THE DESIGN OF ATE FOR TESTING SUB-ASSEMBLIES

Peter James Duggan

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfillment of the requirements of the degree of Master of Science in Engineering

Johannesburg, 1985

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University. The work was performed on the premises of the author's employer, Kentron (Pty) Ltd, and much of the reference material was obtained within Kentron.

[illegible]

Signed ____________ on this 24th day of November [illegible]

ABSTRACT

The testing of sub-assemblies is the least developed of all the fields of automatic test equipment and is in some ways different from the others. Sub-assemblies usually need a relatively small, yet often unusual and specialised, set of tests and test equipment. This investigation analyses the specific requirements of sub-assembly test equipment and considers methods of overcoming the main problems.

[illegible]

[illegible]

- [illegible]
- [illegible]
- [illegible]
- [illegible] to write test programs
- Separation of the functions of the test equipment specialist from those of the other system users.

ACKNOWLEDGEMENTS

I would like to thank my supervisor , Professor M Rodd, for his patience, advice, and criticism during the course of this investigation.

Thanks are also due to Kentron (Pty) Ltd, on whose premises, and using whose equipment, this work was done. Special thanks are due to my colleagues in the test equipment department at Kentron, who served as guinea pigs for many of the experiments performed.

ACKNOWLEDGEMENTS

I would like to thank my supervisor , Professor M Rodd, for his patience, advice, and criticism during the course of this investigation.

Thanks are also due to Kentron (Pty) Ltd, on whose premises, and using whose equipment, this work was done. Special thanks are due to my colleagues in the test equipment department at Kentron, who served as guinea pigs for many of the experiments performed.

CONTENTS

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ATE	Automatic Test Equipment
ATLAS	- Automatic Test Language for All Systems
DACU	- Data Acquisition and Control Unit
DVM	- Digital Voltmeter
IEEE	- Institute of Electrical and Electronics Engineers (USA)
IEEE 488 Bus	- An instrumentation bus standardised by the IEEE
PCB	- Printed Circuit Board
UUT	- Unit Under Test

1 INTRODUCTION

This dissertation describes the requirements analysis and development of an automatic test system and a test equipment design philosophy specifically targeted at the testing of sub-assemblies.

Sub-assemblies can be defined as intermediate stages in the production process. They consist typically of both electrical/electronic components and mechanical, optical or other components. They will commonly perform a single major system function. Testing of this type of item is poorly supported by currently available automatic test equipment. A theory is developed to account for this, and a practical design philosophy is presented to overcome these difficulties.

The complete design and implementation of a system was undertaken, and used to test the validity of the philosophy presented. Among the unusual features that were specially designed and implemented for this system are a multitasking operating system using a desktop paradigm, an extendible test language that is completely English-like in both vocabulary and syntax (and is uniquely specified to be oriented towards the requirements of the reader rather than the writer of the test program), a test program writing aid that enables the novice user to write tests automatically, and a software utility library that supports the concept of a

completely open and user extendible architecture.

The system described was developed in response to the overall theory and philosophy of sub-assembly testing that was developed during this investigation. Before describing this theory however, it is necessary to look at the the field of automatic testing in general, and sub-assembly testing in particular.

1.1 Automatic Test Equipment

The production of complex equipment demands correspondingly complex testing of that equipment before sale to the customer. For various reasons, it is often desirable to automate the process of testing. The test process may need to be performed very quickly, it may be extremely laborious, it may be dangerous, or it may be complex and prone to human error if done manually. Other advantages flow from the use of automatic test equipment; a uniform and repeatable test procedure is enforced, integration with production information systems is enhanced, the number of personnel required may be reduced, and the test personnel may be relieved of a great deal of boring and repetitious work. The operator of an automated test station does not need the level of skill and training that would be required from a technician doing the same job manually. The use of automatic test equipment may also be required for integration with other factory automation systems.

The cost of achieving these advantages is often high. Not only is the capital cost of Automatic Test Equipment high, but the cost of the software needed to program the equipment, the cost of the interfacing, and the subsequent maintenance costs are all so high that an analysis of the life-cycle cost of any automatic test system is necessary before a decision can be made to automate. The cost saving achieved from automation must be sufficient to justify the cost of automation.

1.2 Test Equipment Life Cycle

The life cycle of automatic test equipment (ATE) can be conveniently divided into three phases; development, installation, and maintenance.

The development phase consists of specifying the tests to be performed; specifying the jigs, interfaces and instrumentation required to perform the tests; designing and building or purchasing the test equipment hardware; writing the test software; and finally debugging the equipment. This process is often complicated by the parallel development of the unit to be tested. As the unit to be tested is developed, changes in its testing requirements occur, necessitating redesign of the test equipment. To avoid delaying the production of the unit the test equipment must often be over-designed and capable of more than is eventually required, so that

these changes can be made quickly and simply. To be successful in a high pressure development environment, test equipment must be flexible.

