THE UNIVERSITY *of* EDINBURGH

MICROCOMPUTER BASED SIMULATION

by

Andrew Haining, B.Sc.

Doctor of Philosophy
University of Edinburgh
1981

ABSTRACT

Digital simulation is a useful tool in many scientific areas. Interactive simulation can provide the user with a better appreciation of a problem area. With the introduction of large scale integrated circuits and in particular the advent of the microprocessor, a large amount of computing power is available at low cost. The aim of this project therefore was to investigate the feasibility of producing a minimum cost, easy to use, interactive digital simulation system.

A hardware microcomputer system was constructed to test simulation program concepts and an interactive program was designed and developed for this system. By the use of a set of commands and subsequent interactive dialogue, the program allows the user to enter and perform simulation tasks. The simulation program is unusual in that it does not require a sophisticated operating system or other system programs such as compilers. The program does not require any backup memory devices such as magnetic disc or tape and indeed could be stored in ROM or EPROM. The program is designed to be flexible and extendable and could be easily modified to run with a variety of hardware configurations. The highly interactive nature of the system means that its operation requires very little programming experience.

The microcomputer hardware system uses two microprocessors together with specially designed interfaces. One was dedicated to the implementation of the simulation equations, and the other provided an input/output capability including a low cost CRT display.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. J.R. Jordan and everybody else from the Department of Electrical Engineering and the Institute of Terrestrial Ecology without whose help this project would not have been possible.

CONTENTS

1.1 Motivation

The advent of large scale integrated circuits and the development of microprocessors has meant that considerable computing power is now available at low cost and with small size. Simulation is a powerful tool used for investigation and prediction in a wide variety of fields including engineering and natural resources. Digital simulation of continuous systems has in the past been only available on large and expensive computer installations.

The aim of this project was therefore to investigate the possibilities of constructing a microprocessor based simulation system for the digital simulation of continuous systems. This proposed system would provide a useful level of simulation facilities at low cost. A small single-user interactive system should enable a user with little or no programming experience to quickly develop and use simulation models. The original area for simulation problems whose consideration led to this project, was the natural resource field. A considerable number of problems in this field have a large number of relatively simple equations. This means that although the individual equations are quite easy to simulate, their large number combined with the long time scales generally found in these problems, require considerable computing time. With a conventional time sharing computer system these simulation models can be very expensive to develop and use, especially if an interactive system is used. An interactive simulation system greatly speeds the development of the required models by both providing the user with a feel or insight for the problem and enabling the user to terminate simulation runs when

the model is obviously producing erroneous results. A microprocessor based system offers the advantage of interactive use with low initial and running costs. The microprocessor will be much slower in computations than a mainframe computer, but the overall simulation system response time might not be significantly slower than a busy multi-access system, and has the advantage of being always available.

The simulation system developed and described in chapters three and four, allows a user with little computing experience to develop and use models in a highly interactive mode. Printed and graphic output is provided together with data input facilities. The size of the models usable is dependent on the amount of memory the system has, and is easily extendable.

1.2 Microcomputer System

A microcomputer is just a very small computer which uses a microprocessor as its Central Processing Unit (CPU). The microprocessor performs arithmetic and logical functions, and controls the flow of data to and from the other components which make up the microcomputer. The most basic microcomputer system which could be used for even the simplest simulation tasks would require not only the microprocessor, but also sufficient memory to store the simulation program and its associated data together with a two way interface to allow the user to communicate with the simulation program.

The type and amount of memory required depends on the type of simulation system. There are two basic types of main memory, these are Read Only Memory (ROM) whose contents are permanently fixed and can therefore be used to hold important programs like

monitors and interpreters, and read write memory usually called Random Access Memory (RAM) which can be used to hold temporary programs and data. All variables and results produced while a program is running must be stored in RAM and since microcomputers almost exclusively use semiconductor memory which is volatile, then they are lost if the power is switched off unless they are stored in secondary, or backup, memory. For simulation results this is generally quite acceptable, but for the programs themselves this can be very inconvenient as they have to be loaded up before each session. An alternative is to store the programs in some form of read only memory like Erasable Programmable Read Only Memory (EPROM) which can be wiped clean and reused. ROM and EPROM are fine providing the data need not be changed, so they are more suited to the systems programs and look-up data tables.

In order to get the microcomputer to do any useful work, the user must be able to interact with it, to tell it what to do and to follow its progress. The most common and indeed most flexible way for this is by using an alphanumeric keyboard and display. The display could be either a Visual Display Unit (VDU) or a printer. Alternative input and output devices, like joysticks and oscilloscopes, can be advantageous for some speciallized applications.

No present microprocessors can perform floating point arithmetic directly, and most cannot even perform integer multiplication and division. Therefore subroutines must be provided for the level of arithmetic required for any given application. Integer arithmetic can be used in multiples of the microprocessors word length to provide any accuracy, but for general purpose simulation

where a wide dynamic range is needed together with moderate accuracy, then integer arithmetic becomes very expensive in memory space and the more compact floating point can be used. Special purpose hardware can be used to relieve the arithmetic subroutines of some of their calculations.

To develop simulation programs of reasonable complexity, several development and debugging aids are necessary in order to ensure that the time and effort involved in producing the program is not out of all proportion to the benefits derived from it. Typical programs that may be required are monitors, editors, assemblers, interpreters and compilers. To enter the program into the microcomputer, either for development or use, some form of secondary storage is required with the appropriate loading facilities. Secondary storage methods commonly used are paper tape, and magnetic media such as cassettes and floppy discs.

1.3 Natural Resource Applications

The original motivation behind the microprocessor based simulation system arose from an examination of a simulation problem concerning the water flow in a forest. The problem concerned both the internal water flow pathways inside the forest, including the transport mechanisms within the tree itself, and the net effect of the forest on the surrounding area. The Institute of Terrestrial Ecology has been measuring the development and functioning of a plantation of sitka spruce, and among these measurements were some of the factors which affect the water balance of the forest. The accurate measurement of environmental

factors is difficult to achieve because of the extremes of weather and the long timescales over which the measurements are required. The measurement of physiological details of the trees themselves is even more difficult due to the small quantitative changes involved, and their long timescales. The initial need for simulation would be to allow the action of various theoretical models to be examined, so that scarce resources could be allocated to measuring the most useful parameters. The experimental results could then be used to generate new models, and the whole process would be repeated.

Another use of simulation would be to produce a model of the gross action of the forest, at all stages of development, so that its effect on possible planting sites could be examined. The changes in the water flows caused by the afforestation of a tract of land is very important because the sites suitable for the forest are often reservoir catchment areas.

Due perhaps to the difficulties of collecting accurate data, the model equations resulting from research are in themselves quite simple. However the large quantity of equations needed for an overall water flow simulation and the long timescales over which the model has to be run, combine to produce a simulation model which requires a lot of computing time. The cost of such computing time using a mainframe computer to organisations like the Institute of Terrestrial Ecology, which have very scare resources, means that such simulation models cannot be used to their maximum benefit. A microprocessor based simulation system would however have a very low running cost, and its interactive facilities would also help to reduce the effort and cost of developing the models

in the first place. The slower calculation speed of the microprocessor need not be a disadvantage since, after the user has checked to see that the model is functioning properly, the system could be left overnight to produce the results.

1.4 Engineering Applications

Simulation is a very useful tool in all fields of engineering but, because of the effort required to run a simulation, it has generally been reserved for problems which are difficult either to analyze or to test practically. Small systems, especially electrical and electronic analogue circuits, are quite often designed using a simplified theory and then modified to obtain the correct operation after being built and tested. Although inelegant, this method is often the cheapest and most cost effective. However if a cheap and easy to use simulation system was available, then simulation could be used effectively to reduce the required testing and modification stages, as well as allowing the designer to explore a greater variety of possible solutions. For most problems a graphic output would be preferable, with only a relatively low accuracy result being required.

Another area where simulation could be used profitably is in the teaching of engineering, and in particular analogue electronics. The conceptual difficulty in relating theory to experimental work hampers a student when the theory has to be applied to a real problem. The use of simulation to illustrate the theory could help to break down the conceptual difficulties and enable a student to accept the theory more easily. The need

here would be for graphic presentation of the simulation results which could easily be related to the oscilloscope waveforms encountered in the laboratory.

1.5 Numerical Integration

The present version of the SIMUPROG simulation system program does not provide a built in integration routine, but instead allows the user to implement the most suitable method for the problem under consideration. While this does provide great flexibility, a built in integration method or methods could usefully be included in expanded versions of the system.

Considering a first order differential equation, the numerical integration methods suitable for solving the equation fall into two main categories. These are the single step methods, such as Euler and the Runge Kutta families, and the multistep methods like the Adams Bashforth predictors and Adams Moulton correctors. A full explanation of these and other methods has been given by Gear[1].

Considering a discrete time step h, the value of the variable y at time t+h can be expressed in terms of the values of its derivatives at time t by the following Taylor Series expansion.

$$y(t+h) = y(t) + h*\dot{y}(t) + \frac{h^2}{2} *\ddot{y}(t) + \frac{h^3}{3} *\dddot{y}(t) + ...$$

An integration method is said to be of order N if it produces component terms which match the first N+1 terms of the above series.

Higher order differential equations can either be broken down into a set of first order ones, or in some cases treated directly.

Gear[1] gives examples using the truncated Taylor Series expansion directly and also using the Nyström formula for special second order equations.

The accuracy of any simulation run is dependent on both the integration method and the accuracy of the arithmetic. The arithmetic performed on the GIMINI microcomputer used 32 bit floating point representation, with 23 bits for the mantissa, 8 bits for the binary exponent and one sign bit. Since the results of a calculation are truncated to get them back to the 23 bits, the worst case error produced is 2^{-23} times the exponent of the result. This error is therefore between 2^{-23} and 2^{-22} times the normalised result value. If it is assumed that the error is randomly distributed with a rectangular distribution, then the mean error will be 2^{-24} to 2^{-23} times the result value for each operation performed. For example after 10000 operations the truncation error could be expected to be between .0006 and .0012 times the result value. Another source of error is that the numbers entered into the system are only held with 23 bit significant accuracy. Thus more cumulative errors may be introduced if, for example, a value like the step size is inaccurate. The result of these errors can be amplified if unsuitable equations are chosen, i.e. if a small variation in a large value number is critical.

Ideally any integration method can be made more accurate by decreasing the step size, unfortunately, as the step size is decreased, the truncation errors become more significant until they are completely dominant. Each integration method will have its own error curve, and a typical example is shown in figure 1.1.

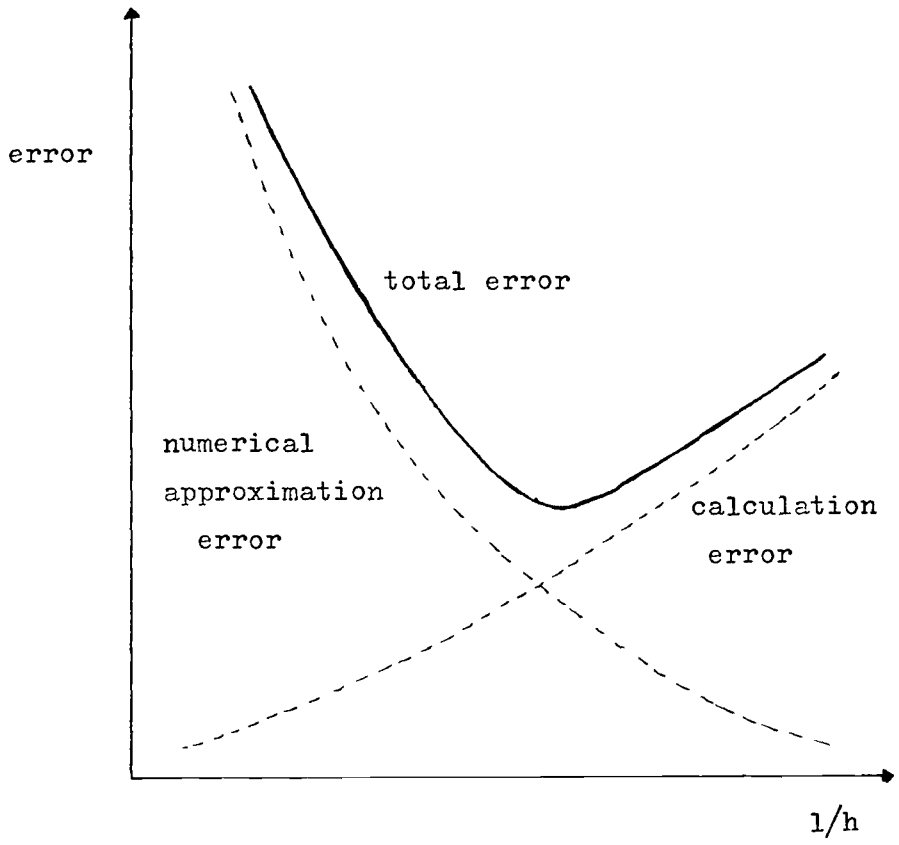


Figure 1.1 Integration Errors

The choice of integration method will depend on both the simulation equations and the accuracy of the results required. Euler integration is the simplest to implement, but being first order it is also the least accurate, so that a large number of steps using high accuracy arithmetic will be needed to produce a very accurate result. Also, because of the large number of steps, it may be slower to run. In spite of this, if a quick low accuracy solution is required, Euler may be very convenient. The one parameter family of second order Runge Kutta methods given by Benyon[2] can be rewritten as the predictor-corrector family given below.

$$y^p_{n+\alpha} = y_n + \alpha h \dot{y}_n$$

$$\dot{y}^p_{n+\alpha} = f(y^p_{n+\alpha}, t_{n+\alpha})$$

$$y_{n+1} = y_n + h((1 - \frac{1}{2\alpha})\dot{y}_n + \frac{1}{2\alpha}\dot{y}^p_{n+\alpha}$$

$$\dot{y}_{n+1} = f(y_{n+1}, t_{n+1})$$

When $\alpha = 1$ the above method is sometimes known as Heun's method or the Euler-trapezoidal method. The other second order Runge Kutta method tested in section 5 is obtained when $\alpha = 0.5$ and is sometimes known as the Improved Polygon method. These methods are quite easy to implement. Both require two derivative evaluations per step, but provide greater accuracy than Euler, since they are second order. James[3] gives the third order Runge Kutta also used in section 5. The classical fourth order Runge Kutta, which is also used, is described by Gear[1].

Multistep integration methods are not as convenient to use as single step methods, because at the start of a simulation run they have to be provided with past values of variables. These values can be estimated before the run and provided as initial conditions, although this normally means estimating a new set for each step size used. Alternatively a single step method can be used to provide the first few values to start the multistep method, but this has the disadvantage of requiring two separate integration methods. The predictor correctors such as the Adams Moulton require twice as many derivative evaluations per step as the predictor only methods such as the Adams Bashforth methods, but should provide better stability.

A comparison of relative computation time per step for various methods is given by Benyon[2], and although the arithmetic calculation times performed by the GIMINI software are not the same, the overall relationships appear roughly similar.

The use of an integration method with a variable step size could mean that less steps would be needed to attain a given accuracy, because the smaller step sizes would only need to be used for fast changing parts of the solution. Variable step size is difficult and not very satisfactory when used with predictor correctors, as pointed out by Martens[4]. Each time the step size is changed, the predictor corrector has to be restarted using a single step method. The Runge Kutta Merson fourth order method can generate an estimate of the integration error, sometimes called truncation error, for each step and so control the step size. For other methods, a possible way of generating an estimate of the error size is the

Richardson method explained by James[3]. In this method two solutions are generated at each step, one forthe full step size and the other using two half steps. The two values are then used to produce an error estimate. One disadvantage of using a variable step method apart from its complexity is that it is difficult to produce regular outputs of results.

Smith[5] describes the use of compensation to increase the accuracy of low order integrators. For a known result the compensation could be adjusted to reduce the error, but when the result is not known in advance, Smith does describe a variable phase integrator which applies the compensation as it proceeds.

2. SIMULATION

2.1 Continous Simulation

The simulation of continuous systems can be described as experimenting with mathematical models of a system instead of the system itself. Simulation can therefore be used both to predict the behaviour of a system and to aid the development of better system models. A continuous simulation problem generally involves the solution of a set of differential equations from initial conditions, and both analogue and digital methods can be used to perform these simulations.

When using an analogue computer an electronic circuit analogue of the required system is constructed from individual component circuit blocks. Parameters and initial conditions are set by potentiometers and the output is either on a Cathode Ray Oscilloscope (CRO) or x-y plotter. Analogue computers are fast, and the essential parallelism of their circuits means that their speed is more or less independent of the problem size, but they have a limited accuracy due to their components (0.01 to 0.05%[6]), and such high accuracy components are very expensive.

Simulation using a digital computer involves dividing the continuous solution into small discrete steps and using the values from the preceeding steps to estimate the result of the present step. Digital simulation can be performed to a very high accuracy using a general purpose digital computer, however since each equation of a parallel system has to be evaluated sequentially the speed of operation suffers. Hybrid computers have been developed which combine the speed of the analogue computer with the ease

of setting up of the digital computer, but they are very expensive and the close interaction between the user and the analogue computer is normally lost.

Analogue computer problems have to be scaled because of the limited analogue voltage range. Digital computers do not require scaling if floating point arithmetic is used, but do if the faster fixed point arithmetic is used. Since the user can manually alter the values of the parameters and initial conditions of an analogue solution and immediately see their effect, the user can obtain an insight or 'feel' for the simulation problem. Most hybrid computers and batch processing digital computers lose this benefit.

Programs run on large mainframe computers are either batch processed or run interactively. Since most interactive programs only use the computers Central Processing Unit (CPU) a fraction of the time, several programs are usually time shared to provide a multi-access facility. Most medium and large minicomputers can also run in both interactive and batch modes and are cheaper but less powerful than mainframes. Small minicomputers and microcomputers are normally used interactively for just one user.

Simulations can be written in a general purpose high level computer language like FORTRAN[7] or BASIC[8], but a low level assembly language would be considerably more difficult to use and to modify. Since most simulations have common features like integration and result output, a lot of programming time can be saved by using a simulation system which takes care of the repetitive functions. Such simulation systems fall into two categories, those which accept an explicit set of differential equations and those where the

model is entered by specifying the interconnections between functional blocks similar to those of an analogue computer. Both the equation oriented and block diagram systems can be used either interactively or in batch mode.

For a small digital system an interactive approach should be able to provide the analogue computer feel for the problem at a much lower cost than an analogue or hybrid computer. A batch processing system makes more efficient use of the computers central processing unit (CPU) than an interactive system since in the interactive mode the CPU will spend significant time waiting for the user to enter new commands. In a large computer system the CPU is an expensive part of the computer so time sharing is often used to cut this cost, but in a microcomputer system the CPU may only cost £10-£40 which is considerably less than most of the other system components. This means that the user's time is far more important than that of the CPU, therefore a microcomputer system is very suitable for interactive use with just one user.

2.2 Simulation Problem Implementation

To perform a simulation task interactively on a microcomputer three main actions were required, firstly the initial model must be entered into the computer, next the model must be implemented and results produced, and finally the model can be edited prior to being run again. To implement a simulation problem either a special program can be written in a suitable general purpose language, or a more specialised simulation language or system may be used.

Using a general purpose language for each simulation problem encountered implies that the programmer must have a good knowledge of the programming language used, and also implies that a good deal of the programmers effort is wasted in rewriting almost similarly functioning sections like data input and result output. High level languages offer the best approach for writing a special program as assembly languages require considerably more programming effort and greatly obscure the relationship between the program and the simulation problem. The high level program must be translated into the actions of the computer required to implement the specified task, and this may be performed either by a compiler or an interpreter. For interactive use on a small computer such as a microcomputer the interpreter approach is simpler, since it requires a smaller program than a compiler and does not need to store a machine code version of the program. On the other hand the compiler's machine code program is considerably faster in execution than the interpreter. While the high level compilers implemented on microcomputers are considerably more limited than those available on larger computers, they still require relatively sophisticated operating systems with secondary storage.

Several high level languages are available for microcomputer use, with BASIC being the most common. Most microcomputers have interpreted BASIC like the SWTP 6800[8] although some BASIC compilers have been produced. Various compilers using versions of IBM's PL/1 language have also been produced like PL/M[9] by Intel and the PLZ[10] family of languages produced by Zilog. The newer Pascal language[11,12,13] is gaining in popularity and by using the

intermediate P-code, both compilers and interpreters have been produced[14], and even a new microcomputer has been designed to operate on the P-code directly[15].

Korn[16] proposed a block diagram language in which a sequence of prewritten subroutines are called to perform any required task. The inputs and outputs of each subroutine called would be specified as parameters with the program stored in an array as threaded code. Threaded code means that the program is represented by the addresses of the subroutines and variables and is implemented by indexed subroutine jumps and indexed data operations respectively. The language Forth[17] uses indirect threaded code with a dictionary of subroutine blocks to implement the program. Forth is also unusual in that postfix or reverse Polish notation is used and that the user has full control of the stack. The threaded code operations refer to entries in the dictionary which contains the addresses and other details of the subroutines, this dictionary structure enables the programmer to define new operations using a combination of existing ones, and even other user defined operations. Forth is a very compact language, and with its reverse Polish notation and explicit processor operations including the use of assembly language it is obviously designed more for system use than general purpose programming.

Of the available high level languages BASIC is probably about the easiest to learn, but the more structured languages like Pascal allow the programmer to produce a program which is more closely related to the simulation problem and thus more comprehensible. The more complicated languages like FORTRAN[7] and the PL/1 variants

are more difficult to learn and are written to use quite a large microcomputer system with secondary storage. The reverse Polish notation and explicit stack manipulation could make Forth difficult to use for an inexperienced programmer. The main problem with using high level languages is that much of the programming effort is used in producing repeatedly used sections like result tabulation and integration. The presence of these sections in the users program also tends to obscure the simulation model. One solution to these problems could be to have prewritten sections or subroutines which perform the often used functions, but most functions like output and Runge Kutta integration cannot be performed by just one subroutine call. Therefore if simplified output and integration instructions are to be used, then the simulation program itself must be running under the control of another program which keeps track of the progress of the simulation problem and performs the actions required by the simplified instructions. Such a control program would form the basis of a simulation system. Korn[6] describes a package of FORTRAN subroutines which can be used on any computer with a suitable FORTRAN IV compiler. The model equations are written as a FORTRAN subroutine on punched cards and combined, with the required subroutines from the package, to form the simulation program which is then run in batch mode.

Since historically, digital simulation developed after analogue computer simulation was well established, many block diagram simulation systems have been written for mainframe computers which were designed to ape an analogue computer. DAS[18] and KALDAS[19] are examples of such systems which have attempted to produce a

digital version of the analogue computer, but since they are batch processed the analogue computer interaction or 'feel' is lost. PACTOLUS[20] is a similar block diagram system,but which can be operated interactively. These block diagram systems require that the mathematical models to be used are first rewritten as an appropriate block diagram, using analogue type components, before entering into the simulation system. The MARSYAS system[21] is a block diagram system designed for aerospace simulation on a large computer installation. The MARSYAS blocks can however be considerably more complicated than simple analogue computer blocks and can include high order transfer functions and complete subsystems which have been previously defined as block diagrams.

Equation oriented systems allow the simulation model to be entered algebraically in the form of differential equations. Most equation oriented systems conform roughly to the conventions of the Society for Computer Simulation's CSSL Committee[22]. One such system is DARE P[6] which is an equation oriented system designed for batch processing and written in FORTRAN IV for portability.

Interactive operation of large computers is very expensive, so some systems have been written for minicomputers of various sizes. The DARE/ELEVEN system[6], which was developed from earlier DARE systems at the University of Arizona to run interactively on a PDP-11/40 minicomputer, is unusual because it provides an equation oriented language together with a fast block diagram language. DARE/ELEVEN can accept an equation oriented CSSL type program which can be entered using an interactive editor. The system translates the simulation program into a FORTRAN program which is then compiled by the computers standard FORTRAN compiler to

produce an object code module. This module is then linked together with any required library routines to form the object program which is then run to produce the required results. The complete process is run under the control of the standard operating system on a PDP-11/40 with 28K of memory and a fast magnetic disc. Since the memory is not big enough to hold all the programs required,overlay techniques are used. Memory overlay is the process of overwriting selected parts of the computers memory with new programs. The previous programs are lost, but copies of all the programs are held on the magnetic disc. The DARE/ELEVEN block diagram language uses fixed point arithmetic for speed, and can even be used in different parts of the same simulation as the equation oriented language. The block diagram is entered by specifying the required blocks together with their inputs and outputs. The DARE/ELEVEN system then orders the blocks in procedural order and uses the computers own macro assembler to produce the object code.

Micro-DARE BASIC/RT 11[23] is a block structured simulation language similar to that used in DARE/ELEVEN which can be used on a PDP-11 or LSI-11 with at least 16K of memory and the RT 11 operating system, but needs no system disc. Micro DARE BASIC/RT 11 uses a dialect of BASIC for initialisation and control of simulation runs, and a block diagram language with fixed point arithmetic for the integration loop. The block diagram language is compiled into a threaded subroutine structure and the BASIC statements are semicompiled on entry to compress them, with an expansion routine provided to facilitate editing.

ISL-8[24,25] is a block diagram language which runs on a PDP-8

minicomputer with as little as 4K words of memory. The fixed point arithmetic used in ISL-8 requires time and amplitude scaling. Newer versions of ISL[26] can be used with other minicomputers and for hybrid simulations.

BEDSOCS[27] is a CSSL specification equation language which uses BASIC instead of FORTRAN as its procedural language. By using interpreted BASIC the system is designed to run on a Hewlett-Packard 2100A minicomputer with 8K of core store.

Hay[28] describes three interactive simulation system implementations on a PDP-8 minicomputer with 28K words of fast memory, a magnetic disc and a floating point processor. All three implementations provide equation oriented simulation languages. One scheme makes extensive use of the PDP-8 system programs like editor, monitor and FORTRAN compiler. The simulation program is translated into FORTRAN statements to be compiled by the FORTRAN compiler into the required object code. The PDP-8 batch processor, under the control of a command file is used to provide the interactive facilities, and extensive overlaying is required. The other two approaches use self contained programs, one converts the simulation program into an intermediate code which is interpreted at run time, and the other has a further stage of compilation to produce a machine code program. More recent versions of the interpreter implementation called ISIS[29,30] are written in FORTRAN and can be used on a variety of computers. ISIS conforms to the CSSL specification but does not translate the simulation program into a procedural language like FORTRAN as most CSSL systems do, instead it checks and semicompiles each line as soon as it has been

entered. The semicompiled form is stored for later translation into its interpreter instruction code at the start of a run.

SIMEX[31] is an equation oriented language which was written for a PDP-9 minicomputer with 24K words of core memory and high speed bulk storage. Instead of compiling the complete simulation program just before it is run as most equation oriented systems do, SIMEX compiles each line separately as it is entered. The system provides for interactive editing by storing the source code on the bulk storage medium. To obtain a fast calculation speed fixed point arithmetic is used, and by restricting the time steps to powers of two, and using Euler integration, the system can perform the integration calculations using just shifts and additions. An automatic scaling system is implemented to relieve some of the burden of scaling the simulation problem, and this uses scale factors of powers of two so that only shifts are needed.

For a user inexperienced in computer programming, writing simulation programs in a purely high level language has two main drawbacks. The first is the need to learn the particular high level language used, and the second is the need to know the appropriate programming techniques required for result output and interactive control of the simulation. Using a simulation system relieves the second and most important drawback. Of the systems described only DARE/ELEVEN, BEDSOCS, ISIS and SIMEX provide high level type languages designed for simulation work. DARE/ELEVEN and ISIS provide FORTRAN features for more advanced simulation programs and BEDSOCS provides BASIC. Only BEDSOCS and SIMEX are core resident programs as the others require overlay techniques,

but SIMEX does require fast secondary storage for source programs.

Block diagram language systems are easier to implement on a small computer since the code to be executed is already written as blocks or subroutines and only the interconnections need be specified. This also means that no source programs need be stored for the execution phase. Micro-DARE/RT11 and ISL-8 provide block diagram languages which do not require fast secondary storage, and Micro-DARE/RT11 also provides BASIC statements for initialisation and control of simulation runs.

A problem with block diagram languages is that users unfamiliar with analogue computing have to learn to convert their problems from the familiar mathematical expressions to an unfamiliar block diagram representation of the problem. Therefore a simple microprocessor based system would preferably be of the equation oriented type.

The previously mentioned equation oriented systems all require the simulation problem defined as a program using the systems special language usually based on a high level programming language. This means that not only does the particular special language used have to be learnt, but the system has to store a copy of the source program for future interactive editing. If instead of this a highly interactive dialogue could be maintained between the computer and the user, then the simulation system could at least partly explain how to enter the required information. Auslander[32] describes two implementations of a structured data simulation system. Both run on a PDP-7 minicomputer with 8K words of memory and instead of writing a special program the simulation model information is entered

using a series of commands. The two implementations differ in the data structure used to store the models, but both require that the model equations are entered in a very sty-lised format with nonlinear functions implemented as machine code subroutines. One implementation uses a linked data block structure with directories used to reference blocks of information about state variables. As well as the names and values of each variable, the blocks also contain a set of pointers to other variables which make up the stored equation. The other implementation uses a node incidence matrix to describe the topology of a directed linear graph. Each equation stored in this system can be represented by a simple directional graph, and the node incidence matrix contains the binary information about the presence or absence of each possible individual branch in the graph.

The command structure used by Auslander does not offer much dialogue so the user has to remember what information has to be entered. The entry format for the equations is awkward and involves extra user effort, with the resulting equations being rather restricted.

It was therefore decided to implement a microprocessor based system which was equation oriented and had the simulation problem held as recoverable data. The system should also have as much interactive dialogue as possible to guide the user in using the system.

2.3 Microcomputer Considerations

2.3.1 Man-Machine Interface

The methods of communication which the user has with the simulation system are very important since the system has to be

easy to use for people with little experience of computers. The entry of data and control instructions should be straightforward, and the microcomputer should be able to output helpful guidance as well as results. An alphanumeric keyboard together with an alphanumeric printer or Visual Display Unit (VDU) is the most common and versatile method used to provide interactive control of a computer. Other input and output devices can provide very useful extra facilities, but they are not absolutely necessary. So that the minimum microcomputer system implementation can be usefully used, the simulation system program was designed so that the alphanumeric input and output devices could provide data input, result output and simulation control. Alternative data input and result output devices were also catered for, but the simulation system can still be used if they are not available.

Difficulties occuring in the communication between user and microcomputer program are caused by the organisation of the interactive dialogue and not by the hardware involved. The present simulation system uses a teleprinter as the main user interface. While the slow speed of 10 characters per second is fine for user input, it greatly limits the amount of text which can be printed out without the user becoming impatient. While an inexperienced user will tolerate lengthy printouts which provide helpful guidance, these printouts will become annoying and timewasting to a user who already knows how to operate the system. A way round this would be to switch between short rather cryptic dialogue and expanded explanatory dialogue depending on the users preference. Useful as this method could be, the memory limit in the present system prohibits its use.

Therefore a compromise was reached for the present implementation so that only one set of dialogue was needed. The function of the simulation system was broken down into a selection of commands, the present version of the program has 20 and each consists of a name or a mnemonic up to 4 letters long. When a command has been activated, the required function is implemented using a controlled dialogue between the user and the program. The simulation model and control information is stored in a unified database and each command has independent access to the appropriate parts of the database. Several formats for the controlled dialogue were tried for different commands.

The use of a higher speed VDU for the text output would remove the speed limit from the size of the program messages, and would also allow alternative dialogue formats to be used. An example of this would be the use of a menu of the command names, a brief explanation of each command could also be included.

2.3.2 Arithmetic

For general purpose use, the simulation system should be able to accept a wide range of input values so that the user does not need to spend extra time and effort scaling the simulation problem. This wide dynamic range must be combined with sufficient accuracy to ensure that the cumulative errors remain within the required limits, even after a large number of steps.

Integer arithmetic is the simplest to implement, but very long integers would be needed to cope with the wide dynamic range needed and this would require a large amount of storage, most of which would

be wasted since the great majority of the values would contain either leading zero's or else more bits of information than the initial accuracy of the variables could justify. Scaling techniques can be used to reduce the size of the integers required, Gakhal[33] describes some scaling methods for some discrete fourier transforms. However for the simulation system, the variety of types of equations that can enter means that an automatic scaling system would be very complex and difficult to implement. Errors in solving differential equations can be reduced by increasing the accuracy of the numerical integration calculations. This can be done by either holding the state variables with a higher precision, or else using residue retention as described by Baker[34], however this would complicate the simulation system since the program would have to detect and keep track of variables being integrated.

Floating point arithmetic can provide both a wide dynamic range and sufficient accuracy for the simulation system. Floating point arithmetic software is slower and more complex than integer software, but is no more difficult to use once the software has been written. An alternative number system suggested by Edgar[35,36] called FOCUS uses a logarithmic representation which is claimed to be significantly faster than floating point for the same accuracy. While multiplication and division are easy in FOCUS, addition and subtaction are slower and more complex. The input and output conversion is also more complex and FOCUS would probably require more memory than floating point. For real time systems where speed is at a premium and calculation errors cease to become significant after

a relatively few number of steps, the FOCUS number system with input and output conversion hardware could be useful.

Arithmetic hardware can both reduce the amount of software and speed up the arithmetic operations. Integer multiplication and division can be used to speed up both integer and floating point arithmetic, and can use either Read Only Memories or digital hardware. The recent introduction of single chip floating point processors means that floating point arithmetic is much easier to implement. The National Semiconductor 57109[37] can provide BCD floating point arithmetic together with trigometric, logarithmic, and exponential functions. This device however uses PMOS technology and is actually significantly slower than the floating point software used with the CP 1600 microprocessor. The Advanced Micro Devices AM9511 arithmetic processing unit was only available at the end of this project, and can provide 16 and 32 bit integer arithmetic as well as floating point arithmetic including trigonometric, logarithmic and exponential functions. The Am9511 uses binary floating point and is faster than the CP 1600 binary floating point software which was actually used for the simulation system. The 32 bit floating point was felt to be the optimum for the simulation system as 16 bit floating point could not provide the accuracy and dynamic range needed, and both 48 and 64 bit floating point would have taken more storage and would also have slowed the simulation system down unnecessarily.

2.3.3 Languages

The choice of language used to write the simulation system program depended not only on the need to produce an efficient and

comprehensible program, but also depended on the physical limitations of the microcomputer system itself. While small programs can usefully be written directly in machine code, it is impractical to write and debug a large complex program without either assembly or high level language program aids.

Assembly language provides mnemonic representation of the microprocessors machine code instructions together with labels for program jumps and variable storage. Programs written in assembly language are much easier to understand than pure machine code because the mnemonic labels and instructions, together with the comment text, allow the program structure to be discerned more clearly. The assembly language program is translated into a machine code program which is then stored to be run later. Even though each assembly statement directly represents one machine code instruction, they cannot be translated in isolation because of references to labels appearing later in the program. The assembler overcomes this by making two or more passes through the program, the first pass builds up a table of the addresses of the labels used in the program and subsequent passes are used to generate the required machine code program.

High level languages are easier to understand than assembly language because each high level statement is the equivalent of several machine code instructions, and can therefore be designed to perform a function much closer to the programmers concept of the basic operations from which the program is constructed. The high level program can be translated into machine code by a compiler which, although more complex, operates in a similar manner to the assembler. An alternative method of implementing a high level

language is to use an interpreter. The interpreter does not produce machine code, but instead it examines each line of the program as it is being run and performs the same actions as the user's program would have if it had been compiled. An interpreted program is therefore considerably slower than a compiled one, but the interpreter does have the advantages that it is easier to use and does not need to store a machine code program. Therefore in programs involving lengthy loops, such as simulation programs, the calculation time taken by an interpreter could be at least an order of magnitude greater than a previously compiled program would take.

Assemblers and high level language compilers can be run either on the microcomputer itself or on another computer altogether. To run an assembler or compiler on the microcomputer itself requires secondary storage for the source and object programs since it would be prohibitive to hold them in main memory. If a cross-assembler or cross-compiler is run on another computer, the microcomputer only needs enough memory for the resulting machine code simulation system program. Compilers are more complex and require more memory than assemblers. At the start of this project very few high level languages were available for microprocessors, and even these required at least a floppy disc drive for secondary storage. By exploiting fully the architecture of the microprocessor, a program written in assembly language should be smaller and faster than the alternative high level program, but the programming effort needed to produce the assembly language program would be much greater. A way to combine the two approaches could be to use a high level language which allowed critical program segments to be written in assembly language,

however there still remains the problem of integrating the actions of the high and low level sections. The CP 1600 superassembly language, finally used to write the simulation system program, overcomes this problem by providing high level programming enhancements to an otherwise standard assembly language. Although the superassembly language is not as sophisticated as a true high level language, it makes it much easier to make full use of the architecture of the microprocessor with little more requirements than a standard assembler.

2.4 Microcomputer System Implementation

The limited memory size and relatively slow speed of the microprocessor based system envisaged means that the simulation model equations need to be stored in as compact a form as possible, without sacrificing execution speed. Since the equations which will be entered into the system do not have a fixed format and can make use of several layers of parenthesis, an equally flexible system is required to store the equations so that they form a suitable instruction stream for run time calculations. Lawson[38] describes the main types of instruction streams which can be used. Polish notation is probably the most compact form of instruction stream, and since the stack operation times, using the microprocessor stack pointer, are very much less than floating point calculation times, there is very little loss in execution speed. The trailing operator or reverse Polish notation (RPN) form of instruction stream is the easiest to evaluate, and so was chosen for equation s[illegible]age. To compress the equation further, the RPN instruction

stream was coded so that each entry consisted of a single 16 bit word. Equations entered into the simulation system can be represented as (variable)=(expression) where the '=' represents an assignment operation not an equality. An expression is coded using positive integers to represent variables whose values are to be pushed onto the stack, and negative integers to represent arithmetic operators and functions which operate on the stack. A positive integer refers to the position of the required variable in a list which, as well as containing the value of the variable, also contains its name so that stored equations can be printed out. The negative integer refers to the position of the starting address of the required arithmetic subroutine in a list which also contains the operator symbol or function name needed to print out the stored equation. The RPN expression is terminated by a zero entry and the variable to which the value of the expression, now held in the stack, is to be assigned is found in a separate list. The reason for this is twofold, firstly it simplifies the evaluation of the expression since it eliminates the need to enter the address of the unknown variable in the stored instruction stream, and secondly it allows for prewritten system functions and user entered functions to be implemented, in future versions, in the same way as model equations.

