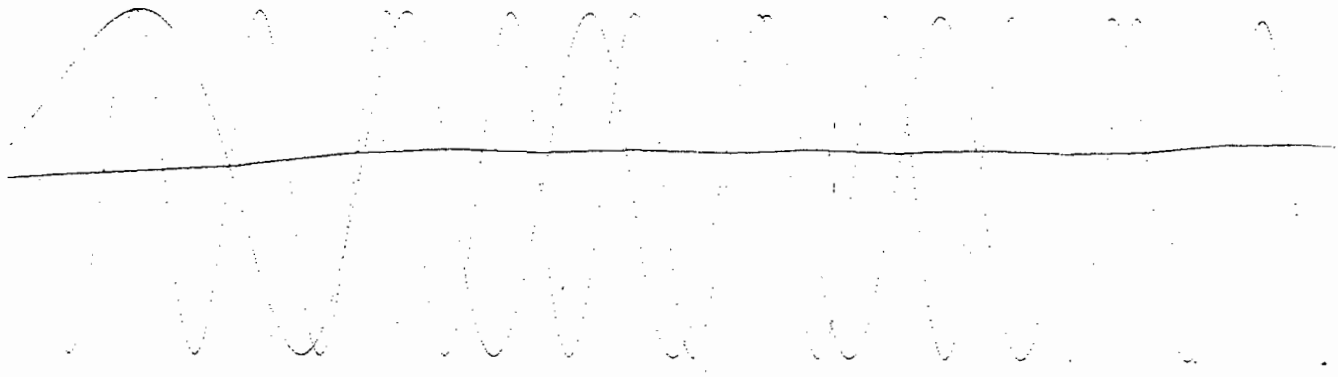
PHASE-LOCKED LOOP

Some of the concepts involved in the PLL are difficult for beginners to grasp. One of these is lock-in. A computer demonstration of this makes the concepts clearer, as students can see the lock-in as though it was in slow motion. One can also monitor the output of different parts of the system simultaneously. Another concept which becomes clear is distortion that results when a simple gaussian filter is used. The PLL does lock in but there is slight oscillation in the phase difference. This is

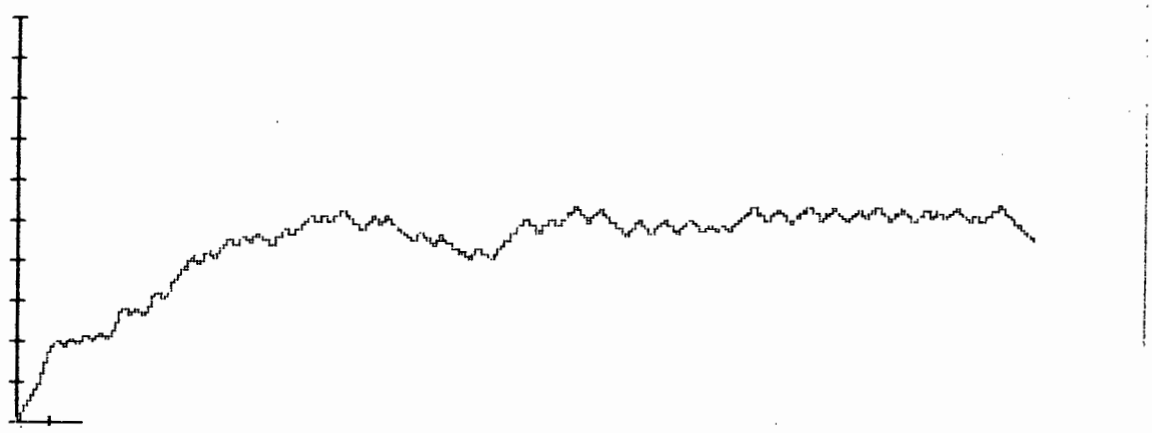
easy to show. The net-list:

```
1 SNW(10)
2 DET 2 4
3 LPF(1) 2
4 VCO(1 30) 3
```

Was used to produce this output:



Also, in a very time-compressed plot of the output of the filter, the phase-oscillations can be seen.



TRANSIENT CAPTURE

The MULTI-PURPOSE ANALOGUE INPUT OUTPUT CONNECTOR supplied with the **User Interface**, can be used to input and output data to various devices:

1. **XY-Plotters**. This can be to output data (since it is far faster than a printer for a hard copy). It can also be used in a measurements course, to determine the frequency and step response of the plotter. This is a neat practical, since the output is in hard copy and correctly scaled.

2. **Oscilloscopes**. The computer can output waveforms of arbitrary complexity. Unfortunately, this application is limited by the low frequency (up to about 100 Hz maximum) of operation of the computer, which does not produce good pictures on most 'scopes. There is more potential when an 8087 card is used.

3. **Signal Generators.** If these are used as inputs to the computer, they ease the burden of calculation considerably, especially with sinusoids. They have useful application, for example, in a tutorial demonstrating the Nyquist sampling rate. The computer is used to sample the signal at different rates. The samples are then put through a low-pass filter. As the Nyquist frequency approaches, aliasing can be seen.

4. **Transient Outputs.** One can use **SOFTWIRE** in a practical where slow transients are to be captured. This could be in capacitor discharge, or with starting of electrical motors, where the transient may be up to a few seconds (here, one could measure current, voltage, speed etc). It could also be used to monitor the step response of control systems. The two analogue inputs on the connector could (for example) be connected to potential dividers at the input knob and output motor of a servo-system.

15.2.1. FURTHER POSSIBILITIES

A number of alternatives exist with other ideas. These are described briefly.

NAME: RECTIFICATION

DESCRIPTION: Use the "DIO" (diode) module to rectify different signals and observe the result.

POSSIBLE MODES: Best done as a demonstration during lectures, when rectification is first dealt with.

NAME: DAMPING

DESCRIPTION: Implement a second-order differential equation solution, and show the effect and significance of the damping factor.

POSSIBLE MODES: This is a good demonstration when network theory is being introduced. It has enough substance to be a short practical exercise at the start of a laboratory involving a second-order filter design and implementation.

NAME: AMPLITUDE MODULATION

DESCRIPTION: Show what results when a low frequency signal amplitude-modulates a high frequency one.

POSSIBLE MODES: This is best done as a successive demonstration - during a lecture the lecturer can discuss different aspects of modulation, and illustrate them with a quick example.

NAME: FREQUENCY MODULATION

DESCRIPTION: Show what results when a low frequency signal frequency-modulates a high frequency one.

POSSIBLE MODES: It would be useful to allow students to design their own frequency or phase modulators. They could then do experiments with them, and this could continue as a hardware project to build what they had designed.

NAME: SOLUTION OF DIFFERENTIAL EQUATIONS

DESCRIPTION: This has been one of the traditional uses of analogue computers, and there is no reason why it should not be included as one of the possible uses of this system, as the mathematics required by electrical engineers includes the modelling of physical situations into differential equations, and then calculating the solution of these equations in the time domain).

POSSIBLE MODES: A practical in an early networks course, or in a physics or an applied mathematics course for electrical engineers.

EXAMPLE In a typical scenario, the students are first introduced to an RC circuit and an LC circuit, and shown how to derive

and solve differential equations. They may then be given a circuit containing a capacitor, an inductor and a resistor, and asked to derive the network equation, and solve it. They may also enter the equation into **SOFTWIRE** and observe the response of the circuit to different inputs. More complex circuits may also be attempted.

NAME: OSCILLATORS

DESCRIPTION: A wide range of integrators are implemented and compared.

POSSIBLE MODES: A practical in a more specialised course on system design.

NAME: FOURIER SYNTHESIS

DESCRIPTION: Any number of sinusoidal outputs of any frequency can be scaled and summed, and the final ouput curve seen.

POSSIBLE MODES: This may be used briefly as a demonstration when Fourier analysis is introduced, or as a tutorial before it is introduced formally in lectures. The full tutorial given at the beginning of this chapter explains a possible way of introducing it.

NAME: DIPOLE FIELD PATTERNS : POINT/LINE CHARGE ELECTRIC FIELD

DESCRIPTION: Both the above field patterns can be shown for different conditions, exploiting the "y vs x" mode of depicting data.

POSSIBLE MODES: This is best done in open-ended exploration, with the student being given guidelines as to what types of sources etc to explore with. Making the student write the implementation program is a very helpful step in aiding understanding.