The installation phase begins when the test equipment is integrated with the rest of the production process, and the first production or pre-production units are manufactured and tested. This phase is generally characterised by the poor quality of the production process. The faults in the process, faults in the design of the production items and faults in the test equipment must be detected, isolated and eliminated. The place where all these faults surface is at the point of test. It is necessary that the test methods used and the limitations of the test equipment should be clearly visible at this stage so that the origin of test failures can be clearly identified. Often, test methods will be changed to identify certain process problems. To be successful in this phase of its life-cycle, test equipment must be both flexible and understandable.

The installation phase will gradually give over to the maintenance phase. As the production process matures, the number of tests performed will generally drop. Improved test methods will be implemented as the production yield statistics stabilise. Maintenance of the test software will consist of adapting it to suit these new circumstances. The test equipment hardware will also need maintenance in the conventional sense. This phase is generally the longest and most important

these changes can be made quickly and simply. To be successful in a high pressure development environment, test equipment must be flexible.

The installation phase begins when the test equipment is integrated with the rest of the production process, and the first production or pre-production units are manufactured and tested. This phase is generally characterised by the poor quality of the production process. The faults in the process, faults in the design of the production items and faults in the test equipment must be detected, isolated and eliminated. The place where all these faults surface is the point of test. It is necessary that the test method and the limitations of the test equipment should be clearly visible at this stage so that the origin of test failures can be clearly identified. Often, test methods will be changed to identify certain process problems. To be successful in this phase of its life-cycle, test equipment must be both flexible and understandable.

The installation phase will gradually give over to the maintenance phase. As the production process matures, the number of tests performed will generally drop. Improved test methods will be implemented as the production yield statistics stabilise. Maintenance of the test software will consist of adapting it to suit these new circumstances. The test equipment hardware will also need maintenance in the conventional sense. This phase is generally the longest and most important

phase of the test-equipment life-cycle. Ease of maintenance is thus one of the most important characteristics that test equipment must display.

In summary, ATE needs to be flexible in order to accommodate the changes that will occur over its life cycle; understandable so that the various users that will use the ATE over its life cycle can use it effectively; and maintainable, since the ATE will spend most of its life cycle in the maintenance phase.

1.3 The Users of Automatic Test Equipment

Over the life of a piece of test equipment, a variety of people will come into contact with it. These people can all be loosely termed the users of test equipment. Each of these users has his own requirements. Many of these requirements are conflicting.

The most important users of automatic test equipment are the following

- the designer of the unit to be tested
- the designer of the test equipment
- the person who specifies and applies the test equipment to test a particular unit
- the test writer
- the production engineer
- the quality assurance inspector
- the test equipment maintenance team
- the test equipment operator

Each of these users has his own special requirements. These will be discussed later (in chapter 2). However, it is worth noting that the requirement to be able to read and understand the test program is common to all the users, and is in fact the primary requirement of most of them.

1.4 The Structure of Automatic Test Equipment

There are several different classes of ATE, depending on the type of unit that is to be tested.

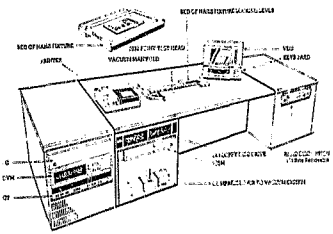
Component manufacturers test integrated circuits and other components on test equipment optimised for this type of work. Printed circuit boards (PCBs) can be tested for faults before components are inserted (bare board testers). The presence and orientation of many components can be checked without powering up the board using a passive in-circuit PCB tester. The overall functioning of the PCB can then be tested using a functional tester. These are the areas in which the science of ATE has progressed the furthest. In all these cases, the testers are characterised by a requirement to handle high volumes of basically electrical units.

At the other end of the scale are the special to purpose ATE systems used to test assemblies and sub-assemblies. These are characterised by relatively lower volumes of units to be tested, together with more complex tests

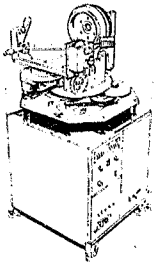
which are often non-electrical in nature.

Some examples of various types of ATE are shown in figure 1.4a.

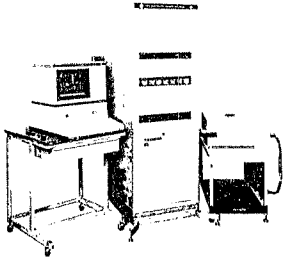
Although different classes of ATE can be identified, they all share some common characteristics. There is typically an interface to the Unit Under Test (UUT), which can vary from a simple connector to a complex jig containing optical, electrical and motion systems. There are usually various measuring instruments, power-supplies, and sources of electrical, optical, or other stimuli. A switching matrix is usually used in some form to connect the instruments, power supplies etc to the UUT in various configurations. The switching matrix is the heart of the ATE: virtually all measurements and stimuli are routed via the switching matrix. Controlling the ATE there is usually a computer. Control is achieved via a communication system such as the IEEE 488 interface bus. This is shown diagramatically in figure 1.4b.