Since the entered equations are not of fixed length, the zero which terminates their equivalent stored instruction stream effectively terminates a variable length list. All the data specifying the simulation models is effectively held in lists, and this provides an effective method for storing variable amounts of data without

keeping track of separate pointers and counters. The set of model equations are themselves held in a list, in the order they were entered. To increase flexibility of operation, the order of the list of equations does not determine the order of operation, and instead the order of operation is specified by a separate list. Not only does the operation list mean that the order of operation can easily be changed, but equations can be dropped from the list or even appear several times without disturbing the original set of equations. The actual list of equations is held in a two dimensional array or matrix. Alternative storage organisations could be implemented to make use of the space otherwise wasted by equations shorter than the maximum length. Storing the equations as linked lists could make use of the space but since the only replacement operations required would involve whole equations, the extra pointers required to implement linked lists would make the system very inefficient. The individual equations could also be stored as segments in a single data space with an array of pointers indicating the start of each equation. This system would make better use of the memory space than the matrix method, and should be just as fast. The only main drawback would be that replacement of individual equations would be more difficult. The main reason that the matrix method was used was that it was easier to implement and debug, the segmented method could then be introduced later and debugging would be much easier when the rest of the program was validated.

All the information needed to run the simulation is included in the database including run time controls and output lists.

This means that the simulation run information is independent of the program and can be stored and rerun at a later date. The size of the database, and hence the maximum size of a simulation model, is limited only by the memory size of the microcomputer and can easily be adjusted to suit the available memory.

Each command used to control the simulation system is implemented as an independent subroutine so that commands can be added to or deleted from the system as required. Each command subroutine operates independently on the database although, to keep the program compact, several command subroutines may themselves make use of common subroutines. Once a command has been initiated, the user is guided through the command sequence by questions requiring yes or no answers and prompts for entering data. As well as a line editor for user entered input, the commands are designed with options so that an entry error by the user does not require too much effort to correct. The equation entry subroutines accept equations in normal algebraic form, with parenthesis, and converts them to the required reverse Polish notation form. Some checks are also made as to the validity of the equations, and a limited error diagnosis is produced before the user is asked to enter the equation again.

It was not felt appropriate to include facilities for automatic sorting of equations or for dealing with implicit loops.

3. MICROCOMPUTER SYSTEM

3.1 System Requirements

3.1.1 Microprocessor

There is now a wide range of microprocessors available, with differing capabilities and designed to suit a variety of applications. Since the microprocessor performs the required calculations and controls the other system components, it is the main limiting factor determining a microcomputers capabilities. The usefulness of a particular microcomputer configuration for a given application is determined by a variety of factors. The principle factors are the speed of the microprocessor, its address range, its instruction set, the type and size of memory used, the hardware and software experience and backup available, and the hardware configuration.

The simulation program will require a considerable amount of calculation, and the programs themselves may be quite lengthy. The microprocessor should therefore be able to address sufficient memory to hold the largest program required, although the use of overlay techniques, with secondary memory, can reduce the requirement. While just about any microcomputer can perform floating point arithmetic if properly programmed, the instruction speed and the word length of the microprocessor, combined with the design of its instruction set and architecture, will determine how fast the calculations are performed.

The most suitable microprocessors are the more recent 'general purpose' 8 and 16 bit single chip microprocessors. The smaller 4 bit single chip microprocessors and microcomputers are designed

for smaller and less memory intensive systems, and their limited memory space combined with the more complicated programs required for calculation and data manipulation means that they are not suitable for the simulation system. Older general purpose microcomputers using PMOS (P-channel MOS) and the earlier NMOS (N-channel MOS) devices are slower, and most also have interfacing disadvantages. Other specialised microprocessors usually have disadvantages, bit slice microprocessors can generally achieve a far greater speed than single chip devices, but they introduce another level of complexity since the actual instruction set to be used in writing the simulation programs has first to be designed and developed. This does not rule out their use, but it does mean that the fast speed has to be balanced against the extra time, effort and cost that they entail. A possible use of bit slice machines would be in a multimicroprocessor system where they could perform a small repetitive section of program at high speed. The single chip microcomputers have memory and input/output lines contained with the microprocessor in one integrated circuit. Those with ROM or PROM memory are designed for volume production, and even those with EPROM or only RAM usually have a limited address space and are not suitable.

Cost is a major factor in deciding on the hardware to be used, but the cost of the microprocessor itself is only a small part of the total hardware cost. Indeed when considering the development costs, unless for high volume production, the cost of the microprocessor itself is usually insignificant. The actual microprocessor used has however a large indirect effect on the costs, since it determines

the hardware needed for the system as well as the effort involved in developing the programs. The development costs can be divided into two main areas, hardware and software. Taking hardware first, the existence of compatable families of microcomputer components means that the construction of the required hardware configuration can be done without an extensive knowledge of digital hardware and computer techniques, providing that the constructor uses only the family components and that the documentation is adequate. Most microcomputer families are incomplete, so that often components from other manufacturers and components not specifically designed for use with a particular microcomputer are needed to attain the required configuration. An alternative to building a microcomputer from scratch is to buy a commercially available microcomputer system. Most of these systems which range from simple single board computers with hexadecimal keyboards and light emitting diode (LED) displays, to sophisticated systems with a visual display unit (VDU) and magnetic disc storage, are designed for the hobbies or small business markets. The resulting high volumes of sales, especially of the single board computers, means that the finished product is very competitively priced compared to the cost of the hardware components, without even considering the development costs to build the hardware. Therefore for the development of a simulation system, the most cost effective approach would be to use an available microcomputer and extend its hardware as required. However if appreciable quantities of systems are to be produced then a microcomputer system can be designed to suit the application. Another advantage of using available microcomputer boards, apart

from not having to design and debug the hardware, is that many of them are supplied with monitor programs in ROM or EPROM. Some of the newer single board microcomputers even have BASIC interpreters in ROM. Most of these microcomputers also have libraries of software available either from the manufacturer or independent software companies.

The main microcomputer used in the simulation system was the General Instrument Microelectronics CP-1600, which is a 16 bit single chip microprocessor. The CP-1600[39] described in more detail in section 3.2.2, was purchased as a complete GIMINI microcomputer system. The GIMINI microcomputer was chosen because it offered very good value for money at the time, when a limited budget was available. The GIMINI also has the advantages that all the system software was provided with the microcomputer, as well as local technical backup being available. The CP-1600 is an NMOS (N-channel MOS) single chip device and, with its regular architecture modelled on that of the PDP-11, its features were indicative of the trends in microprocessor evolution. Therefore the results and experience gained with the CP-1600 would be useful in considering future microprocessors. These trends have certainly continued with the introduction of the Z8000 and M68000 which employ very regular architectures with general purpose register sets. At the same time a smaller 8 bit microcomputer system was built to compare with the capabilities of the 16 bit machine. This microcomputer used a Motorola M6800 microprocessor and was based on their D1 development board. Later on in the project another small microcomputer was built, and this used the 8 bit Z-80. The later system was based

on the NASCOM-1 single board microcomputer which was designed for the hobbies market.

The tasks to be performed by a microprocessor in running a simulation system can often be reduced by extra hardware, such as a floating point processor, or additional microprocessors for input and output or parallel processing. If floating point hardware is used then a large portion of the arithmetic requirement is removed. Intelligent input/output devices and peripherals reduce the need for input and output lines from the main microprocessor as well as freeing it from most of the input and output control tasks, this results in a speeding up of the system's operation. Any interactive simulation system could be divided into the three main activity areas of, user interfacing, data handling for updating and using models, and calculations required for a simulation run. For user interfacing the input/output mechanisms of a microprocessor can effect both the hardware complexity and the software overheads. All the currently available 8 and 16 bit microprocessors can handle the input and output requirements of the simulation system, but the interfacing hardware needs to be designed specifically for a microprocessor in order to achieve the best efficiency. Data handling is an important requirement for the simulation system because its structure involves lists and pointers in an address space which can be varied according to the complexity of simulation tasks to be handled. Therefore address handling with arithmetic operations on the addresses over the full address range is required together with the ability to indirectly address data from a previously calculated address for several levels. The address range required

is dependent on the size of data base used, but for the system envisaged the address size would preferably be 16 bits which is normally the maximum address space of current general purpose microprocessors. An efficient subroutine call and return structure is an advantage because the program can then easily be divided into modules which make it easier to write, change and debug,if the modules are suitably written. Floating point arithmetic is used by the simulation system, and since no present microprocessors can directly perform floating point arithmetic, then special software has to be written. Recently hardware, in the form of an auxiliary arithmetic processor, has become available for use with microprocessors, but is still relatively expensive. This processor would however relieve the microprocessor of the actual calaculations, as it would then only be required to transfer data to and from the auxiliary processor. The simulation system implements its equations in reverse Polish form, so therefore requires a separate arithmetic stack from that used to hold subroutine return addresses. This stack need not be physically separate from the normal stack as long as the program can distinguish which quantity is which. The extra computing required to disentangle arithmetic data from return addresses would however add to the size of the program as well as reducing calculation speed.

Considering first the 8 bit microprocessors suitable for a simulation system. The Intel 8080[40] was the first of the NMOS devices to achieve popularity, and was designed to provide software compatability with their preceeding 8008. The Intel 8085[41] is basically an 8080 with hardware improvements including a single

supply rail and more interrupts. The Zilog Z-80[42] not only has software compatibility with the 8080, but also has additional registers, including two index registers, and extra instructions. The maintenance of upward compatibility from the 8008 has produced rather irregular instruction sets for the 8085 and Z-80. This together with the lack of certain instruction types, such as direct addition from memory to accumulator, means that they are more difficult to program than would otherwise be the case. The Zilog Z-80 with its much expanded instruction set and extra registers is more suitable for this application than either the 8080 or 8085.

The Motorola M6800[43] microprocessor has a more regular instruction set than the Z-80, including direct arithmetic operations, so is therefore easier to program. The 6800 also has the advantage of two accumulators, but does not have a set of general purpose registers like the 8085 and Z-80. The 6800, while lacking some of the Z-80's sophisticated instructions, has the ability to perform some operations directly on memory without requiring the use of the registers and this can often offset or even outweight the Z-80's extra registers. The Z-80 in common with the 8085 has separate memory and input/output buses, whereas the 6800 has a combined bus. This means that the 6800 performs input/ouput operations in an identical fashion to memory references, thus allowing the use of it's direct memroy operations. The 8085 does have some input/output devices designed to operate as memory, and the only disadvantage of this method is that, especially with partial address decoding, the maximum memory size is reduced, but with the smallish simulation systems envisaged this would not matter.

The MOS Technology 6502[44] microprocessor has a very similar instruction set to the 6800, but has only one accumulator together with extra addressing modes. Whereas the 6800, Z-80 and 8080 originally became popular through being provided as chip sets and sophisticated development systems, the 6502 has become well known because of it's use in built up microcomputers like the PET and the Superboard.

The Fairchild F8[45] seems to be designed primarily for control and other low memory requirement applications, as the lack of an on chip program counter complicates it's use in memory intensive situations. Although the 6800, Z-80, 6502, 8080 and 8085 are the most popular 8 bit microcomputers a variety of others are available. Some of these are slower like the Signetics 2650[46] and the General Instrument Microelectronics LP8000 whereas others have more limited instruction sets like National's SC/MP. This does not preclude their use but they may incur speed and programming penalties. RCA's COSMAC[47] microprocessor is unusual in that it's CMOS technology means a low power requirement, and it also has an unusual architecture with a set of sixteen 16 bit registers any one of which can be defined as a program counter and any other as a data pointer.

Some of the newer single chip microcomputers are being designed to cope with memory intensive applications, and although most are designed for high volume and use mask programmed ROM, some are also available as prototype versions with either EPROM or no ROM at all. If they are based on an existing microprocessor then there is unlikely to be any advantage.

Microprocessors like the Z-80 have some limited 16 bit data

handling capabilities which are designed primarily for address calculations. The Motorola 6809 is an attempt to bridge the gap between 8 and 16 bit microcomputers. The 6809[48] has the same architecture as the 6800 and shares its support devices, but has an extra index register and stack pointer together with more addressing modes and 16 bit arithmetic including multiplication.

Sixteen bit single chip microprocessors, because of their greater internal complexity, are more difficult to produce and consequentially have been slower to appear than the 8 bit machines. The General Instrument CP-1600 previously mentioned is modelled on the PDP-11 minicomputer but is not software compatable. The existence of large amounts of software written for minicomputers, and the increasing competition from the microprocessors produced by the semiconductor companies, has induced the minicomputer manufacturers to produce their own microprocessors which are software compatable with their minicomputers. The Texas Instruments 9900[49], which has the same instruction set as their 990 minicomputer, has an unusual memory to memory architecture with a set of working registers being maintained in RAM memroy instead of in the processor itself. This means that interrupts can be handled very quickly by moving to a new section of memory, but has the disadvantage that computation speed is dependent on the speed of the memory which is usually slower than internal registers. Digital Equipment Corporations's LSI-11 is not a single chip microprocessor but a multiple chip set which can use a lot of the software produced for the PDP-11. The Intersil IM6100[50] is a CMOS microprocessor with a 12 bit word length and the instruction set of the PDP-8 minicomputer. This means that

PDP-8 software can be run on the IM6100, but the programmer is limited to the PDP-8's rather primitive instruction set and architecture. The instruction set of the Data General Nova minicomputer is used for their own Micro Nova as well as Fairchilds 9400[51] microprocessor. The Nova, although not as well known in this country as the PDP-11, also has a large software base. The use of an existing minicomputer instruction set limits the architecture and features which can be added to a new microprocessor, and since established minicomputers generally were designed to be implemented in either discrete components or small scale integrated circuits they are unlikely to make the most efficient use of current large scale integration technology. Most semicondcutor manufacturers who are not also in the minicomputer business, have opted to design their own architectures and instruction sets. Motorola and Zilog have made a complete break with their previous 8 bit microcomputers in attempts to produce architectures which will serve them for a future series of upward and downward compatable microprocessors.

The Zilog Z8000[52,53] is a 16 bit microcomputer, in that its hardware handles sixteen bits in parallel, but it can handle data types of 8, 16, 32 and for some instructions even 64 bits as well as single bits. The Z8000 also has sixteen general purpose registers which can be used with most instructions as accumulators, data storage, index registers, or memory pointers. The Z8000 is available in two versions, one has a 40 pin package and can address 64 kilobytes of memory, and the other has a 48 pin package and provides for the use of 128 segments each of 64 k bytes to give a total memory space of 8 Me-gabytes. The Z8000 has two modes of

operation, each with its own stack pointer. The system mode has full control of the microprocessor, but the user mode has a restricted instruction set with input/output, interrupts, traps and mode changes being prohibited. If any of the prohibited instructions are used in user mode, a trap into system mode occurs which returns control to program using system mode. While this facility would be of no particular use in the presently implemented simulation system, any future expansion aimed at providing facilities for user written subroutines or procedures could greatly benefit from the protection afforded by this facility.

The architecture of Motorola's M68000[54] is designed to be 32 bits wide but is currently implemented as 16 bits. The M68000 has sixteen 32 bit registers, eight of which are designed primarily as address registers. Data of 1, 8, 16 and 32 bits can be accessed from a memory space of 16 Megabytes. The M68000 also has a privileged system mode. Some instructions, such as floating point arithmetic, are specified in the architecture but not implemented, and when these instructions are used a trap occurs which can be used to provide software implementations of instructions not implemented in hardware. Unlike Z8000 the M68000 does not have multiplexed address and data buses, but does therefore have a larger package.

The Intel 8086[55,56] is a sixteen bit microprocessor which can handle quanitities of 1, 8 and 16 bits, and which has eight 16 bit general purpose registers. Although most of the instructions can use all the registers, some of the addresssing modes are restricted to using certain registers and the multiply and divide instructions can only use the accumulator. The 8086 can address one megabyte

of memory and has a sophisticated memory segmentation system. The 8086, unlike the Z8000 and M68000, has been designed to retain compatability with the preceeding 8080, although hardware compatabiity including interrupts is not maintained. This limited compatability may be useful for upgrading previous 8080 programs, but the constraints it imposes on the architecture of the microprocessor, and the instruction set together with its mnemonic representation, are a disadvantage for writing new programs and could even add to the cost of developing new software.

3.1.2 Memory

The microcomputer's main memory holds the simulation programs together with any required data. Since the simulation program developed stores all the simulation model equations and structure as data, the actual program remains fixed when in use. While all the simulation data has to be held in read-write memory, except for fixed data or constants used by the program, the finished program could be stored in either read only memory (ROM) or in read-write memory which is more commonly called random access memory (RAM). Execution of the simulation program needs to be performed for two distinct purposes, firstly to develop and debug the program itself and secondly to use the program to perform simulation tasks. The development of a complex program like the simulation system program usually proceeds in a cyclic fashion by first finding that some feature of the program does not work, then debugging the program to find the cause, followed by rewriting part of the program to correct the fault, then testing the program to find a new fault

which was masked by the previous one, and so on until the program works properly. For this development the program has to be altered both to correct the errors and also, all be it temporarily, to help in the debugging itself. For the development phase the program will therefore have to be held in RAM.

Semiconductor RAM is almost exclusively used in microcomputers because of its low cost and high speed. RAM memories can be divided into two categories, static RAM which retains its information as long as power is supplied, and dynamic RAM which has to be refreshed every few milliseconds or else the information is lost. Dynamic RAM is cheaper than the static type, but normally requires extra circuits to perform the refresh, although the Z-80 microprocessor has the circuitry built in. Some manufacturers produce pseudostatic memories which are dynamic memories complete with refresh circuits on chip so that they act like static devices. Some memories have power down facilities whereby a small amount of memory can be kept active by standby batteries and therefore important information can be kept during power down. With CMOS memories, the power required by these devices when not being accessed is so small that sizable sections of memory can be kept active from small batteries. The penalty of CMOS memories are their greater cost, but in some systems they may eliminate the need for expensive secondary storage. A two kilobyte CMOS memory was implemented for the NASCOM-1 based system and this had optional write disables so that it could be used as an EPROM simulator for the development of the graphics program.

Dynamic memories, and to a lesser extent static memories, tend to suffer from occasional errors produced by alpha particles

or cosmic radiation, and most large minicomputers and mainframe computers employ error correcting codes when using dynamic memories. For the simulation system, the error rate for the size of memory used will be very low, indeed it is likely to be less than errors caused by faulty equations entered by the operator. Some microprocessor manufacturers produce memory specifically designed for a particular microprocessor, but these tend to be dearer than memory devices produced in larger volumes for more general use. These generalised memory devices, although they need extra decoding circuitry, are usually quite easy to interface providing that they are fast enough for the intended microprocessor operating speed.

ROM is available in a variety of forms for different uses. Mask programmed ROM has its information set at manufacture by a mask used to deposit an aluminium pattern on the integrated circuit, but is only suitable for volume production runs. Programmable read only memory (PROM) is programmed after manufacture and can therefore be used for one off designs, with the information being set by blowing or not blowing fuses with a special PROM programmer device. PROM's once programmed cannot be changed, but erasable programmable read only memories (EPROMS) can be erased using ultra violet light and then reprogrammed. The EPROMS use trapped charge in MOS devices to hold the information, which can be retained for many hundreds of years. Some other reprogrammable memories are available but offer no advantages for the simulation system and have higher costs.

The current development rate of semiconductor memories and their competitive pricing means that memory costs have fallen dramatically in the last few years, thus changing the balance of

hardware to software costs. This means that a greater program size can be tolerated if it cuts down the development cost of the software.

3.1.3 Secondary Storage

Secondary storage can provide non-volatile storage for large amounts of programs and data, at a lower cost per bit than main or primary memory. The low cost per bit is achieved mainly by the sequential nature of the secondary storage devices, this means that the read write electronics are for the most part independent of the amount of data stored and the data itself is stored on a low cost medium such as plastic with a magnetic coating.

For the simulation system developed in this project the main contender on the grounds of cost alone is the audio cassette recorder. In order to provide a reasonably short load up time for a RAM only microcomputer a data rate of at least 1200 baud would be needed. If the simulation system program was in ROM or EPROM then the lower 300 baud could be used for reloading just the simulation model data. The digital mini-cassette, with its facility for total computer control of the cassette drive, would be a contender for a more professional system. An extended simulation system with result storage and additional data processing programs could use floppy discs or bubble memories. The usefulness of the bubble memories is very dependent on the pricing of these memory systems, but they should prove cheaper than mini floppy discs for small quantities of storage. Although the mini floppy is slower than the standard one its lower cost would make it preferable, even if the required secondary storage capacity was greater than could be held in one minifloppy, as it

would be advantageous and only fractionally more expensive to use two minifloppys instead of a standard drive. The two drives would make it much easier to copy discs and thus provide backup copies for use in the event of disc failure, since floppy discs only have a limited life span in use.

3.1.4 Input and Output Devices

The full benefit of an interactive simulation system can only be obtained if the input and output facilities are easy to understand and use. Therefore control dialogue must be as unambiguous and informative as possible bearing in mind the constraints of the microcomputer system itself. Presentation of the results is very important and a visual presentation in graphic form would be very desirable.

There are two distinct areas of use for input and output devices, which are system program development and simulation system use. During program development editors and assemblers will be used so full alphanumeric input and output is required, with hard copy being almost essential. The situation is rather different when the simulation system is actually being used, since additional input and output devices could usefully be employed. Considering first the simulation system output, the user has to be guided through the sequences required to set up the simulation problem as well having feedback from the microcomputer on the progress of each piece of command dialogue. Other output requirements are to display the simulation results in numerical and graphic forms.

The inputs required when using the simulation consist of commands, equations and numerical values. The alphanumeric keyboard is probably the most versatile input device available, and for a minimum system configuration is the only human input device needed.

Ideally during command dialogue the microcomputer should provide as much information as possible to enable the user to fully operate the simulation system without having to remember all the command names and control sequences. A fairly high data output rate is therefore preferable to prevent annoying delays. Visual display units (VDU's) can provide high data rates, often up to 96,000 baud, and can also simultaneously display several lines of information. Commercial visual display units (VDU's) including keyboards can be obtained for about £500, but more restricted VDU's produced for the hobbies computer market are considerably cheaper.

While a VDU with its high speed is ideal for setting up a simulation task, final results of a simulation would be more useful in a permanent hard copy form. A tabulated numerical printout would provide the most accurate output, but for most applications a graphic output would be preferable.

Alphanumeric VDU's and printers can be used to provide very low resolution graphics using standard characters, and this graphic capability can be extended by using a graphics character ROM, where the space taken up by an alphanumeric character is subdivided to enable higher resolutions to be obtained. Using a 3 by 2 subdivision on a low cost display such as the NASCOM-1 gives a resolution of about 48 vertical points and 96 horizontal points, and this would be sufficient resolution for a rough check

of the model. Much higher resolution can be obtained by storing individual points of the raster display which can be produced either on a video monitor in a similar way to alphanumerics, or on an oscilloscope (CRO) using high speed digital to analogue converters to produce the raster in the same way as McLennen[57]. Storing individual points requires a large amount of storage for high resolutions, for example a 256 by 256 display would need 64k bits of memory. An alternative which was finally chosen for the simulation system is to draw the graphic output directly from the microcomputer either to an X-Y plotter or to an oscilloscope. The hardware is very cheap as most of the work is done by a microprocessor which is used as a peripheral of the main simulation microprocessor. The Z80 used in the simulation system can draw four graphs each of 256 points sequentially on an oscilloscope and still produce a flicker free picture with a refresh rate of over 30 frames a second. Since this display is software controlled, different display formats can be chosen with data being stored only for the actual points needing displayed. The direct output can also be used to drive an X-Y plotter simply by slowing the display rate down and eliminating the refresh. Hard copy can be obtained from the oscilloscope simply by using a camera such as a simple polaroid hand held instant camera.

3.2 System Hardware

3.2.1 General Description

The initial simulation system programs were developed using the GIMINI microcomputer alone, but later versions used the full

system including the NASCOM-1 microcomputer and extra hardware. Figure 3.1 shows the total simulation system as it was used, with the exception of the LAN display oscilloscope, and figure 3.2 gives the block diagram of the total system. The GIMINI microcomputer consists of a card frame with four cards, processor, memory, monitor, and input/output. The processor card contains the General Instrument Microelectronics CP 1600 microprocessor together with clock circuits and buffering. The memory card has 8k by 16 bit words of dynamic RAM memory together with the refresh circuits. One problem with this card is that the refresh circuits are driven by the signals from the microprocessor, therefore if the microprocessor halts, the refreshing is stopped causing loss of data. The input/output card provides both a slow speed asynchronous serial interface for a teletype, and a higher speed parallel interface for a paper tape punch and reader. A Fortronics optical reader was used, but no punch was available so the slow speed teletype punch had to be used. The monitor card contains software for the teletype based operation of the GIMINI as well as software and hardware to operate the front control panel, which contrary to appearances is not hardware operated, but software operated and thus able to cope with the dynamic memory. The monitor provides hexadecimal debugging facilities, a relocating loader, and some input and output utility routines.

The NASCOM-1 is a single board microcomputer which uses a Mostek Z80 microprocessor. The NASCOM-1 has 2k bytes of static RAM memory, half of which is used for a memory mapped visual display. The display provides 16 lines of 48 characters which can either drive a video monitor or, with the on-board UHF modulator, a

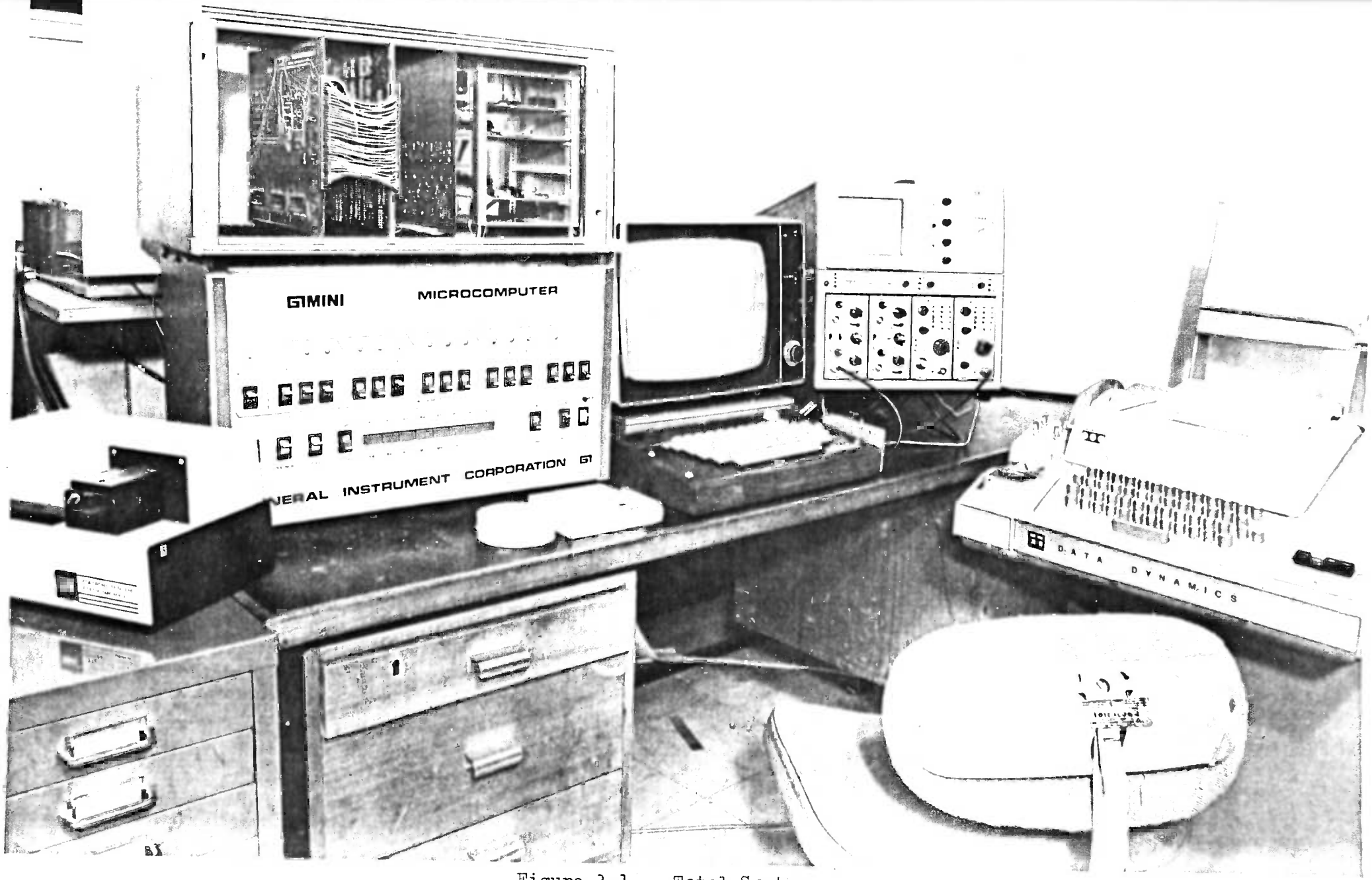


Figure 3.1 Total System

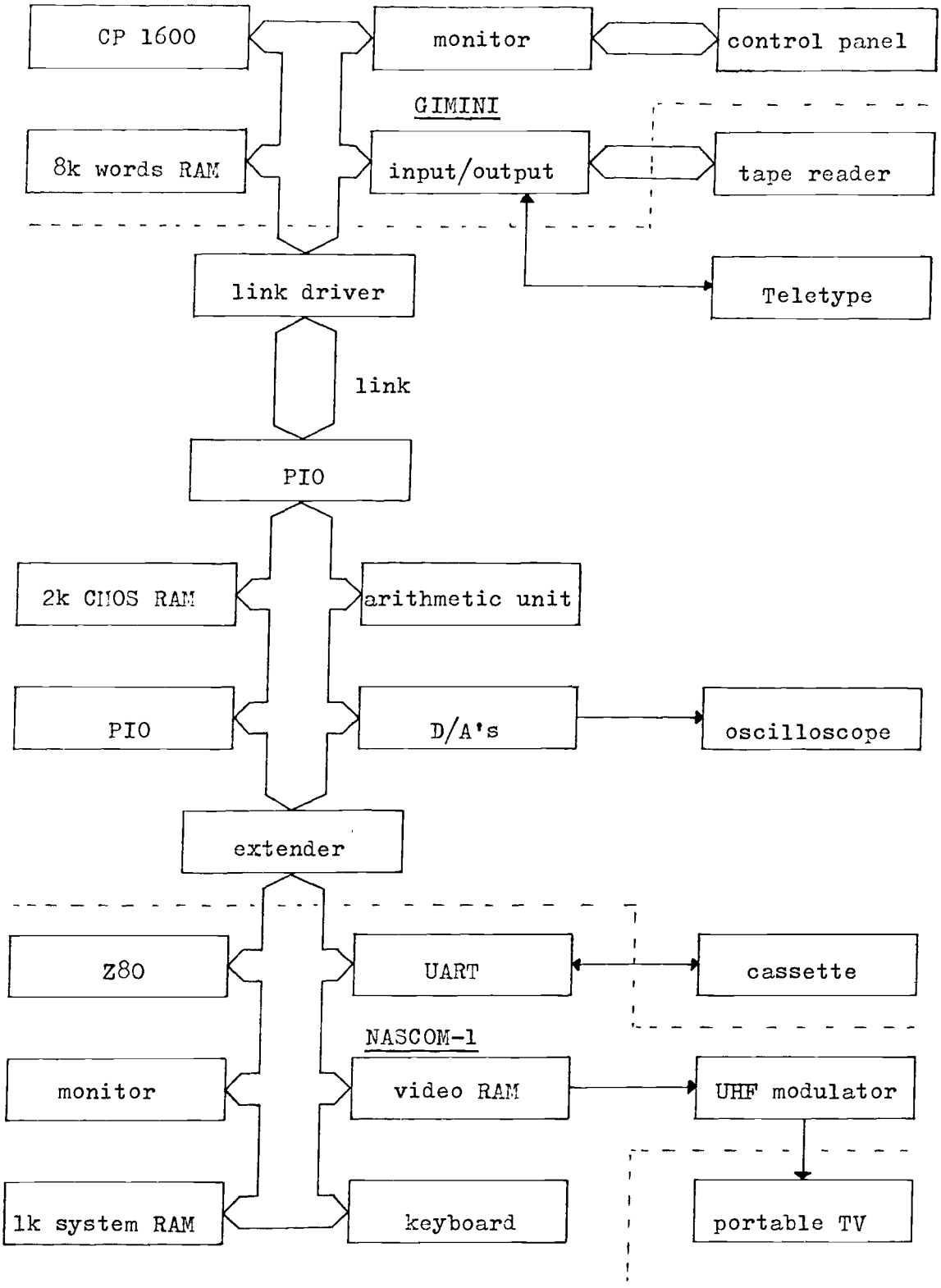


Figure 3.2 Total System Diagram

standard 625 line television set. A software polled alphanumeric keyboard is used and the monitor provides debugging facilities, input and output utilities, and a memory dump and load facility. The UART can be used to dump the data serially to an audio cassette recorder, or an asynchronous terminal such as a teleprinter. The NASCOM-1 hardware and software manual[58] gives further details of the hardware operation and the monitor program. The Series 1600 system documatation[39] gives details of the hardware and software of the GIMINI microcomputer.

Extra hardware was designed and constructed to extend the capabilities of the NASCOM-1 and provide a link between the two microcomputers. The parallel link utilizes handshaking techniques and is designed specifically for master-slave operation in the simulation system. Two 8 bit D/A's are used to provide a refreshed graphics display on an oscilloscope. The 2k byte CMOS RAM memory has battery backup and can be write protected to provide an ROM simulator for program development. A floating point processor was also interfaced, but insufficient time was available to integrate its operation into the simulation system. An uncommitted peripheral input output device (PIO) was also included for future extension such as direct analogue data input using an analogue to digital converter.

3.2.2 Microprocessors

The General Instrument Microelectronics CP 1600 microprocessor is a 16 bit single chip device which has a combined memory and input/output address space of 64k words. Since all the registers including the program counter (R7) can use all the CP 1600's

arithmetic and addressing modes, an unusual and powerful set of branch and jump instructions can be achieved. The Series 1600 system documentation[39] gives the instruction set of the CP 1600, and operations can either be on one register, between registers, or between memory and a register. The 5 addressing modes used are: register, direct, register indirect, relative, and immediate. Any register can be used for single register operations, with the exception that shifts and rotates can only be performed on registers 0 to 3. The direct address is normally a full 16 bits, but can be reduced to 10 bits if full width memory is not used. The 10 bit wide instructions combined with the CP 1600's ability to handle 16 bit data as two bytes means that programs can often be stored in 10 bit wide memory. Register indirect means that the contents of a register can be used as a 16 bit address. R0 cannot be used for indirection as these particular codes are used for direct addressing. Immediate addressing is achieved by using register indirect with the program counter R7. R6 can be used as a stack pointer and registers R4 and R5 can be used as autoincrementing data pointers with the register indirect mode. The subroutine call structure is unusual for a microprocessor because it does not use a stack. When a subroutine is called the return address is stored in one of three registers R4, R5, or R6 depending on the instruction used, and when a return from subroutine is required the return address is simply moved from the storage register to R7. One of the benefits of this subroutine call system is that passing parameters to subroutines is very easy since the address storage registers are autoincrementing. The branch instructions using relative addressing provide a full

16 bit displacement together with a sign bit to enable jumps between any two points in the 64k word address space. As well as conditional branches on the state of internal flags set by preceeding instructions, the CP 1600 can also perform conditional branches on the state of 16 external flags. Direct memory access and both maskable and non-maskable vectored interrupts are provided, with provision for daisy chaining the interrupts to provide a priority structure.

The Mostek Z80 is a second sourced version of Zilog's Z80 single chip 8 bit microprocessor. Unlike the CP 1600, the Z80 has separate memory and input/output address spaces, with 64k bytes of memory and 256 input or output ports available. Most Z80 arithmetic or logical operations require the use of the single 8 bit accumulator, but the main and alternate register sets can be swapped to provide a fast interrupt response. The general purpose registers are mostly used as single 8 bit or double 16 bit registers for temporary storage or addressing. Sixteen bit registers are provided for a program counter, a stack pointer, and two index registers. The interrupt vector register supplies the top 8 bits of the interrupt vector if used, with the bottom 8 bits being provided by the interrupting device itself. Dynamic memory can be supported by the Z80 without additional hardware because the Z80 uses the refresh register to provide the addresses necessary to periodically refresh the memory. The interrupt system of the Z80 is more complex than the CP 1600. The non-maskable interrupt line provides a jump to a fixed address at the bottom of memory, but the maskable interrupt line as well as jumping to another fixed address has two other modes of operation. In addition to the vectored interrupt previously mentioned, the

maskable interrupt can operate by accepting a restart code from the interrupting device and then implementing that instruction as if it was in a program step. The relative addressing mode is used for conditional branches over a short range because only 8 bits are used to give a displacement of +127 to -128. The register indirect addressing mode uses pairs of general purpose registers to provide the address for data transfer. Indexed addressing for the Z80 is an extension of register indirect whereby a 8 bit two's complement displacement is added to either index register to give the required data address. In addition to the normal range of operations expected for a microcomputer, the Z80 also provides for manipulation of individual bits as well as providing some useful block search and transfer instructions.

While input and output can be performed by simple 3-state buffers and latches, full use of the Z80's interrupt structure can be obtained by using interface devices designed specifically to operate with the Z80. The PIO, which stands for parallel input/output, is a programmable input/output device which provides 2 ports each of 8 input/output lines, 2 handshake lines, and interrupt control circuits. There are four modes of operation for the PIO ports all of which have been designed to use interrupts. The ports can be used independently as either latched inputs or outputs with handshaking signals for the external interfaced device. The PIO does not contain any microprocessor readable flags indicating the condition of the handshake lines so the PIO's vectored interrupts have to be used. Another available mode is the bit mode in which the individual bits of a port can be set either to be inputs or

outputs. This mode does not use handshaking signals but does provide an interrupt facility using extra mask registers in the PIO which generate an interrupt when a specific bit pattern appears at the port. The final mode is only available on one port with the other in bit mode, and provides bidirectional data transfer with independent handshake signals. Both pairs of handshake lines are used in this mode to allow the bidirectional data transfer with both vectored interrupts being used, one for data input, and one for data output. A PIO operating in bidirectional mode is used in the link circuitry between the two microcomputers, and here the second port operating in bit mode is used to provide software controlled flags for communication between the two microcomputers. PIO's can be daisy chained to provide a priority interrupt structure, up to 4 without any extra circuitry. An unusual feature of the Z80 is that, unlike the CP 1600, it does not issue an end of interrupt signal, so the PIO decodes the return from interrupt instruction itself.