NAME: CHARACTERISTICS OF SOLAR CELLS

DESCRIPTION: A solar cell is placed outside, and the voltage from it, as well as from a light-intensity meter are fed into the **User Interface** (after suitable scaling). Readings are taken for about eighteen hours, at half-minute intervals. The results can be used to plot solar radiation cycle over time. They could also be used to relate radiation and voltage.

Another possibility is to use the **SOFTWIRE** to provide a variable load, and to plot the V vs I characteristics.

NAME: SERVO-CONTROLLER

DESCRIPTION: Output from a potential divider measuring the position of a servo motor is input to a **SOFTWIRE** system which uses a control algorithm to control the motor. The system initially applies a step change to the motor. It stops after a fixed length of time and gives a plot of the results on the screen. The experiment can be repeated with various parameters in the algorithm, or with various algorithms.

NAME: SERVO-CONTROLLER MODELLING

DESCRIPTION: The above experiment is modelled completely in **SOFTWIRE** by using block diagrams, and the experimental and actual results compared.

NAME: SERVO-CONTROLLER MODELLING BY DIFFERENTIAL EQUATIONS

DESCRIPTION: The above exercise is repeated but starting with a differential equation. The significance of the parameters in this equation can be compared with the parameters in the previous example.

NAME: STRANGE ATTRACTORS / CHAOTIC SYSTEMS

DESCRIPTION: This interesting field of study -usually in

difference equations- is easily studied with the aid of **SOFTWIRE.** Although serious study is better done with a specialised program (for reasons of speed), demonstrations of this phenomenon are simple to set up. For example, try the famous "Hennon attractor", by doing a "y vs x" plot of the difference equation:

$$x_{n+1} = 1 + 0.3\ x_{n-1} - (1.4\ x_n)^2$$

Use the delay element to obtain the previous x value, and plot over the range ±1.3 for an overview, before zooming in on smaller windows.

CHAPTER 16

CONCLUSIONS

A system, **SOFTWIRE**, has been designed, and different parts have been tested and progressively refined until it can now be used with a degree of confidence as a tool to enhance learning in Electrical Engineering curricula.

In evaluation, it would be helpful to compare **SOFTWIRE**'s performance with the original aims, and also to point the way for improvements. This will now be done.

16.1. COMPARISON WITH BROAD AIMS

It was stated that the system should be resource-effective, enabling more learning to be achieved at present budgets. This has been achieved through the substitution of some of the time of paid lecturers by a computer -freeing the lecturer to spend time more fruitfully with students. Students have come more often with problems related to course content itself than with distractions (as was the case before). Certain modes (such as demonstration and group work) also ensure that the computer is more optimally utilised, by ensuring that more than one student can be serviced by each computer. A big advantage of simulation is that more useful material is developed in a short time, than with modes such as computer-assisted instruction, where a few minutes of lesson time require a few hours of preparation time.

The system was intended to have low initial lecturer training and capital outlay. By further increasing the potential use of computers already installed, and by making most of the hardware

demands of the system optional (rather than pre-requisites), present computing resources will obviate most of the capital outlay. The advantage comes because the equipment required is already commonplace, and not specialised. Only the **User Interface** is specialised. However, the system is still effective without it. Also, it has been very simply designed, and cost should be low. With the diagrams provided, departments will be able to manufacture their own, if need be. Further, if a class was to use **SOFTWIRE** as part of their laboratories, not all the computers would require interface units.

Since the **User Interface** is very adaptable, it appears it can adequately replace much of the laboratory equipment. It can also emulate equipment which (because of its expense) would not be accessible to each student for hands-on experience. The student would be receiving more than a "simulation" experience, because **SOFTWIRE** would merely be doing mathematically what the equipment did internally anyhow.

Regarding staff training, becoming familiar with the system would take the lecturer about the same time as the student, initially. The lecturer may begin to use it without understanding it fully. On the pedagogical side, although the system does involve descriptions of educational strategies, lecturers are not obliged to consult these. It is expected they would prefer to adapt their previous techniques. To include the strategies mentioned in PART 3 would be optimal, but not essential.

Desire was expressed to have a system that was <u>easily progressively implementable</u>. This is a requirement that **SOFTWIRE** fulfills completely. A suggestion is that lecturers begin using the program for demonstrations. Some demonstrations, especially those provided in PART 3, or by other lecturers, would be possible with

very little preparation. The lecturer could gain enthusiasm in this way. If he simultaneously distributed the software to enthusiastic members of the class, interest would become even higher.

The lecturer could then progress by adding a project on **SOFTWIRE** to a list of options for a small class design project. This may be preferable to trying it out on the whole class at once! He could also use one of the tutorials provided in PART 3. The system thus appears to be conducive to slow introduction of its use, allowing minimal disturbance to previous routine at each level.

16.2. COMPARISON WITH SPECIFIC AIMS

How the system has achieved the specific aims mentioned in the INTRODUCTION will now be discussed in turn:

1. "To encourage a modular, functional orientation toward the design of electronic systems."

SOFTWIRE has fulfilled this aim by its intrinsically modular, functional nature. To specify a system as a net-list, one **has to** represent the system modularly and describe it in the net-list in terms of these modules, each with a specific function. The modular orientation is not encouraged, but forced, and in using **SOFTWIRE**, the user develops the required approach.

The next step in modularity is to take the modular concept further, and introduce "top-down" design. By adding the facility for taking a net-list which performs a specific function, and defining this whole list as a submodule, one could enhance the

top-down approach, although the onus would be more on the lecturer to encourage this.

2. "To allow the student to experiment with systems, with maximum transparency. This means minimising distraction due to the interface with the computer, and simplifying syntactic requirements of the description language, as well as making required interaction as intuitively obvious as possible."

In this area, **SOFTWIRE** has been a very definite success. Primarily graphic output representation has made it possible for the user to rapidly interpret the output. The **User Interface** has enhanced this by increasing the intuitive value of physical representation of output, and it has also made interactive input by the user during a simulation more intuitively obvious and hence less distracting (no one needs training to use a "volume control" potentiometer).

It is felt, though, that the **User Interface** would have even more intuitive value if the analogue sections were converted from linear to logarithmic representations. This is because logarithmic scales are more closely related to the physical world, and especially our physical senses. For instance, a change from 0.001 to 0.002 volts has more significance than a change from 100.001 to 100.002 volts. This is consistant with a logarithmic scale (but negated on a linear scale), and hence such a scale is of more value in imparting a sound approach to students.

The use of default values and sequenced input prompting has meant that the computer requires less data from a novice in order to produce a meaningful simulation, and at the same time ensures no

important data remains unspecified. The more experienced user can bypass these options for greater flexibility.

The source of distraction resulting from slowness in a computation-intensive simulation does not seem to be a problem in the majority of simulations. In fact there are many applications where the present speed is actually useful, as the rate-of-change of output becomes very graphic when a fast-changing plot appears to move at higher speed on the screen than a slow-changing one. This adds an information dimension to the output. A faster plot may lose this dimension. However, some net-lists, and many real-time applications are at present distractingly slow. There is not much one can do about this from a design point of view. However, the use of a mathematics co-processor (the **8087**) was tried, and engenders a significant improvement.

Improvements to **SOFTWIRE** which could further this aim are now discussed briefly. A possible enhancement would be to improve information presentation on the screen. The screen could consist of the block-diagram representation, with windows containing graphic output next to each required module. Another improvement would be for the user to input a block diagram directly, and for the conversion to net-list to be done automatically by the computer. These suggestions may not be as good as they first appear. Firstly, they contradict the fourth aim listed (giving practice in making net-lists). Secondly, it is good for students to have to draw their own diagrams and keep other information on paper. This improves student interaction and involvement. The conclusion would be that (subject to personal taste) the present system has more educational value.

The use of **SOFTWIRE** could be further enhanced through the use of context-sensitive help files at each point in the program.

Another useful addition would be a single text line below the menu, providing constant information on what data was required next. This would prevent students from becoming stuck as to where to proceed. However, experience so far is that the present system is unambiguous enough that students rarely get stuck, and the use of a good tutorial sheet reduced this problem, by anticipating these areas for novices, and giving more complete instruction where applicable.