Printed Circuit Board Tester



An Electro-Mechanical Test Jig



A Component Test System

Figure 1.4a: Examples of Different Types of ATE

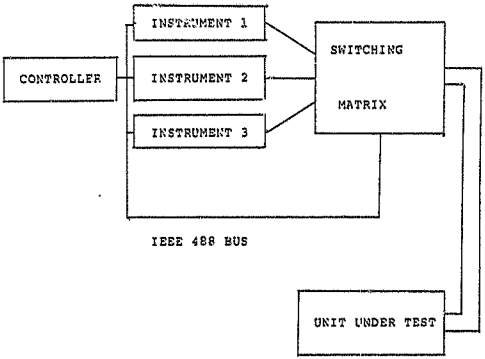


Figure 1.4b: ATE Block Diagram

1.5 The Role of Software in Automatic Test Equipment

In order to perform a test, a series of actions must be performed by the ATE. It is the function of the software running in the controlling computer to order and structure these actions. The software can be divided into two areas; the utilities that enable the computer to be used effectively as an ATE controller, and the test programs which specify exactly which actions are to be performed to test the UUT.

Often the test programs will make use of pre-programmed routines which perform commonly used functions, such as the setting up and reading of a digital voltmeter (DVM). Other pre-programmed functions might include the routines to print result sheets. Some of these tools may be part of the Operating System of the computer, and others may be specially written as part of the overall test program. In general, the tools used to enable the operator to use the computer will be part of the operating system, and the tools used to interface the computer to the rest of the ATE will be specially written.

Other tools that may be added are routines to assist in writing test programs, debugging programs, or in other ways assisting the user to use the ATE. These latter routines are usually not part of the operating system but have been specially written for the test environment. They are often referred to as a secondary

operating system. The test programs themselves may be written in a common computer language such as BASIC, or in a language specially designed for testing, such as ATLAS. The program that interprets or compiles the test programs is often included as part of a secondary operating system.

These software sub-systems (the operating system, secondary operating system, interface tools etc.) will usually be written by different people, with different goals, using different languages. This often makes maintenance of ATE software a difficult and expensive task. The complexity of the structures found in ATE software, and the large number of different structures, is due in part to the large number of users of ATE. Each different class of user needs a different set of tools.

Where an ATE and its software have been designed together, the complexity of the software can be greatly reduced. The operating system can be designed specifically for the requirements of ATE, the test language can be tailored to the exact instruments used, and the other tools necessary can be included in the system in a logical and consistent fashion. This is the route followed by the specialist ATE manufacturers.

1.6 In-House Development of Automatic Test Equipment

Automatic test equipment is expensive to buy. For this reason it may appear attractive to develop such

equipment in-house. However, the initial cost saving is often swamped by the later maintenance costs. These costs, as will be shown later, are primarily due to software maintenance. It is usually cheaper to buy ATE than develop it.

The purchase of an ATE, however, has disadvantages. The off-the-shelf system is likely to be structured in an inflexible fashion; if it does not fit the needs exactly, it may be very difficult to modify to include extra functions.

The need to include special functions becomes especially important when the ATE is to be used at levels of testing higher than PC board level, that is to test sub-assemblies or assemblies. Often mechanical jigs need to be controlled. The UUT itself may need complex stimuli, which may not be electrical in nature but optical or mechanical. Unusual instrumentation is often required. Paradoxically, the number of tests at higher levels of assembly is often smaller than at PC board level, with a requirement for relatively few but highly specialised resources from the ATE. Thus the bought-in ATE will often not meet the technical requirements of a UUT, implying that it will have to be modified. On the other hand many of the existing resources of the bought-in ATE will not be utilised, although they will of course have to be paid for.

In-house development of ATE is thus indicated when the

UUT is an electro-optical or electro-mechanical device with special stimulus or measurement requirements. It is often not possible to purchase a suitable ATE for this type of device. In addition, when the device is a relatively simple assembly or sub-assembly it may be cost effective to design an in-house ATE simply because of the cost savings that can be achieved by using a small set of carefully chosen ATE resources, rather than a large, general purpose ATE system. The cost savings must however be weighed against the high maintenance cost that can be expected from in-house designed ATE.

1.7 Types of UUT : PCBs, Sub-assemblies and Assemblies

A typical production process will manufacture printed circuit boards (PCBs) and test these before further assembly. These will then be assembled into sub-assemblies, which will generally perform some major function of the final product. The various sub-assemblies are then integrated into the final assembly.

The testing process at each stage is usually restricted to testing the previous process. Therefore testing PCBs will primarily involve testing whether or not the PCB has been assembled correctly rather than whether or not the design is correct, or whether the components meet their individual specifications.