3.2.3 CMOS Memory

The 2k byte CMOS memory board whose circuit diagram is shown in figure 3.3 was designed to provide non-volatile storage of data and programs. The memory is organised as two blocks each of 1k bytes either of which can be write disabled to simulate EPROM or ROM. Each block consists of 8 Intersil IM6508C (1k by 1 bit) CMOS RAM memories with battery backup provided by rechargeable nickel-cadmium batteries. This board was originally designed to work with a Motorola M6800 microprocessor running at 1 MHz clock rate, but was used unmodified for the Z80 with a 2 MHz clock. While the

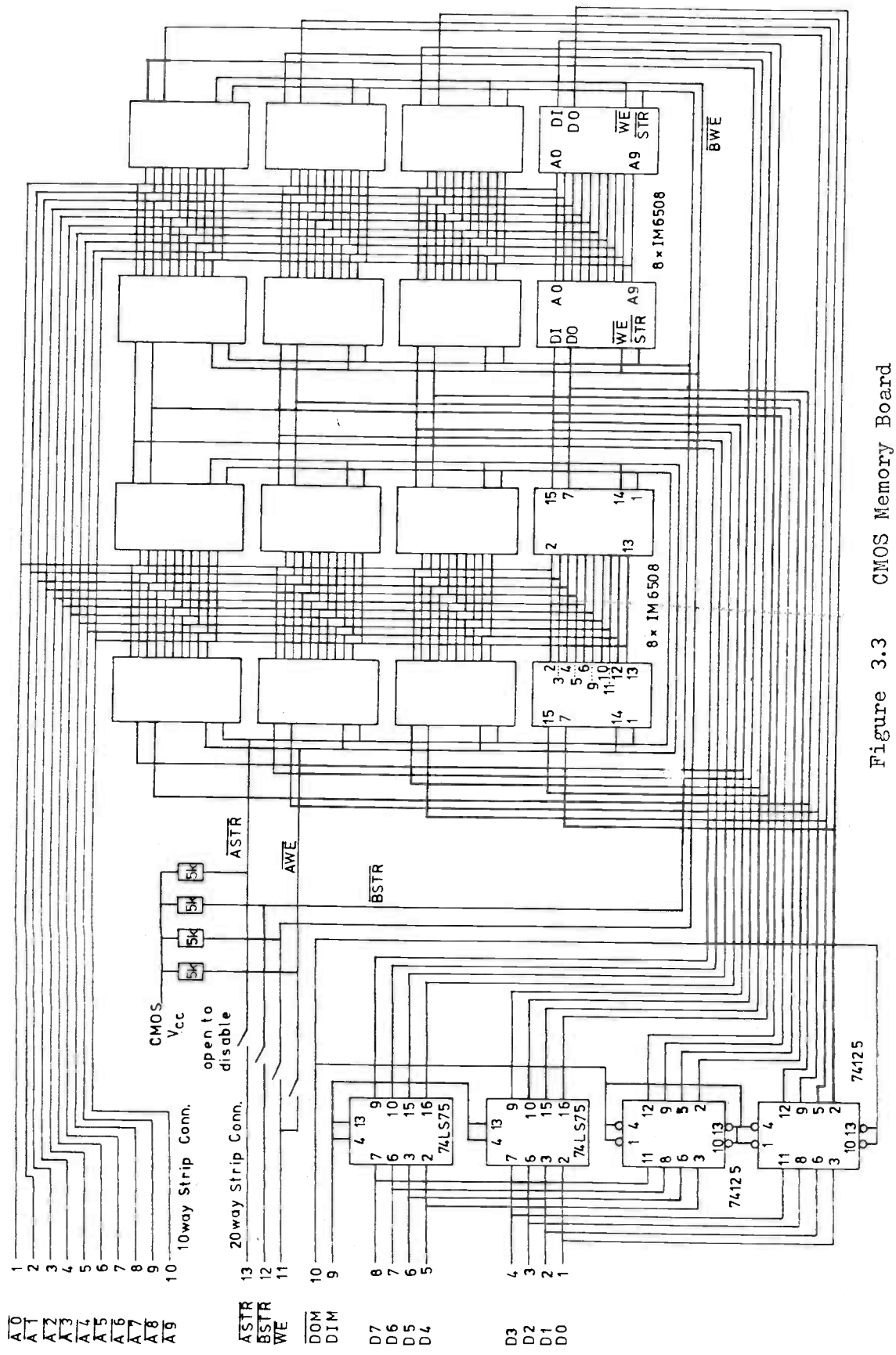


Figure 3.3 CMOS Memory Board

IM6508C is fast enough to work with the Z80, it is not fast enough to work directly with the M6800 at 1 MHz because of the normal addressing method used by the M6800. Rather than slowing down the M6800, or using dearer memories, it was decided to add extra circuitry to alter the addressing method since the cycle time of the memory was actually less than minimum time between memory accesses by the microprocessor. The read timing was met by activating the memory select lines ($\overline{ASTR}$ or $\overline{BSTR}$) in advance of the normal M6800 memory strobe. The write timing was met by stretching the memory select signals and latching the data into the two 74LS75 latches on the memory board. The normal M6800 memory write signals are used to provide the $\overline{DIM}$ latch pulse. The 3-state outputs of the memories do not have suitable timing for direct connection to the microcomputer's bus, so two 74125 quad 3-state drivers are used to drive the data bus during read operations. Schottky TTL devices are used to drive the address and data lines, because their outputs are effectively low when they are unpowered and therefore reduce any possible interference. The strobe and write lines are standard TTL with additional 5k pull-up resistors which hold these lines high in a power down mode, since they are connected to the battery backup circuits unlike the latches and 3-state buffers. The switches allow either the complete access or just the write access to be disabled for each memory block. The battery backup circuit is the same as described by Intersil[50] and the 500 mAh batteries used can keep the memory data intact for at least a month. When used with the Z80 microprocessor, no special timing considerations are required, and the connections to the memory board are a 10 and a 20 way strip connector which was found to be more reliable than the single sided VERO edge connectors previously tried.

3.2.4 Arithmetic Processor

The arithmetic board shown in figure 3.4 used the Advanced Micro Devices Am9511 single chip floating point processor. The Am9511 provides 32 bit floating point arithmetic complete with trigometric and logarithmic functions. This unit also performs 16 and 32 bit integer arithmetic. The arithmetic processor is stack based and is operated by first writing all the required data for a calculation to the processors internal stack in 8 bit bytes, and then writing the required arithmetic command to the processor which will then perform the calculation. The microprocessor can detect the end of a calculation either by reading a status word in the Am9511 or by a hardware interrupt if implemented. Since the Z80 was already being interrupt driven by the CP 1600, software monitoring of the calculations was chosen. The maximum time required for floating point operations varies between 84 μs for a multiply and 4.6 ms for X to the power Y. Results are retrieved by reading the internal stack of the processor, again in bytes. When the AM9511 is not ready to transfer data, including status information and commands, it issues a wait signal to halt the Z80's memory access until the arithmetic processor is ready.

The read and write signals to the AM9511 are straightforward, with the wait signal operating in the normal fashion for a Z80 system. A reset signal derived from the master reset is used to ensure that the processor is idle until a command has been issued. One peculiarity of the AM9511 not mentioned in the data sheet is that the chip select must be used since the select pulses are used to reset internal circuitry. The timing of the chip select is

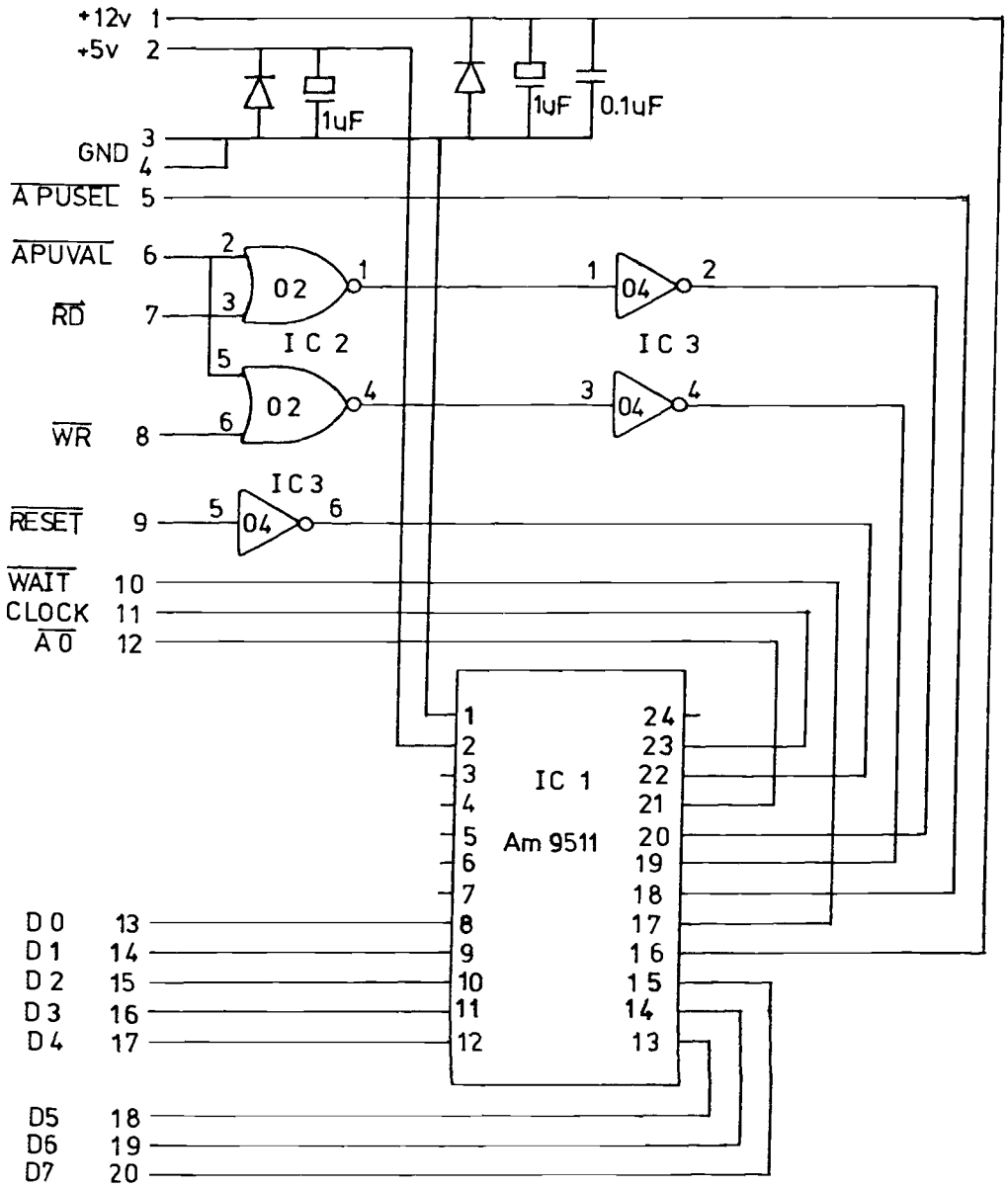


Figure 3.4 Arithmetic Board

unsuitable for using the Z80's normal $\overline{\text{IOREQ}}$ signal. The solution was to decode the port address whether it was valid or not and use this for the chip select. The occasional chip selects without any data transfer have no detrimental effect on the performance of the arithmetic processor.

3.2.5 NASCOM-1 Extender

The Extender circuit shown in figure 3.5 provides the buffering and decoding required to operate the additional memory and peripheral devices. The address lines are buffered by IC's 1 to 4 and the data lines by the 3-state bus transievers IC 8 and 9. To avoid contention on the main microprocessor bus, the 3-state buffers always drive into the extender except when the microprocessor requires to read data from either the memory, the arithmetic processor, or a PIO. Note that not only does the microprocessor read data from the PIO's, it also reads the interrupt vectors when an interrupt request is made. The NASCOM-1's internal decoding has to be disabled to allow extension, so two decoded signals MEXT and IOEXT are returned to the NASCOM-1 to operate its internal memory and input/output. Figure 3.6 gives the memory map for the expanded Z80 system and figure 3.7 gives the input/output map. To ease future expansion, full decoding is used for memory. The extender PIO's are daisy chained to give interrupt priorities, but since no access is given to the daisy chain signals of the PIO on the NASCOM-1, it was removed. The arithmetic processor is considered to be an input/output device since there is no speed advantage in using it as a memory mapped device. The access time of the arithmetic processor is at least as long as the Z80's input/output cycle with its automatic extra wait state.

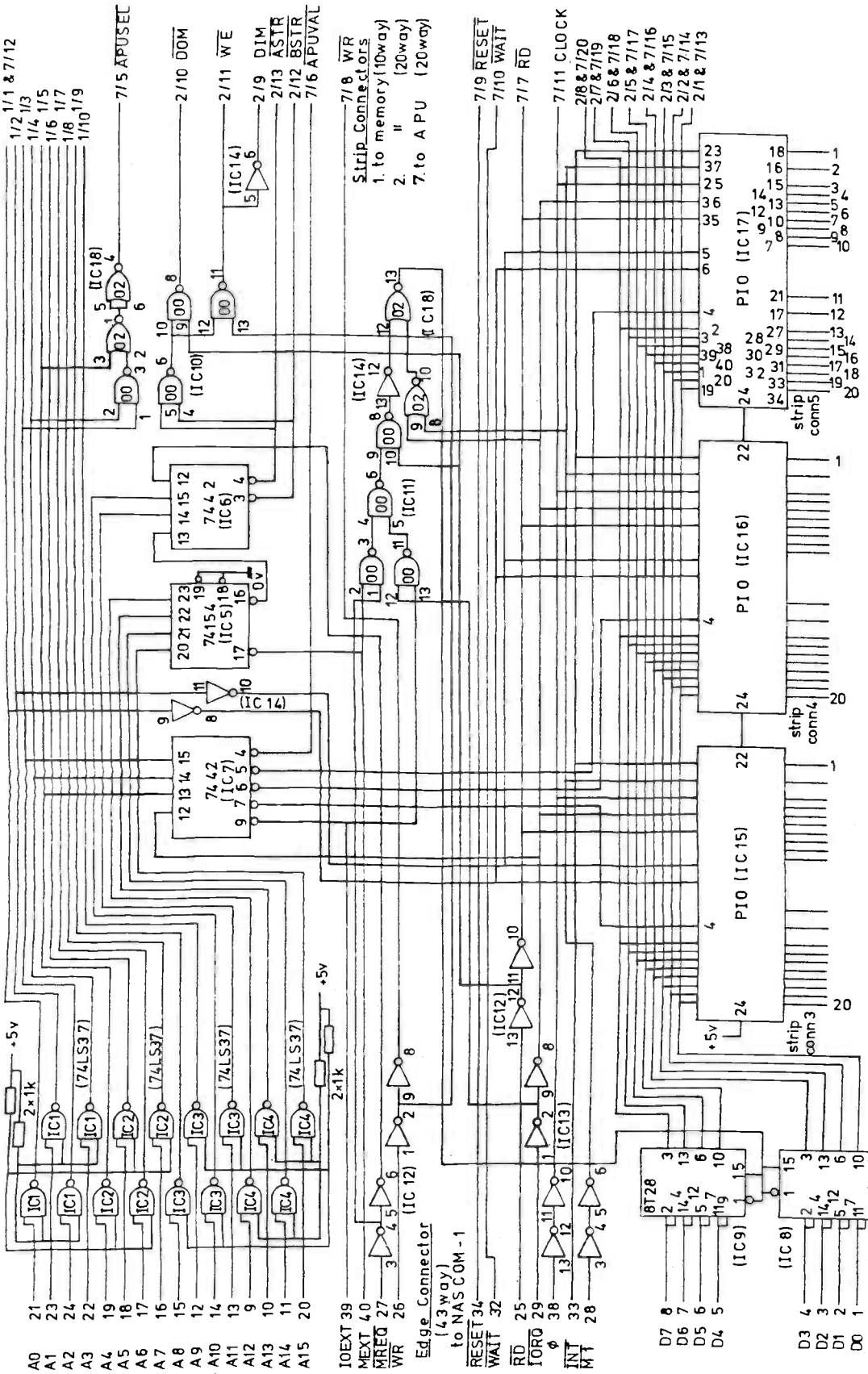


Figure 3.5 NASCOM-1 Extender Board

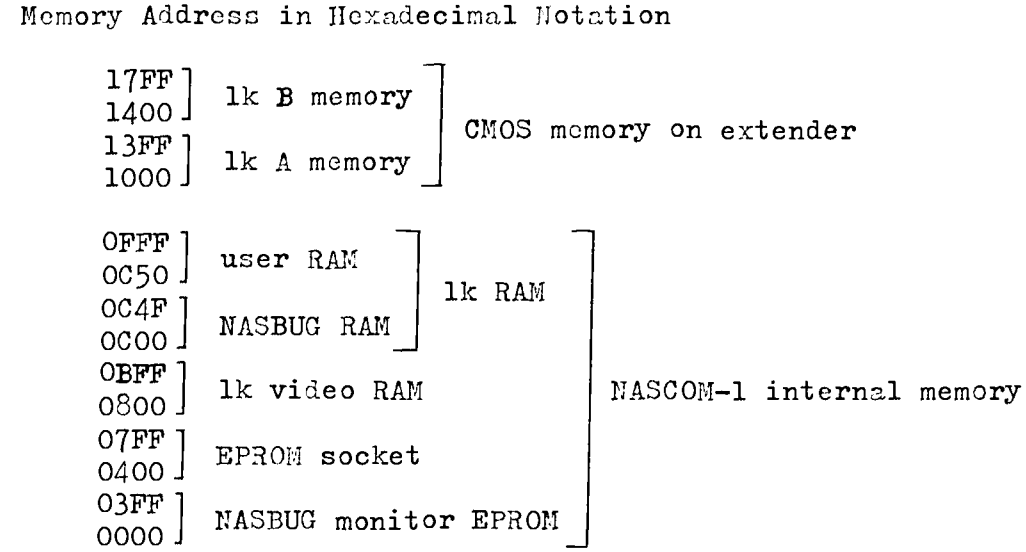


Figure 3.6 Memory Address Map

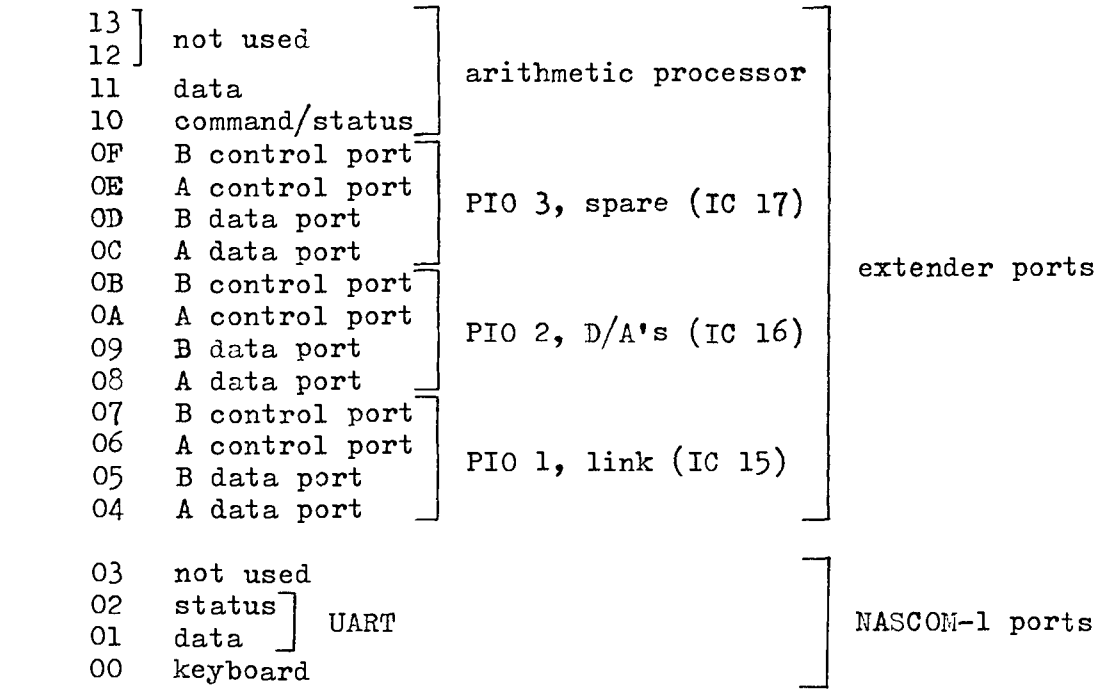


Figure 3.7 Input/Output Address Map

All the control signals used by the extender card are buffered to avoid loading the Z80. Full input/output decoding is not provided but the input/output devices used have mutually exclusive addresses.

3.2.6 Link Circuit

The link circuit which provides the communication between the two microcomputers was constructed on two small circuit boards which plug into one of the card positions in the GIMINI card frame. Figure 3.8 shows the link data board which provides 3-state buffering between the GIMINI's data bus and PIO 1 of the extended NASCOM-1. IC 1 and 2 are 3-state buffers for the bidirectional data transfer to the PIO A port. Data transfers are performed by read and write operations of the CP 1600 with the data for both directions being latched in the PIO. The $\overline{ZTC}$ signal enables data from the PIO onto the GIMINI's data bus and also causes the A ready line to be cleared and an output interrupt request to be issued. The $\overline{CTZ}$ signal gates data from the GIMINI data bus to the PIO and latches it, this signal also clears the B ready flag and issues and interrupt request. The two ready flags together with a master software controlled 'Z80 ready' flag are fed to the CP 1600's external flag intputs so that software conditional branches can be used to perform the CP 1600's part of the handshaking control. Four other data lines from the CP 1600 are fed to the PIO B port, which is in the bit control mode, and these data bits are used to specify the type of operation the Z80 is required to perform with the supplied data. The further 4 data lines latched from the CP 1600 and 3 from the PIO

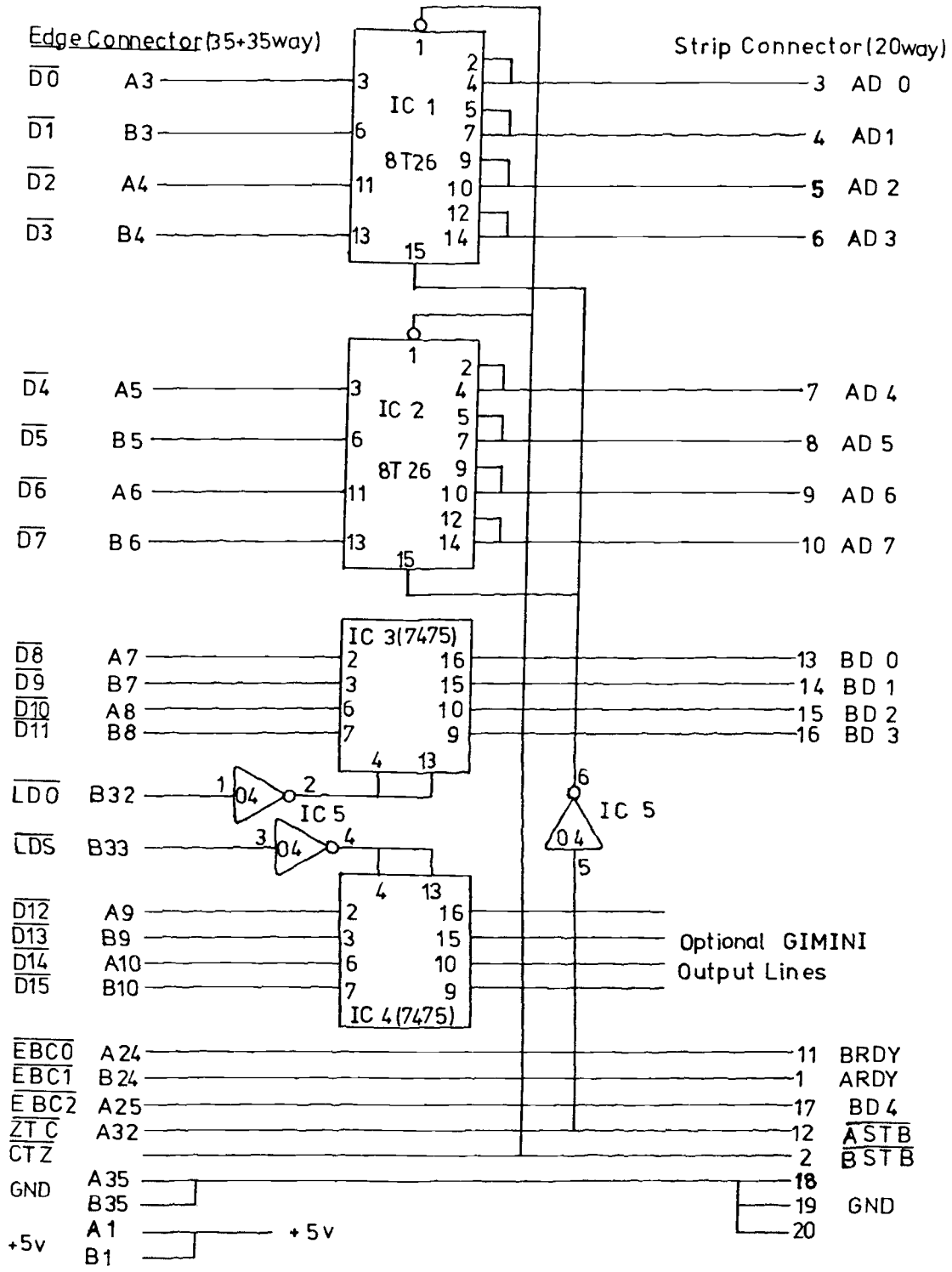


Figure 3.8 Link Data Board

are uncommitted and can be used for other purposes. The link control board shown in figure 3.9 provides the address decoding and control signals for data transfer. The $\overline{DTB}$ and $\overline{DWS}$ signals are the GIMINI microcomputers read and write pulses respectively.

3.2.7 Analogue Output

The digital to analogue board shown in figure 3.10 contains two 8 bit digital to analogue converters (D/A's) to provide a graphic output on an oscilloscope or an X-Y plotter. Two Ferranti ZN425E D/A's buffered with ZN 424 operational amplifiers were used, and although these are low cost devices they are fast enough to cope with the maximum output speed of the Z80, which with a 2 MHz clock, is about 5.5 μs per output byte. The D/A's work off ±5v supplies, and were set up to provide an output range of 0 to 3.5v. A 74123 retrigerrable monostable is used to blank the display so that the oscilloscope tube does not get burned when the display is stationary. The TTL output level of the monostable is sufficient to operate the Z-axis modulation of the LAN display oscilloscope normally used with the simulation system. Other scopes would probably need buffers to produce suitable Z modulation signals. Although the analogue output board is driven for convenience by a PIO, straightforward latches would be sufficient.

3.3 System Software

3.3.1 Program Development

After the initial structure of the simulation system program was designed, a trial version of the program was written in the

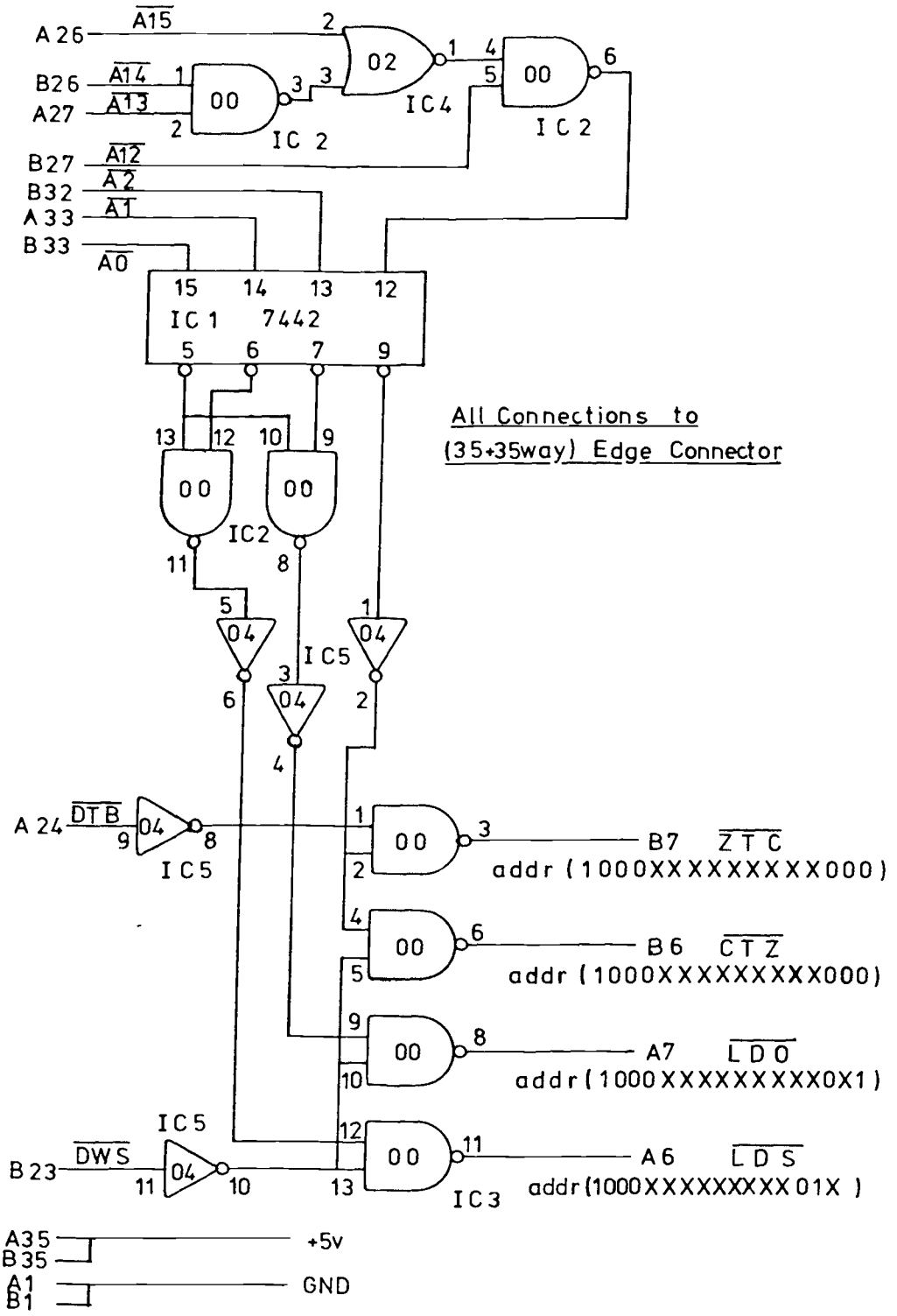


Figure 3.9 Link Control Board

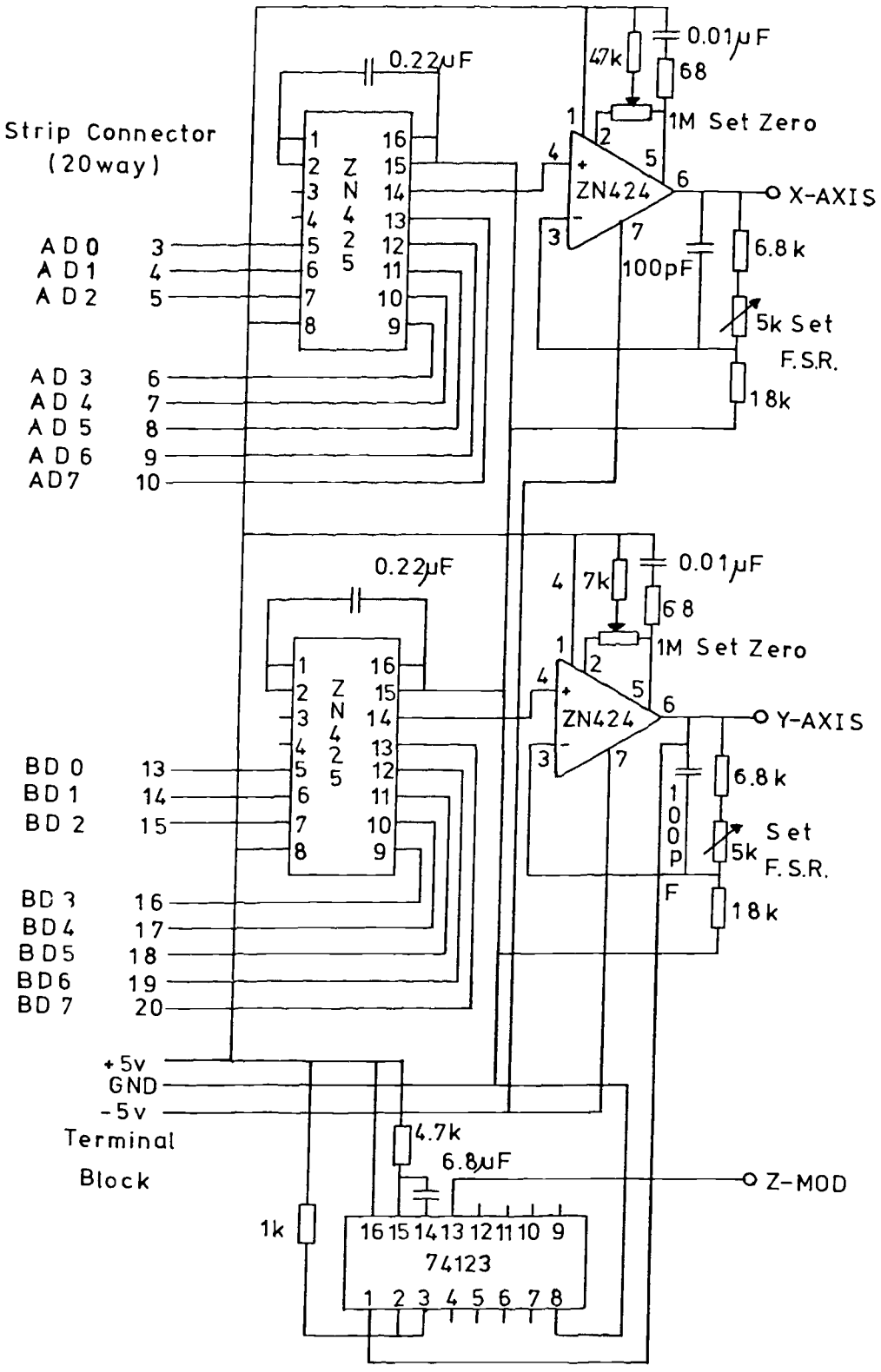


Figure 3.10 Digital to Analogue Board

high level language IMP[59] on a large ICL 4-75 mainframe computer which had the EMAS[60] multiaccess operating system. Since the simulation system program was designed to be highly modular, the structure of the linkages between the modules forms a most important part of the program. The high level program validated the program structure and provided some useful information on the use of control dialogue with a scrolled VDU. Since the program structure was retained, the modules of the program which were implemented as subroutines could be rewritten individually to perform the same functions. Changes had to be made in individual subroutines to take account of the different input and output procedures as well as the need to use an explicit floating point arithmetic package. This version of the program was written for the GIMINI microcomputer in a mixture of CP 1600 assembly language[39] and CP 1600 superassembly language[61]. Because of the more direct access available to the microcomputers memory, changes were also made to the array storage, parameter passing, and text strings to enable more efficient use to be made of the microcomputers more limited memory and execution speed. In general the higher levels of the program had a high percentage of super assembly statements, with the ratio decreasing until the lowest levels of the program were almost pure assembly language statements.

The program was then subjected to continued development with more features being added and improvements being made to existing subroutines. The 8K word size of the microcomputer memory exerted a severe constraint on the program size, and in order to get even the program let alone the simulation data into the memory, the program

had to be considerably compressed. The program size was reduced in a variety of ways, mainly by reorganising data storage and rewriting subroutines. For ease of implementation, the text strings used for microcomputer dialogue were originally held with one byte per 16 bit word and terminated by a zero word, but this was changed to storing two characters per word. While replacing high level statements with assembly language ones makes the program considerably harder to understand, the greater use of the CP 1600's register set enables considerable savings to be made in memory requirements and execution time. The debugging aids used in developing the microcomputer program were very primitive compared to those of a large computer system, so initially almost every subroutine had its own set of variables which greatly aided debugging and extension of the program. However in the later stages of development, when each new version of the program was a modification of the previous version, the variables used were rationalised to free extra space. Changes were also made to the organisation of some of the subroutines so that advantageous use could be made of the CP 1600's instruction set. One feature used to good effect in saving memory space was the CP 1600's ability to manipulate the contents of the program counter. The main savings obtained by this facility were in the command subroutines where the yes or no decisions and dialogue were performed by a subroutine which could return conditionally to differing points in the program. The present SIMUPROG V3 with data storage and floating point software uses the entire 8K word memory.

Since the CP 1600 super assembler used to translate the SIMUPROG programs can only handle about 180 symbols with the present memory,

the program had to be broken down into 5 segments not including the floating point arithmetic subroutines. The resulting object code modules, together with the floating point software, was linked together to form one object program tape by the object module linker[39]. The object module linker resolves the global references between modules, and this enables the individual modules to be written and assembled separately. The source programs, consisting of assembler and super asssembler statements, are created using the GIMINI's text editor[39]. The size of the GIMINI's memory limits the amount of text that can be stored in memory by the text editor, so the source code for the simulation program modules had to be subdivided into segments which could be handled by the text editor. All the programs used to produce SIMUPROG ran on the GIMINI microcomputer itself.

The programs for the NASCOM-1 microcomputer were hand coded into machine code for use, but a listing was produced later on a Zilog development system. This development system was disc based and had no other loading facilities so a source code program was retrospectively written from the debugged machine code program and typed into the Zilog development system. The machine code program was entered as hexadecimal digits into the NASCOM-1 microcomputers memory using the systems NASBUG monitor.

Handling the various text segments on paper tape was very time consuming, especially since only the slow speed teletype punch was available. The punch proved to be unreliable and had great difficulty in producing an error free object program and even produced errors in the shorter source and object programs of the individual modules. While these tapes sometimes had to be repunched, it was often possible

to correct the tape by splicing parts of two or more versions together. Printing out a listing of the program modules was done by the super assembler and was also time consuming, but if major changes were made to the program the listings are essential to keep track of the program development and debugging.

The super assembler is a two pass assembler, this means that the source program is read twice by the assembler to produce the object code tape. The first pass creates a symbol table which contains all the labels and variable names including arrays and external references, and the second pass uses the table to resolve the jumps and memory accesses and so produce a relocatable object program tape. The second pass has to be repeated to produce a listing since the teletype was used for both printing text and punching object code paper tape.

The facilities provided by the super assembler include sub-routine calls with parameters, conditional jumps, assignment statements with 16 bit integer arithmetic, single dimensional arrays, repetitive loops and conditional program blocks. The loops are similar to those used by FORTRAN and can be controlled either by counting a variable or checking that a condition holds. With the version of the assembler available, these DO loops could only be nested to a depth of two so when more nested repetative loops were needed, they either had to be explicitly written in assembly language or incorporated into a subroutine which is called within a loop. The conditional blocks are similar to those of ALGOL and are either of the IF-THEN or IF-THEN-ELSE forms, but cannot be nested. Any nested conditions required can be implemented by using IF-GOTO conditional jumps

inside a conditional block. Since the super assembler was written specially for the CP 1600 architecture, the high level statements are very efficient for memory to memory operation so the savings achieved by using pure assembly code derive mainly from the greater and more efficient use of the CP 1600's general purpose register set.