As discussed before, hardware improvements in microcomputer technology would be of benefit. Specifically, faster processors would be helpful, for reasons discussed earlier in this section. Improved graphics resolution would also help, as it would assist in a less limited presentation format for data on the screen. It would ultimately be useful to list all useful parameters and the net-list on the screen at once, together with graphic output on more than one set of axes.

3. "To tie together simulation based on idealised component implementation (e.g. traditional analogue computation) and that based on an extended set of functional primitives, in which real-world limitations and complications can be progressively introduced, thus firmly rooting simulation in a real-world context."

This idea was found to be best implemented via a **sequenced tutorial** mode: starting from an idealised representation and progressing to a complex and more realistic one. An example with integrators, given in PART 3, shows the potential of the mode, and culminates in an actual hardware implementation on the **User Interface**. The best way that this aim could be furthered, was by the building up of files (or software "libraries") containing the

code for different levels of idealization of various functions.

A similar useful procedure also involved the students' modifying the ideal model themselves, using other function blocks. This procedure could be enhanced by the implementation of submodules, as mentioned under the first aim.

4. "To provide students with an early acquaintance with a linguistic (net-list) alternative (of growing importance) to graphic-based descriptions such as the block diagram."

The syntax used in **SOFTWIRE** is seen to be a useful starting point for teaching the idea of using a net-list. It is useful because of its simplicity, and simple correspondence with the block diagram. It is simple enough to only require a few minutes for the learner to grasp the syntax, and the use of only a few examples makes it very clear. It also corresponds with the block diagram closely enough that, given a block diagram, there is a one-to-one correspondence with information on the block diagram, and information in the net-list. This means minimal complexity in performing the conversion. After a time of use with **SOFTWIRE**, the user becomes familiar in working with net-lists. This familiarity is useful in a conversion to a more complex program such as **SPICE**, which has so many more parameters etc. in each net-list entry.

As mentioned, the net-list was considered important in ensuring students did not do exercises without proper interaction. If they had to hand in a report on a tutorial which included block diagrams, they were forced to mentally process each block when they made the conversion, and what they had to hand in could not be merely copied from the screen.

5. "To provide a vehicle for the rapid creation of demonstrations of basic signal-processing operations and techniques, and the dynamic performance of systems."

The creation and use of demonstrations has been the mode which has caught on the most rapidly amongst users. Since each simple demonstration only takes a few minutes to set up, in many cases it can be less time-consuming than preparing an overhead transparency. It has the advantage over overheads of being dynamic. Parameters may be easily changed, and new results shown. This is another area where the building up of a library of demonstrations would enhance effectiveness.

6. "To provide a simulation language optimised specifically for electrical engineering, and an associated system with a hardware laboratory interface, so that simulations could be closely associated with (and progressively substituted by) actual laboratory experiments."

Four features of **SOFTWIRE** make this aim simple to achieve. The first is the ability to simulate an entire laboratory in software. The second is the laboratory interface, which has been made relatively transparent to the user. (Transparency is achieved by treating purely software modules and hardware modules identically. A net-list can have a mix of the two types of modules, and only the mnemonics reveal the category. Substitution is thus trivial.) The third feature is the ability of **SOFTWIRE** to be used as an instrument in a laboratory. Finally, one can display, manipulate and record results easily.

Extensions which would enhance this mode, would be to devise a set of cards with different transducers, thus facilitating **SOFTWIRE**'s application in performing and learning about measurements. A significant enhancement would be a feature to display recorded data, and to add a level of data-processing to this option, such as filtering or frequency-domain analysis.

7. "To provide a system for laboratory prototyping, in which part of the system could be constructed in hardware, and part simulated by **SOFTWIRE** modules, in an integrated system."

This aim has been simple to achieve, since it is a matter of course to set up a test-bed, with net-lists to generate the input and process the output. The prototype boards are then plugged into the **User Interface**, and since binary and analogue I/O and ±12V and ±5V power rails are provided, the prototype can be constructed without distraction in these areas. Any part of the card which has yet to be constructed may be simulated.

To improve the usefulness of **SOFTWIRE** in this regard, it would be help to have provision for working with current as well as voltage. This would enable the more advanced students to progress further into the realm of electronic implementation. Of course, the user may still work in the current realm with the present system, simply by inserting an appropriate op-amp circuit at the input and/or output of his board to make the conversion.

8. "To provide an extendable simulation language and system, in which new functional primitives can be incorporated by the user."

The language has been shown to be extendable, since modules were constantly being added on since the system was first developed.

It takes an average of five minutes or less to define a new module (excluding function coding). The length of time to code the function of the new module depends on its complexity. A simple one would take one or two minutes.

Although it is easy to add a module once the procedure is known, it is felt that at present it takes too long to learn how to add a module. This results in module extensions only being done by real enthusiasts, and it would need a more sophisticated module creation system to encourage others to make extensions. A useful extension would be the use of a module file. This file would be a table such as the one below:

Name	Inputs	Parameters	Def'd	Extra Mem	Comments
VIN	0	2	Y	N	Voltage input from keybd
SUM	2	0	N	N	Sum 2 inpts
DLY	1	1	N	Y	Delay by time in param

A short intermediate program would convert the table into source code, a process which would not be difficult, since the above "syntax" maps directly onto the data structures used by **SOFTWIRE**. The name in the first column is stored as a string, the next few as sets, and the comment is ignored. The table has the advantage of being self-documenting to the user, and learning how to add modules becomes very simple. One of the fields could also be the name of a file containing the code, and the program would then **include** that file during compilation.

16.3. BUILDING UP A USER BASE

The program also requires more testing before assertions can be made about its place in Engineering Education, and really the final test is in the marketplace -Faculty. This is a very difficult place to predict success, since there are more

considerations taken into account in this realm than objective feasibility!

It is the dream of every producer in computing to build up a user base. Once this process has begun, factors come into play which entrench and improve the product.

It was noteworthy that this software was distributed freely to students during testing, and that a few of them (especially those with their own computers) were already suggesting changes, and examining and adding to the source code. This enthusiasm outstripped the expectations of the tests done.

One of the factors, which needs to be enhanced, is the easy exchange of ideas. This includes advertising by word-of-mouth, as well as describing problems run into, giving helpful hints and producing support material. Bulletin boards greatly facilitate this. **SOFTWIRE** could benefit by having a shareware base which encourages correspondence.

Support material comes in the form of public domain extensions created by enthusiastic users. Especially helpful would be well-written function modules and simple net-lists.

These could be used to build up libraries of specific applications. One could, for example have a whole library of integrator implementations. There is also scope for libraries of ideal-real transitions, or of modules with highly specific functions, useful for a single demonstration or tutorial only. Libraries need not be software -one could have hardware libraries of schematics for cards. This process would require disciplined sorting and description. The libraries could be created by single people at one sitting, or they could be created over time as they proved themselves to be effective.

One way to improve the user base for **SOFTWIRE** would be to use it in other disciplines such as applied mathematics or physics. Although the system design was not optimised for them, there is enough overlap for it to be useful, it would help students to have standardization if it is used in different courses, and it would considerably broaden the user base.

16.4. CONCLUSION

PART 1 of this dissertation laid a historical backdrop and overview which resulted (together with other considerations) in the motivation for devising a set of aims, as well as for the means of implementing them.

In PART 2, the technical aspects underlying the creation of **SOFTWIRE** were discussed, and in PART 3 the application to teaching was covered.

This chapter has shown in what ways the system has achieved the set of aims set out for it. The next step would be to simultaneously make improvements to **SOFTWIRE**, such as the one's recommended, and then to use that framework on which to hang a curriculum involving the fulfillment of the pedagogical aims mentioned. The previous section outlined a beginning strategy for this.

However, it is felt that the present implementation of **SOFTWIRE** is already fully capable of being profitably used, and that improvements would be marginal. The modules already implemented and the areas of useage already covered have already proved their contribution to enhancing learning in the Electrical Engineering curriculum.

APPENDIX A
SOFTWIRE MANUAL

[N.B. This manual has been shortened to avoid unnecessary duplication with the rest of the dissertation]

1. INTRODUCTION

SOFTWIRE is an educationally orientated simulation program which is intended as a basic teaching resource to be used in several contexts. A few of these are listed:-

-Teaching the fundamentals of a modular, functional approach to electronics.

-Allowing students to experiment with systems.

-Introducing concepts of idealising and modelling, and giving assistance in the transition to the complexities of "real world" implementations of systems.