Testing the final assembly similarly involves testing

the process of integration of sub-assemblies that has just occurred. Often, it is unnecessary at this stage to test the exact specification of system parameters. For example, the final testing of a TV set may simply involve a *functional test and inspection*, rather than a parametric test of RF sensitivity, tube alignment, and so on. The parametric tests are most commonly performed at sub-assembly level.

The reason for performing parametric tests at sub-assembly level is primarily economic. Sometimes there are also technical reasons; it may be impossible to access a certain parameter after final assembly. It is generally true to say that it is usually most economical to test a parameter at as early a stage in the production process as possible. This is because the cost of failure is reduced; the amount of disassembly and wasted assembly costs are minimised by this philosophy. The *earliest stage* that the primary system functions can be tested is generally the sub-assembly level.

1.8 Overall Aims of the Study

This study was aimed at defining and satisfying the requirements of the users of *in-house designed* ATE systems for sub-assemblies. The overall characteristics of such ATE systems are that they must be low cost relative to the bought-in alternative; they must be

flexible in order to meet the special requirements of sub-assembly testing; and they must meet the special requirements of all the users of ATE in a consistent manner conducive to low development and maintenance costs over the entire ATE life-cycle. Reducing the cost of maintaining in-house developed ATE was seen to be an especially important goal.

In its initial phase, the study was directed at more closely defining these requirements. At this point a system specification was defined and feasibility studies undertaken. A system was then implemented and evaluated using a real UUT. The validity of the original requirements analysis was then investigated and conclusions drawn as to the requirements of the users of this type of ATE and useful techniques for achieving them.

The final outputs of the study were therefore a set of recommendations for the design of ATE for testing sub-assemblies, and a working prototype of such a system.

1.9 Structure of this Dissertation

The overall structure of this dissertation attempts to lead the reader through an introduction to ATE in general, a discussion of the specific problems of sub-assembly ATE, a narrative description of the progress of the investigation, a more detailed analysis

of some of the primary technical areas, a description of an actual implementation of the ideas developed in the study, and an analysis of the results obtained with the system. Finally, conclusions are drawn as to the validity of the goals of the study and the usefulness of the techniques used to achieve these goals. Appendices are attached containing material that may be of interest. These appendices are referred to where appropriate in the body of the dissertation.

2 TESTING SUB-ASSEMBLIES

The problem of testing sub-assemblies can be stated briefly as follows:

Testing a sub-assembly usually involves testing a relatively small number of parameters. At this level, the UUT has normally only one major function; however this function often requires the use of one or more special-to-purpose instruments to test it. Off-the-shelf ATE systems usually need to be extended to cater for these special functions. On the other hand, the built in functions of such a bought-in ATE systems are often under-utilised in testing a sub-assembly. It is therefore attractive to build a special-to-purpose ATE system for a particular sub-assembly. In practice however, such in-house ATE systems have proven to be costly to develop and maintain.

2.1 Existing Solutions

Two classes of solution to the general production ATE problem are available. The first is the bought in ATE, and the second is the in-house developed ATE.

Bought-in ATEs vary from systems costing around R40 000 such as the Beaver Major and the Wayne-Kerr A8000, to systems costing well over R200 000, such as those manufactured by Membrane, Telodyne, Fairchild, Olivetti, and others (1984 prices).

The majority of ATE systems on the market are aimed at the testing of either printed circuit boards or electronic components, especially digital components. These systems are in general useless for the type of application under investigation. This is primarily because a typical sub-assembly tester must do more than apply simple electronic stimuli and measure digital or analogue voltages.

Often the UUT will need to be moved (eg on a rate table); pneumatic or hydraulic systems will require the measurement of rates of flow, pressures, torques, displacement, etc; optical systems may need irradiance measurements, scanning jigs, focusing and alignment control systems. In general, testing this type of unit requires the application of non-electronic stimuli, and the measurement of non-electronic outputs, often with some form of process control functions required. Systems that can be used in this environment usually contain programmable D/A and A/D modules, relay modules, multiplexed switching systems, and a facility for controlling ancillary instrumentation via the IEEE 488 bus.

Test programs (on bought-in ATEs) are usually written in a relatively high-level test language, such as ATLAS. In some systems software writing aids, such as menus, are provided. However, these writing aids are usually of little help if special instruments must be added to the

system; it is often difficult to add new instruments to bought-in ATEs.

An in-house developed system is normally designed around an instrument controller, such as a HP9826 desktop computer, and a selected set of instruments. Most commonly, the instruments are controlled via the IEEE 488 bus. The instruments are usually housed in a rack or console. Each instrument in the system will need software to be written for it, since the software control of instruments is not standardised, and is often complex. In addition, software will need to be written to perform "house-keeping" functions such as generating result-sheets, storing high and low limits for each result, and so on. The tests themselves will usually be written in the language native to the controller; this is usually BASIC.