3.3.2 Program Debugging Aids

The GIMINI microcomputer has only a small operating system, stored in read only memory, which provides facilities for loading and dumping programs, utility subroutines, memory access capability and simple machine code monitor. Paper tapes produced by the CP 1600 assemblers are loaded using the relocating loader in the operating system firmware, but program modules with external references cannot be loaded so either they have to be linked into one self contained object tape by the object module linker or loaded using the relocating linking loader[39] which is supplied on paper tape. Since the relocating linking loader has to reside in memory, it restricts the size of program which can be loaded. The program dump is only a copy of the memory contents and is therefore an absolute object program since it cannot be relocated or moved to another position in memory. The utility subroutines provided input and output facilities for the high speed tape reader as well as the teletype.

The monitor program, as well as providing facilities for inspecting and modifying the contents of memory addresses, also provides up to 8 breakpoints and a single stepping facility. The breakpoints and single stepping are software controlled, and as

such can only be used with RAM memory. The instructions where breakpoints are set have their own instruction codes replaced by software interrupt instructions so that when an active breakpoint is reached in a program, control is returned to the monitor by the software interrupt instruction. The monitor keeps track of the breakpoints so that they are transparent to the user which means that when the memory location is accessed, it is the original instruction which is read not the software interrupt. Care has to be taken, when a program has been exited at a breakpoint, to ensure that the breakpoints are removed so that no software interrupts are left when the program is re-run from the start. The single step facility operates by temporarily inserting a breakpoint just ahead of the instruction to be single stepped. Both breakpoints and single step operations store the contents of the microprocessors registers so that they can be read and even altered before continuing the program.

Debugging was occasionally made more time consuming by the fact that the contents of the dynamic RAM memory were lost if the microprocessor stopped. Since the HALT instruction code for the CP 1600 is all zero's, any fault causing a jump outside the program code area of memory is likely to encounter a halt instruction. When the program is lost due to a spurious halt instruction, no indication is left of the origin of the fault. The procedure used when this happened was to attempt to deduce approximately which area of the simulation system program the fault was in, then divide that area up using the breakpoints. By proceeding from breakpoint to breakpoint, the position of the fault can be narrowed down. This process can

be repeated for smaller segments until either the fault is found from examining the program listing or a small area of program can be single stepped to find the fault.

Each superassembly statement used generates several machine code instructions, and although an expanded listing can be generated by the assembler it takes considerably longer to printout and is more difficult to follow. This means that debugging at a machine code level is difficult without an expanded listing, so it was found that the best way normally to find a fault, if it does not crash the program, is to feed into the program a variety of inputs designed to highlight how a particular fault is operating. By examining the program in conjunction with test results and by using strategically placed breakpoints to allow examination of the variables held in memory, the actual behaviour of the program can be found and the fault corrected. Most of the faults found in the simulation system programs were typing or transcription errors and these were usually found simply by examining the program listing when the area of the fault had been narrowed down by examining the interactive responses of the program. Testing and verifying the programs was an iterative task since one fault could easily bypass several otherswhich would only appear when the initial fault was corrected. The modular structure of the simulation system program was extremely valuable in fault finding because it makes isolating the component parts of the program much easier. An example of this is the equation translation, where separate subroutines are used to store and retrieve equations from a common data area. Here if the output did not correspond to the input, the program could be stopped after

data input and the data area examined using the monitor. This procedure checks the input operation independently, and by using the monitor to set up data in the data area the output operation can be checked even if the input is not working.

The programs used for the NASCOM-1 microcomputer were much simpler than the main simulation program, so it was possible to hand code them directly into Z80 machine code which was entered into the NASCOM-1 by the NASBUG monitor. The NASBUG monitor uses hexadecimal notation unlike the GIMINI's monitor which uses octal. The NASBUG monitor which is stored in EPROM also has single stepping, but has only one breakpoint which is again software implemented. The NASBUG monitor lacks sophisticated loaders but instead has facilities for loading and dumping memory using a domestic audio cassette recorder. The Z80 program was intimately linked with the hardware and made extensive use of interrupts, so the breakpoint and single step were used for fault finding together with a logic analyser for checking the hardware responses.

3.3.3 Floating Point Package

A floating point arithmetic package[62] was supplied for the CP 1600 microprocessor. A 32 bit floating point representation is used with a 23 bit mantissa, an eight bit binary exponent, and an overall sign bit. The mantissa is usually normalised to preserve accuracy and is a fraction of one. The exponent is held in excess 128 notation so that subtracting 128 from the value gives the power of two which the mantissa is multiplied by. The arithmetic operations provided by the package were addition, subtraction,

multiplication, division and square root. Also provided was fixed point decimal input and output routines as well as conversion routines between floating point and 16 bit 2's complement integer notation .

Unfortunately the floating point package contained several errors which produced results varying from slightly inaccurate to absurd. Most of the routines had to be modified to produce acceptable results. The resulting software has been extensively tested and proved satisfactory. The accuracy of the addition and subtraction was doubled, but since these calculations still used truncation rather than rounding a further increase in accuracy could be expected if rounding was adopted.

4. MICROCOMPUTER SIMULATION SYSTEM

4.1. Overview

The microcomputer simulation system program described in this section was designed to provide an interactive system for entering and running simulation equations on a relatively small microcomputer system. The simulation system was designed to be both easy to use and forgiving, with facilities to lead the user through certain operations and to allow recovery from user error with the minimum effort. The program was also designed to use the microcomputer's limited memory and computing power efficiently for calculation and display. This was done by designing the program to make efficient use of the architecture of the CP 1600 microprocessor as well as the special hardware involved with the Z 80 microprocessor used for displaying results.

The GIMINI microcomputer used for this implementation contained 8 k words of user RAM memory as well as a simple octal monitor program. Since this microcomputer has no operating system with fast secondary storage, the program had to be self contained so that it could be loaded complete in one go using the GIMINI's paper tape loader. It was also desirable for the program to be easily extendable. These aims were achieved by storing the equations set, together with any other quantities which would need to be changed for a given simulation task, as data. The entering, displaying and changing of the data specifying a simulation task is performed by interactive subroutines operating on the stored data. Even running the simulation problem is achieved by another subroutine which interacts with the data input device and display. These subroutines are invoked by interactively entering commands. Since the system had to be flexible in the types, size and number of equations stored, and also as fast as possible in

execution, a precoded form of reverse Polish notation was used to store the equations in a matrix. Thus the calculation of the equations uses a stack and a set of arithmetic subroutines which operate on that stack. Since the simulation system was designed for use by people with little or no programming experience, the equation entry subroutines contain a translator which converts the more common algebraic equation form into the reverse Polish form. Having all the commands as individual subroutines has several advantages. Each subroutine can be written separately, which makes them easier to develop and debug as well as easing the problems encountered in translating the program for another microcomputer. Additonal subroutines can be easily added to expand the simulation system facilities.

The microcomputer simulation system program consists of four main parts, the command level, the database, the command subroutines and the utility subroutines. The command level interacts with the user to select the required command and command subroutines perform the actions of the individual commands on the database, which in turn contains all the information about the simulation model and running conditions. The utility subroutines provide basic facilities, such as input/output and floating point arithmetic, for the command subroutines. The input and output utility subroutines make use of the GIMINI monitor's own input and output facilities. The program and data presently runs in RAM memory, but can easily be changed to operate in ROM or EPROM memory with RAM memory only for the database and other variables. This would have the advantage of not needing reloading when the microcomputer is switched on.

The SIMUPROG V3 program together with the floating point package forms a stand alone system for the GIMINI. If the graphics display is required then the Z80 based input and output facilities must be added. Listings of both the Z80 display program and SIMUPROG V3 are available as a departmental research report [63].

The command level program first reads in the user entered command name, the program then searches the list of command names, XS, for a match and if found jumps to corresponding subroutine start address held in list SRAD. If a name is not found or a command has finished then the program reprompts the user and reads in the next name. This structure means that commands can be added simply by appending their names and subroutine addresses to the lists. Alternatively if space is limited, commands can be removed by removing their names and addresses.

The main elements of the database are shown in figure 4.1. The operate list OPER specifies the order in which the equations are evaluated, but note that an equation can appear several times in the list. The equation matrix EQN holds the set of equations in the coded form of reverse Polish notation, where a positive integer indicates a variable and a negative integer an operation or function. The equation is terminated either when the maximum length has been reached or a zero entry encountered. The unknown variable for each equation is held in the VARP pointer list, and the use of this seperate list simplifies the stack operations associated with equation evaluation. The names of the variables are stored in list SS with the initial values held in list TVAR, and the present values held in list VAR. The copy of the initial values enables a simulation problem to be rerun from the same initial conditions. The names of the operators

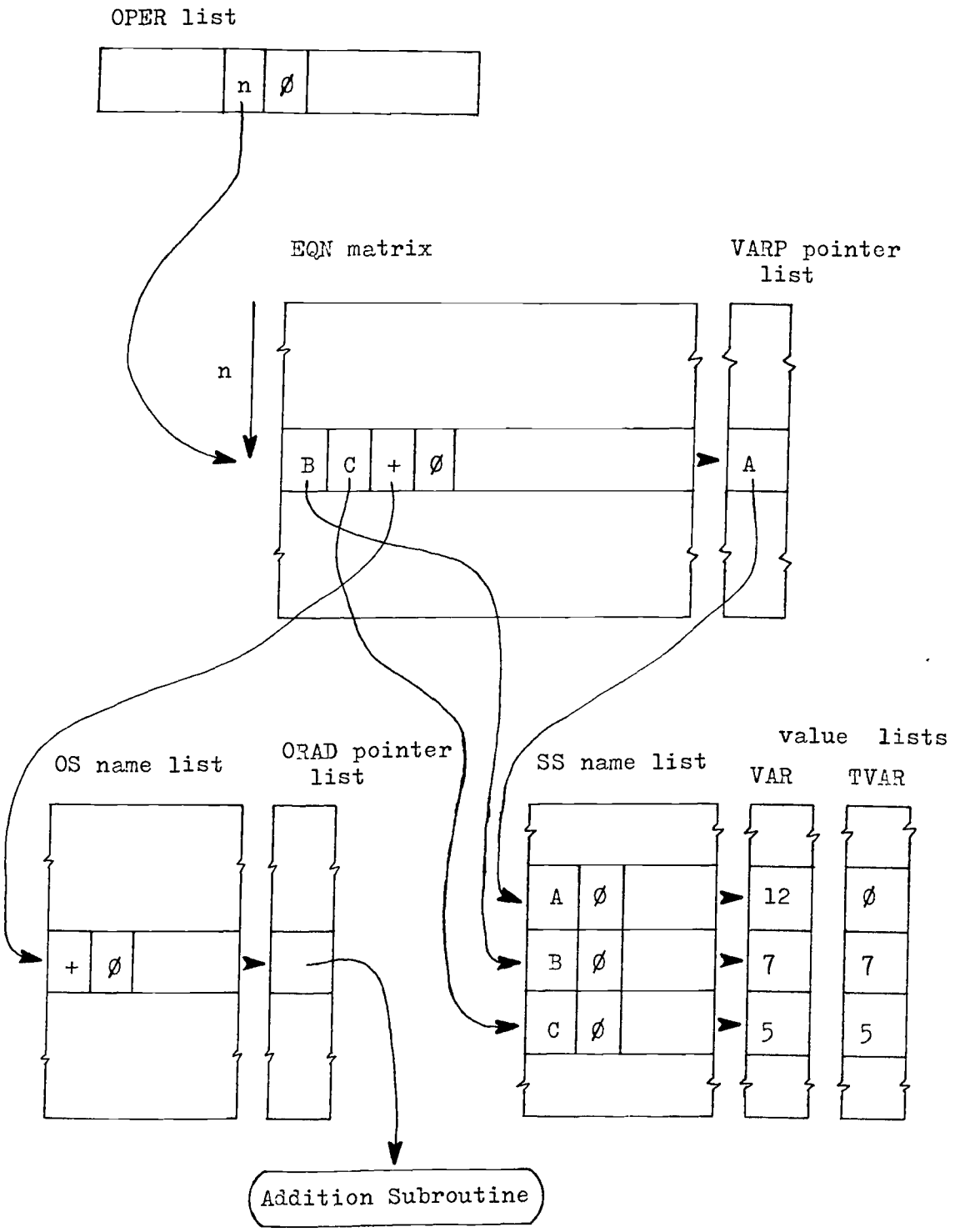


Figure 4.1 Database Main Elements

and functions are stored in list OS with their subroutine start addresses in list ORAD. The magnitude of an equation entry gives the position of the variable or function in its respective list. For the purposes of this program a list in the database is defined as a one dimensional array which can be terminated before its maximum length by a zero entry. Thus the equation matrix and the name lists OS and SS are actually lists of lists, but because of their functions, they will be referred to as a matrix and lists respectively. An additional list of all the parameters used in the equation set is called PARM. A parameter is a variable which does not apppear on the left hand side of an equation, and thus can only be changed by the user. The variables to be tabulated and plotted are held in the lists TAB and GRAF respectively, with the graph scale factors held in GSCALE. Also in the database are the run time control variables: STEPS is the number of steps of simulation performed; TABIN is the interval between printouts, with TDEST defining the output device; MODE is the data input mode, with the number of streams defined by MAXIN and the interval by INSTEP; GMODE defines the graphic mode, and GINTR the interval between plotted points.

To provide for easy expansion of the database, the sizes of the various lists are specified by only four numbers, so by changing these numbers and reassembling the program, the database can be expanded to suit memory availability.

The conversion algorithm for translating algebraic equations to reverse Polish form is shown in figure 4.2, with the table of precedences used shown in figure 4.3. The algorithm and precedences used were similar to those described by Abramson[64], but were modified to allow for the use of the arithmetic and control functions and their

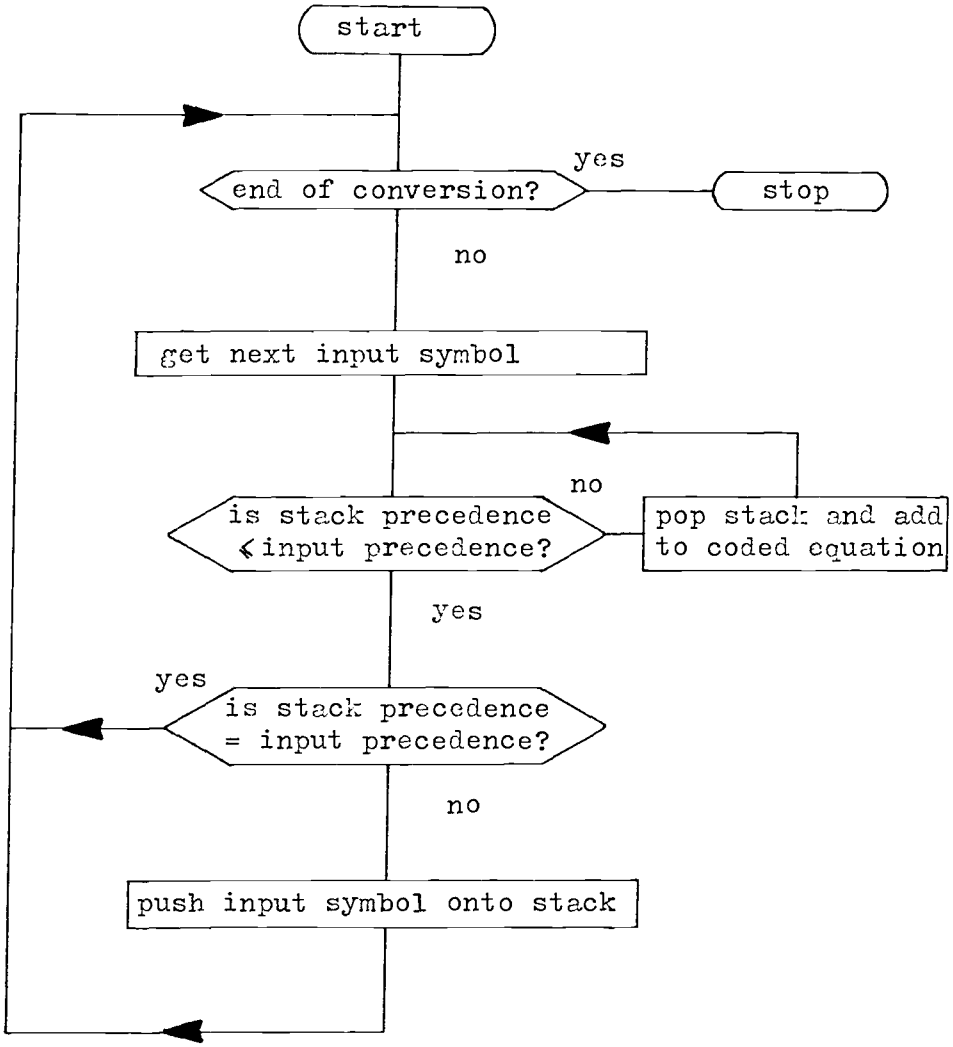


Figure 4.2 Reverse Polish Notation Conversion Algorithm

symbol	input precedence	stack precedence
+ or -	1	2
* or /	3	4
variable	5	6
function	7	8
(	9	ø
)	ø	-
,	ø	-

Figure 4.3 Reverse Polish Notation Precedences

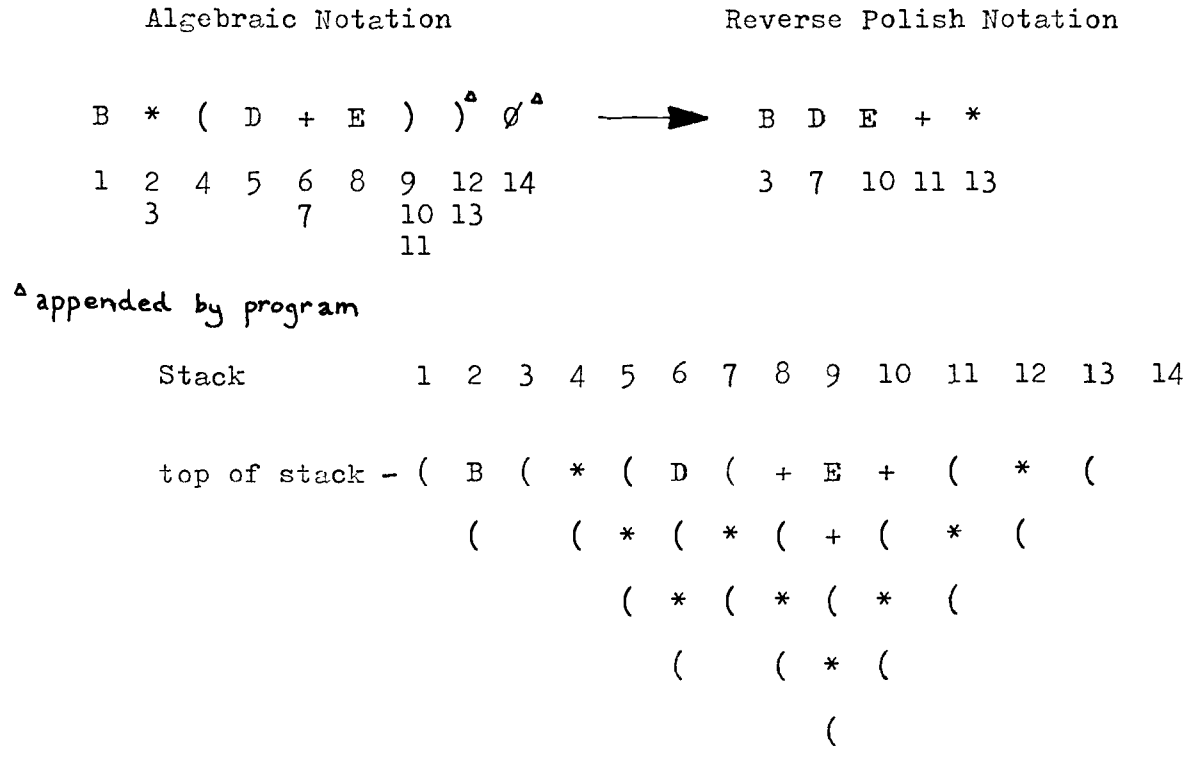


Figure 4.4 Conversion Example

parameters. In the conversion, input symbols are pushed onto the stack providing they have a higher precedence than the top of the stack. If they have a lower precedence then the stack is popped up and used to build the equation until the input symbol can eventually be pushed onto the stack. The left bracket effectively initiates a new stack until a right bracket appears as the input symbol. Both brackets are eventually dumped because they have equal precedence and the stack is restored to its position before the left bracket appeared. An example of the conversion of the simple operation B*(D+E) is given in figure 4.4.

Initially a left bracket is pushed onto the stack and a right bracket followed by a zero is appended onto the entered equation. The zero is used to terminate the conversion and the pair of brackets insure that the stack is empty when the zero is reached. The conversion takes 14 stages and the contents of the stack are shown at each stage. The input symbols are also numbered, as are the symbols making up the converted form of the equation. Note that when symbols are being pulled (popped) off the stack, the input symbol does not change as in stages 3,7,10,11 and 13. Since functions can have more than one parameter, these parameters are separated by commas. In order to allow complete expressions to be used as parameters without having to bother enclosing them in further brackets, the conversion algorithm was modified so that a comma is treated as a right bracket except that the left bracket is retained on the stack. This means that the left bracket enclosing the function's parameters remains valid.

A simplified initial version of the program was first written in a high level language called IMP [59, 65] on a large multi-access

computer system called EMAS [60]. This proved that the program structure was sound and allowed investigation into the use of a VDU display for interactive simulation. The program was then translated, subroutine by subroutine, into a mixture of GIMINI super assembly and assembly language statements. Some alterations were required to compensate for the change from a large computer with a sophisticated operating system to a microcomputer with no operating system. The rewritten program was called SIMUPROG version 1 and was assembled using the superassembler and debugged with the aid of the GIMINI's resident monitor. The program was then extended and restructured to provide more efficient code, and this formed version 2 of the program. In the present version 3, extra features were added for manipulating the equation set and for running the NASCOM-1 graphic extension. Because of the 8k word limitation of the RAM memory, the program had to be extensively revised and the same functions compressed into smaller spaces. This revision had no real effect on the program structure, but it did mean that even more assembly language was used to replace the higher level super assembly statements. The superassembler produces code which is just as efficient as the equivalent assembly language statements. The space saving obtained by using assembly statements comes from the fact that the superassembler statements operate from memory to memory whereas the ordinary assembler statements need not and therefore can make good use of the CP 1600's general purpose registers. The memory size imposed restrictions on the size of program segments which can be assembled, and the size of text segments which can be held in memory for editing. For assembly, version 3 was broken into 5 segments which were assembled seperately and

then linked together along with the floating point package to form the SIMPUROG version 3 object program which is held on paper tape and loaded using the GIMINI's resident tape loader.

4.2 Command Level

Once the SIMUPROG V3 object program is loaded into RAM it is started using the monitor, and can later be stopped and started without loss of data which is very valuable for debug purposes. A flow diagram of the command level of the program is given in figure 4.5. To save time when loading the simulation system program the database area is not loaded, only the first entry of the operate list OPER is loaded. If data has been entered into the database the initial zero in OPER will be overwritten so the database will only be cleared when operate list is empty. The stack pointer is set so that the stack grows upward from the end of the database. An introduction to the program is then printed by the microcomputer and the user is prompted by a 'c>' to enter a command. If a valid command is entered, then program control is passed to the appropriate command subroutine. When the command subroutine is finished control is returned to the start of the command level loop. The program is stopped by entering the command STOP which transfers control to the GIMINI's monitor.

There are three main groups of commands, those involved with entering the equation set, those controlling the simulation, and those running the actual model.

4.3. Equation Entry

The commands CHEQ and INEQ shown in figure 4.6 are used to enter a new equation set into the equation matrix. The CHEQ command

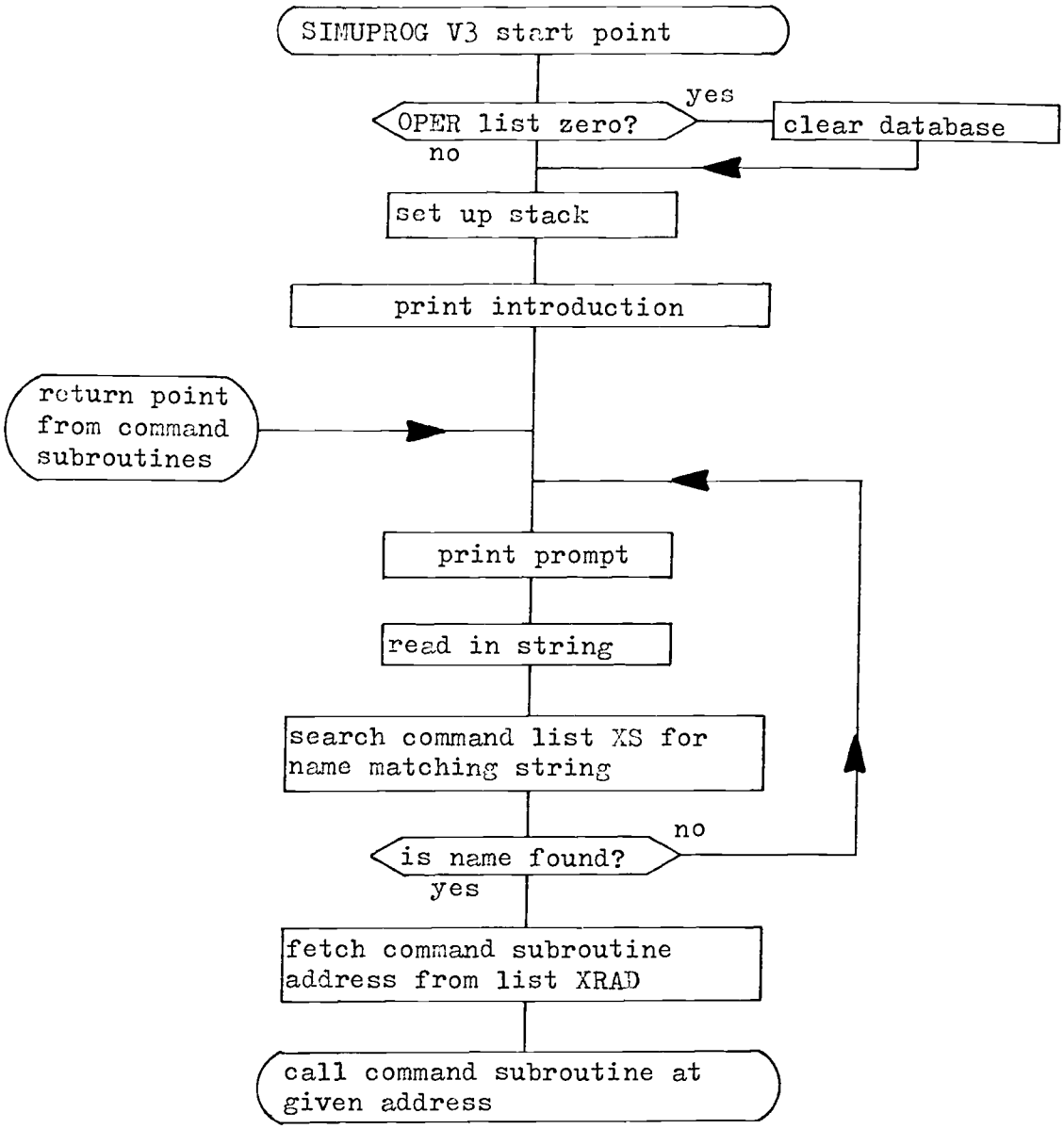


Figure 4.5 Command Level Program

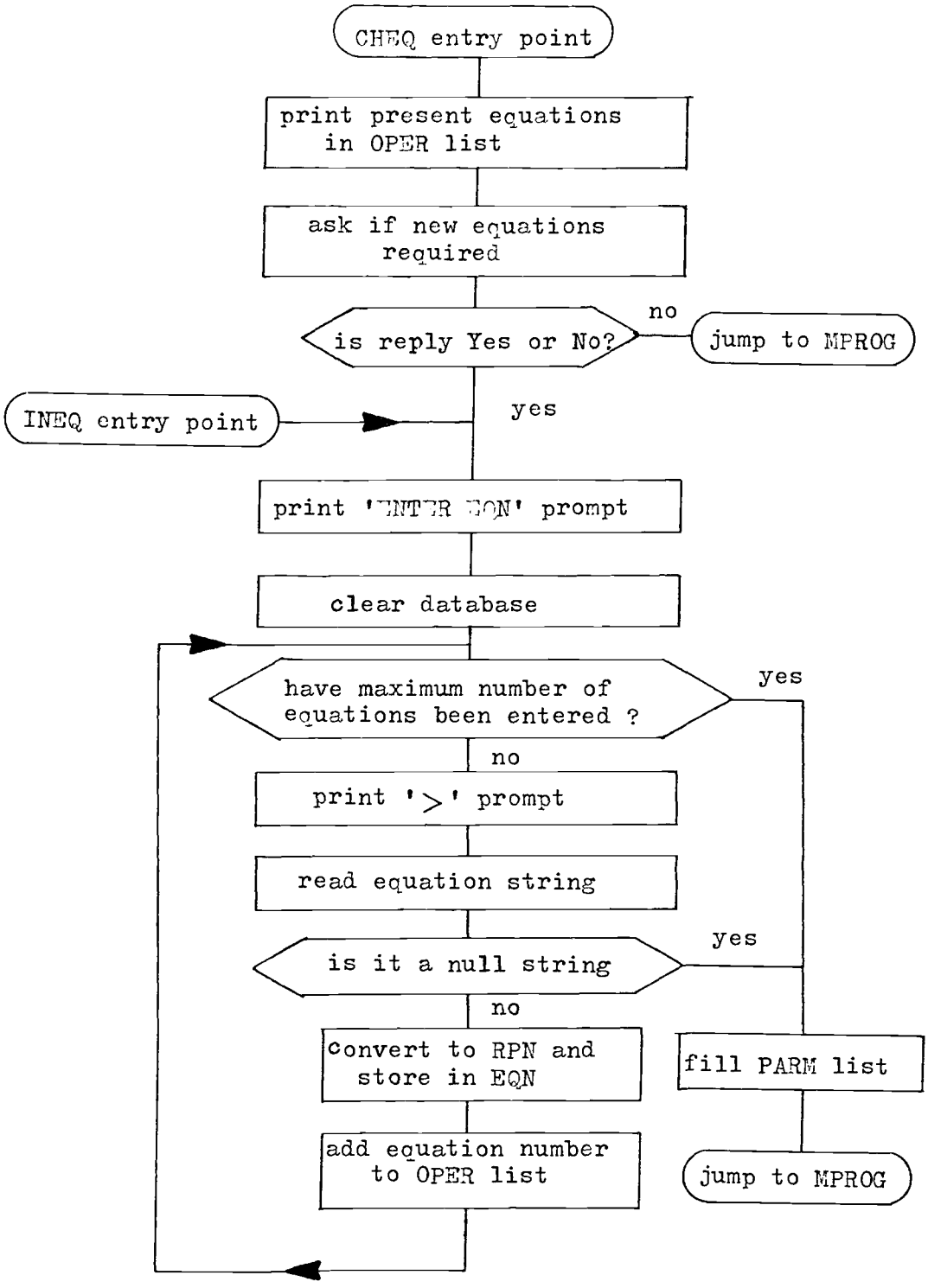


Figure 4.6 CHEQ and INEQ Subroutines

subroutine first prints the set of equations presently held in the equation matrix. After asking if a new set of equations is required, the user reply 'N' (indicating no) returns control to the command loop at MPROG. A reply of 'Y' allows the subroutine to perform the INEQ command, and any other character causes the program to reprompt for a 'Y' or 'N' reply.

The INEQ command part of the subroutine first prints a prompt to enter an equation and then clears the database elements in preparation for the new equation set. The equations are entered by the user in response to a prompt, and are subsequently converted and stored in the equation matrix. The number of the completed equations is then added to the OPER list and the entry loop is repeated. The entry of equations is terminated when either the maximum has been reached, or a null string corresponding to a carriage return only has been entered to indicate the end of the equation set. Once the equation matrix has been completed, a list PARM of all the parameters used is made up by comparing the entries in the OPER list with the full list of all variable names SS. Control is then returned to the commandentry loop.

The APEQ command shown in figure 4.7 is used to add extra equations to the matrix. The subroutine first checks to see if any space is available, and if not prints a message and returns to the command level. The number of the free equation is appended to the OPER list and its position in the matrix is cleared. A prompt is issued to user to enter a new equation and once entered, the equation is converted and stored in the matrix. The previous equation list is cleared and a new one constructed before the program control is

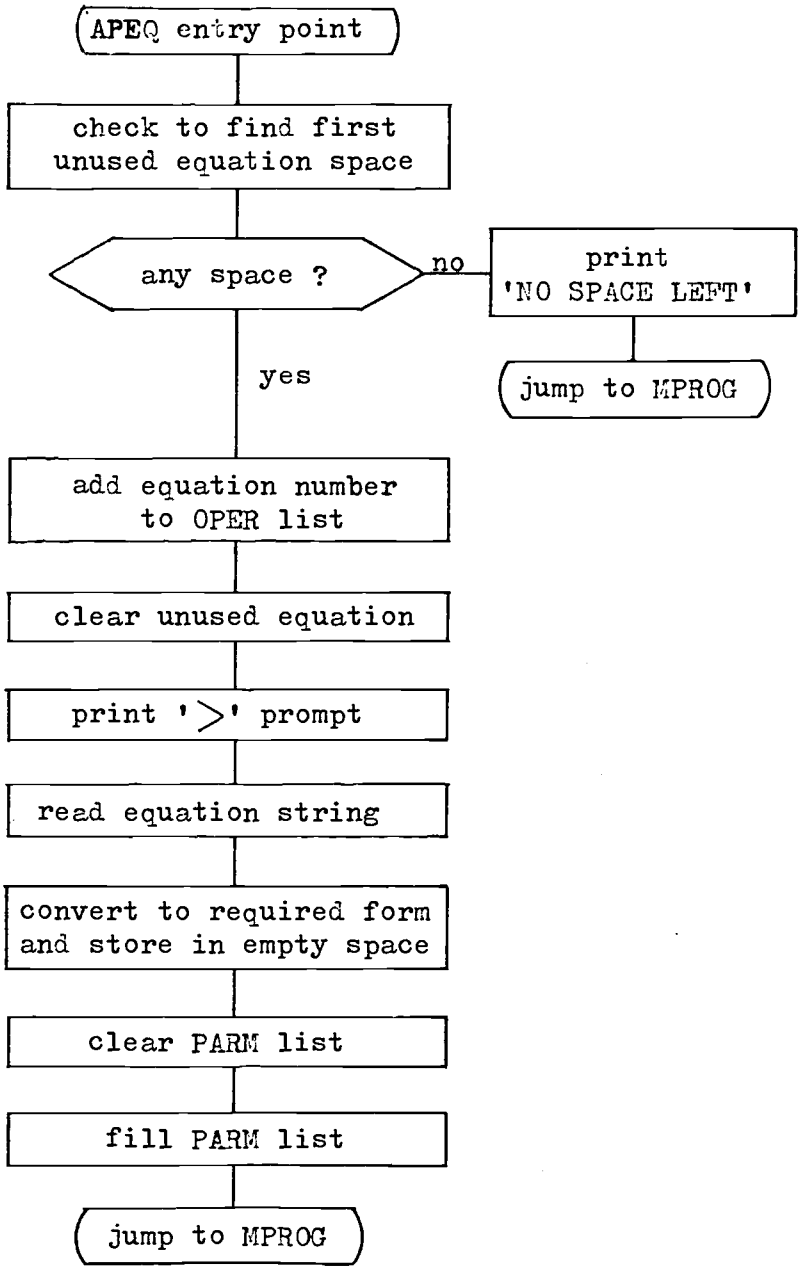


Figure 4.7 APEQ Subroutine

returned to the command level.

The REEQ command shown in figure 4.8 is used to replace an existing equation with a new one. The program first prompts the user to enter an equation number. If the number is valid and that equation exists, the existing equation is printed out, otherwise the user is reprompted. To allow for an incorrect entry or change of mind, the user is asked if the replacement process is to proceed and if not, control is returned directly to the command level. The equation is then cleared and the user prompted to enter a new equation. The equation is then converted and stored in the matrix, and after a new PARM list is generated, control is returned to the command level. The REEQ command at present does not remove the names of parameters only appearing in the old equation, as this would require a larger subroutine to ensure that parameters appearing in other equations were not removed.

The DAEQ command displays all the equations in the matrix, and not just those appearing in the OPER list. After the equations, the present OPER list is also printed and control is returned to the command level. The equations are printed in reverse Polish notation except that the unknown variable is printed first, with the equals sign, for clarity and for ease of checking the actual order of the equations.