-Providing the opportunity for active problem-solving in a simulated environment;

-Setting up demonstrations for a class, with very little preparation time.

-Providing a testing and prototyping environment for hardware constructed in design projects.

2. DESCRIPTION

The program **SOFTWIRE** is similar to an analogue computer in its operation (though not its purpose). Modules are defined in the program, each one representing a particular hardware function such as integration, zero-order-hold, level detection etc. The modules can range in complexity from simple summers, to voltage-controlled oscillators, threshold detectors with hysteresis and phase-locked loops. Some of the blocks may even be actual hardware circuits interfaced via a general purpose I/O port which has been made, which includes some built-in hardware for input (such as potential dividers and switches) and output (such as meters and LED's). The user can include these blocks in any valid topological combination, and the program computes the voltage levels in the circuit. These levels may then be plotted against time or against other outputs, in a variety of formats. System creation and modification, and output format specification are quickly and easily accomplished.

It is important to note that the system includes a hardware interfacing unit. **SOFTWIRE** may be used without this unit, but none of the hardware features mentioned will be available.

3. USES

SOFTWIRE can be applied to many areas of electronics learning. Here are examples of three areas where it is likely to be useful:

3.1 EXAMINING SIMPLE ELECTRONIC PRINCIPLES

One may watch and see how a simple RC circuit integrates or differentiates. It is a simple matter to explore the results of a wide range of inputs, and RC parameters. By using different

4.1 STRUCTURE

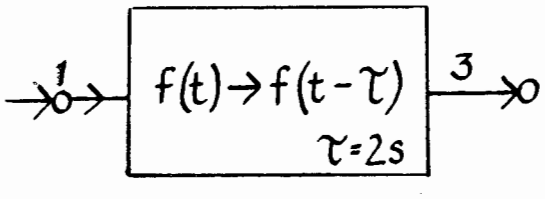
The structure specifies the modules which are to be used, and the way they will be interconnected. The system should be drawn out on paper, and node numbers assigned to the output of each functional unit (module) such as an integrator, adder etc. One module per line is typed in. Each module description is of the form:-

N module_name(parameter1 parameter2) input1 input2[1]

where N specifies the output number of that module. The module_name specifies the function of that module, taken from the library of available functions. The inputs are then specified using the N values of other modules. For example,

3 DLY(2) 1

takes the output of module 1 and delays it by 2s to give a value at output number 3. The order of module input is not significant. The module entered could be described thus:-



SOFTWIRE continues to accept the net-list, line by line, until an empty line (i.e. a <Retrn> key at the beginning of the line) is given to indicate the end of the data.

[1] The formal **Backaus-Naur Form** is

N' 'module_name('('param1(',param2)')')' '(input1(' 'input2))

However this is an unnecessary complication at this stage!

If a mistake on a line is detected by the computer, the line is rejected and must be entered again.

Comments may be included on a line following the entry. They must be seperated by a space. They are not stored, and so are only useful if entry is from a file.

4.2 PLOT RANGE

The approximate voltage range of the modules to be plotted must be specified. Rather use bigger values the first time, and then focus in on the amplitude range being examined. The screen has 200 dots in the vertical scale. If the values are out of range, the calculations will not be affected at all, but the out-of-range values will be clipped.

The range is specified by two numbers -a minimum and maximum- which are prompted for. For instance, the values:-

Minimum y-axis value ("?" for help)=**-3.6**
Maximum y-axis value ("?" for help)=**4.2**

would specify that the plot must be scaled between -3.6 and 4.2 volts.

4.3 PLOT MODULES

A list must be provided of all the modules for which a plot is required. Up to seven plots may be done simultaneously, but the computer only plots in three colours, so this may make recognition of curves difficult. The first plot specified appears in white, the second in blue, and the third in red. The fourth is then white again and so on. The modules are specified by listing

their numbers, delimiting them with spaces. For instance:-

3 6 23 1

would result in modules 3, 6, 23 and 1 being plotted.

4.4 TIMING RANGE

The simulation runs from t=0s until a time specified by the user. The user must also specify the time increments used in the calculation. Smaller increments result in more accurate results, but it takes longer to run the simulation. Further, if the increments are too small, the results become so accurate that the resolution of the screen becomes the source of error. **SOFTWIRE** scales the time axis to fit the entire run onto one screen, so a long time period will make the plot cramped. The screen has 320 dots across it. If the simulation is done in the infinite time mode, the run continues indefinitely (as will be discussed later) and the plot is scaled at one horizontal dot per time increment. However, a time period must still be specified initially.

The time range is specified by two numbers separated by a space. The first number is the time increment, the second the time period. For instance:-

```
Time increment ("?" for help)=>0.04
  Maximum Time ("?" for help)=>10
```

will result in a run with the time being incremented in steps of 0.04 seconds, continuing from t=0 seconds to t=10 seconds. This will result in (10/0.04)=250 sets of calculations.

The simulation normally runs faster than real time. This is not the case if a very small time increment is used, or with a very

large net-list. The option also exists to specify a real-time simulation, if the hardware interface is plugged into the computer.

4.5 STRUCTURAL CHANGE

This heading is given if the user requires to edit the net-list. The format for entry is identical to the initial STRUCTURE format. If a module number is specified, and that module already exists, the old module is replaced by the new one. If not, it is added to the net-list.

4.6 PARAMETERS

This heading is given if one requires to change only the parameters. To specify a change in a parameter, the module number is first given, followed by the parameter(s). A space separates each entry on a line. A new line is used for each module. An empty line followed by a <Retrn> key indicates the end of the input. For instance:-

```
4 (3.0)
2 (1.1 2)
<return>  {note this empty line indicating end of parameter list}
```

specifies that module 4 must have its parameter changed to 3, and module 2 has its first parameter set to 1.1 and its second to 2.

5. MODULE FUNCTIONS

5.1. LIST OF FUNCTIONS

A list of all the functions provided is not entirely useful, since the user may program any number of his own functions in. Furthermore, one only uses the functions which are suitable to

the given application. For instance, a lesson on logic functions will use AND, OR, FLP etc. A few modules have been included here to give examples of the types of functions that can exist, and the variety of numbers of inputs and parameters possible, as well as a simple way to tabulate the functions. When the program is used, it must be accompanied by a list of the modules being used, and their functions. If the user types in a non-existent function, **SOFTWIRE** will list all the module names available, but will not describe the functions.

NO INPUTS -NO PARAMETERS:

***time** -this is automatically given as the output of module 0.

***constant** -e.g. 2 CON(4.3) gives a constant output of 4.3 from module 2.

NO INPUTS -WITH PARAMETERS:

***sinusoid** -e.g 7 SNW(6.2) gives a sinusoidal output at module 7 of amplitude = -1..1 and frequency 6.2 Hz.

***cosinusoid** -e.g 5 CSW(0.1) gives a cosinusoidal output at module 5 of amplitude= -1..1 and frequency 0.1 Hz.

***rectangular pulse** -e.g 1 PLS(5.2,6.4) gives a zero output at module 1 for all times except when 5.2 <= t <=6.4 (seconds), when it gives an output of 1.

***square function** -e.g 2 SQW(6) gives a "square wave" output at module 2 of amplitude = -1..1 and frequency 6 Hz. If a negative frequency is specified, the output begins as a negative value, otherwise it always begins at +1.

***sawtooth** -e.g 3 SAW(12) gives a sawtooth output at module 3 of amplitude = -1..1 and frequency 12 Hz.

***triangular function** -e.g 7 TRW(5.6) gives a "triangular wave" output at module 7 of amplitude = -1..1 and frequency 5.6 Hz.

***modulated width pulse** -e.g 10 SNW(30 0.4) gives an rectangular output at module 10 of amplitude = -1..1, frequency 30 Hz and duty cycle is 40%.

***user-defined function** -e.g 1 WFM(2.3) gives an output at module 1 of amplitude = -1..1 and frequency 2.3 Hz. The output is defined by the file "WFM.INS".

***rectangularly random function** -e.g 3 RND(0.3 1.3) yields random output levels at module 3 with uniform weighting between 0.3 and 1.3.

***gaussian random function** -e.g. 7 GAU(0.5 0.3) yields random output levels at module 7 with gaussian weighting, of mean 0.5 and standard deviation 0.3.

***keyboard-controlled input** -e.g. 5 VIN(2 0.2) will cause module 5 to output a level of 2 initially, but the level will change by increments of 0.2 when the user pushes the grey '+' or '-' keys.