Whether the system is developed in-house or is bought-in, special to purpose interfacing will need to be developed. In addition, there is usually a mechanical jig that will typically contain electrical, mechanical or optical systems that will need to be controlled by the ATE. The software to do this will need to be written before the system can be used to test UUTs.

Development Phase	Installation Phase	Maintenance Phase
- Modify Equipment	- [illegible]	- Isolate Hardware Failures
- Build/Purchase Equipment	- Test Preproduction Hardware	- Use ATE for Production Process Monitoring/ Fault Finding
- Modify Tests	- Interface ATE Faults	- ATE and Diagnostic Procedures
- Write Tests	- Correct Production Process	- Debug/Modify Test Procedures
- Debug ATE	- Test Early Production Units	- Reliability/Maintainability Improvement Program
	- Improve Production Process	
	- Improve ATE/Tests	

2.2 Problems Perceived with Existing Solutions

As has been shown, there are two classes of solution. The first of these, the bought-in ATE system, is often not suitable for reasons that have already been given. The second class of solution, the in-house developed ATE is often expensive to develop and to maintain. These are major problems; underlying them, however, is a set of problems that is related to the design methodology adopted by the designers of both in-house and bought-in ATE systems. These problems are more fundamental than those already discussed.

In order to appreciate these problems, it is necessary to recall the ATE life-cycle mentioned earlier. This life-cycle is shown diagramatically in figure 2.2. The problems are related to the users of the ATE. In each stage of the ATE life-cycle the users are different, and their problems are different.

During the development phase the the designer of the UUT will decide on the parameters that will need to be tested during production. He will usually write an Acceptance Test Specification which will detail the parameters that must be tested and the acceptable performance limits that each UUT must achieve in order to be acceptable. Deciding on these limits involves more than simply analysing the functional requirements of the UUT. Such factors as the expected degradation over the lifetime of the UUT, the accuracy of the test equipment,

the expected statistical distribution around the nominal of production units, and the percentage customer risk or producer risk that is prescribed must all be taken into account. Often these factors are unknown, or only imprecisely known, during development. It is therefore to be expected that the acceptance limits of a UUT will be subject to change.

It is clear that the UUT designer must have a clear understanding of exactly how and to what accuracy the the ATE is testing the UUT. Often, subtle problems in the test procedure will cause inaccurate results. The ATE designer and the UUT designer must be able to communicate in a language that both can understand. Often the UUT designer will have little or no knowledge of software. He must then rely on the ATE designer to translate the software for him. Two problems arise: the pressure of timescales will often result in the UUT designer simply accepting the word of the ATE designer; and any subtle errors in test methods that the ATE designer may have made will not be picked up by the UUT designer since he cannot read the test program without assistance from the writer of the program.

This requirement for a detailed understanding of the test procedure is shared by the quality assurance inspector and the production engineer at a later phase of the ATE life-cycle. These people are even less likely to be able to follow the software, even if it is written in so common a language as BASIC. They are

specialists in other areas, and should have no need to become experts in software as well. They do however need to have a detailed understanding of how the UUT is being tested. For example, it may be necessary to evaluate the measurement accuracy of the ATE at some point in a test. It is necessary to know not only which instrument is being used, but on on what function and range as well.

The test equipment maintenance team find themselves with two problem areas. In the first place there is the normal maintenance of the test equipment hardware. In the second place they will often be required to perform software maintenance.

Maintenance of the hardware implies being able to isolate a fault to a particular wire, relay, or instrument in the ATE. In order to do this the ATE maintainer needs to have a detailed knowledge of exactly what the ATE was doing when the fault occurred. He needs to know more than that it was performing Test 3. He needs to know precisely what point on the UUT was connected to what instrument and via what relay. His only source of this information is the test software.

The other type of maintenance, software maintenance, arises from normal improvements in the production process that occur as the process matures. The ATE maintenance team find themselves in the position of test writers.

The test writer needs to translate a test procedure into

a test program. This is not usually a very difficult task. However the existing solutions require the test writer to learn a software language. In some cases, especially the case of bought in software, these languages are relatively easy to learn and there may even be writing aids that assist this process. However, as has been discussed, the major part of a sub-assembly test usually requires the use of ancillary equipment that has been added to the the bought-in ATE. The use of this equipment is in general not supported by the software aids purchased with the ATE. The writing and maintenance of test software, whether for in-house or bought-in ATE systems, is therefore often done by scarce and expensive manpower.

The designer of an ATE (whether he is attached to a specialist ATE design company or is configuring an in-house ATE system) will be a specialist in his field. Much of the software he writes will be at the operating system level. This type of software is best written using specialist tools. Often the systems level programmer will write software in assembler, or a specialist high-level language such as C or FORTH. The operating systems of bought in ATEs will be of this kind. This is usually not a problem, except that these systems are not extendible by the user.