The REOR command shown in figure 4.9 is used to generate a new OPER operate list. After the existing list is printed, the program asks if a new list is required, and this allows the command to be used to inspect the list. An 'N' reply returns control to the command level. The program then clears the OPER list and

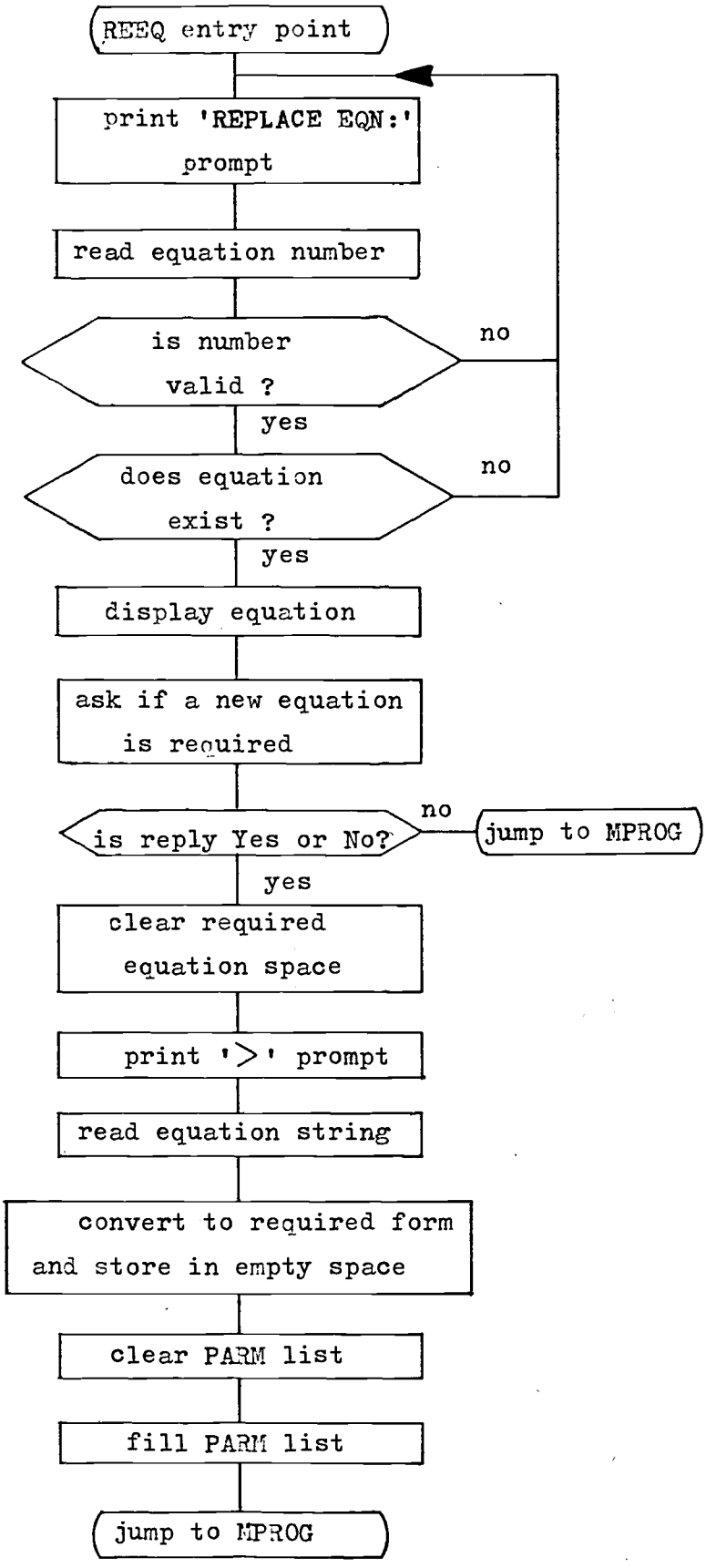


Figure 4.8 REEQ Subroutine

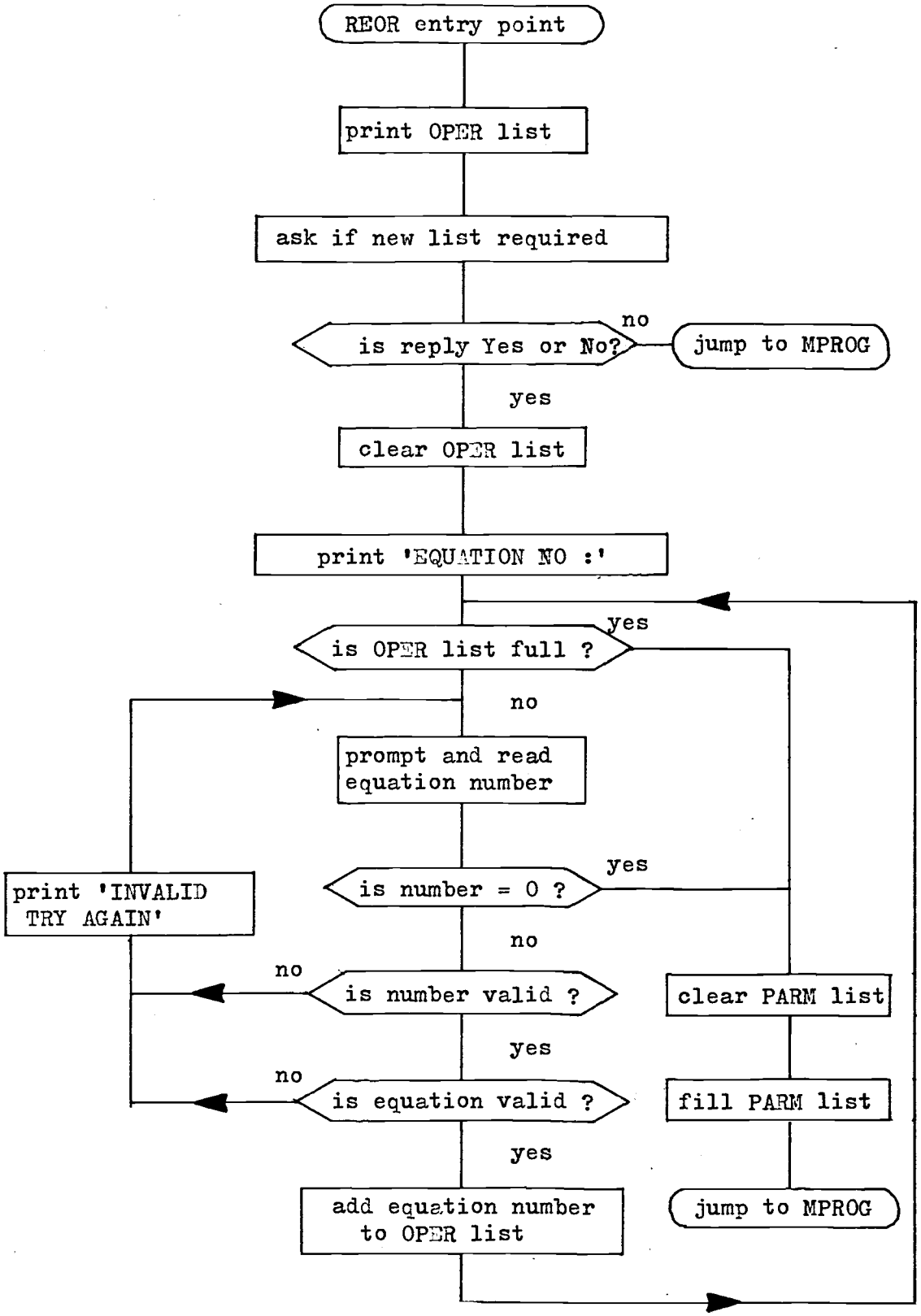


Figure 4.9 REOR Subroutine

prompts the user to enter an equation list. After each number is read a check is made to see if it is a valid equation number and that the equation exists, if not an error message is printed and the user reprompted. Valid equation numbers are appended to the OPER list until either the list is full or else a zero entry, caused by the user replying with only a carriage return, is received. When the OPER list is completed a new PARM list is constructed before control is returned to the command level.

Figure 4.10 shows the subroutine CONV which performs the equation conversion for the CHEQ, INEQ, APEQ and REEQ commands. First the program stores a temporary copy TSS of the variables name list SS so that if the entered equation string is invalid, the equation can be reentered without disrupting the database. The name of the unknown varible is extracted from the input string and list TSS is searched for a match. If not found, the name which may consist of an alphabetic character followed by up to three alphanumeric characters is added to the TSS name list. The variables number is then entered in the VARP pointer list corresponding to the equation. This list links the expression held in the equation matrix with its unknown variable. The pointer to the present symbol in the equation string is incremented to skip the equals sign. The rest of the equation is converted by a further subroutine RPOL. If the equation is valid then names list SS is updated by copying from TSS, and control is returned to the command level. If the equation is invalid the RPOL subroutine causes program control to jump back to the start of the CONV conversion subroutine.

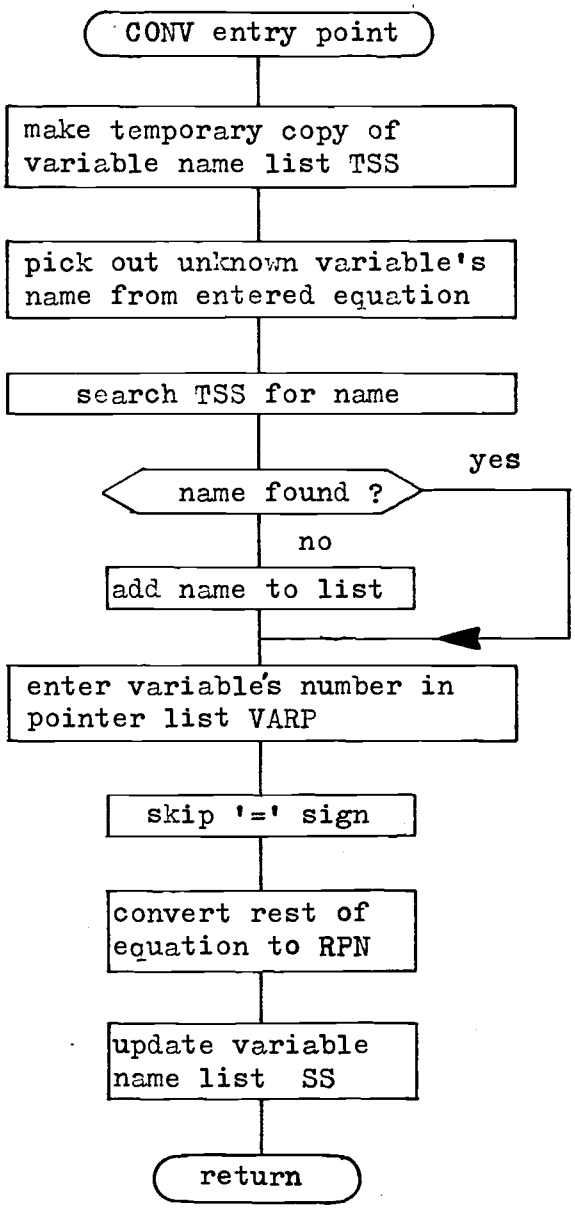


Figure 4.10 Equation Conversion

The RPOL subroutine is shown in figure 4.11 and this subroutine itself uses other subroutines. One such subroutine is used to establish the precedence of a symbol according to the table given in figure 4.3, both when it is inputted and when it is pulled off the stack. The other subroutine, used by the CONV subroutine, extracts a name from the equation string. After setting up the conditions for the conversion, the program then enters the main decoding loop. First the next character, indicated by the pointer into the equation string, is copied into the input symbol variable SY. If the symbol is zero or a space then the equation string is finished, and if the stack is then empty, the equation has been converted correctly and control is returned to the calling subroutine CONV. If an error is detected then an error number is printed and the user is prompted to retype the equation. Since the CP 1600's subroutine mechanism does not use the stack, the program jumps directly to the start of the CONV subroutine which means that any actions of RPOL and CONV preceeding the detection of the error are wiped out and the conversion is started afresh. Another effect of the call mechanism is that the main processor stack can be used during the conversion, which is more efficient than creating a software one. A comma is used to separate the parameters of a function and this has the property of allowing any of the function parameters to be an expression, without the use of extra pairs of brackets.

If the symbol is alphabetic then it is the start of a variable name, and the full name is then read. List TSS is searched for the name and if not found, it is added to the list. If the symbol is a '%' sign then the following name is that of a function. After

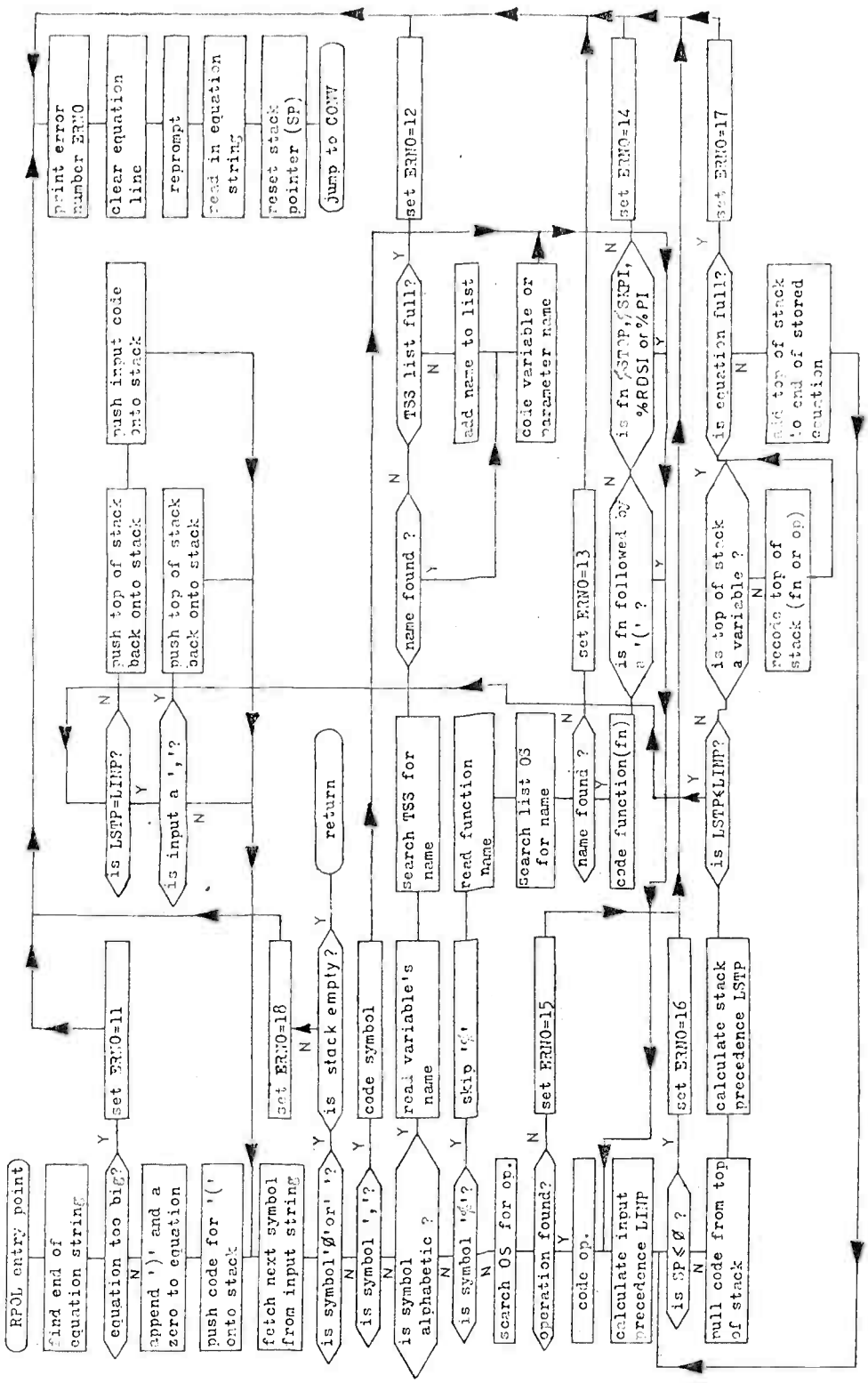


Figure 4.11 RPN Conversion Subroutine

skiping the '%', the function name is read and the list of functions OS is searched for a match. A check is made to see if the function is followed by brackets enclosing the parameters. The functions %STOP, %SKPI, %RDSI and %PI do not need parameters, so are exempted from this check. Any valid symbol left at this point must be an operation, and list OS is searched for a match. Any symbol not found is invalid and will generate an error. All the symbols are converted to an intermediate code for use during the conversion and finally changed to the actual code prior to storing. The stored coding represents variables by positive integers and functions by negative numbers, but as this is inconvenient for calculating the precedence, the intermediate code represents functions as integers in excess of 200.

The input precedence is calculated according to the input symbol. After checking that the stack is not empty, the top symbol is pulled from the stack and its precedence calculated. If the precedence of the symbol from the top of the stack is greater than that of the input symbol, the symbol from the top of the stack is added to the converted equation. The two precedences are equal when the top of the stack is a left bracket and the input symbol is a right bracket, in which case both brackets are discarded, or a comma, in which case the left bracket is reinserted onto the stack and only the comma discarded. If the input precedence is less than that of the stack then both the top of the stack and the input symbol are successively pushed onto the stack.

4.4. System Control

The SPAR command shown in figure 4.12 allows the user to change the parameter values of the model equations. First the present parameter names and values are printed after which the user is asked if new values are to be entered. If new values are to be entered then the program proceeds through the parameter list, printing each name followed by a prompt for the user to enter a value. This value is converted to the 32 bit floating point format and stored in both the VAR and TVAR value lists. If the user does not enter any digits before pressing the carriage return, the present value of the parameter is retained. During the floating point conversion checks are made as to the validity of the conversion. If an entry proves to be invalid the user is again prompted to enter an new value.

The SIC command allows the user to set up the initial conditions for a model run. This subroutine is very similar to that for the SPAR command, with the only difference being the addressing method for the required variables. For the SPAR subroutine, the parameter list PARM points directly to the required variable names and values. However for the SIC subroutine, the operate list OPER points to the required equations and the associated VARP pointer array entries. The VARP entries point to the required variable names and values.

The command SDT shown in figure 4.13 can be used to change the value of a parameter called 'DT'. The subroutine first searches the variable name list SS for the name 'DT', and if found prints its value. The user is then prompted to enter a new value and this is subsequently converted and stored in the location for DT.

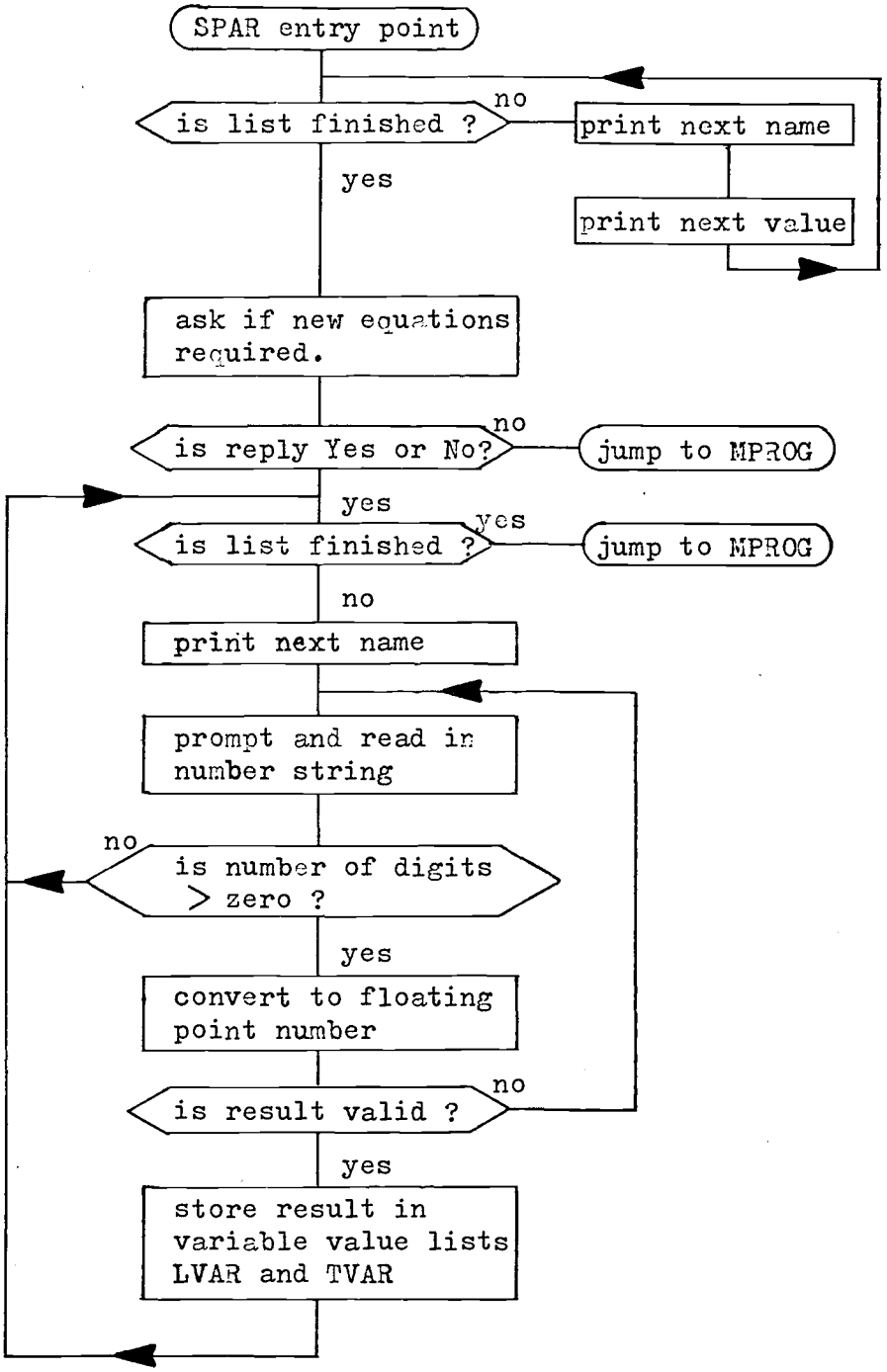


Figure 4.12 SPAR Subroutine

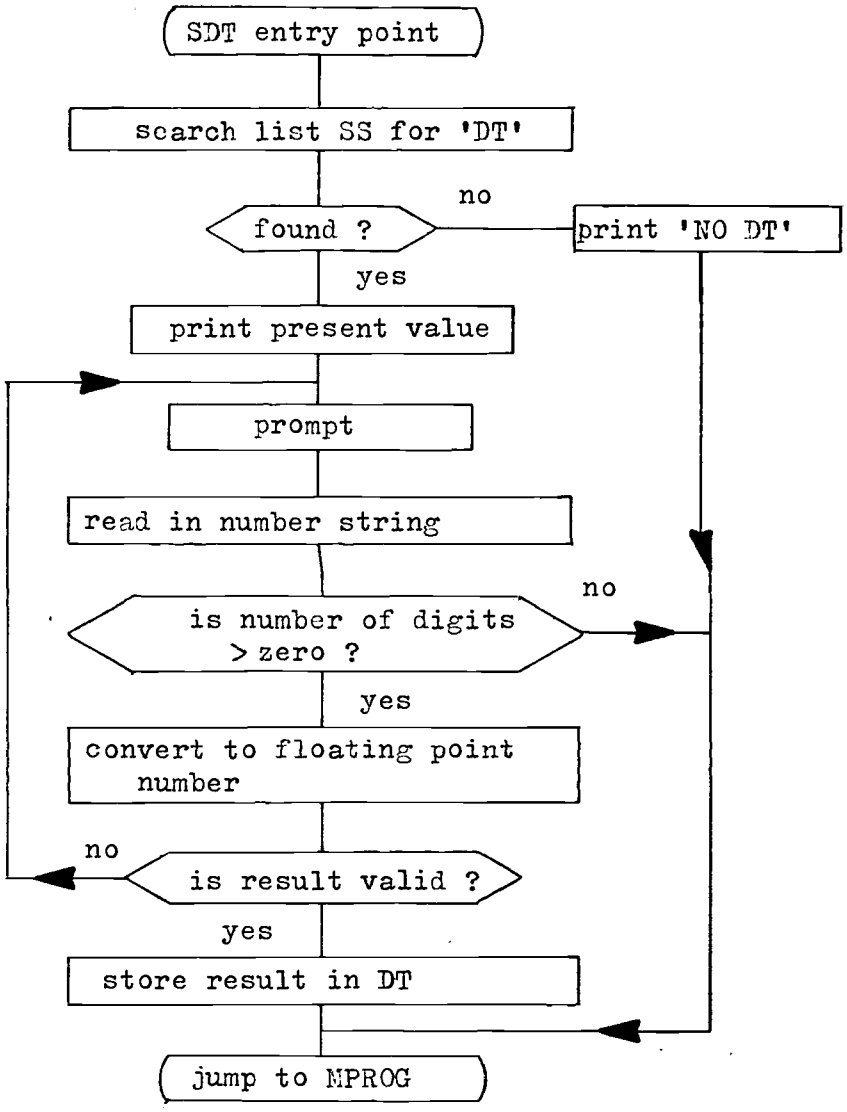


Figure 4.13 SDT Subroutine

The STPS command shown in figure 4.14 sets the maximum number of steps for which the model will run. After first printing the present maximum number of steps, the subroutine asks if a new value is required. If a new value is required then the user enters a suitable integer value which is then stored. The 16 bit integer values are stored in 2's complement form and give a maximum value of 32678.

The STAB command shown in figure 4.15 is used to set up the tabulation list for printing results. After first printing the present tabulation list, the user is asked if a new list is required. If a new list is required, the previous list is cleared and a new list built up. In response to a prompt, the user enters a name which is compared with list SS for a match, and if found it is added to the tabulation list. The tabulation list TAB is terminated either when it is full or when a null entry consisting of just a carriage return character is received.

The TINT command sets the interval at which tabulation printouts occur , and the subroutine operates in an identical way to STPS.

The SIM command shown in figure 4.16 allows the user to change the data input mode. To save space the interactive 'Y' or 'N' (yes or no) decisions are made by a separate subroutine which is also used by other commands. Since all the command subroutines return control to the same point, straightforward jumps are used instead of subroutine calls and returns. After printing the present input mode and asking if a new mode is needed, the program if required allows the user to enter the new input mode in response to a series of questions and prompts. Serial input mode just means that the user is allowed

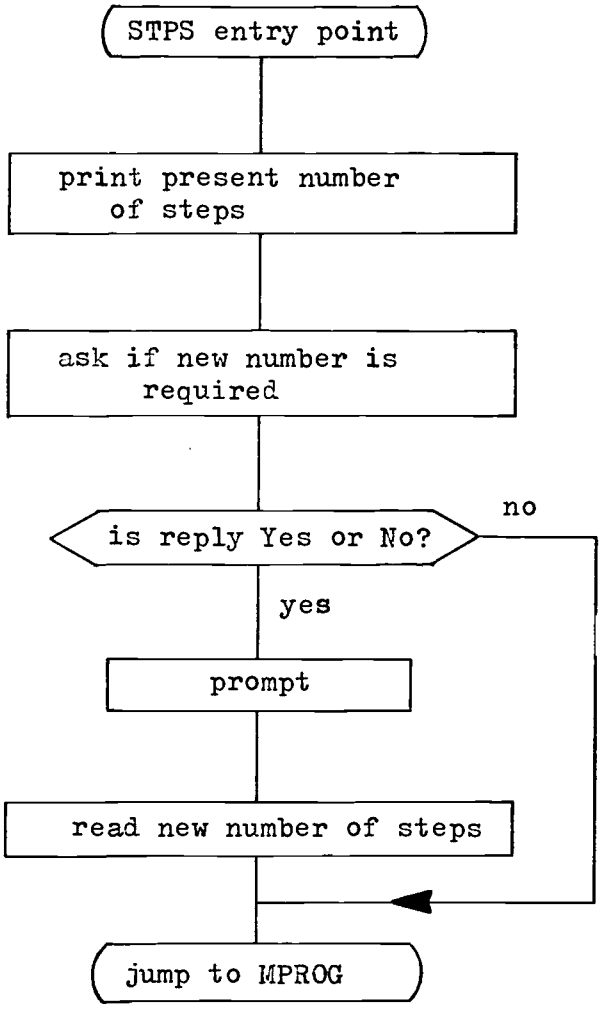


Figure 4.14 STPS Subroutine

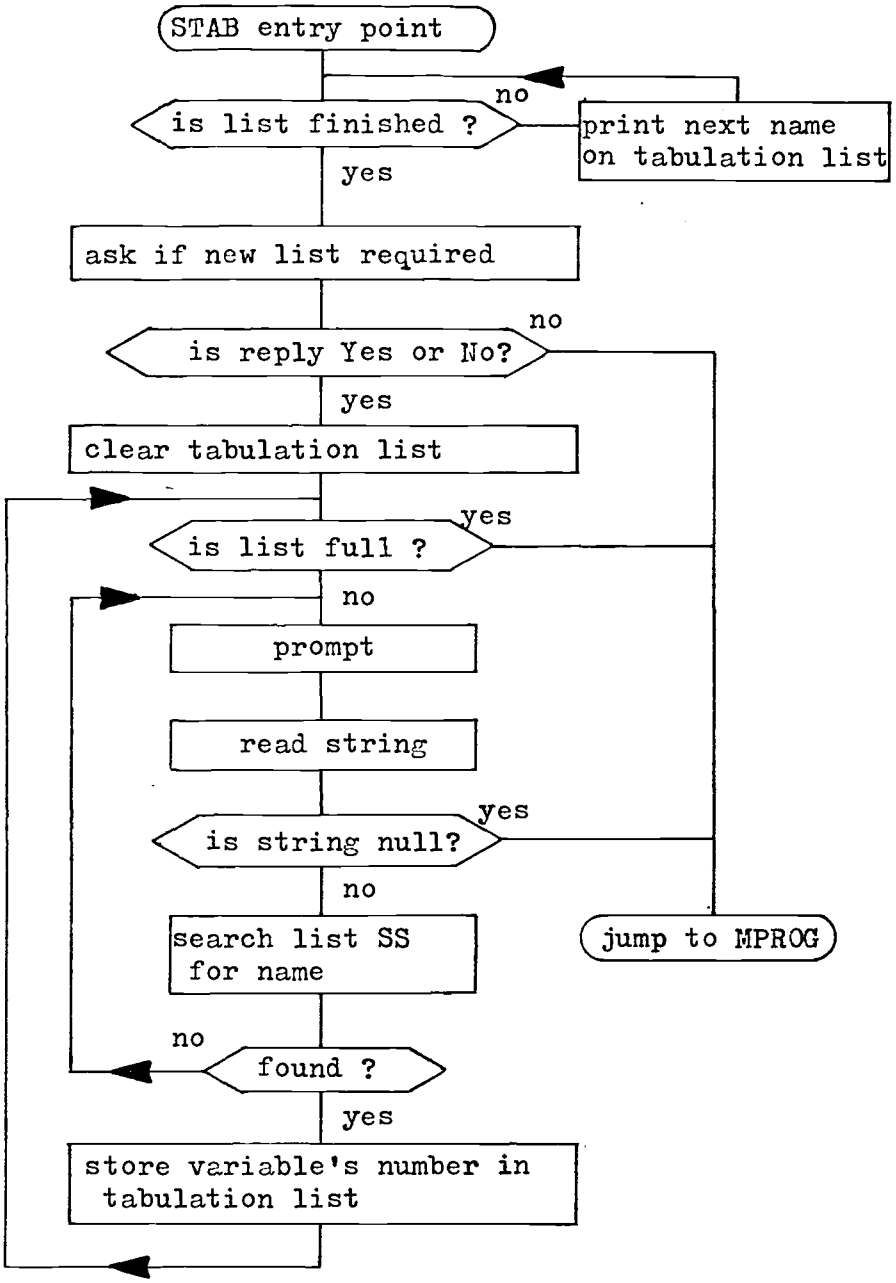


Figure 4.15 STAB Subroutine

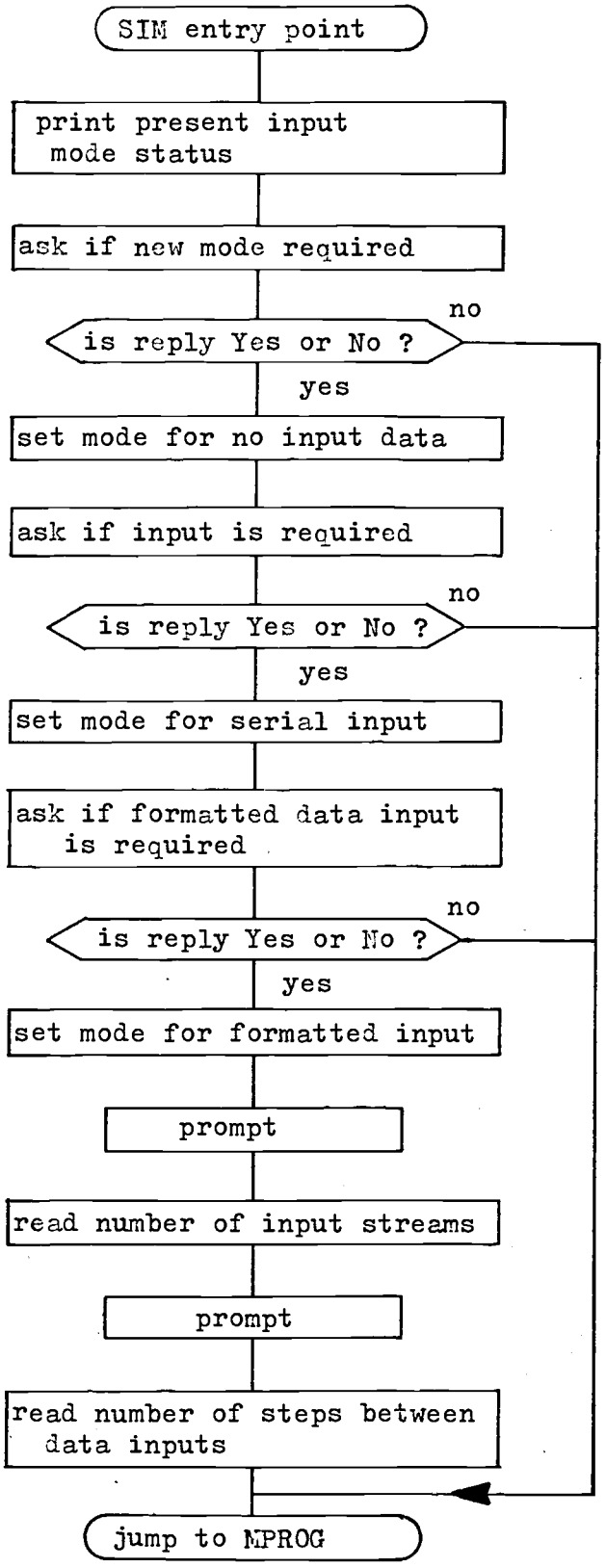


Figure 4.16 SIM Subroutine

to use the %RDSI function in the model equation to input data. If the mode is set for formatted data input, the program automatically reads data from the input device set by the user. The user also sets the number of streams of data values entered each time, and these are stored in a temporary array until they are updated. The %READ function is used to access these values.

A list of variables whose values are to be plotted as graphs can be set up using the GRAM command shown in figure 4.17. The present graph list, if any, is printed and the user asked if a new graphic mode is required. A graph list is entered by typing the name of a variable in response to the graph number. The variable name list SS is searched for the name, and if found, the name is added to the graph list. The graph mode which indicates the number of graphs in operation is also incremented.

The GINT command works in a similar fashion to the STPS command and is used to enter the interval between graphic output points.

The scale factors used by the program when plotting the graphs can be set up using the GRSC command shown in figure 4.18. If a graphic output mode is set, then the entries on the graph list are printed together with their present scale factors and the user is prompted to enter a new value. When a number string is entered, it is converted to floating point form, and if valid, is stored as the required scale factor. If the user enters a carriage return without any preceeding digits, the existing value of the scale factor is kept so that new values need not be entered.

The TABD command allows the user to select the destination for the tabulation printout. The subroutine asks if the VDU or Teletype is required and sets the tabulation destination flag before returning to the command level.