***User Interface-controlled analogue input** -e.g. 2 PT1(-4 6) will result in an output ranging linearly from -4 to +6, corresponding to the position of the leftmost potential divider on the **User Interface** cabinet. (The rightmost pott. is referred to as PT2, and operates in an identical fashion to PT1).

***User Interface-controlled binary input** -e.g. 3 SW1 will output a 0 at module 3, except when the leftmost button on the **User Interface** cabinet is pressed, when it will output a 1. SW2 works in the same way except that it is a toggle switch. SW3 is the same as SW2, except bi-directional, giving outputs of -1, 0 or +1.

***User Interface-card analogue input** -e.g. 5 AD1 will cause module 5 to output a value ranging linearly from -1 to +1, corresponding to the voltage (of range -5V to +5V) on analogue channel 1 of the **User Interface** slot. (Channel 2 is called AD2, and operates in an identical fashion to AD1).

***User Interface-card binary input** -e.g. 4 DI1 will output a 0 at module 4, except when the level on the **User Interface** binary channel 1 is high, when it will output a 1. (SW2 works in the same way except that it monitors channel 2).

SINGLE INPUT -NO PARAMETERS:

***sin** -e.g. 23 SIN 4 outputs at module 23 the sin of output 4 (in radians).

***cos** -e.g. 11 COS 9 outputs at module 11 the cos of output 9 (in radians).

***differentiate** -e.g. 45 DIF 1 outputs at module 45 the derivative of output 1.

***exponent** -e.g. 22 EXP 4 outputs at module 22 the exponent of output 4.

***arithmetic inverter** -e.g. 6 INV 3 takes the output from module 3, and subtracts it from 1 to give the output at module 6. This action resembles an inverting amplifier.

***User Interface-card analogue output** -e.g. 5 DA1 7 will output a voltage of range ±5V, corresponding to an input of ±1, to the **User Interface** analogue channel 1. (DA2 works in the same way for channel 2).

***User Interface-card binary output** -e.g. 4 DO1 6 will output a TTL "high" level on the **User Interface** binary channel 1, if the output of module 6 high, otherwise it will output a "low". (DO2 works in the same way except that it controls channel 2).

***logical NOT** -e.g. 22 NOT 4 outputs at module 22 the logical inverse of output 4 (a level of "1" is high, "0" is low).

***natural logarithm**-e.g. 41 LIN 3 outputs at module 41 the natural logarithm of output 3.

***negative** -e.g. 45 NEG 1 outputs at module 45 the negative of output 1.

***absolute value** -e.g. 4 ABS 12 outputs at module 4 the absolute value of output 12.

***square** -e.g. 5 SQR 21 outputs at module 5 the square of output 21.

***arctan** -e.g. 2 ATN 4 outputs at module 2 the arctan of output 4.

***comparator** -e.g. 5 COM 14 outputs a 1 at module 5 if output 4 is greater than 0, else it outputs a -1.

***idealised diode** -e.g. 5 DIO 4 module 5 follows output 4 when it is positive, otherwise the output is zero.

***user-defined (tabled) output function** -e.g. 7 TAB 2 will use the output from module 2 to access a table defined in twenty steps from 0 to 1. This is defined in the file "TABLE.INS". It will use this, and a linear interpolator, to output a value.

SINGLE INPUT -WITH PARAMETERS:

***z-o-hold** -e.g. 10 ZOH(7.23) 4 outputs at module 10 the value of output 4 i.e. 10 follows 4, but the output is held at its last value, and only updated at the frequency 7.23 Hz. [NB The sample period should obviously be smaller than the time increment steps being used in running the simulation, and preferably a lot smaller, otherwise appreciable inaccuracies will occur].

***gain** -e.g. 3 GAI(2.3) 2 multiplies the output at module 2 by 2.3 and gives the result at module 3.

***integrator** -e.g 2 INT(4.1) 3 integrates module 3's output (using the Euler formula) and outputs the value at module 2. Initial value (at t=0) is 4.1 The output has no threshold (actually it is at ±32700V) and does not invert. i.e. It is an idealized integrator. (NT1 operates in exactly the same way, except that it users a Fourth Order integration formula. NT2 is the same as NT1, except that it saturates at ±1, and

inverts the input. Nt3 is the same as NT2, except that it has a slight drift, and a small amount of noise superimposed)

***threshold detector** -e.g. 5 THR(0.7) 12 tests the output of module 12. If this value is >0.7 then module 5 sits at 1V else it sits at -1V

***threshold with hysteresis** -The first parameter specifies the threshold level as a fraction from 0..1 The second specifies the starting output value, and must be +1 or -1. If for instance,the threshold value is 0.2, and the initial value is -1, the module will behave as THR acts. However when the output swings to +1, the threshold then becomes -0.2. e.g. 2 THH(0.1,-1) 7 will start with 2 at -1V, and will test output 7. If it goes above 0.1V, then 2 will go to +1, until 7 goes below -0.1 etc.

***delay** -e.g. 5 DLY(0.4) 6 means that module 5 follows module 6 with a delay of 0.4 seconds. [NB Ensure that the delay required is definitely not smaller than (and preferably not close to, unless it is exactly integral to) the time increment specified for running the simulation].

***integrator(saturates/inverse)** -e.g. 7 RNT(4) 2 will integrate the output of 2 and give the result at 7. However it inverts the output (changes the sign) and saturates at 5V, thus simulating an op-amp integrator. The parameter specifies the initial output value, which in this example is 4V.

***monostable** -e.g 23 MON(3.7) 6 will output at module 23 a rectangular pulse of duration 3.7 seconds and amplitude 1V, every time the output of 6 goes above 1V. The monostable module only tests after the previous pulse is complete (i.e. it is a "non-retriggerable" monostable).

***voltmeter** -e.g. 7 MT1(-5 10) 8 outputs the value of output 8 on panel meter 1, with the range of the meter defined as -5V to 10V.

***low-pass gaussian filter** -e.g. 5 LPF(0.4) 6 means that module 5 follows module 6, but with an RC filtering effect, with τ = 0.4s.

***User Interface LED** -e.g. 7 LT1(0.4) 3 will monitor the level of module 3, and light the first LED if it exceeds 0.4. (LT2, LT3 and LT4 work the same way, for the second, third and fourth LED's).

***logarithm** -e.g. 41 LOG(10) 3 outputs at module 41 the logarithm to base 10 of output 3.

***deadspace** -e.g. 5 DSP(-1 2) 4 module 5 follows output 4 when it is outside the range -1 to 2, otherwise the output is zero.

***voltage-controlled oscillator** -e.g. 12 VCO(10 2.5) 3 will cause module 12 to output a sinusoid of centre frequency 10 Hz, and modulated by the output from module 3, by 2.5 Hz per unit.

TWO INPUTS -NO PARAMETERS:

***sum** -e.g. 6 SUM 1 2 sums the outputs of modules 1 and 2 and outputs the result at module 6.

***mul** -e.g. 5 MUL 3 8 multiplies the outputs of modules 3 and 8 and outputs the result at module 6.

***quotient** -e.g. 6 QUO 7 2 divides the output of module 7 by the output of module 2 and outputs the result at module 6. If the denominator is zero then it checks the numerator. If this is zero it outputs unity. If this is negative it outputs the maximum negative number. If this is positive it outputs the maximum positive number.

***modulus** -e.g. 5 MOD 3 8 takes the modulus of the outputs of modules 3 and 8 and outputs the result at module 6.

***logical AND** -e.g. 4 AND 1 2 will logically AND the outputs from modules 1 and 2. A level of 1 is true, and 0 is false.

***logical OR** -e.g. 3 ORR 5 2 will logically OR the outputs from modules 5 and 2. A level of 1 is true, and 0 is false.

***logical XOR** -e.g. 7 XOR 1 3 will logically XOR the outputs from modules 1 and 3. A level of 1 is true, and 0 is false.

***logical NAND** -e.g. 4 NND 1 2 will logically NAND the outputs from modules 1 and 2. A level of 1 is true, and 0 is false.

***logical NOR** -e.g. 6 NOR 5 8 will logically NOR the outputs from modules 5 and 8. A level of 1 is true, and 0 is false.