The in-house developed ATE however has a larger problem. Since the development cost of in-house ATE must be amortised over relatively few systems, the cost of

writing and debugging such software becomes high.

To combat both these problems (inextensibilty of bought in operating systems, and high development cost of in-house operating systems), non-specialist languages such as BASIC are often used. Companies such as Tektronix and Hewlett-Packard sell ATE systems with secondary operating systems written in BASIC. The disadvantage of such bought-in systems is their clumsiness. It is to be expected that an operating system written in BASIC will be slow and lacking in power. Typically the tests will need to be written in BASIC as well, since the operating system does not support higher level languages or sophisticated writing aids. Such systems are aimed primarily at the laboratory environment, but do not address the needs of production testing.

The use of BASIC in in-house developed ATEs is usually due to the lack of specialists who are competent in the use of more powerful tools. Despite its easy-to-write reputation, BASIC is notoriously difficult to maintain. Thus any savings made during the writing process are negated by the maintenance costs. These systems do not in any case address what is becoming obvious as the main problem: more people spend more time reading test programs than writing them.

Configuration of a given ATE to suit the needs of a particular UUT involves the design of interfaces and

mechanical jigs, as well as the integration of special to purpose instruments. The engineer performing this function is faced with two problems. The first is the physical integration of the system. The second is the integration of the new parts with the existing software. The first problem is usually solved by such systems as the IEEE 488 bus. The second problem is less amenable to solution. In the case of bought in ATE systems, the solution may be that all software involving the extensions be written and maintained by a specialist. Often, bought in ATEs provide extremely primitive tools to support such extensions. Maintenance of this kind of software is expensive. In-house ATE systems offer significant advantages in this area. The special instruments and jigs can be integrated with the ATE in a consistent fashion, and any special software can be written by the specialists who wrote the original ATE software.

The test equipment operator has a completely different set of problems. He must use the ATE to test UUTs. He is therefore concerned primarily with the "user friendliness" of the system. His primary interface with the ATE ' provided by the software, in particular the operating system. He will be using the system for long periods, and so will generally get to know the system well. However, the turnover in test equipment operators is relatively high, since it is not a particularly challenging or interesting job. The operating system

must *therefore be easy to use (low training cost)*, but also not patronise or irritate the operator. The same operating system that allows the operator to use the system easily must not get in the way of the other users of the system.

It is clear that the problem field lies primarily in the software area. An ATE system is held together by software, and all the users of the ATE use it via software. In particular, the ATE operating systems and the test languages do not address the needs of all the users. Operating systems are structured to aid the writers of software rather than the readers. Most of the software aids that are provided are there to assist the software writer. As has been shown, the majority of users are more interested in reading the test software than in writing it. Thus the needs of the software reader are seen to be more important than those of the writer.

However, the needs of the software writer must also be addressed. It appears that there are two types of software writer in question. These are the specialist (who writes the system level software) and the non-specialist (who writes and maintains the test software).

The existing solutions cater to one or the other of these users: either the system is very easy for the non-specialist to use but impossible for the specialist

to extend or modify (most bought in ATEs fall into this category), or the system allows the specialist to tailor it to meet any testing requirement precisely, but requires the specialist to also maintain the resulting system (this is typical of in-house ATE systems).

The fundamental problem is thus seen to be this; ATE systems, whether bought-in or designed in-house are designed for the INITIAL users of the systems, rather than all the users. In the case of bought-in ATEs, the initial user is the test writer. Bought-in systems provide easy-to-write software systems. The initial user of in-house ATE is the ATE configuration engineer. These systems typically have operating systems written in BASIC, which is usually the language the engineer knows.

Thus the sub-assembly ATE field is characterised by

- the use of inappropriate software: operating systems written in BASIC, test programs written in FORTRAN;
- the use of inappropriate manpower: operating systems written by instrumentation engineers, trivial test programs written and maintained by engineers;
- the use of inappropriate hardware: it is cheaper to use ten percent of the resources of an ATE costing R100 000, than to design and build a simple ATE system in-house;
- inflexibility: bought-in systems are sold as

complete products, in-house developed systems are too expensive to change once in production;

- user dissatisfaction: the maintenance technician finds it impossible to understand the hardware, due to unreadable software; the Quality Assurance inspector mistrusts the ATE, since he has no way of knowing what it is doing;

- reduced usability; the power of an ATE is under-utilised because it is too expensive to configure for new tasks, such as statistical sampling of a problem parameter on the production line.

2.3 Work Done in Related Fields

In the previous section it was shown that the problem lies primarily in the field of software. It is therefore to be expected that developments in the field of software system design would be of use in this study, as well as work done in the ATE field itself. An overview of the field is given here. The references are, of course, far from exhaustive, but serve to highlight the important areas that have provided an input to this study.