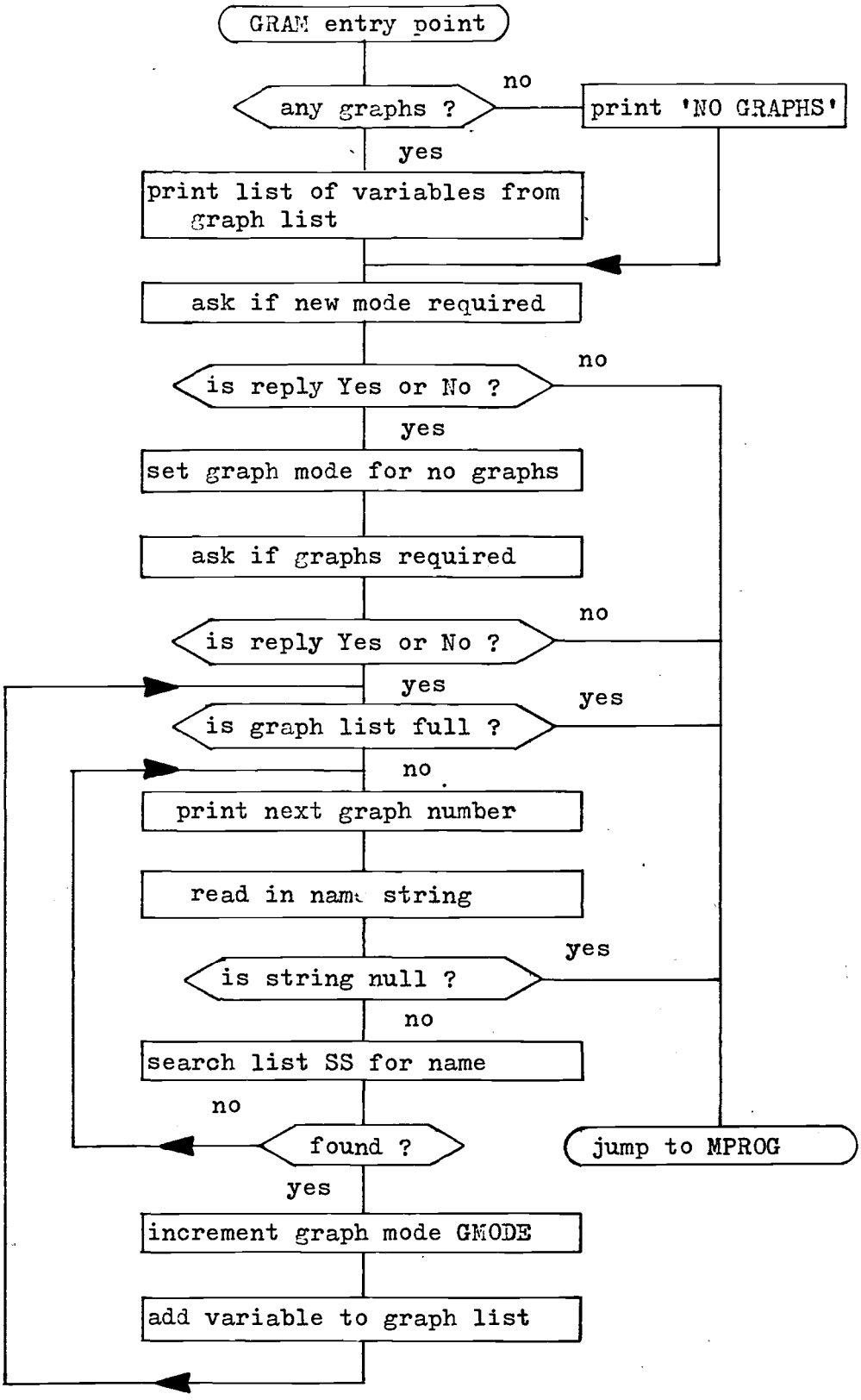


Figure 4.17 GRAM Subroutine

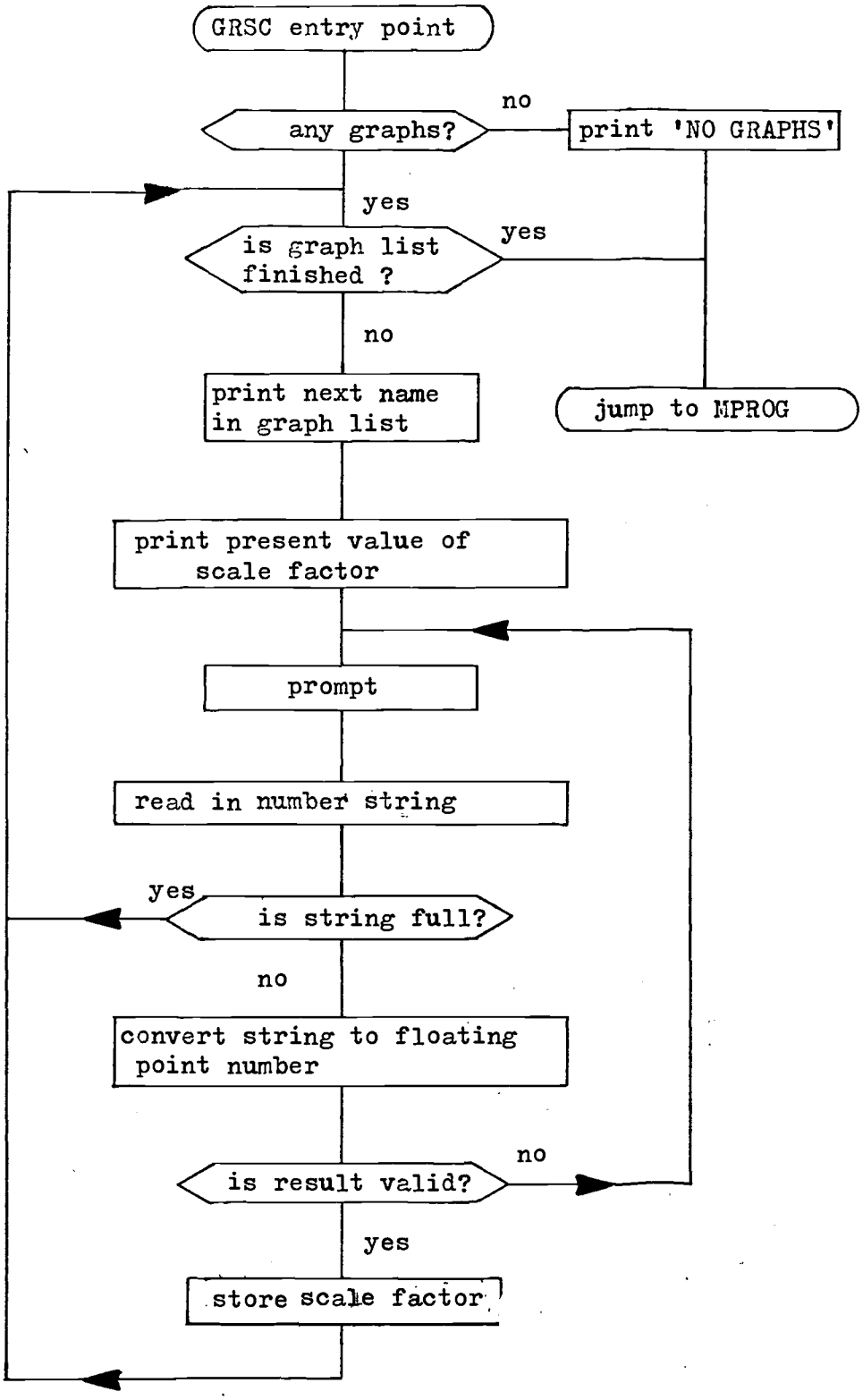


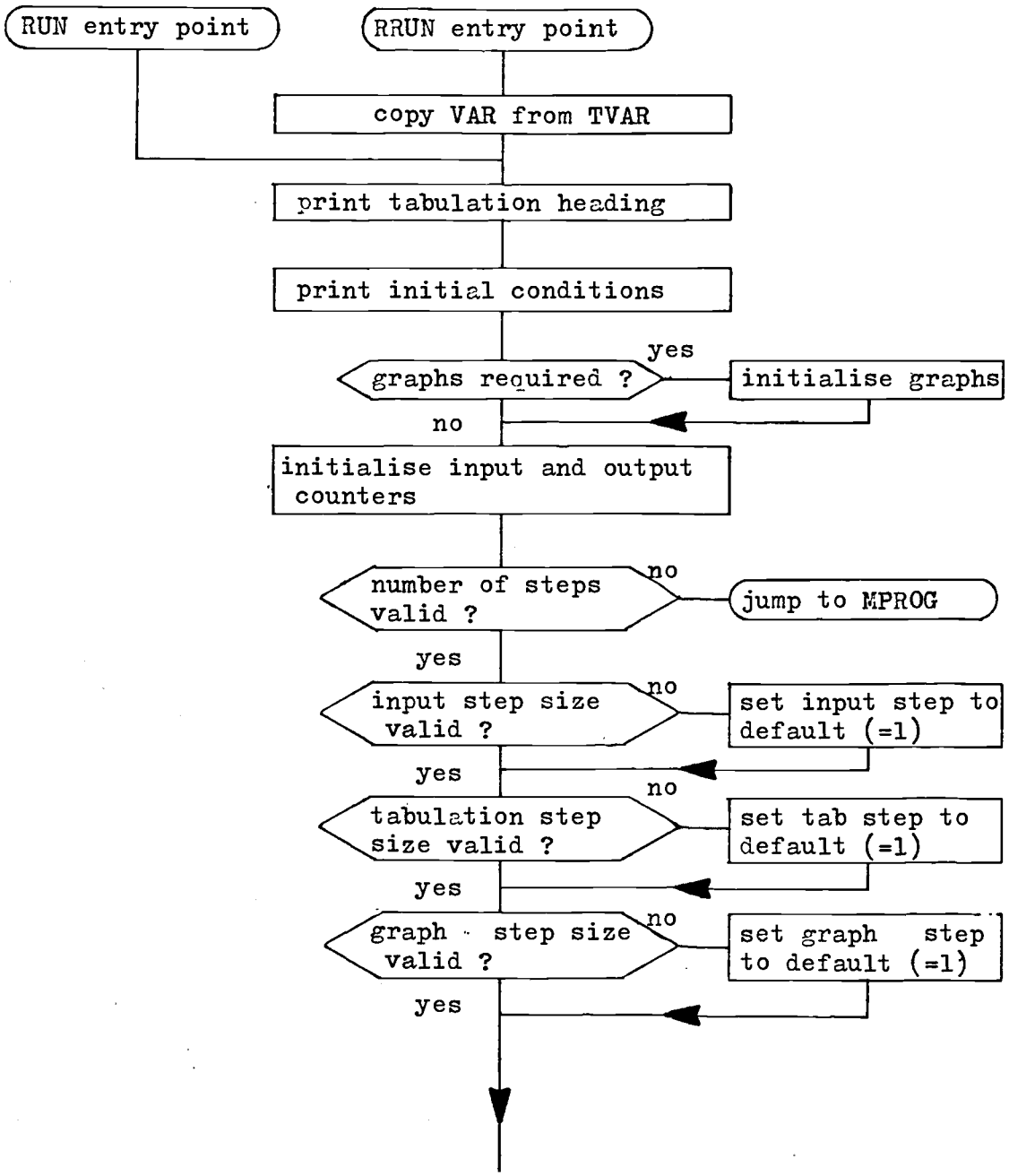
Figure 4.18 GRSC Subroutine

4.5 Running Equations

Figures 4.19a and 4.19b show the RUN and RRUN subroutines which are used to implement the stored equation set. The RUN and RRUN commands differ only in that the RUN command uses the final values of the variables from the last run as its initial conditions, and the RRUN command uses the last entered set of initial conditions held in TVAR. Since the SIC command changes both the TVAR and VAR value lists, the use of this command fixes the set of initial conditions whether or not the values have been changed.

The common part of the subroutine first prints the names and initial values of any variables in the tabulation list, so that they form the heading and first line of a table. The required output control counters are then initialised, and the run control parameters checked for validity. The step sizes for input, tabulated output and graphic output all have default values of one. Note that the initialisation of the graphs also includes plotting the initial values of the variables.

The main calculation loop shown in figure 4.19b repeats each calculation step until all the steps have been performed, or the run has been stopped prematurely by a %STOP command or by the user entering a 'control C' character. If a formatted data input mode has been set, then each step is automatically checked to see if it is a multiple of the data input step size. If it is then the program attempts to enter and store data from the high speed tape reader. Invalid data on the tape will produce an error message and terminate the run. The program then proceeds sequentially through the operate list OPER and evaluates the corresponding equations held in the



(continued on next page)

Figure 4.19a RUN and RRUN Subroutines

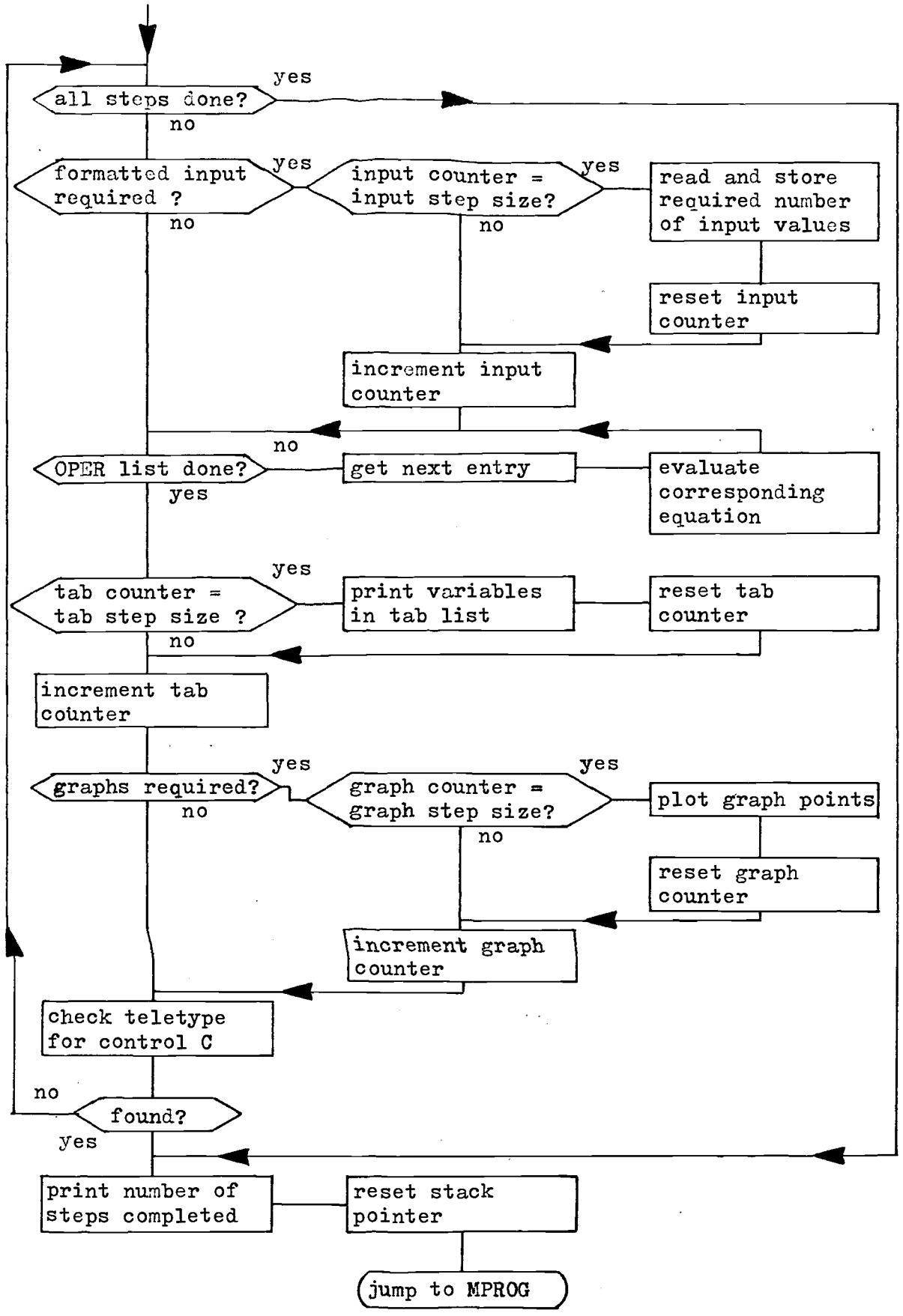


Figure 4.19b RUN and RRUN Subroutines

equation matrix EQN. If the number of steps so far performed happens to be a multiple of the tabulation step size, the present values of the tabulation list(TAB) variables are printed. If the steps are a multiple of the graph step size, then the value of the variables in the graphic output list GRAF are multiplied by the associated scale factors and are then converted to 8 bit integers for transfer to the Z 80 microprocessor based graphic display. At the end of a step the program checks in case a control C character has been entered on the teletype to terminate the run. If an arithmetic error is encountered, such as an overflow or a division by zero, and error message is printed indicating the arithmetic operation in progress at the time and the run is terminated. The number of steps of the run completed is printed only if the full number has been reached or a control C is encountered.

Figure 4.20 shows more detail of the equation evaluation, which is performed by a subroutine CALEQ. The subroutine examines each entry of the equation in turn until the equation is terminated either by a zero entry, or when the end of a line in the equation matrix is reached. A negative entry indicates an operation or function, and after 2's complementing, the entry is added to the base address of table ORAD. ORAD holds the addresses of the arithmetic subroutines and a software constructed subroutine call is used to transfer program control to the address specified in the ORAD table. If an equation entry is positive, it is doubled and added to the base address of the variable value list VAR. The 32 bit value is then pushed onto the stack for use in a later calculation. All entries are checked for validity. When the equation is terminated, the result is pulled from the stack and stored in the variable indicated by the VARP

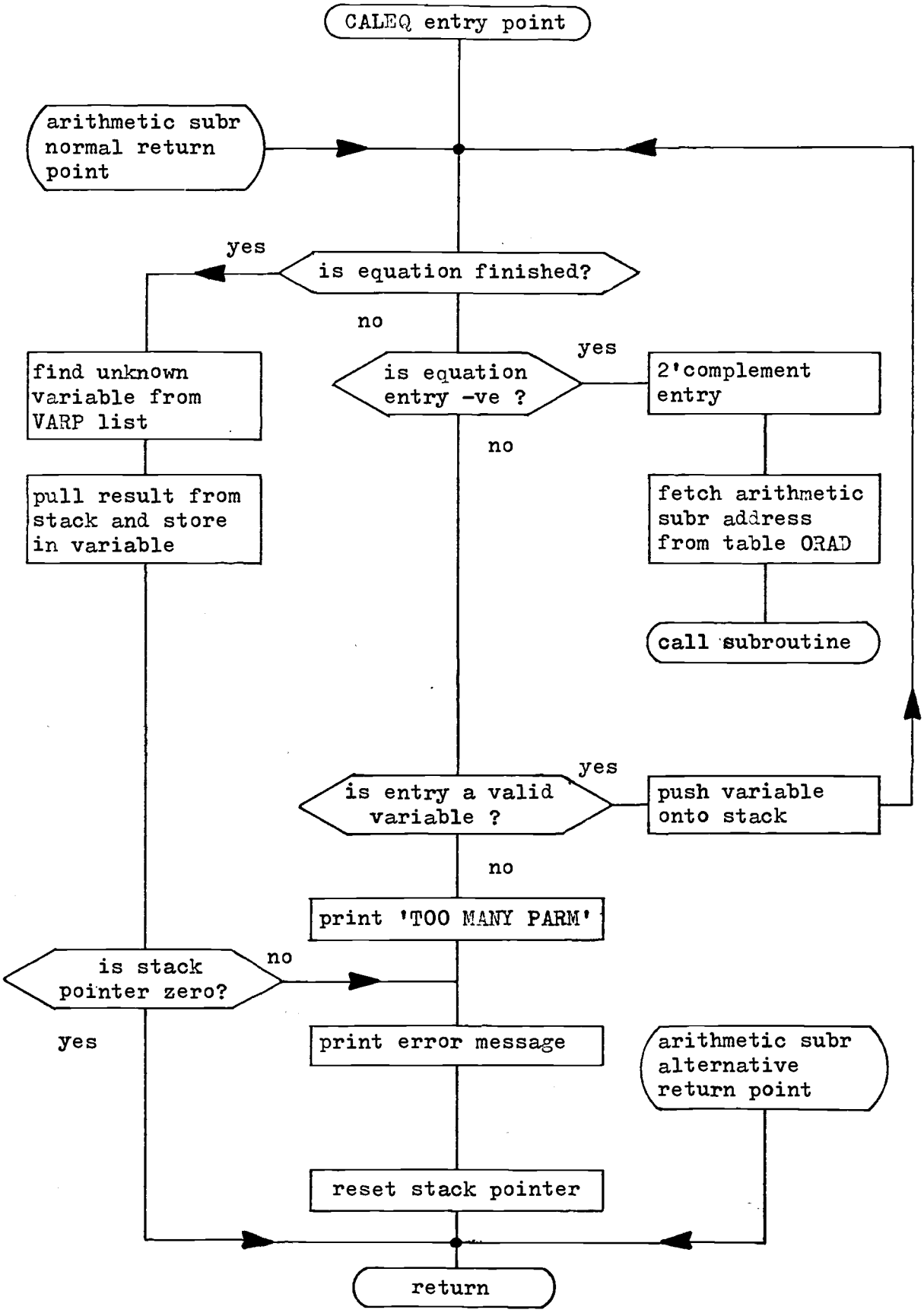


Figure 4.20 Equation Calculation

variable pointer list. If the stack is not empty when the result has been pulled, then the equation is invalid and the result is not stored in the variable. The arithmetic subroutines used for conditional equation evaluation return control to the beginning of CALEQ only when their conditions for evaluation are satisfied. When the conditions are not satisfied, the equation is terminated immediately and control is returned to the end of the CALEQ subroutine.

4.6. Displaying Results

The headings for the tabulated output are printed by a subroutine PHEAD shown in figure 4.21. If the VDU has been specified by the user, using the TABD command, the link available flag is checked to see if the Z80 system is ready to operate the link. If the link is not available, the program changes the tabulation destination to Teletype thus overriding the TABD command. The Z80 is prepared to receive the heading by sending the required code to the link control latch. The heading is made up of the names of the variables in the tabulation list TAB. Output on the Teletype is performed with the aid of a printing subroutine in the GIMINI's monitor and some utility subroutines built into the SIMUPROG program. For output using the Z80's memory mapped VDU, the individual characters are transferred by handshaking to the Z80 input hardware.

The actual tabulation of the values of the variables is performed by the PVARS subroutine which operates in a similar way to PHEAD except that instead of names, the actual values are printed. The values are printed in fixed point format and a flag (FLAG) is used by the floating point Teletype printout subroutine (PRINTF) to tell

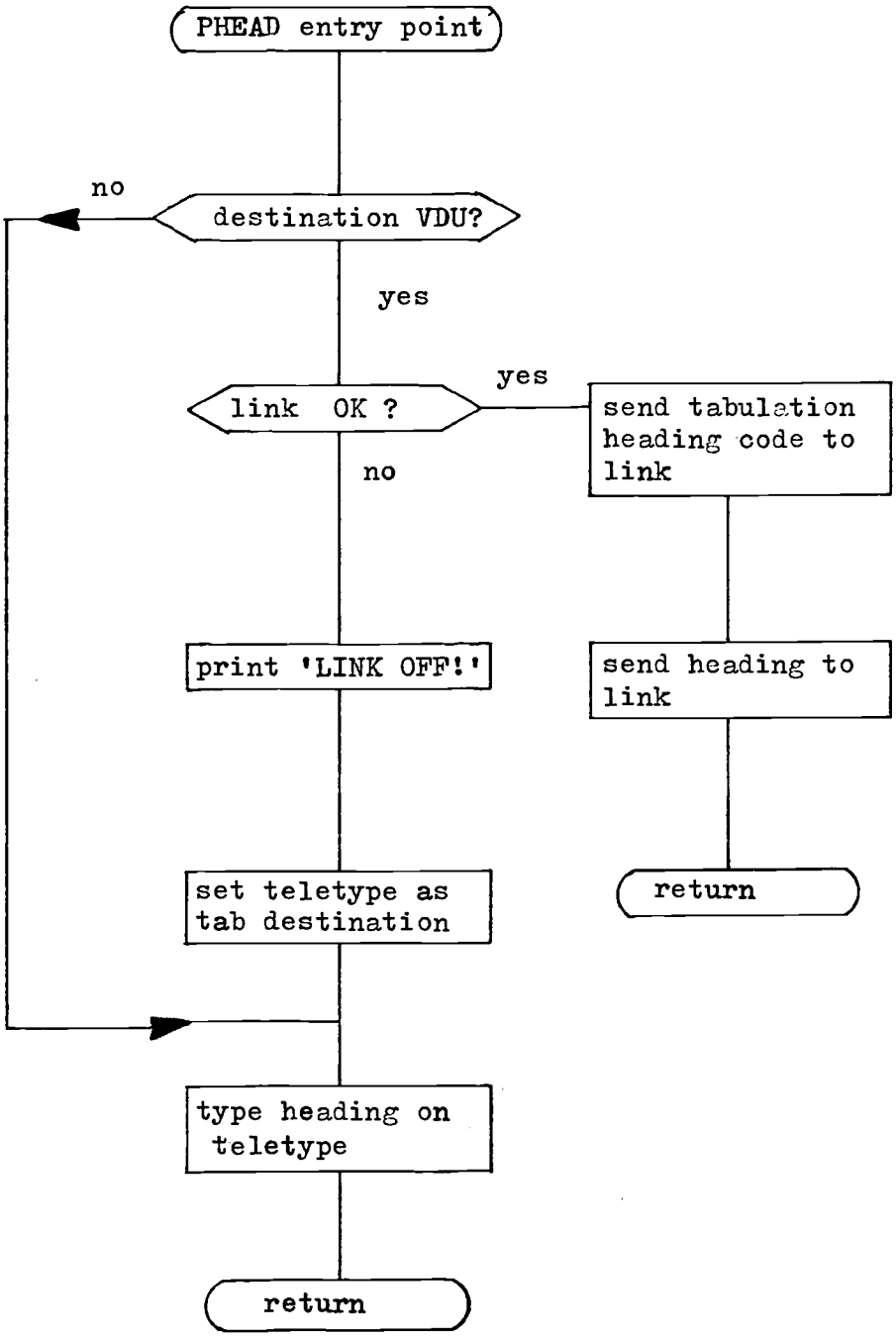


Figure 4.21 Result Tabulation

the PVARS subroutine how many characters have been printed so that column alignment can be maintained by printing extra spaces. For transfer to the VDU the subroutine itself counts the characters and adds the required spaces. Note that the VDU is only 48 characters wide, so that if more than 4 variables are on the TAB list then each printout will take two lines and the columns will become obscured.

Figure 4.22 shows the GZERO and GPLOT subroutines used for graphic output. The GPLOT subroutine converts the values of the variables, indicated by the graphic output list GRAF, into 8 bit integers and sends them to the Z80 for display. In additon to this, the GZERO subroutine also clears the previous graphs. Both subroutines first check the link available flag to see if the Z80 is ready, and if not a message is printed and the graph mode changed to no graphs. For GZERO the graphs are cleared by sending the code for a graphic command to the link control latch, and then sending a zero to the data link under the control of the handshaking. The use of an extra data byte for the graph command is provided both as a protection device and to facilitate future expansion where graph formats and possibly scaling are under control of the simulation model. The handshaking involved in data transfer is a combination of hardware and software. The Z80 side of the link is purely hardware whereas the CP 1600 uses software to read the ready flags from the Z80.

Both subroutines plot the data points in the same way. For each entry in the graph output list, the code for that graph is first sent to link control latch and then the converted data point is sent to the Z80 using handshaking to control the data transfer. The 8 bit

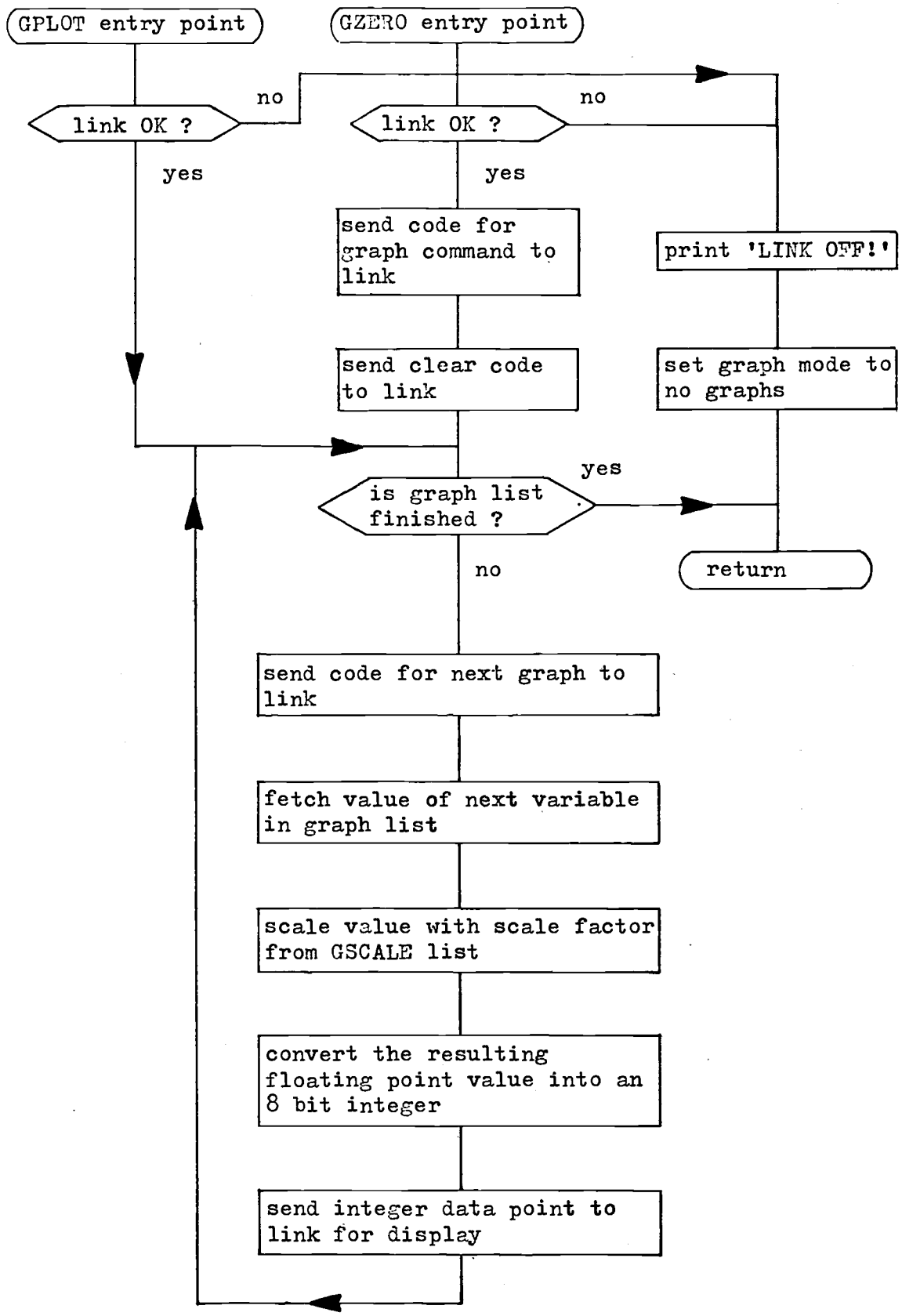


Figure 4.22 Graphic Output

resolution graphic display is used to represent values of +1 and -1, so the value of the variable for display has to be divided by the previously entered scale factor to get it within this range. The resulting value is then multiplied by 127 and converted to an integer. The final 8 bit value is then formed by adding 128 to the integer since the output of the D/A converters do not accept negative values. If any variable value is outside the display range then it is limited to either the maximum or the minimum on the display.

The actual refreshed display of the graphs is performed by the Z80 program shown in figure 4.23. After first setting up the D/A converters and the hardware for the data link to the CP1600 the program then repeatedly displays the graphs stored in memory so that the display appears static. There are at present two different display modes and these are controlled by the NASCOM-1's keyboard under control of the display program itself, so that only the graphic mode control keys will have any effect on the program. The first mode displays up to four graphs simultaneously against time, and the second mode displays the first two graphs against each other. For the plot against time, each graph is in turn plotted by updating the Y and X axis D/A's with the values from the graph data and an incrementing counter respectively. After each complete set of graphs has been displayed, the program checks the NASCOM-1 keyboard to see if a 'P' character has been entered indicating that the program is required to switch to a phase type plot. Since the complete display of all the graphs takes less than 1/30 of a second, there is no flicker on the display and no need to latch the keyboard entries.

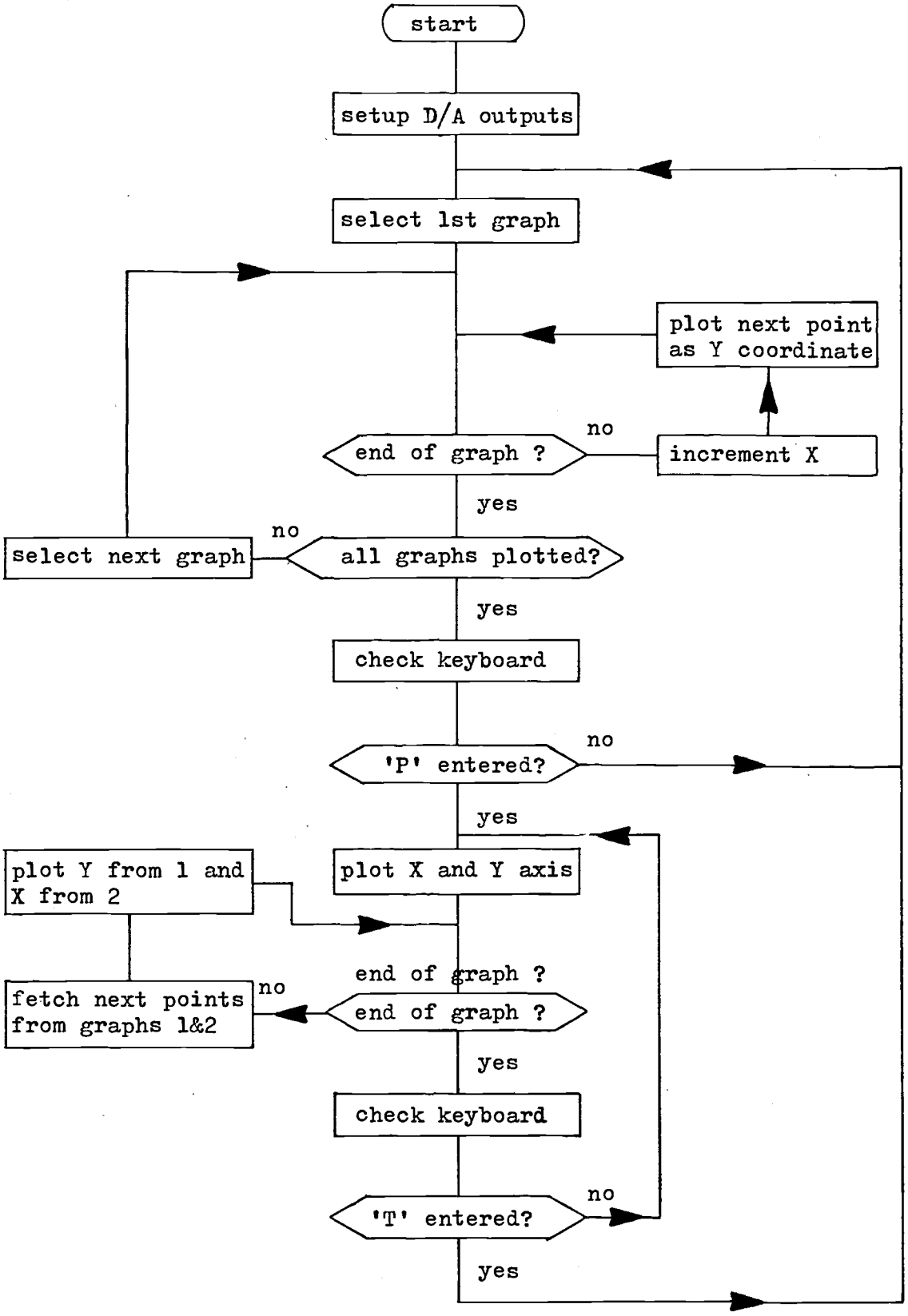


Figure 4.23 Refresh Graphics

Originally the time plots also had the X-Y axes but this was dropped as it tended to obscure the graphs. For the phase plot, the axes are plotted and the graph displayed by plotting each point with its Y value taken from graph 1 and the X value taken from graph 2. After both graph and axes have been plotted, the keyboard is checked for a 'T' entry indicating a return to a time plot.

The data for the graphs is updated by an interrupt service routine which is invoked by the link hardware to handle a data transfer. On recept of an interrupt indicating the transfer of data to the Z80, program control is transferred from the display program to an interrupt handling routine. The handling routine reads the link control latch and then calls up a subroutine to deal with the operation specified by the link control code. With the availability of the graphics, the limited output tabulation on the VDU was not used, so the latter was omitted from the present version of the display program.

5. USING THE SIMULATION SYSTEM

5.1 Operational Details

The SIMUPROG V3 object program is loaded into the GIMINI using the built in relocating paper tape loader. If a previously generated model is to be used, then that is also loaded after the simulation program so that it overwrites the cleared database. The program is started by using the monitor to enter the start address which was previously printed out by the loader.

On startup the program announces itself and prompts the user to enter a command, the present list of commands is given in figure 5.1. All commands and data, except for simple yes or no answers, are entered using a line editor which enables the user to delete characters one at a time by entering a 'rubout' code. Entire lines can also be deleted provided that a '←' character is entered instead of the carriage return which normally indicates the end of an input line. For questions which require a yes or no answer, a single character reply of 'Y' or 'N' is entered. Wherever possible, a carriage return only reply to a prompt for a numerical value means that the present value is retained.

New equation sets are entered in algebraic form using either the INEQ or CHEQ commands. The CHEQ command can also be used to print the present equation set in reverse Polish notation. The DAEQ command can be used to printout all the equation held in the equation matrix, and not only those presently involved in the equation set specified by the operate list. Equations can be changed by using the REEQ command, and extra ones added with the aid of the APEQ command. Equations cannot be deleted, but they can be overwritten.

name	function of command
INEQ -	input new equation set
CHEQ -	change equation set after displaying old set
DAEQ -	display all stored equations
APEQ -	append an extra equation
REEQ -	replace a single equation
REOR -	reorder equation set
SIC -	set initial conditions of variables
SPAR -	set parameter values
SDT -	set the value of a variable called DT
SIM -	set data input mode
STPS -	set number of steps to be run
RUN -	run equation set using present variable values
RRUN -	rerun equation set from initial conditions
STAB -	set tabulation printout list
TABD -	set tabulation printout destination
TINT -	set tabulation printout interval
GRAM -	set graph mode
GRSC -	set graph scale factors
GINT -	set graph plot interval
STOP -	stop program and return to monitor

Figure 5.1 SIMUPROG V3 Commands

By using the REOR command, which generates a new operate list, they can however be dropped from the present equation set. When new values are required when using the SIC, SPAR, SDT, and GRSC commands, they are entered as fixed point decimal numbers with up to 12 digits and having a magnitude not exceeding 2^{23}. Numbers are entered using the line editor and are terminated by a carriage return.

The tabulation list is entered using the STAB command and up to six names can be entered. The TABD command used to set the tabulation printout need not be used, as the default destination is the teletype. The values entered for the TINT, STPS, and GINT commands must be integers in the range 1 to 32768. The GRAM command is used to set up the graphic output if the NASCOM-1 and graphic hardware are connected. Up to four variables can be plotted and the GRSC command is used to enter the associated scale factors.

The RUN command starts a simulation run using as its initial conditions the values of the variables left by the previous run. If it is desired to rerun the simulation from the last set of initial conditions, the RRUN command is used.

Since the simulation system equations are entered as assignments rather than equalities, the differential equations should first be rearranged so that the highest order derivative only is on the left hand side. The algebraic equations entered consist of operations, functions, and variables. Any constants must be entered as parameters, which are variables whose values do not change during a run. Variables are specified by a four character name consisting of an alphabetic character followed by up to three alphanumeric characters. The available operators are the four basic arithmetic operations together

with brackets and the comma which is used to separate function parameters. Functions consist of a % sign followed by a name similar to a variable, and the present available functions are given in figure 5.2. The control functions allow conditional implementation of equations and can be used to provide such operations as limiters and nonlinear functions. During the conversion of the equations, numerous checks are made on the validity of the equations and figure 5.3 gives the errors which are flagged with numbers. Error messages are given with any other errors discovered, and the numbers are only used to save memory space. A simple example is given in figure 5.4 of the squares of the integers from one to ten. Note that the user entries are underlined.

Two problems were used to investigate the computational accuracy and speed of a variety of integration methods. The first problem given in section 5.2 was the step response of a simple first order system, and this was used with a variety of Runge Kutta single step integration methods. Multiple step predictor-corrector methods could not be used on their own for this example because of the discontinuous nature of the step input. The second problem, given in section 5.3, of a linear oscillator allowed the comparison of both single and multiple step methods. The use and capabilities of the simulation system were further investigated in section 5.4. The results for the error comparisons of the integration methods were obtained using tabulated printouts on the teletype. The other graphs were obtained by photographing the screen of an oscilloscope directly with a polaroid camera. A Tektronix 7704 oscilloscope was used for all the pictures except for figures 5.27

arithmetic functions

%SQRT(X) : $\sqrt{X}$

%NEG(X) : -X

%INT(X) : integer part of X

%PI : π

%INTG(X1,X2,DT) : (X1+X2)/2*DT

%RDSI : next serial data input value

%READ(N) : present value of formatted input stream number N

control functions

%SKPI :skip next serial data input value

%STOP :stop run and return to command level

%IFT1(L,X,U)
%IFT2(L,X,U)
%IFT3(L,X,U)
%IFT4(L,X,U)
} continue equation only if { $L < X < U$; $L < X \leqslant U$; $L \leqslant X < U$; $L \leqslant X \leqslant U$

%IFF1(L,X,U)
%IFF2(L,X,U)
%IFF3(L,X,U)
%IFF4(L,X,U)
} abandon equation only if { $L < X < U$; $L < X \leqslant U$; $L \leqslant X < U$; $L \leqslant X \leqslant U$

Figure 5.2 Arithmetic and Control Functions

error number

11 : equation too long

12 : variable list full

13 : invalid operator or function name

14 : parameter missing from function

15 : invalid character in equation string

16 : stack -ve (too many pop operations)

17 : stored equation space full

18 : stack not empty (too many push operations)

Figure 5.3 Conversion Error Codes

```
C >INEQ
ENTER EQN
 >X=X+ONE
 >Y=X*X
 >

C >SPAR
PRESENT PARAM
ONE=   0.0

NEW Y/N?: Y
ONE= >1

C >SIC
PRESENT INITIAL COND.
X=   0.0
Y=   0.0

NEW Y/N?: N

C >STAB
PRESENT OUTPUT IS

NEW Y/N?: Y
TAB:>X
TAB:>Y
TAB:>

C >STPS
STEP=   1275

NEW Y/N?: Y
STEP= >10

C >TINT
PRINT INTERVAL IS   1

NEW Y/N?: N

C >RUN
X              Y
 0.0            0.0
 1.0            1.0
 2.0            4.0
 3.0            9.0
 4.0            16.0
 5.0            25.0
 6.0            36.0
 7.0            49.0
 8.0            64.0
 9.0            81.0
 10.0           100.0
AFTER     10   STEPS


C >
```

Figure 5.4 Simple Example

and 5.28 where an Advance OS 3000 was used. For general use a larger LAN display oscilloscope was found to be useful although its limited bandwidth meant that fast changing graphs were not necessarily accurately represented.