***polar coordinate argument** -e.g. 2 ARG 3 4 will accept the input from module 3 as a horizontal coordinate in a rectangular system, and the output from module 4 as the vertical component. It will then output the angle in (2-D) polar coordinates

***polar coordinate length** -e.g. 2 LEN 3 4 will accept the output from module 3 as a horizontal coordinate in a rectangular system, and the output from module 4 as the vertical component. It will output the length in (2-D) polar coordinates.

***rectangular coordinate horizontal component** -e.g. 5 HOR 9 8 will accept the input from module 9 as a modulus component in a polar system, and the output from module 8 as the argument. It will then output the horizontal component in (2-D) rectangular coordinates.

***rectangular coordinate vertical component** -e.g. 5 HOR 9 8 will accept the input from module 9 as a modulus component in a polar system, and the output from module 8 as the argument. It will then output the vertical component in (2-D) rectangular coordinates.

TWO INPUTS -WITH PARAMETERS:

***error monitor** -e.g. 9 CTL(2) 3 5 will subtract the level at module 5 from module 3 to output an error level for a controller. The initial value is set to 2 (this value is important as a "starting point" in a system with feedback).

***phase detector** -e.g. 5 DET(1) 6 8 detects the zero-crossing of modules 6 and 8, and XOR's these to give a duty cycle proportional to the phase difference of the two signals. An initial value of 1 is output here. (this value is important as a "starting point" in a system with feedback).

5.2. CUSTOMIZING FUNCTIONS IN SOFTWIRE.

(This section is intended primarily for the instructor using **SOFTWIRE**, rather than the student. It is not necessary for the user who does not wish to customize **SOFTWIRE** to read it)

A powerful feature of **SOFTWIRE** is that one may customize the functions that the program uses. This is not done under program control, but requires changes to the Turbo Pascal source code. Thus, a knowledge of Turbo Pascal, as well as a copy of the Turbo Pascal 3.0 editor and compiler is required. However, the actual programming required has been greatly systematized to minimize errors. The procedure is as follows:-

1. Edit the file **Modecl.Ins**.

This is where the function modules are declared. First, the number of modules needs to be updated. This is given by the CONST called max modactions.

Next, a three-letter name has to be given to the module. This is the name that the user will use to identify the module in his net-list.. This is given as an element of the CONST array opname. N.B. the number of elements of opname **must equal** the value of max modactions. There is no restriction on the characters that may be used in these names[2]. The module TIM (the timing module) must never be excluded, as it is part of the system.

One now moves to the TYPE declarations. The first is TYPE mod actions. This contains a list of scalar names as they are to

[2]Except for the very unlikely case where a single quote is used, in which case the normal rule in Pasca applies, that two quotes must be typed in juxtaposition, to represent the single quote within the string. This is to avoid confusion with the single quotes used to delimit the string.

be used by **SOFTWIRE** itself. These names need not be the same as the list in opname. However, it is recommended that they be kept the same to avoid confusion. An important limitation here is that the names used must be unique. This means that:-

a) They may not be Pascal reserved words.

b) They may not be Pascal Standard Identifiers if these identifiers are used elsewhere in the program with a different meaning. i.e. If the word 'cos' is used to define a cosine module, this will be accepted by Pascal, but then the user may not use the standard function cos() later on.

c) They may not be any identifiers which have another meaning globally.

Remember that the module TIM must always be included.

Another group of CONST declarations is now encountered. This consists of a long list of sets which specify the structure of the module to be defined. The module's scalar name (as defined in mod actions) must be inserted in every set that satisfies the description of the module. The different options are as follows: zerinp,oneinp and twoinp are the modules with zero, one or two inputs respectively i.e a random generator module would have no inputs, a NOT gate one input, and an AND gate two. Then zerparm, oneparm and twoparm are the modules with zero, one or two parameters respectively. The set defndatzero specifies the modules which are defined at t=0 (i.e they have an initial value). If a module requires extra variables in which to store miscellaneous information, the module name may be entered in the set scratchusers. A example of this would be a delay module. In order to provide the delay, a number of previous values would have to be stored. To do this, the module would be included in the set scratchusers. Then, every time a user specified a delay module in his netlist, the module would be created, along with an

array of type real.

SOFTWIRE initialized all outputs to zero, unless otherwise specified. Sometimes this is not required. For example, an integration module would be initialized to a user-specified initial value. Such modules are included in the next set, initnotzero.

The declarations are now complete, and what remains is to specify the function which relates the output of the module to its input(s).

2. Edit the file **Calc.Ins**.

Before this is done, it is important to have an idea of what each module is, from the point of view of the program.

MODTYPE
OpNode
Action
inpt1,inpt2
Param1,Param2
Scratch_Ptr
NextMod

The record for each module

```
modtype=record
    OpNode:ModRange;
    action:mod_actions;
    inpt1,inpt2:ModRange;
    param1,param2 :real;
    scratch_ptr:^scratchpad;
    nextMod:ptrModType;
end; {modtype}
```

Pascal Code

There are three integers here, opnode, inpt1 and inpt2. They are in fact indexes of the array mod vals. This array contains the values of all the output nodes of the modules, during the execution of the simulation. The two variables param1 and param2 contain the parameters of the module, and the identifier action

describes the function. If the module requires extra memory, it is available in the real array pointed to by scratch ptr.

Further understanding can be obtained by working through some examples. The file **Calc.Ins** contains a procedure Calc. Inside this procedure is a large case statement. This contains a statement for every module function that is required. The first module is TIM, the time module. This is always included. After that, the required modules may be included. The program has been written so that when the Case statement is reached. The next module to be specified has already been selected. This means that one need only specify the required field of the module record, and not refer to the module at all. In other words, one refers directly to inpt1,action etc.

A simple example will first be considered first:-.

```
SUM: mod_vals[OpNode]:=mod_vals[inpt1]+mod_vals[inpt2];
```

This would be a line inserted in the Case statement. If the user had added to the net-list a line such as

6 SUM 3 5

The module in question would then have

OpNode=6
Inpt1=3
Inpt2=5
Action=SUM

Since the action was SUM, the Pascal line shown above would be the one selected. It would then be read as

```
mod_vals[6]:=mod_vals[3]+mod_vals[5];
```

i.e SUM the values of elements 3 and 5, and put the result in element 6.

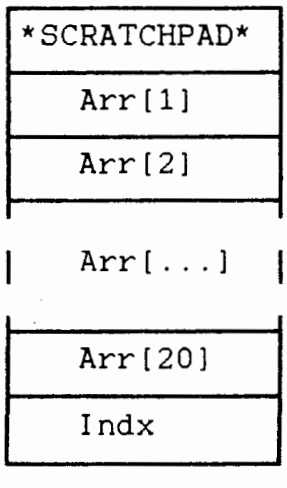
Another example, which uses the parameters and the time variable **T**, is

```
PLS: if (T>=param1) and (T<=param2) then
                mod_vals[OpNode]:=1
          else mod_vals[OpNode]:=0;
```

Here, the time **T** is checked against the two parameters, and the output is zero or one, depending on the outcome.
The use of the optional extra memory is demonstrated by the following example:-

```
DLY: begin
   with scratch_ptr^ do begin
       mod_vals[OpNode]:=arr[indx+1];
       indx:=indx+1; if indx=scratchp_rng then indx:=0;
       if indx>=trunc(param1/delta_t) then
             arr[indx+1-trunc(param1/delta_t)]:=
                                  mod_vals[inpt1]
       else arr[scratchp_rng+1+indx-
                 trunc(param1/delta_t)]:=mod_vals[inpt1];
    end; {with}
end; {DLY}
```

Each module that requires extra memory is assigned one record. The record is arranged thus:-

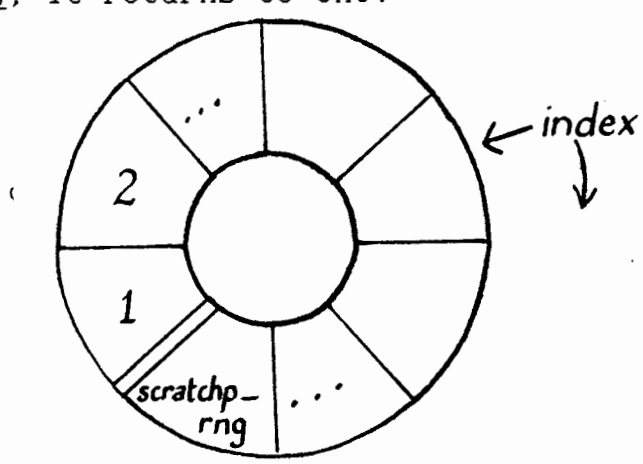



```
const scratchp_rng=20;  {range for each
                         module's scratchpad
                                  elements}
type scratchpad=record
   arr:array[1..scratchp_rng] of real;
   indx:integer;
end; {scratchpad}
```

The module record contains a pointer, scratch ptr, which points to the place on the heap where the "scratchpad" is. Thus, any accessing of the scratchpad is enclosed by the lines:-

```
with scratch_ptr^ do begin
     :
     :   {place code here}
     :
end; {with}
```

This is demonstrated in the delay module source code, as shown previously. The next line of the code outputs to Mod vals. How it does this has already been covered in previous examples. The following line is more interesting. It uses the identifier indx, which is the index to the scratchpad array. The actual array is called arr[]. There is also a reference made to the variable delta t. This is the time increment used by **SOFTWIRE**. The way the delay module works is best considered by thinking of the array as being circular. When the index reaches the maximum value, given by scratchp rng, it returns to one.