ATE technology is most advanced in the field of component and printed circuit board testing. Sharma and Avila (1983) have given a description of the

characteristics of ATE software in the LSI testing environment, and describe methods of automatic test software generation for this type of application. The solutions they propose are not readily applicable to the field of sub-assembly ATE.

The differences between sub-assembly testing and PC board and component testing have been well described by Turino (1983). He has also shown that in a mature technology such as the component test technology, bought in systems are significantly cheaper than in-house developed systems. He postulates that for bought-in sub-assembly ATE to reach the same maturity level as board and component level ATE great flexibility will be required; he discusses using robotics to solve this problem.

The economic advantages of modular, flexible ATE systems over dedicated ATEs have recently been described by Adams (1985). The development of modular test systems around personal computers has been described by Elder and Potts (1984), who show that there are technical and cost advantages to be obtained from these systems when compared to more classical instrument and controller based systems.

Mandelzweig (1980) has given an excellent description of the development a particular in-house ATE system for testing gyroscopes. He gives the reasons for using ATE, and for developing the ATE in-house in that particular

case. This particular ATE is reasonably typical of in-house designed ATE systems. It uses an Apple computer, the IEEE 488 bus, several special-to-purpose instruments, and a sophisticated electro-mechanical interface. It utilises a secondary operating system written in BASIC. The test programs are also written in BASIC.

The maintenance phase of this particular ATE system was investigated as part of this study. The results of this investigation generally supported the problem description given in the previous chapters. While an excellent example of hardware and software design from the point of view of testing gyroscopes, the cost of software maintenance has been high. Even relatively trivial changes to test methods require specialist attention. A simpler gyroscope, for a different application, has also been developed, and an ATE designed to test it. The software used in the earlier ATE was modified for use in the new system. The cost of this modification was not signifigantly smaller than the original development cost. This was primarily due to the test procedures being quite different. As will be shown later, the cost of writing tests can be made a trivial part of the ATE cost, if appropriate methods are used.

A discussion of the importance and difficulties of software maintenance in the military ATE field has been given by Forss and Lundin (1986). Their discussion emphasises the conflicts between the test writer,

operating system writer, and test system maintainer. The problem of obtaining operating system support over long periods from ATE vendors is also discussed.

The best known standard for test software is the language ATLAS (Abbreviated Test Language for All Systems), which is described in IEEE Std 416-1981. A subset knowns as Common ATLAS or C/ATLAS is described in IEEE Std 716-1982. As will be discussed, ATLAS has some serious shortcomings and does not meet the requirements developed in this study.

The field of software operating systems has undergone a revolution in recent years. Specific attention has been paid to making systems easier to use without penalising the expert user. It is generally accepted that work done at Rank Xerox Palo Alto research centre has been in the forefront of developments. A good description of of the work done there has been given by Tesler (1981). This work has entered the mainstream of computer hardware and software design via such machines as the Apple Macintosh computer, the language Modula-2, and the Lilith computer (Wirth,1981; Ohran,1984)

Johnson (1982), and Stewart (1982) have each discussed automatic program generation with examples of available systems. Bailey (1984) has given a good overview of computer ergonomics and the design of "user friendly" systems.

2.4 Specific Goals of the Study

After the initial investigation of the problem had been completed, a specific set of goals was identified. These are stated here to provide a basis for the further discussion. The rationale behind these goals will be developed in later sections.

- to develop a test language with the following characteristics
  - English vocabulary
  - English syntax
  - oriented towards the hardware (instruments and UUT), rather than abstracts such as voltage ("measure dc signal") - extendible in a consistent fashion to include special instruments, functions, or procedures
- to develop an operating system that
  - is non-directive (ie the computer responds to commands from the operator, rather than the operator responding to demands from the computer)
  - allows both the specialist and non-specialist to function effectively
  - is multitasking in nature
  - provides menus without enforcing their use
  - does not require the non-specialist to learn computer jargon

- allows the specialist to use powerful systems level programming tools
- allows the use of non-specialist personnel to write and update tests
- allows the use of specialist personnel to address those problems which by their nature require specialist attention
- allows the tools developed by specialists to be usable by the non-specialist in a consistent fashion
- is optimised for readability

- to develop a test writing aid that allows the non-specialist to write tests easily, since the test language is specified as being reader rather than writer oriented.

- to develop a prototype system that uses the above tools.

The hardware cost, including racks, looms, the controller and switching matrix, but excluding instruments and jigs, and excluding manpower costs, should be less than R20,000. This was a reasonably arbitrary decision, based on the costs of bought in ATE systems. The cost of instruments, jigs etc would vary from UUT to UUT and would be independent of the primary ATE cost. Bought in ATEs would however have some built in instrumentation that would meet the needs of a proportion of UUTs. It was therefore decided to aim at a

capital cost of roughly 50% of the typical low-cost bought-in ATE.