5.2 First Order Step Response

The first order problem chosen was the step response of the differential equation $\tau\dot{y} + y = x$ where y and $\dot{y}$ are the variable and its derivative respectively. The input is a step in x from -1 to +1 at time zero and the time constant τ was set to 1.5. The Euler integration method is the simplest to implement and figure 5.5 shows the entry of the problem using Euler integration. The underlined text is that entered by the user. The resulting graphic output is shown in figure 5.6 and there is no discernable difference between this result and the much more accurate 4th order Runge Kutta which was also used. As well as the Euler, two second order Runge Kutta methods (α = 0.5 and α = 1) and a third and fourth order Runge Kutta were used. In order to compare the accuracies and speed of the integration methods, the results were printed in tabulated form every 0.5 second of model time for a total of 10 seconds. The results were printed to an accuracy of 10 digits and the approximate time to calculate 1000 steps for each method was found using a stop watch. From this, the calculation time for each step was worked out and is shown in figure 5.7. All five integration methods were used with a range of step sizes between 0.0001 and 0.5 and since the exact solution for this problem is $y(t) = 1 - 2e^{-t/\tau}$, the errors of the various integration

```
C >INEQ
ENTER EQN
 >DY=(X-Y)/TAU
 >Y=Y+DY*DT
 >T=T+DT
 >

C >SPAR
PRESENT PARAM
X=   0.0
TAU=   0.0
DT=   0.0

NEW Y/N?: Y
X= >1
TAU= >1.5
DT= >.04

C >SIC
PRESENT INITIAL COND.
DY=   0.0
Y=   0.0
T=   0.0

NEW Y/N?: Y
DY= >
Y= >-1
T= >

C >GRAM
NO GRAPHS!

NEW Y/N?: Y
GRAPHIC OUTPUT Y/N?:Y
GRAPH   1>X
GRAPH   2>Y
GPAPH   3>

C >GRSC
GRAPH SCALES
X=   0.0>1
Y=   0.0>1

C >STPS
STEP=   10

NEW Y/N?: Y
STEP= >255

C >GINT
GRAPH INTERVAL IS   5

NEW Y/N?: Y
INT=>1

C >RUN
```

Figure 5.5 Setup First Order Example

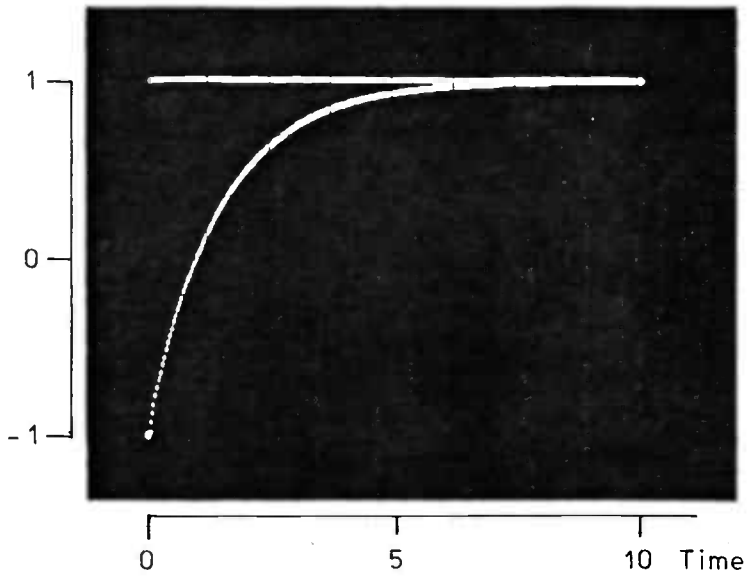


Figure 5.6 Step Response

integration method	h for max error of 0.0001 (s)	calculation time for 1 step (ms)	calculation time for 1 sec (s)
Euler	0.0012	14	12
RK-2 (α=1)	0.06	30	0.5
RK-2 (α=.5)	0.06	29.5	0.5
RK-3	0.27	50	0.2
RK-4	0.5(see text)	62.5	0.13

Figure 5.7 Computation Times

methods can be calculated. Figure 5.8 shows the absolute errors of each integration method plotted against the time step size on a logarithmic scale. From this graph, the maximum step size producing an error not greater than 0.0001 was estimated and is shown in figure 5.7. To compare the speed of the integration methods, the real time taken by the microcomputer to calculate one seconds worth of model time, with the step size shown, was also calculated and is shown in the final column of figure 5.7. As can be seen the higher order integration methods, although containing more equations, are much faster for the same error. It is worthwhile to note that the maximum step size used for the 4th order Runge Kutta method was governed by the need to printout every 0.5 second, and not by insufficient accuracy. The shapes of the individual error graphs correspond to those predicted in section 1.5. Initially the accuracy of a given method increases as the step size decreases until the cumulative arithmetic truncation error, caused by the larger number of steps, becomes the dominant factor.

An unusual feature of figure 5.8 is that the error curves for small step sizes, for all the integration methods, are almost identical. This can be explained by examining the main probable sources of error. Each of the single step methods evaluates the value of the next point with the same general form of equation $y_{n+1} = y_n + hB$ where B is a function of previous values of y and their derivatives. Now if the step size h is small then the value of h*B will be significantly smaller than y_n and hence a considerable part of h*B will be truncated and lost in the addition process. Therefore any increase in accuracy in the h*B term which is obtained

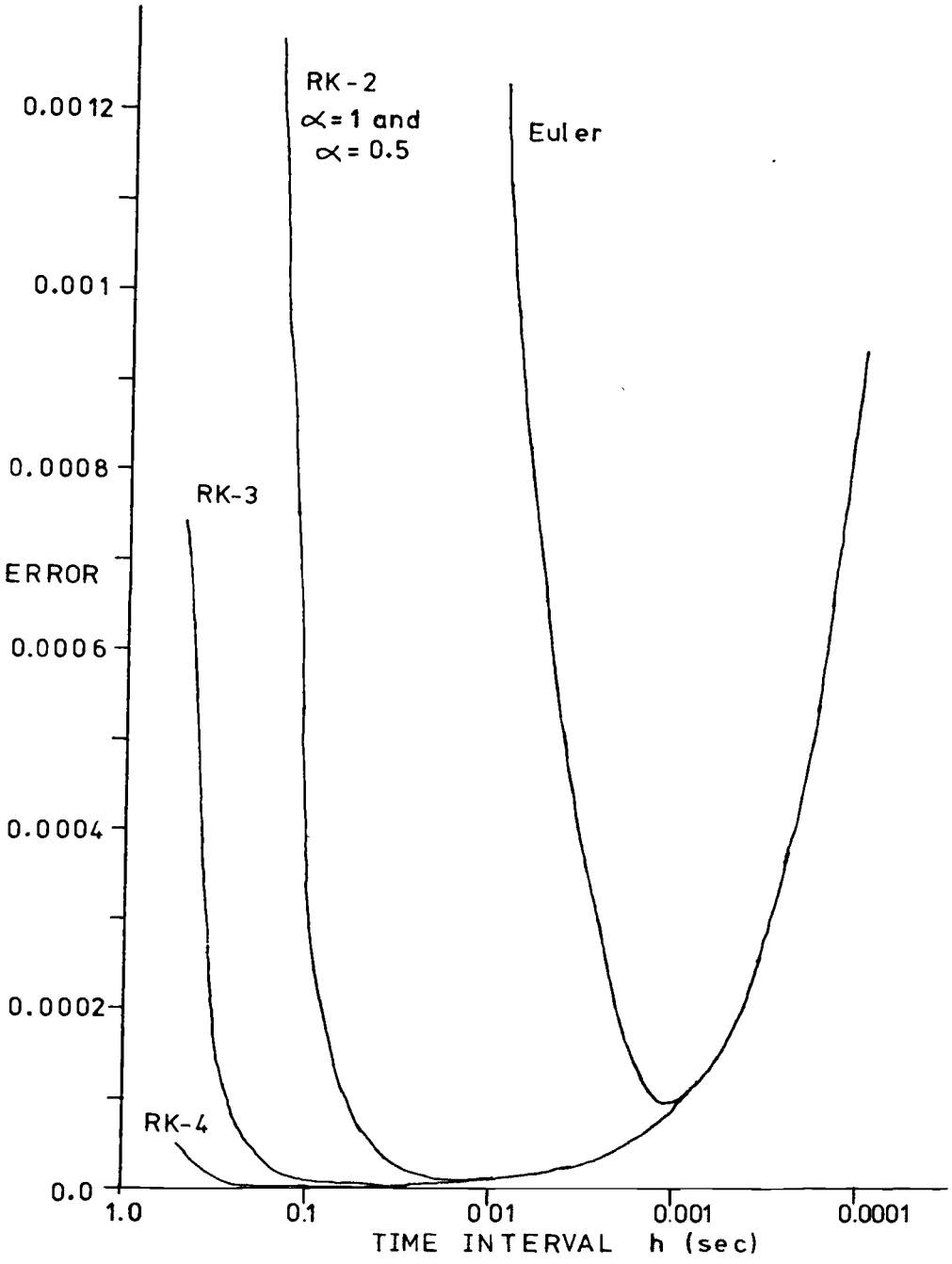


Figure 5.8 Errors in First Order Step Response

by reducing the step size will eventually be lost when the arithmetic magnitude of the h*B term becomes so small that the bits representing the increase in accuracy will be truncated off. Another potential source of error is input conversion of the step size h itself, especially if it is very small, since the errors caused by inaccuracies in step size are cumulative and increase as the number of steps increases. As can be seen from figure 5.8 the 4th order Runge Kutta method provided the greatest accuracy while the Euler method only just reached the 0.0001 error point. It is this greater accuracy which means that the 4th order Runge Kutta is the fastest of the methods used, indeed it is nearly 100 times faster than the Euler method. This test is not an absolute direct comparison of the arithmetic calculation time of each integration method since the times measured include all the overhead processing required to direct the evaluation of the equation set as well as the arithmetic calculations themselves. The calculation times however do not include printed or graphic output as that is independent of integration method. The SIMUPROG program is very efficient in the arithmetic evaluation of individual equations, but the error checks and run time controls for input, output and run length add an extra overhead which is independent of the number and complexity of the equations used. This means that the higher order methods with their smaller number of steps have less microprocessor calculation time wasted by the overhead processing. The test is therefore a comparison of the suitability of integration methods for use with the simulation system. For speed and accuracy the 4th order Runge Kutta was obviously the best choice but it has the disadvantage

of being much more complicated than the lower order methods. A good compromise between speed and efficiency of use would seem to be the 2nd order Runge Kutta methods which produced almost identical results. The Euler method, although the simplest, was much slower and would only be useful either when equation space was limited or when only a low accuracy result was required. For all the methods except Euler, the calculation times shown in figure 5.7, for one second of model time, are considerably less the time to print one set of results.

For comparison a 2nd order Admas-Bashforth predictor and a 4th order Adams-Moulton predictor corrector were tried but without being reinitialised by another integration method after the input step. As expected they did not perform very well, and indeed the 2nd order predictor gave significantly worse results than Euler, especially with larger step sizes. The 4th order predictor corrector gave only marginally better results than Euler.

A variable step 2nd order Runge Kutta integration method was successfully implemented using Richardson's method for error estimation. While the variable step method required fewer steps to obtain the same accuracy as either of the 2nd order fixed step methods, the computation time for each step was considerably greater. This meant that the variable step method actually took longer to perform a run than the fixed step method for a given accuracy. Another disadvantage of the variable step method is that regular printouts are difficult to achieve, so for this application the fixed step methods would be a better choice.

5.3 Linear Oscillator

The second order problem chosen was the linear oscillator given by the equation $\ddot{y} + y = 0$, since this made it relatively easy to use predictor-corrector integration methods. The oscillator was implemented as a pair of first order differential equations using a variety of methods. In addition it was also implemented directly using both the Nyström formula and a Taylor series expansion.

For the implementation as a pair of first order equations, three Runge Kutta methods were used, the two 2nd order methods (α = 1 and α = 0.5) and a 4th order method. The Euler method was again used, as well as two Adams-Bashforth predictors (2nd and 4th order) and a 4th order Adams-Moulton predictor-corrector. The simulation was first run from the initial conditions, of y=1 and $\dot{y}$=0, for 10 seconds with printouts every second. All the integration methods were evaluated for a range of step size between 0.001 and 1 second. The approximate calculation times for 1000 steps of each method were again obtained with a stop watch. The exact solution for this problem for the given initial conditions is y(t) = Cos(t), so the absolute value of the errors generated by each integration method can be evaluated.

Figure 5.9 shows the entry and running of the problem using the 4th order Runge Kutta integration method. The graphic output from the second run in figure 5.9 is shown in figures 5.10 and 5.11. The switch between the time plots of y and $\dot{y}$, and the phase plot of y against $\dot{y}$ is controlled independently by the NASCOM-1 keyboard.

Figure 5.12 shows the errors for the Nyström and Taylor series methods together with those for the 4th order Runge Kutta method, and

```
C >INEQ
ENTER EQN
 >K11=-Y*DT
 >K12=DY*DT
 >K21=-(Y+K12/TWO)*DT
 >K22=(DY+K11/TWO)*DT
 >K31=-(Y+K22/TWO)*DT
 >K32=(DY+K21/TWO)*DT
 >K41=-(Y+K32)*DT
 >K42=(DY+K31)*DT
 >DY=DY+(K11+TWO*K21+TWO*K31+K41)/SIX
 >Y=Y+(K12+TWO*K22+TWO*K32+K42)/SIX
 >T=T+DT
 >

C >SPAR
PRESENT PARAM
DT=   0.0
TWO=   0.0
SIX=   0.0

NEW Y/N?: Y
DT= >.5
TWO= >2
SIX= >6

C >SIC
PRESENT INITIAL COND.
K11=   0.0
K12=   0.0
K21=   0.0
K22=   0.0
K31=   0.0
K32=   0.0
K41=   0.0
K42=   0.0
DY=   0.0
Y=   0.0
T=   0.0

NEW Y/N?: Y
K11= >
K12= >
K21= >
K22= >
K31= >
K32= >
K41= >
K42= >
DY= >
Y= >1
T= >
```

```
C >GRAM
GRAPHS OF

NEW Y/N?: Y
GRAPHIC OUTPUT Y/N?:Y
GRAPH   1>Y
GRAPH   2>DY
GRAPH   3>

C >GRSC
GRAPH SCALES
Y=   0.0>1.5
DY=   0.0>1.5

C >STPS
STEP=   255

NEW Y/N?: N

C >GINT
GRAPH INTERVAL IS   1

NEW Y/N?: N

C >RUN

AFTER     255  STEPS

C >SDT
DT=   0.5>.1

C >RPUN

AFTER     255  STEPS

C >
```

Figure 5.9 Oscillator Setup Example

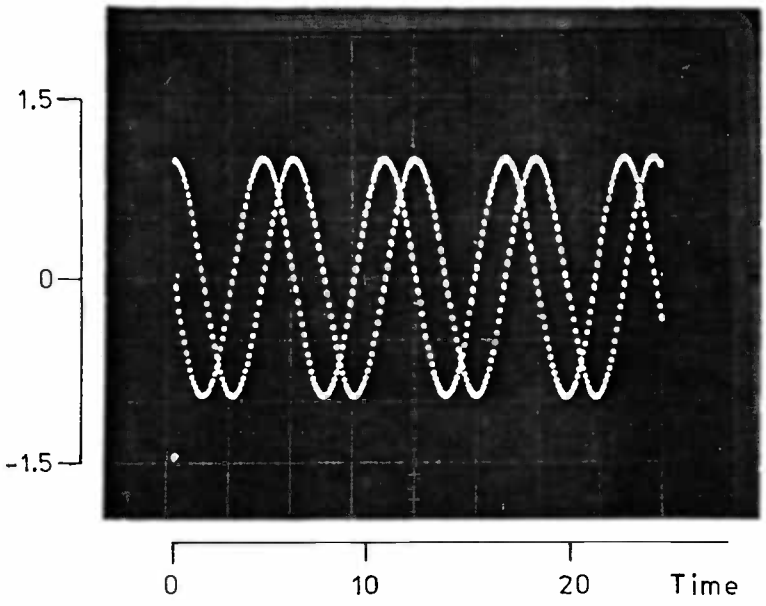


Figure 5.10 Time Plot from Oscillator

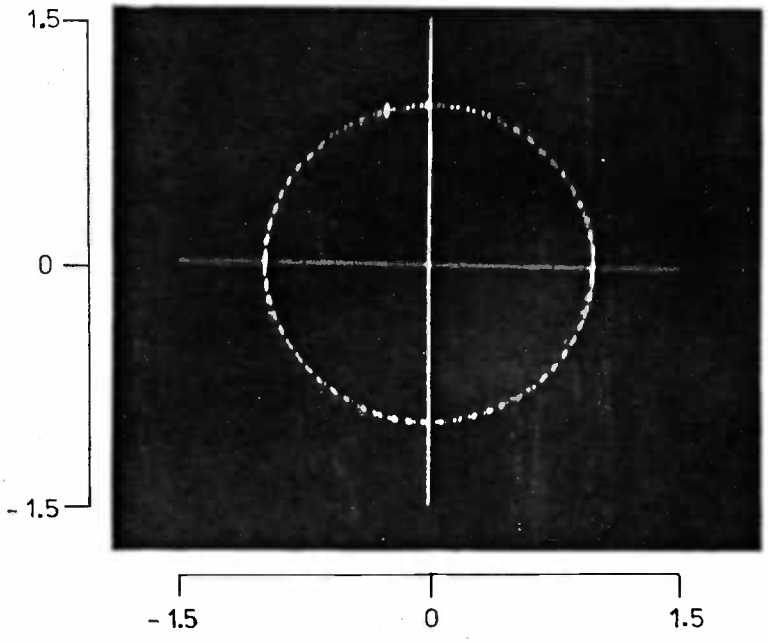


Figure 5.11 Phase Plot from Oscillator

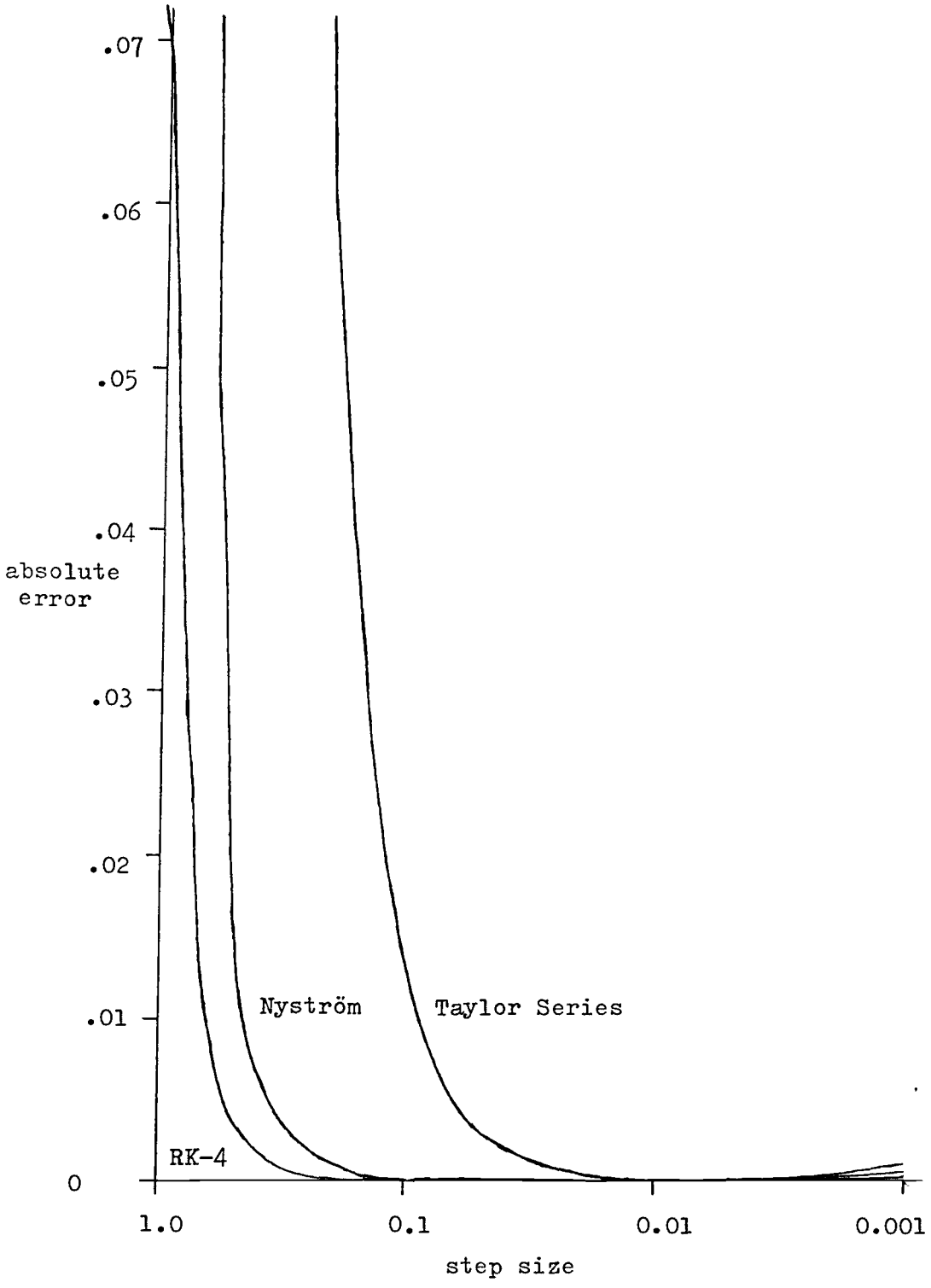


Figure 5.12 Errors in Oscillator (Set 1)

figure 5.13 shows the corresponding plots for the other methods used. The Nyström method performed better than the Taylor series, and this was as expected since the Taylor series is second order and the Nystrom method is essentially third order. The 4th order Runge Kutta method performed better than all the other methods used. Both 2nd order Runge Kutta methods gave almost identical results to each other, and also to the Taylor series method for step sizes above 0.01. Below this step size the 2nd order Runge Kutta methods gave lower errors than both the Taylor series and Nystrom methods. The Nystrom method only gave greater accuracy than the 4th order Adams-Moulton method at step sizes above 0.1. Above a step size of 0.1, the 4th order Adams-Bashforth quickly became unstable. However below this step size the 4th order Adams-Bashforth outperformed the 2nd order Runge Kutta methods which in turn outperformed the 2nd order Adams-Bashforth and Euler methods. At step sizes of below 0.001, all the methods, with the exception of the Euler method and two direct methods, gave similar errors. As with the first order problem, this can be attributed to the arithmetic truncation errors involved in the final calculation of the variable's value at each step. The Euler method does not reach its point of minimum error in the graph. Both direct methods gave higher errors for step sizes below 0.001, and this could be due to the fact that the variable and its derivative were evaluated separately at each step.

Estimates of the largest step size which would produce an error less than 0.01 were made from the graphs for each integration method. The values together with the calculation time per step for each method is given in figure 5.14. Also shown is microprocessor

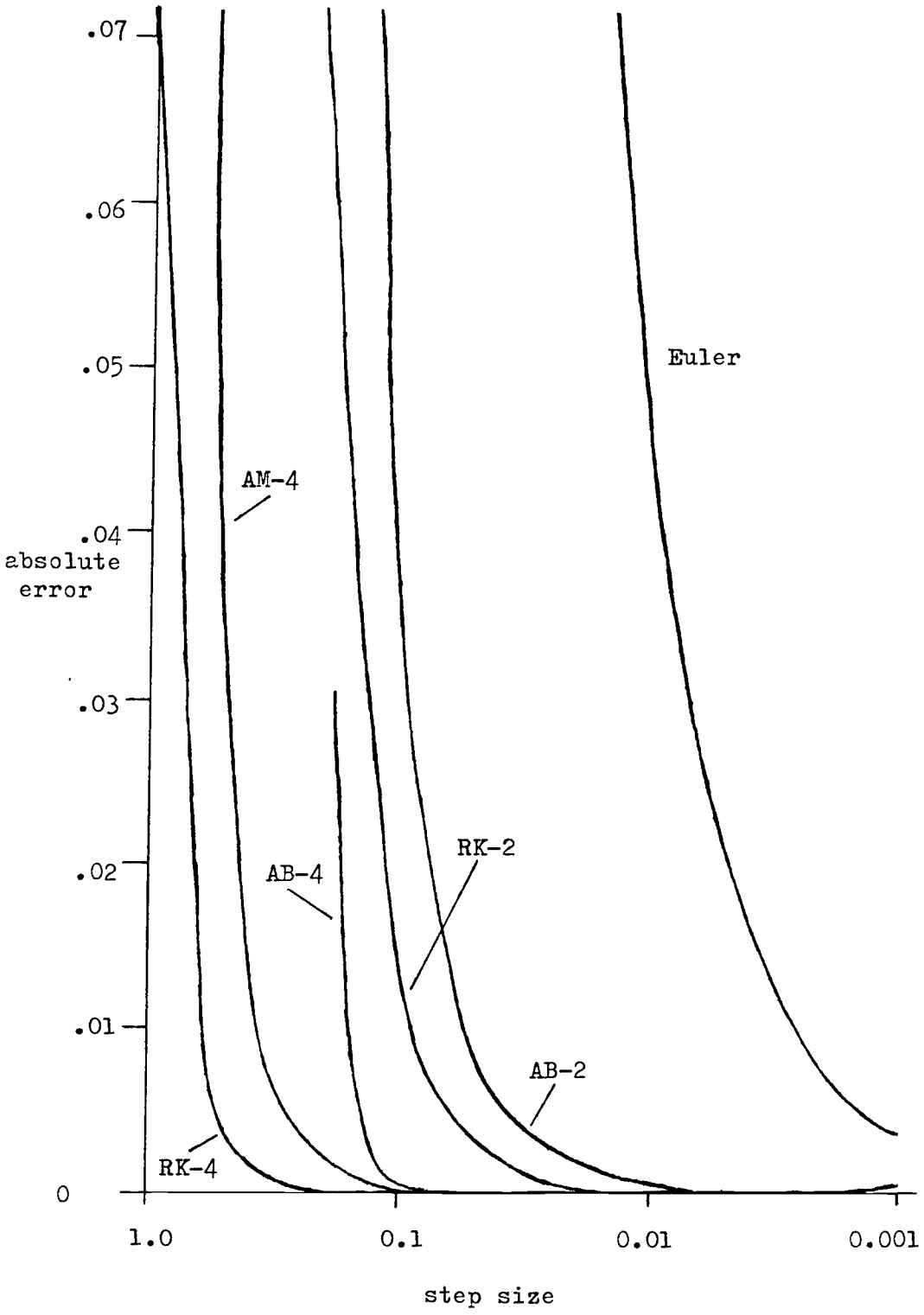


Figure 5.13 Errors in Oscillator (Set 2)

integration method	step size for 0.01 absolute error (s)	calculation time for 1 step (ms)	calculation time for 1 second (s)
Euler	0.0025	17.3	7.0
AB-2	0.054	36.1	0.7
Taylor Series	0.088	44.1	0.5
RK-2 (α =1)	0.088	40.2	0.5
RK-2 (α =0.5)	0.088	38.0	0.4
AB-4	0.16	69.8	0.4
AM-4	0.29	121.5	0.4
Nyström	0.47	68.5	0.2
RK-4	0.61	91.5	0.2

Figure 5.14 Computation Times

integration method	maximum error $500 \leq t \leq 505$ (s)
RK-2 (α =1)	0.84
AB-4	0.018
Nyström	0.0074
AM-4	0.001
RK-4	0.0007

Figure 5.15 Long Term Error

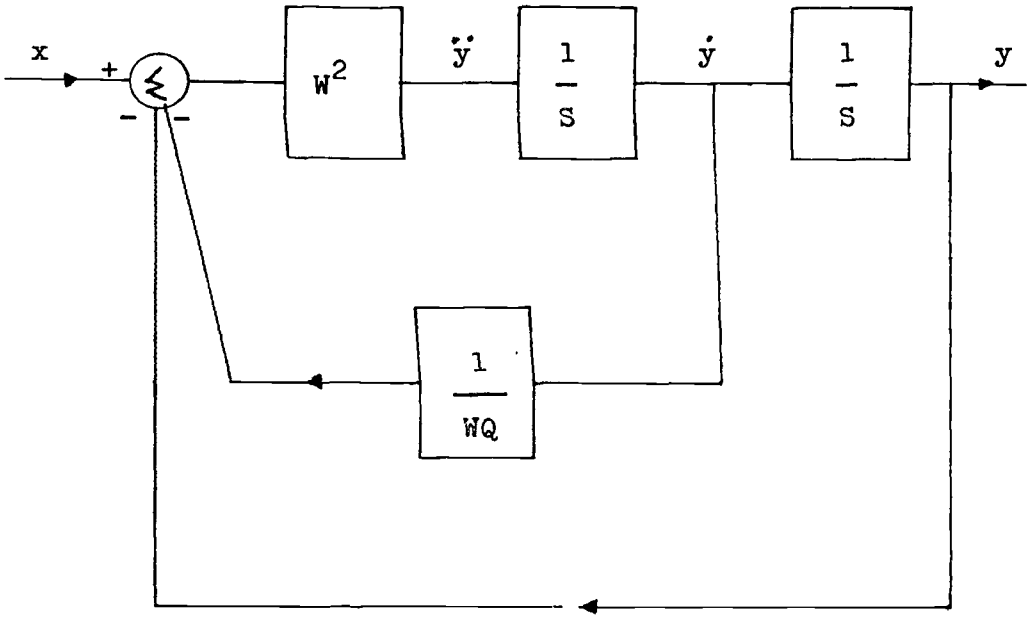
calculation time for one second of model time, and as before this does not include printout time.

To check on the long term stability, some of the integration methods were run for 500 seconds of model time with a step size of 0.1 seconds. The maximum error during the next 5 seconds was obtained and is shown in figure 5.15.

The 4th order Runge Kutta method again proved the fastest of the general purpose methods. For this particular problem, the simpler Nyström method was just as fast as the 4th order Runge Kutta method over the short term, but if long term stability were required, the Nyström method is not as good. The Euler again was much slower than any other method. The 2nd order Adams-Bashforth was the next slowest and, in common with the 4th order Adams-Bashforth, it had stability problems for large step sizes. The 4th order Adams-Moulton predictor corrector had no advantages over the 4th order Runge Kutta method. Indeed since predictor and predictor-corrector methods are not self starting, the need to provide past values of the variable at the start means that they are more awkward to use with this simulation system than the single step methods.

5.4 Other Test Results

As a test example for both evaluation and illustration of the use of the simulation system, the second order system given in figure 5.16 was implemented. Figure 5.17 shows the initial entry of the second order system, using the initial conditions to implement the input step. For such a demonstration it is unlikely that any great accuracy would be required, so the simple Euler



$$\frac{1}{W^2} * \frac{d^2y}{dt^2} = x - y - \frac{dy}{dt} * \frac{1}{WQ}$$

Figure 5.16 Second Order System

```
C >INEQ
ENTER EQN
 >D2Y=W*W*(X-Y)-DY*W/Q
 >Y=Y+DY*DT
 >DY=DY+D2Y*DT
 >T=T+DT
 >

C >SPAR
PRESENT PARAM
W=   0.0
X=   0.0
Q=   0.0
DT=   0.0

NEW Y/N?: Y
W= >10
X= >
Q= >.5
DT= >.01

C >SIC
PRESENT INITIAL COND.
D2Y=   0.0
Y=   0.0
DY=   0.0
T=   0.0

NEW Y/N?: Y
D2Y= >
Y= >-1
DY= >
T= >

C >STPS
STEP=   255

NEW Y/N?: N

C >GRAM
GRAPHS OF

NEW Y/N?: Y
GRAPHIC OUTPUT Y/N?:Y
GRAPH   1>Y
GRAPH   2>

C >GRSC
GRAPH SCALES
Y=   0.0>1

C >RUN
```

Figure 5.17 Setup Second Order System

integration method was used for simplicity. Figures 18, 19 and 20 show the graphic results obtained with a range of damping factors, varying from heavily damped to underdamped.

The system was then altered by inserting a limiter to keep the value of the first derivative between +1 and -1. The system with limiter is shown in figure 5.21, with the required commands to modify the stored model shown in figure 5.22. The limiter is implemented using two conditional equations which are appended to the equation set using the APEQ command. Note that the present version of the SIMUPROG program requires both upper and lower limits for a conditional equation. This restriction means that the upper limit has to be set to an arbritary value which is greater than the variable will reach, in this case 100,000. Figure 5.22 also shows the action of the equation conversion subroutine when an erroneous equation has been entered. Figure 5.23 shows the result obtained with the addition of the limiter. This result is for a Q of 2.5 and is therefore directly comparable with the non limited case shown in figure 5.20.

Input data can be entered into the model using the high speed paper tape reader. Figure 5.24 shows the commands used to initiate the data entry. Formatted data entry is used, so the data input is controlled by the simulation system and not by the model. The APEQ and REOR commands are used to add an equation for reading the data to the start of the equation set. The data input used is a square wave of +0.5 to -0.5, and was used with the second order system without the limiter. The result for a Q of 2.5 is shown in figure 5.25.