With the background supplied in this section, and by examining the documented examples of modules on the disk (in the file MODECL.INS), anyone familiar with Turbo Pascal should be able to set up their own function modules.

6. USING FILES

Files can be used in **SOFTWIRE** to store either a system specification or the results of a simulation.

System Specification: All the data that is used to describe a simulation run, including the net-list, timing, parameters and plot range, are stored. They may be recalled when the program starts up, after the file-retrieve prompt message appears. To save the specification, one uses the **S**ave option of the main memory. **SOFTWIRE** then saves the run exactly as it is then. The program has a default file called **DATA.DAT** which is used if you don't want to give the file a name. This saves time if you are continually saving, retrieving and working on the same file. The program also adds the suffix '.DAT' to the name supplied, to indicate what type of file it is. This may be avoided by ending the name you supply with a period.

Results: These may be stored, examined and printed as a text table (using a text-editor or word-processor), but the subsequent re-displaying from **SOFTWIRE** has not yet been implemented.

7. INTERPRETING THE MENU

Most of the time when **SOFTWIRE** is lying idle and awaiting a command, a menu will appear on the bottom line of the screen, indicating the commands available. There are two levels of menus. The first level indicates the type of command, such as whether to run a simulation or edit your net-list. The next level is used if the command can be broken into many parts, in which case another menu appears with the alternatives. The main menu appears as

follows:-

Disp **E**dit **H**elp **N**ew **P**rnt **Q**uit **R**un **S**av

And one has the following alternatives:-

Disp: Display the results of a previously saved RUN. Not yet implemented.
Edit: Edit some data regarding the simulation. Upon pressing 'E', the second level menu appears:-

Outpt **P**rams **Q**uit **R**ang **S**truct **T**im

Output: Re-specify which module outputs to plot.
Prams : Change module parameters.
Quit : Takes you back to the main menu.
Rang : Change plot range
Struct: Change system structure

Default values: Pressing "D" (or any other letter key except "W") causes the simulation to run immediately. The following default values are used:-

-the output is a "volts vs t" plot

-output goes to the screen only

-the run ends at the time specified earlier under TIMING

Save: Asks you for a filename, and saves the current net-list, range, and timing values under that filename.

8. A SAMPLE PROGRAM

The following is a listing of the exact sequence that would produce a simulation of Amplitude Modulation. To start with, the **SOFTWIRE** disk should be inserted in the boot disk and the computer turned on. Soft type will indicate output to the screen by **SOFTWIRE**, and **bold type** the response from the user. {} brackets indicate a comment inserted only in this Appendix to explain what is going on.

SOFTWIRE
Produced by J. R. Greene and C.D.Geerdts
SOFTWIRE is a teaching aid based on the
simulation of electronic modules

{After another few seconds this will be replaced by the prompt message}

Input from keyboard or file (K/F)=>**K**

{screen clears}

STRUCTURE

1 INT(0) 3 {modules 1 to 3 form a triangle wave generator}
2 GAI(-1) 1
3 THH(0.9 -1) 2
4 CSW(6)
5 MUL 1 4
{one must press an extra <Retrn> here to indicate end of input}

OUTPUT MODULES

1 5

RANGE

Minimum y-axis value ("?" for help)=>**-1**
Maximum y-axis value ("?" for help)=>**1**

TIMING

Time increment ("?" for help)=>**0.003**
Total time ("?" for help)=>**1**

{the menu now appears at the bottom of the page}

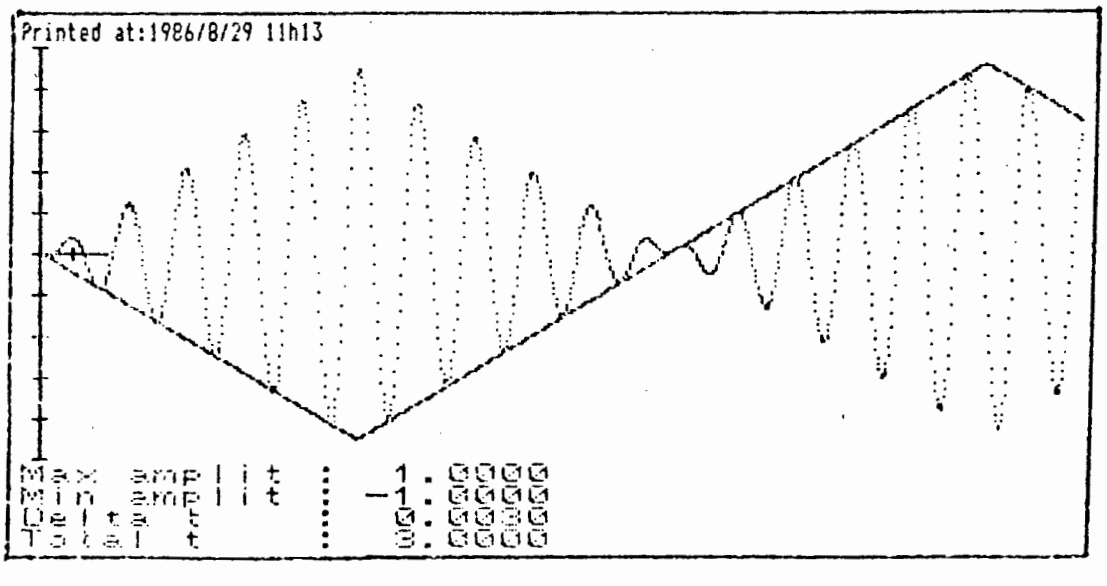
Disp **E**dit **H**lp **N**ew **P**rnt **Q**uit **R**un **S**av

R
{a new menu appears}

Default **W**ith features

D {the default set-up is all we need}

{the following appears on the screen:-}



9. DESCRIPTION OF FILES ON THE DISK

Your disk should contain the following files:-

SOFTWIRE.COM	-the executable program.
DATA.DAT	-the default file in which programs are stored: comes with a sample program in it
SOFTWIRE.PAS	-main file containing source code for SOFTWIRE.COM
MODECL.INS	-insert file called by SOFTWIRE.PAS of source code that contains declarations of the different module functions and descriptions.
CALC.INS	-insert file called by SOFTWIRE.PAS of source code that contains the main calculation routine. Included in this is the code for the calculations involved for each module function.
READ.ME	-file containing text information to supplement this manual.
AUTOEXEC.BAT	-this is executed when the system boots up. It begins execution of **SOFTWIRE**.
COMMAND.COM	-a system file required by DOS, in order to boot up. It is not part of the **SOFTWIRE** package.

SOFTWIRE EVALUATION SHEET

1. INTRODUCTION

SOFTWIRE is an educational program which was conceived and developed at the University of Cape Town's Electronics Engineering Department by J.R.Greene and C.D.Geerdts. It is at present being evaluated and appropriate revisions are being made.

SOFTWIRE is to be used in conjunction with curriculum material, which will demonstrate the different modes of learning in which it will be useful to lecturers. These modes include demonstation, exploration and design testing. The curriculum material is also being evaluated and revised.

This evaluation contains questions which are themselves also being reviewed.

2. INSTRUCTIONS

The Evaluation Sheet is divided into three sections. The first section is this introductory page. This includes general information. The second section has questions about the actual program **SOFTWIRE.** The third section examines the content of the curriculum material and the final section asks for comment on the actual Evaluation Sheet.

You are asked to fill in the questions in the relevant sections, which are attached to this sheet. Fill in only the questions which apply to you. Do not feel restricted by the questions, but make extra comments where necessary.

3. BACKGROUND OF REVIEWER

Name _____

Past qualifications _____

Present pursuits/career:

(undergrad. student/lecturer/Postgrad studentt etc)

_____

SOFTWIRE PROGRAM REVIEW

1. BACKGROUND DETAILS

Date of review _____

Place of review _____

Approximate hours spent using program _____

Reviewer's previous experience (state level)

Electronics _____

Computers _____

Maths _____

Analog computers _____

Digital Simulation _____

2. GENERAL QUESTIONS

What (in your opinion) is the aim of the package _____

_____

_____

What level (or range of levels) do you think it is aimed at:

Junior School _____

High School _____

Junior Undergrad _____

Higher undergrad _____

Graduate _____

Technikon _____

Other (specify) _____

What do you see as pre-requisite skills required (i.e used a computer before etc) _____

_____

3. EVALUATION TABLE

The following questions allow six answer levels, ranging from **1** (very poor) to **6** (exceptional). The answer may also be qualified with three alternatives. These are: 'not applicable', 'not enough information to say', 'not qualified to evaluate'. Each question also has 4 levels of "how appropriate is this question to the subject matter"

6 Exceptional
5 Very Good
4 Good
3 Fair
2 Poor
1 Very Poor
A Item not applicable
B Insufficient information, can't evaluate
C Not qualified to evaluate

Indicate the importance of each feature for this case. Circle the appropriate numbers-

	A. DOCUMENTATION	
6 5 4 3 2 1 A B C	clarity of information in manual	4 3 2 1
6 5 4 3 2 1 A B C	completeness of instructor guides	4 3 2 1
6 5 4 3 2 1 A B C	adequacy of instructions for actually operating SOFTWIRE	4 3 2 1
6 5 4 3 2 1 A B C	overall quality of manual	4 3 2 1
	B. SUPPORT OF THE TEACHING PROCESS	
6 5 4 3 2 1 A B C	ease of integration with course procedures	4 3 2 1
6 5 4 3 2 1 A B C	potential for improving instructor's ability to communicate principles and theories	4 3 2 1
6 5 4 3 2 1 A B C	potential for improving instructor's ability to communicate methods and techniques.	4 3 2 1
6 5 4 3 2 1 A B C	potential for teaching how to interpret and apply results	4 3 2 1
6 5 4 3 2 1 A B C	overall instruction ability	4 3 2 1
	C. FRIENDLINESS	
6 5 4 3 2 1 A B C	error messages clear enough	4 3 2 1
6 5 4 3 2 1 A B C	error messages specific enough	4 3 2 1
6 5 4 3 2 1 A B C	obvious what input is required next (always know what is being asked for)	4 3 2 1
	D. STIMULATION OF STUDENT INTEREST	
6 5 4 3 2 1 A B C	potential for capturing student interest	4 3 2 1
6 5 4 3 2 1 A B C	challenge to student creativity	4 3 2 1
6 5 4 3 2 1 A B C	student choice in patterns of use	4 3 2 1
6 5 4 3 2 1 A B C	appropriateness for student initiated work	4 3 2 1
6 5 4 3 2 1 A B C	overall contribution to student motivation	4 3 2 1
	E. COMPUTER TECHNIQUES/MATERIALS	
6 5 4 3 2 1 A B C	soundness of computer programming methods	4 3 2 1
6 5 4 3 2 1 A B C	completeness of technical documentation	4 3 2 1
6 5 4 3 2 1 A B C	portability (for different IBM-PC configurations)	4 3 2 1
6 5 4 3 2 1 A B C	ease of use	4 3 2 1
6 5 4 3 2 1 A B C	overall quality of computer techniques	4 3 2 1
6 5 4 3 2 1 A B C	easy to add one's own modules to the program	4 3 2 1

OVERALL EVALUATION OF PACKAGE

4. General Comments on package

Do you have any general comments on **SOFTWIRE** _____

_____

_____

_____

_____

SOFTWIRE TUTORIAL REVIEW

1. BACKGROUND DETAILS

Name of the tutorial reviewed _____

Date of review _____

Place of review _____

Approximate hours spent using tutorial _____

Reviewer's previous experience (state level)

Electronics _____

Computers _____

Maths _____

Analog computers _____

Digital Simulation _____

2. GENERAL QUESTIONS

What (in your opinion) is the aim of the tutorial _____

_____

_____

What level (or range of levels) do you think it is aimed at:

Junior School _____

High School _____

Junior Undergrad _____

Higher undergrad _____

Graduate _____

Technikon _____

Other (specify) _____

What do you see as pre-requisite skills required (i.e used computer before etc) _____

_____

3. EVALUATION TABLE

The following questions allow six answer levels, ranging from **1** (very poor) to **6** (exceptional). The answer may also be qualified with three alternatives. These are: 'not applicable', 'not enough information to say', 'not qualified to evaluate'. Each question also has 4 levels of "how appropriate is this question to the subject matter"

6 Exceptional
5 Verry Good
4 Good
3 Fair
2 Poor
1 Very Poor
A Item not applicable
B Insufficient information, can't evaluate
C Not qualified to evaluate

Indicate the importance of each feature for this case. Circle the appropriate number:-

A. INSTRUCTIONS		
6 5 4 3 2 1 / A B C	clarity of instructions in manual	4 3 2 1
6 5 4 3 2 1 / A B C	completeness of instructions	4 3 2 1
6 5 4 3 2 1 / A B C	adequacy of instructions for actually going through the tutorial	4 3 2 1
6 5 4 3 2 1 / A B C	factual accuracy	4 3 2 1
6 5 4 3 2 1 / A B C	overall quality of tutorial instructions	4 3 2 1
6 5 4 3 2 1 / A B C	ease of understanding of the subject matter	4 3 2 1
B. SUPPORT OF THE TEACHING PROCESS		
6 5 4 3 2 1 / A B C	ease of integration with course procedures	4 3 2 1
6 5 4 3 2 1 / A B C	potential for improving instructor's ability to communicate principles andd theories	4 3 2 1
6 5 4 3 2 1 / A B C	potential for improving instructor's ability to communicate methods and techniques.	4 3 2 1
6 5 4 3 2 1 / A B C	potential for teaching how to inteerpret and apply results	4 3 2 1
6 5 4 3 2 1 / A B C	overall instruction ability	4 3 2 1
C. STIMULATION OF STUDENT INTEREST		
6 5 4 3 2 1 / A B C	potential for capturing student interest	4 3 2 1
6 5 4 3 2 1 / A B C	challenge to sttudent creativity	4 3 2 1
6 5 4 3 2 1 / A B C	potential for forcing the student to THINK about the work	4 3 2 1
6 5 4 3 2 1 / A B C	student choice in patterns of use	4 3 2 1
6 5 4 3 2 1 / A B C	appropriateness for student initiated work	4 3 2 1
6 5 4 3 2 1 / A B C	adequacy for a large class	4 3 2 1
6 5 4 3 2 1 / A B C	overall contribution to student motivation	4 3 2 1

OVERALL EVALUATION OF TUTORIAL

4. TUTORIAL REVIEW

What was the instructional intention of the tut (e.g to make one familiar with Fourier synthesis) _____

_____

_____

Is the intent an important one _____

_____

Does the program achieve its educational aim _____

_____

_____

Is the tut too detailed/not detailed enough _____

What was the actual subject matter covered _____

_____

is it long enough to leave the user with useful knowledge

_____

ENVIRONMENT

How much supervision is required _____

Does it cater for differences in ability in a class___

_____

How many people recommended per terminal _____

_____

Do students have to have useful/interesting aspects pointed out to them before it is interesting _____

_____

5. GENERAL COMMENTS ON TUTORIAL

Do you have any general comments on the tutorial _____

_____

_____

_____

_____

REVIEW OF THIS EVALUATION SHEET

Do you have any comments about the way this questionnaire was compiled with regard to :

Ambiguity of questions _____

Thoroughness of questions _____

Redundancy of questions _____

Space to fill things in _____

THANK YOU FOR YOUR HELP