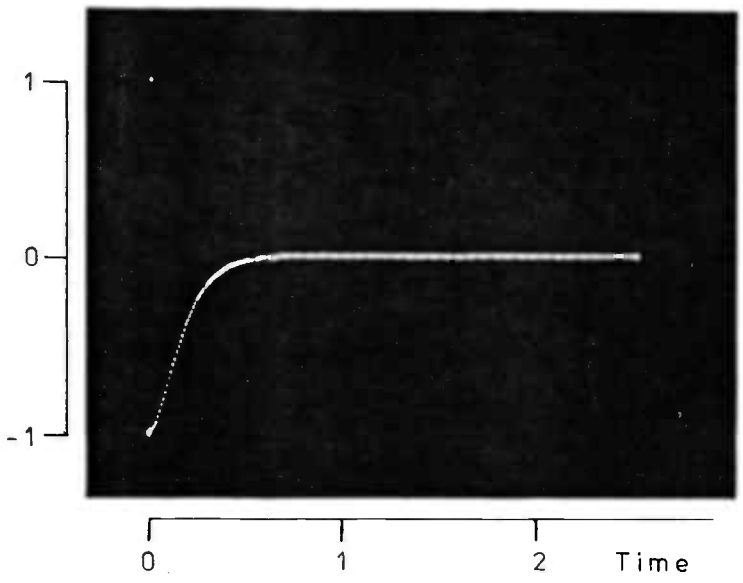


Figure 5.18 Response for Q=0.5

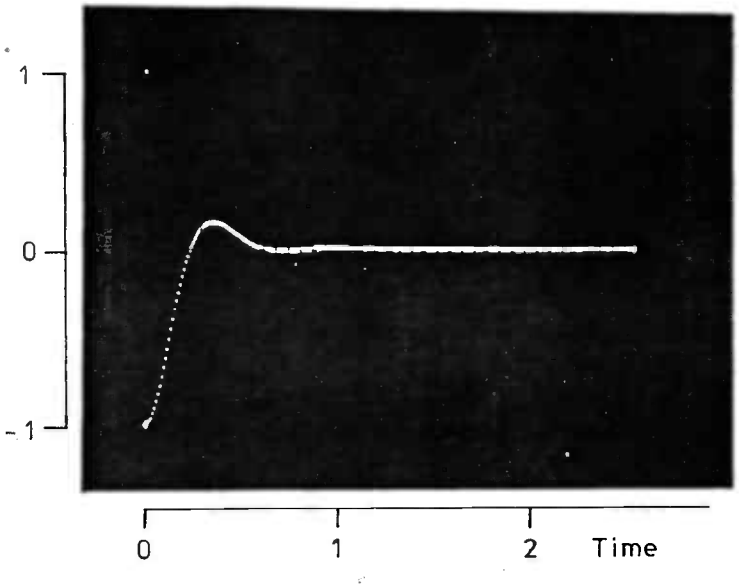


Figure 5.19 Response for Q=1

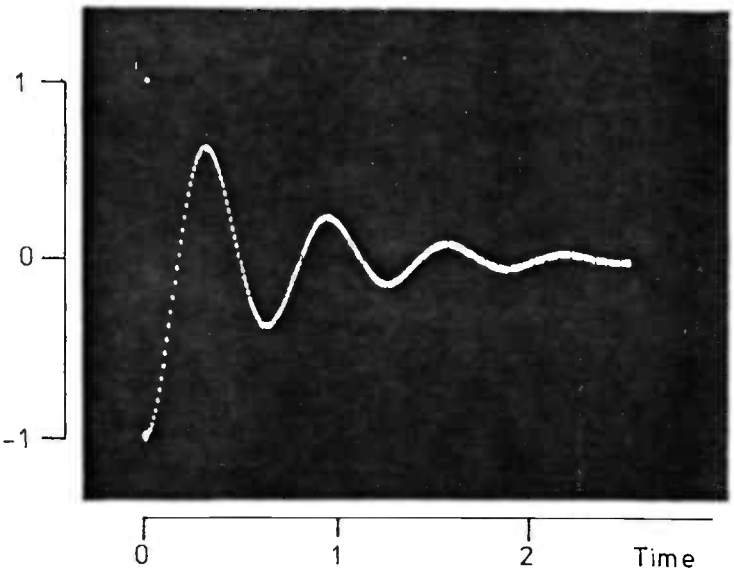


Figure 5.20 Response for Q=2.5

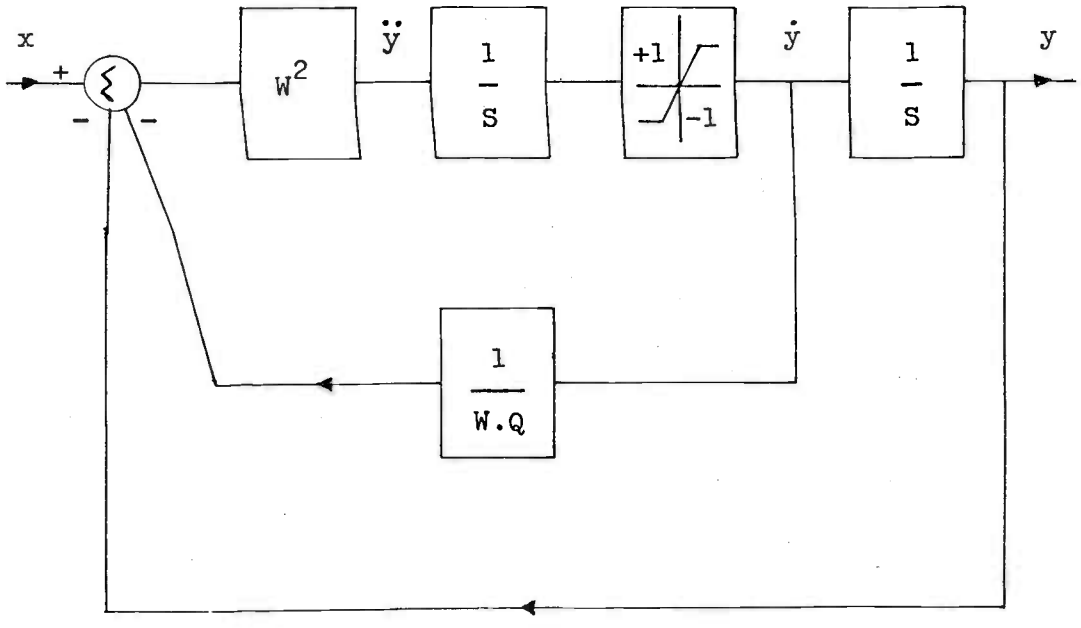


Figure 5.21 Second Order System with Limiter

```
C >APEQ
ENTER EQN
>DY=%IFT1(L,DY,
##ERROR    18 EQN WRONG!, ENTER AGAIN
>DY=%IFT1(ONE,DY,M)ONE

C >APEQ
ENTER EQN
>DY=%IFT1(ONE,%NEG(DY),M)%NEG(ONE)

C >SPAR
PRESENT PARAM
W=   10.0
X=   0.0
Q=   2.5
DT=   0.009999999
ONE=   0.0
M=   0.0

NEW Y/N?: Y
W= >
X= >
Q= >
DT= >
ONE= >1
M= >100000

C >RRUN


AFTER     255   STEPS
```

Figure 5.22 Setup Limiter

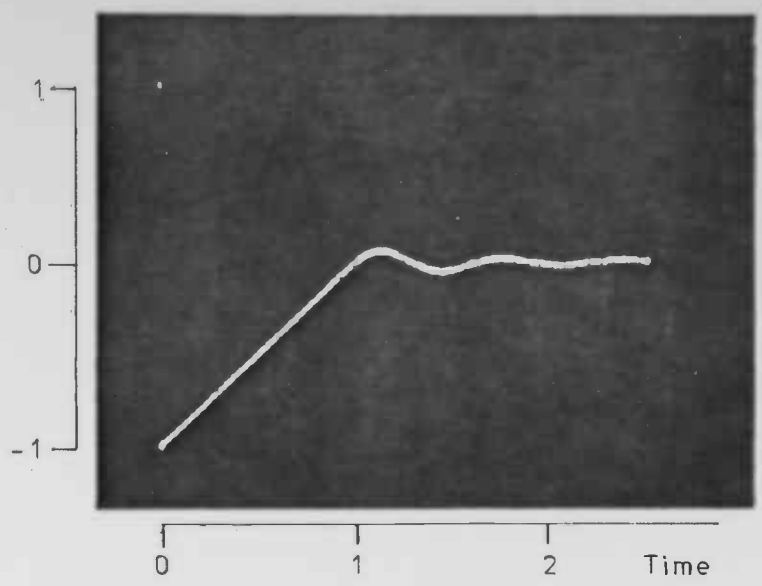


Figure 5.23 Response with Limiter

```
C >SIM
NO  DATA INPUT

NEW Y/N?: Y
DATA INPUT  Y/N?: Y
FORMATTED DATA INPUT  Y/N?: Y
STREAMS = >1
STEPS BETWEEN INPUTS  = >50

C >APEQ
ENTER EQN
>X=%READ(ONE)


C >REOR
OPERATE LIST IS
 1 2 3 4 5 6 7

NEW LIST Y/N?:Y
EQN NUMBERS
>7
>1
>2
>3
>4
>


C >STPS
STEP=  255

NEW Y/N?: Y
STEP= >1020

C >GINT
GRAPH INTERVAL IS  1

NEW Y/N?: Y
INT=>4

C >RRUN


AFTER    1020  STEPS


C >
```

Figure 5.24 Setup Data Input

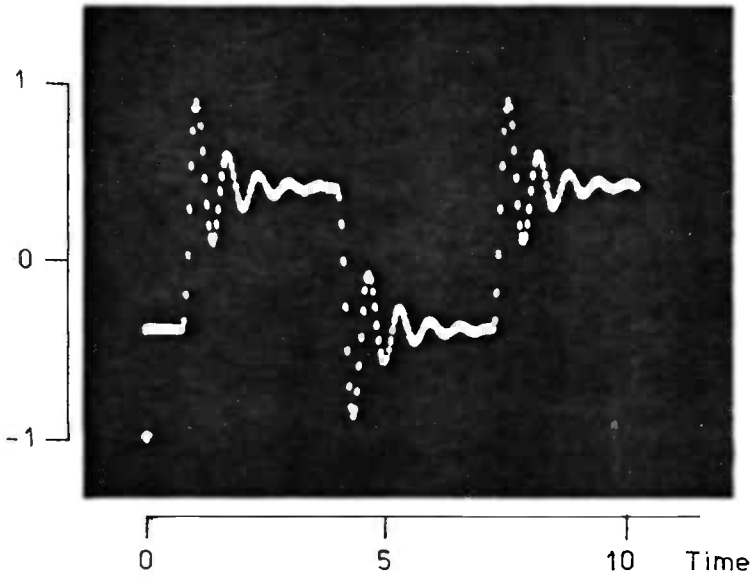


Figure 5.25 Response to Square Wave

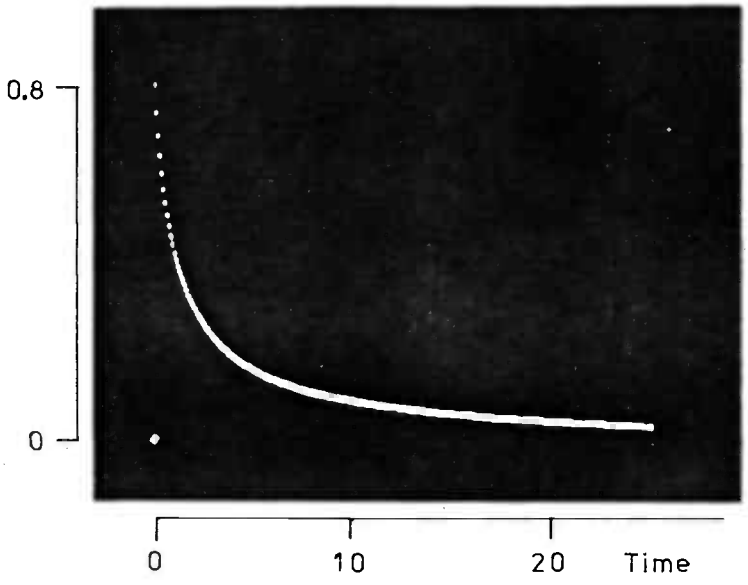


Figure 5.26 Nonlinear Solution

The simulation system is not limited to linear equations, as any algebraic equation can be entered, including discontinuous functions implemented using conditional equations. A simple nonlinear example which does have an analytical solution is the differential equation $\dot{y} = -y^2$, which has the solution $y(t) = y(0)/(1 + y(0)t)$. This was implemented using Euler integration with a step size of 0.01 and the result is shown in figure 5.26. The error produced was less than the minimum resolution of the graphic display which has 256 by 256 points.

Figures 5.27 and 5.28 show the phase plots of the response of the system defined by the differential equation $\ddot{y} = -y - A\dot{y}$, starting from the initial conditions $y=1$, $\dot{y}=0$ and $\ddot{y}=-1$, with a step size of 0.005. Variables y and y are outputted as graphs, and the display is changed to phase plot to give y against $\dot{y}$. Figure 5.27 shows the stable result when A=0.4 and figure 5.28 shows the unstable result when A=-0.05.

Some compensated integrators of the types suggested by Smith[5] were implemented, including a variable phase integrator for the linear oscillator problem. Figure 5.29 shows the result of the linear oscillator using the integrator with the variable phase facility disabled, and figure 5.30 shows the result when the variable phase integration is operational. The actual integrator implemented was a restricted version of Smith's, due to lack of equation space, so some degradation of the result will have occurred. The variable phase integrator does work, but the complexity of implementation, for what is essentially a second order integrator, means that it is not really suitable for the simulation system.

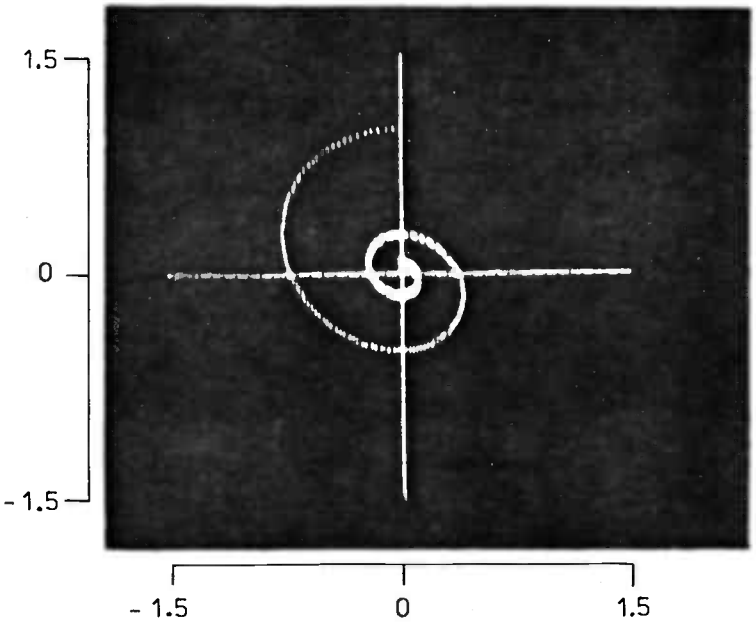


Figure 5.27 Phase Plot of Stable System

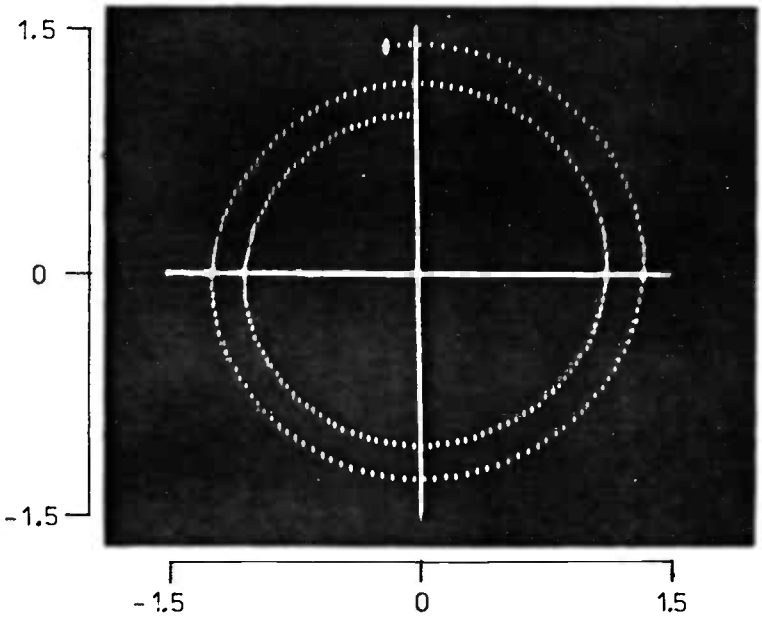


Figure 5.28 Phase Plot of Unstable System

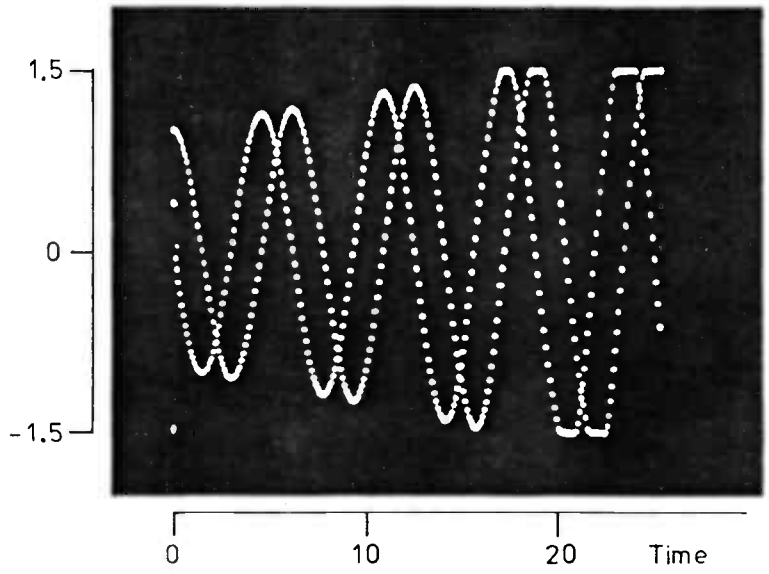


Figure 5.29 Variable Phase Disabled

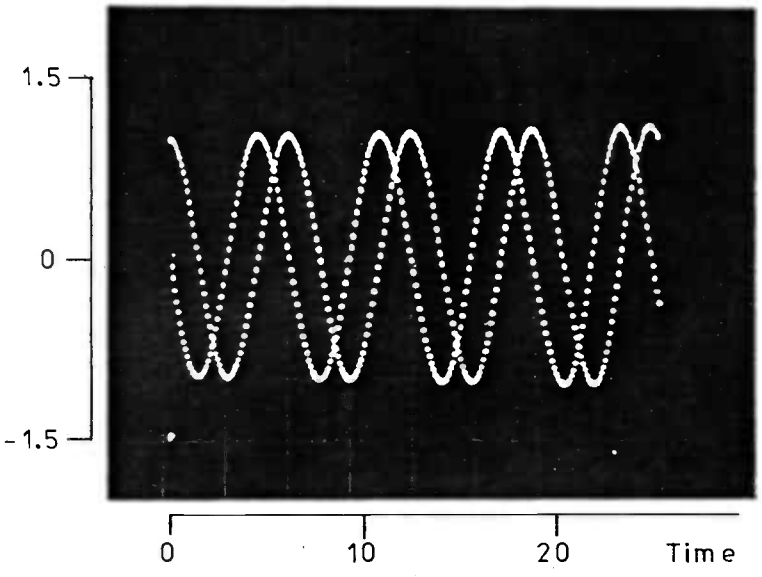


Figure 5.30 Variable Phase Enabled

A simulation of a phase locked loop was performed to investigate the reaction of the circuit to changes in frequency and phase of the input signal. The simulation was based on the design information supplied in the data sheets and applications note[66] for the Motorola MC 4344 and MC 4324 phase locked loop components. Figure 5.31 shows the output frequency response of the phase locked loop to an input frequency step from 120 MHz to 120.01 MHz, and figure 5.32 shows the output frequency response of the phase locked loop to a 180^{o} step change in the phase of the 120 MHz input signal. Both simulations used Euler integration with a time step size of 0.001 seconds.

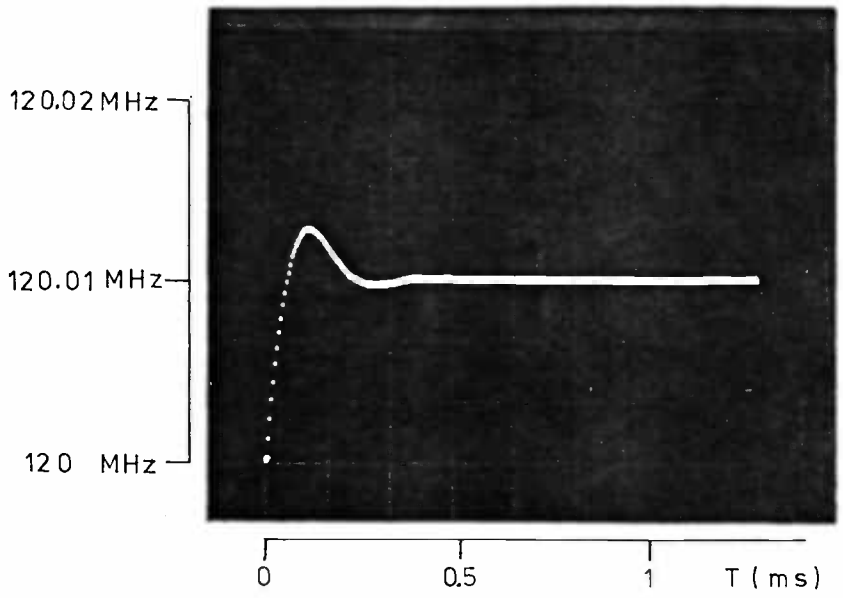


Figure 5.31 Frequency Step Response

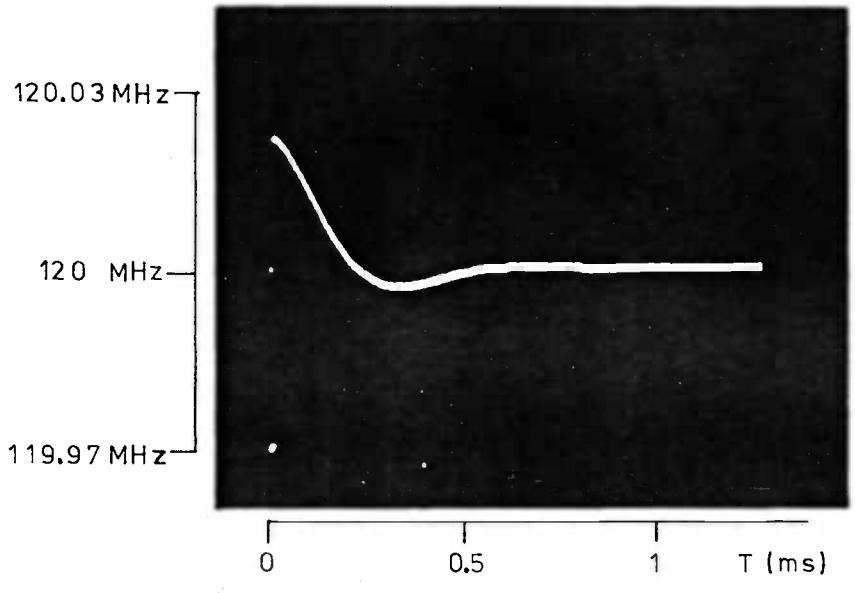


Figure 5.32 Phase Step Response

6. Conclusions

6.1 Present System

The microprocessor based simulation system provides an effective and easy to use method of digitally implementing small continuous simulation problems. For a large range of problems which do not need very high accuracy, the simulation system can produce results very quickly in graphic form. The graphic output on the oscilloscope can be used either for the final result, or to monitor the progress of the simulation run so that the operation of the simulation model can be checked prior to a higher accuracy tabulated printout. The mnemonic command names and the interactive dialogue of the individual commands go a long way to guide the user in operating the simulation system. The simulation system as it stands does not represent the minimum configuration which would be required to run the simulation system since it also had to be used for the more demanding task of program development. If the link was fully utilized to provide control of the simulation system from the VDU of the Nascom-1 microcomputer, then the only components left of the GIMINI microcomputer would be the memory and the microprocessor itself.

The present simulation system program is stored in RAM memory and has to be reloaded at the start of each session. Using the high speed reader the loading takes about three minutes, which would be quite acceptable at the start of each day, but if the Nascom-1's audio cassette was used, then the loading would take about eight imes as long which would be unacceptable. It would therefore be etter if the simulation system program were stored in ROM or EPROM

memory which would give instant availability. Because of the structure of the program, only minor changes are needed for operation in EPROM and these consist of moving the positions of the temporary variables from inside the program to an area of RAM memory beside the database.

The CMOS memory used with the Z80 provided an effective EPROM simulator for the display program, and could also be used to provide a non volatile memory for storing the simulation model data. CMOS memory is still more expensive than NMOS memory but, when the CMOS on sapphire technology becomes established, there should be less difference in cost. CMOS could still only provide a temporary storage, since several simulation task could not be held in the memory together. The Nascom-1's cassette interface was rather prone to interference, so a more sophisticated and reliable system would be needed for data storage.

The teleprinter was used during program development to output the source and object programs on paper tape as well as print the listings. It was very slow and noisy and also proved to be unreliable at punching paper tapes, so unreliable that it often proved quicker to find and correct the errors manually by splicing the tape rather than wait for the punch to produce a correct copy.

The simulation system presently runs in 16 k bytes (8 k words) of memory, including the floating point software which uses 3158 bytes of memory. The actual space used for storing the model equations is 2700 bytes, and this can hold 29 individual equations each of 25 elements together with the names and values of up to 35 variables. Since the simulation system program can cope with any size of memory,

which can be used with the CP 1600, the use of the CP 1600's maximum memory size of 128 k bytes would mean that the system could store over 1200 equations with over 1400 variables. The limiting factor on the length of the equation string entered is the line buffer which was matched to the teletype line length of 72 characters. Once the parenthesis have been deleted, and the variable and function names reduced to their coded form, the resulting equation is very unlikely to exceed a length of 25 elements. Even if the limit is exceeded the simulation system will detect the problem and ask the user to shorten the equation.

The cost of RAM memory has dropped dramatically over the last few years, and this makes it much cheaper to expand the simulation system just by increasing the size of the memory. The cost effectiveness of gaining extra storage space by rewriting the program in a more compact form has therefore been greatly reduced. However the use of a single segmented list to store the equations and the compression of the stored variable names into fewer words would still provide a significant increase in the usable equation storage space for a relatively small amount of programming effort.

The floating point arithmetic package, in its revised form, performed adequately but the lack of trigonometric functions is a handicap. The Am 9511 arithmetic processing unit which was evaluated, but not integrated into the simulation system, would remedy the situation as well as providing an increase in calculation speed. The arithmetic unit could be incorporated by simply changing the operator and function subroutines so that they access the hardware arithmetic unit instead of calling the floating point software.

Since the arithmetic unit is actually a self-contained processor which can operate in parallel with the host microprocessor, then a greater increase in operating speed could be obtained if the microprocessor was setting up the next arithmetic operation while the arithmetic unit was still processing the data from the previous operation. Although this would mean a change in the way the calculations are performed, the equation structure and the arithmetic subroutine calling mechanism would remain unaltered. The use of the arithmetic hardware would also free more memory for equation storage. The arithmetic unit is presently attached to the Z80, and therefore the simulation system can only access it via the link with the aid of a Z80 interrupt routine. This was originally done so that the Z80 could also access the unit for scaling the graphic output. Since the present simulation system scales the graphic output before sending it to the Z80, the arithmetic unit would be more efficiently used if it was attached directly to the CP 1600.

While automatic scaling of the graphic output would be convenient, full range scaling would require five times as much memory as presently used for result storage, if the refresh display were to be maintained. A more limited form of automatic scaling could be obtained by storing 16 bit fixed point values instead of the present 8 bits, and then extracting 8 bits for display, corresponding to the most significant bits of the largest value in the graph to be displayed. This scaling, using integer values, would be used in conjunction with the floating point scaling and integer conversion presently used by the simulation system and would not require the use of the arithmetic unit. The present graphic output represents a vertical scale of +1 to -1 and a single

scale factor is chosen to fit the result data into the range. This is not really satisfactory if the data does not extend equally above and below zero, since an unnecessary loss of resolution will occur. A better method of setting the scale factors would be to enter the lower and upper values of each graph to be displayed, and let the simulation system calculate the offset and scale factor required.

The refresh rate of the graphs was sufficient to produce a flicker free display even when four graphs were displayed simulataneously, High rates of change of the variable being displayed produces noticably dotted lines due to the limited resolution of the display. The only way round this would be either to use a higher resolution which would be more expensive, or to use analogue interpolation which would be both difficult and expensive. For most purposes an increased resolution would only have a cosmetic effect since the human eye and brain are very good at performing complex interpolations.

The teleprinter used for hard copy output was both noisy and slow. While this could be accepted for result output, an alternative output device such as a thermal or dot matrix printer would be preferable since these devices can be both faster and quieter. The Nascom-1's VDU could be used to display the tabulated results, but the availability of the graphic output facility means that the VDU would be rarely used. The main tabulated output requirement is still a high accuracy printout. The present floating point software produces a fixed point decimal output and while this is useful for most problems, an alternative of scientific or engineering format would sometimes be preferable. The alternatives could be achieved by having input and output conversion routines available for all three formats and adding a command which is used to select the required format.

The equation translator worked very well and the lack of a reverse translator, from reverse Polish notation to algebraic notation, did not prove a handicap since the original equations were recorded when they were typed in. The actual equation stored can be printed out in reverse Polish notation and its equivalence to the entered equation can therefore be checked. If the VDU was to be used for entering the equations then a translator for displaying the equations would be helpful, but it would use up a sizeable chunk of memory. An easier alternative, if a printer was available, would be to print out a copy of the equations as they were entered. The lack of numerical constants gave no problems but was inconvenient. Numerical constants could easily be implemented in the same way as variables and parameters except that no name would need to be stored. The equation input section of the program would have to be slightly modified to read in the numerical value instead of the name, but this has been anticipated by ensuring that all variables and parameters must start with an alphabetic character.

The conditional functions were inconvenient on two accounts, firstly the user has to remember which function name corresponds to which condition, and secondly a dummy second limit is required to implement a single sided condition. It would therefore be preferable if the function name only specified whether the equation was to be implemented when the conditions given as parameters were true or false. The initial function names would therefore only be dummies and the program would select the actual subroutine required, according to the conditions given in the parameters.

The simulation system control dialogue seems to be a reasonable

balance between minimising memory space and providing an explanatory dialogue. If more explanation were given to assist the first time user, the time taken for the extra printout would be annoying to a user who is thoroughly familiar with the simulation system. If the VDU was used for control dialogue, then one useful additon would be a menu of system commands.

The implementation of a built in integration routine would be very useful, but would involve extra complexity and add extra restrictions on the format of a variable, so that the simulation system could identify the derivatives. Of the integration methods examined, only the Runge Kutta ones would really be useful for general purpose work. The Euler method was by far the slowest method, and is indeed simple enough to implement without being built in. The higher order Runge Kutta methods are rather complex to implement so one of the second order Runge Kutta methods would be the best compromise. There was nothing to choose between the two second order Runge Kutta methods tried, so the easiestto implement should be chosen.

6.2 Future Developments

The simulation system is by no means limited to using the CP 1600 microprocessor, and the newer and more powerful 16 bit microprocessors, such as the Z8000 and the M68000, could be used to produce a faster system with a bigger memory address range. These processors are primarily designed for multiuser systems and provide for operating system protection and memory management. Their present high cost would detract from the ideal of a cheap single user simulation system. The use

of an arithmetic processor such as the Am9511, removes a lot of the run time processing from the host microprocessor. This would mean that, when used with an arithmetic processor, an 8 bit microprocessor could run a simulation model as fast as a 16 bit machine. The Z80 could be used, but the M6809 would be preferable because of its two stack pointers and its more comprehensive addressing modes.

A M6809 with an Am9511 arithmetic unit could provide the basis for a future low cost system, but to keep the final cost low, the program development time must be kept to a minimum. With memory prices falling all the time, it would be more cost effective if the bulk of the simulation system was written in a high level language and assembly language only used for time critical sections. It would actually be possible to use an interpreter for most of the system, but a structured language such as Pascal would be much better than the unstructured Basic. The Forth language would seem to offer the benefits of a structured high level interpreter with an operating speed much nearer to pure in line machine code operation, and therefore must be a prime candidate.

The usefulness of the simulation system would be increased if the user could write special functions which could be called in the model equations. This facility could be implemented by allowing the user to enter generalised equations, similar to the model equations, which would then be assigned as a block to the required name which could be called in the same way as any present system function. A more powerful alternative to this would be for the user to enter the name of a new function together with the address of a prewritten subroutine

which will perform the required operations. The prewritten function could be in machine code or else written in a high level language for use with an interpreter. An interpreter would be slower but Forth, which itself uses a stack and reverse Polish notation, may be useful and for future systems should be investigated.

To save space, the present simulation system uses the monitor's memory dump facility to save the simulation data, so the format is therefore peculiar to the GIMINI and the memory size. A more general data output which dumped only the model would therefore be preferable since the model data could then be transferred between simulation systems using different memory sizes and even different microprocessors.

The simulation system has at present a data input facility from paper tape, but his could be replaced by an analogue to digital converter and a multiplexer to give a real time data input facility, so that the system could be used as data logger or data analyzer. The analogue output, used for the graphics display, could also be used and the system would be able to perform real time control of slow systems. The simulation system would not really be practicable as a control system, but the control facility could be very useful in a teaching role where the students could see a demonstration of the actual operation of the set of equations entered into the simulation system. The input and output data handlers could either be built into the system or implemented using the user written functions previously mentioned.

To perform real time control effectively, the simulation system would have to be much faster. One way of increasing the speed would be to use scaled integer arithmetic. The simulation would first be

performed using floating point arithmetic to determine the magnitude of the variables in response to the range of input values. These results would then be used to scale the problem so that integer arithmetic could be used for the actual running. Even with integer arithmetic, the relatively slow speed of the microprocessor would still limit the systems use.

Bit slice microprocessor elements could be used to emulate a general purpose microprocessor, but this would be complex and expensive since a program would first have to be written to simulate the general purpose microprocessors instruction set. The memory used would also be a limiting factor, and high speed memory is very expensive. For a real improvement in speed, parallel processing would have to be used. A system could be envisaged where each equation of the simulation system, or at least each integrator, was implemented by a separate microprocessor. Each of these parallel processors could be a general purpose microprocessor or, for extra speed, a bit slice machine. Raamot[67] describes the use of two bit slice machines to simulate a galvanometer. For the Galvanometer application the the control program is fixed in a PROM, but for the simulation system the program would have to be updated. The programs for the parallel microprocessors could be stored completely or partly in RAM and updated when required by a general purpose microprocessor which controls the whole system and interfaces with the user. Apart from the hardware complexity of the parallel processors, a lot of work will have to be done to produce an efficient distribution of the simulation problem between the various microprocessors. The resulting system should however be capable of operating speeds approaching those of an analogue computer for a fraction of the cost.

REFERENCES

1. Gear, C.W., 'Numerical Initial Value Problems in Ordinary Differential Equations', Prentice Hall, 1971.

2. Benyon, P.R., 'A Review of Numerical Methods for Digital Simulation', Simulation, November 1968, pp.219-238.

3. James, M.L., Smith, G.M. and Wolford, J.C., 'Applied Numerical Methods for Digital Computation', Harper & Row, 1967.

4. Martens, H.R., 'A Comparative Study of Digital Integration Methods', Simulation, February 1969, pp.87-94.

5. Smith, J.M., 'Mathematical Modelling and Digital Simulation for Engineers and Scientists', Wiley.

6. Korn, G.A. and Wait, J.V., 'Digital Continuous Simulation', Prentice-Hall, New Jersey, 1978.

7. 'Edinburgh FORTRAN Language Manual', Edinburgh Regional Computing Centre, Edinburgh 1974.

8. Uiterwyk, R.H.,'8K Basic Version 2.0', Southwest Technical Products Corp., 1977.

9. Brown, W., 'Modular Programming in PL/M', Computer, IEEE, March 1978, pp.40-46.

10. Bass, C., 'PLZ : A Family of System Programming Languages for Microprocessors', Computer, IEEE, March 1978, pp.34-39.

11. Ravenel, B.W., 'Toward a Pascal Standard', Computer, IEEE, April, 1979, pp.68-82.

12. Wickham, K., 'Pascal is a "Natural"', IEEE Spectrum, March 1979, pp.35-41.

13. Schneider, G.M., 'Pascal : An Overview', Computer, IEEE, April 1979, pp.61-66.

14. Bate, R.R. and Johnson, D.S., 'Language Extensions, Utilities Boost Pascal's Performance', Electronics, McGraw Hill, June 7 1979 pp.111-121.

15. Posa, J.G., 'Microcomputer Made for Pascal', Electronics, McGraw Hill, October 12 1978, pp.155.

16. Korn, G.A., 'A Proposed Method for Simplified Microcomputer Programming', Computer, IEEE, October 1975, pp.43-52.

17. Hicks, S.M., 'Forth's Forte is Tighter Programming', Electronics, McGraw Hill, March 15 1979, pp.114-118.

18. Baum, M.M., Blake, R.G. and Smale, R.J., 'Use of Digital Analogue Simulator (DAS)', The Computer Journal, The British Computer Society, August 1966, pp.175-180.

19. Dinely, J.L. and Preece, C., 'KALDAS, and Algorithmically Based Digital Simulation of Analogue Computation', The Computer Journal, The British Computer Society, August 1966, pp.181-187.

20. Brennan, R.D. and Sano, H., '"PACTOLUS" - A Digital Analog Simulator Program for the IBM 1620', Proceedings-Fall Joint Computer Conference, 1964, pp.299-312.

21. Trauboth, H. and Prasad, N., 'MARSYAS - A Software System for the Digital Simulation of Physical Systems', Proceedings- Spring Joint Computer Conference, 1970, pp.223-235.

22. Strauss, J.C. (ed), 'The SCi Continuous System Simulation Language (CSSL)', Simulation, December 1968, pp.281-303.

23. Conley, S.W., 'Micro-Dare BASIC/RT11', Microcomputer '77 Conference Record, IEEE, 1977, pp.67-70.

24. Benham, R.D., 'Interactive Simulation Language-8 (ISL-8)', Simulation, March 1971, pp.116-129.

25. Benham, R.D., 'An ISL-8 and ISL-15 Study of the Physiological Simulation Benchmark Experiment', Simulation, April 1972, pp.152-156.

26. Benham, R.D. and Taylor, G.R., 'Interactive Simulation Language for Hybrid Computers', Simulation, February 1977, pp.49-55.

27. Brown, G., 'Multi-User Simulation via a Small Digital Computer', Proc 7th AICA Conference, Prague, 1973, pp.113-116.

28. Hay, J.L., Pearce, J.G. and Narotam, M.D., 'Simulation Language Implementation on Minicomputers', Annales de l'Association internationale pour le Calcul analogique, No.4, October 1975, pp.260-269.

29. Hay, J.L., 'Interactive Simulation on Minicomputers : Part 1- ISIS, a CSSL Language', Simulation, July 1978, pp.1-7.

30. Pearce, J.G., 'Interactive Simulation on Minicomputers : Part 2- Implementation of the ISIS Language', Simulation, August 1978, pp.43-53.

31. Worth, G.A., 'SIMEX-Simulation Executive and Compilers', Proc. SCSC, 1974, pp.61-67.

32. Auslander, D.M., 'A Structured-Data, Interactive Dynamic System Simulation Language Suitable for Mini-Computer Implementation', Journal of Dynamic Systems, Measurement and Control, September 1974, pp.261-268.

33. Gakhal, S.S., 'Scaling Methods for a Digital Processor', Electronic Engineering, Morgan Grampian, London, May 1979, pp.101-105.

34. Baker, P.W., 'The Solution of Differential Equations on Short Word Length Computing Devices', IEEE Transactions on Computers Vol. c-28, No.3, March 1979, pp.205-214.

35. Edgar, A.D. and Lee, S.C., 'FOCUS : A New Number System for Microcomputers', Microcomputer '77 Conference Record, IEEE, 1977, pp.181-185.

36. Edgar, A.D. and Lee, S.C., 'FOCUS Microcomputer Number System', Comm of the ACM Vol.22, No.3, March 1979, pp.166-177.

37. Weissberger, A.J. and Toal, T., 'Tough Mathematical Tasks are Child's Play for Number Cruncher', Electronics, McGraw Hill, February 17 1977, pp.102-107.

38. Lawson, H.W., 'Programming Language Oriented Instruction Streams', IEEE Transactions on Computers Vol. c-17, No.5, May 1968, pp.476-485.

39. 'Series 1600 Microprocessor System Documentation', General Instrument Microelectronics, 1975.

40. 'Intel 8080 Microcomputer System Manual', Intel Corporation, 1975.

41. 'MCS-85 User's Manual', Intel Corporation, 1978.

42. 'Z80-CPU Technical Manual', Zilog.

43. 'M6800 Systems Reference and Data Sheets', Motorola Semiconductor Products Inc., 1975.

44. 'MCS6500 Microcomputer Family Programming Manual', MOS Technology Inc.

45. Losel, M. and Fompe, G., 'Fairchild F8 Microprocessor', Fairchild Semiconductor.

46. 'Signetics 2650 Microprocessor Manual', Signetics Corporation, 1975.

47. 'COSMAC Microprocessor Product Guide', RCA Corporation, 1977.

48. Powers, I., 'MC6809 Microprocessor', Microprocessors, IPC Business Press, Vol.2, No.3, June 1978, pp.162-165.

49. 'TMS 9900 Microprocessor Data Manual', Texas Instruments, 1975.

50. 'Intersil IM6100 CMOS 12 Bit Microprocessor', Intersil.

51. Wilnai, D. and Verhofstabt, W.J., 'One-Chip CPU Packs Power of General Purpose Minicomputers', Electronics, McGraw Hill, June 23 1977, pp.113-117.

52. Shima, M., 'Two Versions of 16 Bit Chip Span Microprocessor, Minicomputer Needs', Electronics, McGraw Hill, December 21 1978, pp.81-88.

53. Peuto, B.L., 'Architecture of a New Microprocessor', Computer, IEEE, February 1979, pp.10-21.

54. Stritter, E. and Gunter, T., 'A Microprocessor Architecture for a Changing World; The Motorola 68000', Computer, IEEE, February 1979, pp.43-52.

55. Morse, S.P., Pohlman, W.B. and Ravenel, B.W., 'The Intel 8086 Microprocessor; A 16 Bit Evolution of the 8080', Computer, IEEE, June 1978, pp.18-27.

56. Kornstein, H., '8086- Its Development and Capability', Microprocessors, IPC Business Press, Vol.2, No.3, June 1978, pp.166-169.

57. MacLennan, I.G., 'A Low Cost Video Display Unit', Students Project Report, Edinburgh University, May 1976.

58. 'NASCOM-1 Handbook', NASCO Sales Ltd., 1978.

59. McLeod, R. (ed), 'Edinburgh IMP Language Manual', Edinburgh Regional Computing Centre, Edinburgh, 1974.

60. McLeod, R., 'EMAS User's Guide', Edinburgh Regional Computing Centre, Edinburgh, 1976.

61. 'Super Assembly Language S16SAL', General Instrument Corporation.

62. 'Floating Point Arithmetic Package Version 2', General Instrument Corporation.

63. Haining, A., 'Microcomputer Simulation Software', Department of Electrical Engineering, University of Edinburgh, September 1979.

64. Abramson, H., 'Theory and Application of a Bottom-Up Syntax Directed Translator', Academic Press, London 1973.

65. Stephens, P.D., 'A Syntactic and Semantic Definition of the IMP Language', Edinburgh Regional Computing Centre, Edinburgh 1974.

66. Nash, G., 'Phase Locked Loop Design Fundamentals', Motorola Semiconductor Products Inc., Application Note AN-535, 1970.

67. Raamot, J., 'Integer Arithmetic Calculation of Phase Plane Trajectories with the Am2901 Bit Slice Microprocessor', Modelling and Simulation, Vol.9, No.4, Proc. of the 9th Annual Pittsburg Conference, April 1978, pp.1451-1457.