The life-cycle cost of the ATE, if it meets the above goals, is expected to be considerably lower than any available ATE system.

3 METHOD OF SOLUTION

This section describes the course of the investigation in narrative fashion. The main objectives and results of each phase of the investigation are described briefly. The major technical trade-offs and decisions that were made will be discussed more fully in later sections. In these cases, the issue is merely touched on here. Certain peripheral issues which were addressed are also described in this section.

3.1 Initial Survey of ATE Users

The first step was to define more clearly the requirements of the potential users of an ATE system. A questionnaire was drawn up (see Appendix A) and distributed among ATE users of all kinds. The results were not used for statistical analysis, but to draw up a list of desirable characteristics that an ATE should have in order to satisfy the needs of all users. This list is given in Appendix A. It can be seen that many of the requirements are conflicting. This survey emphasised and refined the perception that has already been discussed that existing ATEs do not satisfy the needs of all users.

3.2 Specification of a Specific System

Once the requirements had been set out, the next step

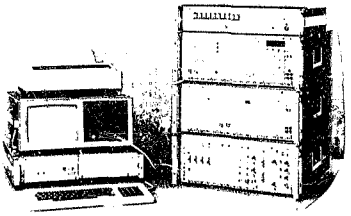
was to investigate ways of meeting the requirements. A general specification of the system was derived from the list obtained previously. Most of the alternative solutions to the requirement were seen to be software alternatives. Among the alternatives considered were different types of test language (BASIC, ATLAS, etc,), different operating system models (conventional DOS systems, Smalltalk, spreadsheet based systems, program generators, forms based systems, etc), various systems level programming languages (BASIC, Forth, C, Pascal, Assembler), different computers (HP9826, Fluke, Apple), and different packaging systems (racks, desks, instrument consoles).

The overall architecture of the system was analysed extensively. It was seen that the systems's potential flexibility and extensibility, as well as its overall lifecycle cost would be greatly affected by the choice of architecture. The final choice of an open, extendible architecture base around IEEE-488 bus compatible instrumentation is motivated later in this chapter.

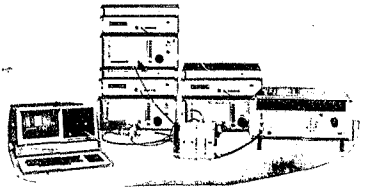
The software decisions are discussed in this and later chapters, but can be summarised as follows. No existing test language was found to be suitable, and it was decided to develop a language that would be English-like both in vocabulary and syntax. The language would be hardware oriented ie it would allow the instruments in use to be set up and read explicitly. An operating

system using a desktop paradigm, menus, and a pointing device was decided on, based on work done at Xerox PARC laboratories. The operating system concept was extended to provide a complete programming environment, rather than a simple set of tools. A decision was made that the system would be multitasking in nature. A writing aid was also specified to assist the writer of tests. The concept of modeless operation was also specified. (Modality can best be described by means of examples; an operating system that accesses the disk via a command interpreter which is not available when an editing facility, for example, is being used, is said to be modal. One is either in Edit mode or in Disk Operations mode).

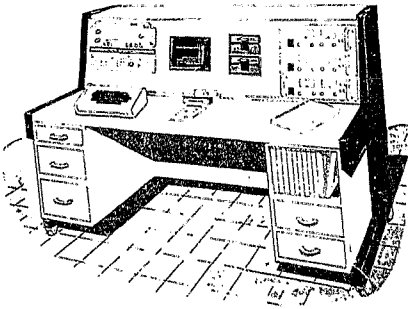
After some study, a decision was made to use the IEEE 488 bus as the instrumentation bus. There is really no viable alternative to this system if a large variety of instrumentation is to be usable.



An Internal Bus Based System



An Extendable VI Crate System



A Closed Architecture

Figure 3.3: Examples of ATF Architectures

3.3 System Architecture

The overall architecture of the ATE was studied, with several alternative architectures considered. It became apparent that three different architectures could be described: the self-contained bus architecture, the closed general purpose ATE architecture, and the uncommitted open architecture. Examples of these are illustrated in figure 3.2.

The self-contained bus systems typically are housed in a card-cage and consist of a central computer module with a set of plug-in peripheral modules. They are usually compact and inexpensive. The Wayne-Kerr A8000 is typical of this type of system. Other systems are marketed for a variety of desktop and personal computers. This type of system has the advantage that the instrumentation is cheap, since it need not provide for manual operation. Since the instrumentation is tightly coupled to the controller on an internal bus, the software interface is potentially easier to implement, as well as being inherently faster.

The major disadvantage of this type of system is that the user is restricted to modules made by the supplier of the system. The busses of these systems are not as yet standardised. Each supplier also has his own software support. In general, the software supplied with these systems does not satisfy the requirements identified in this investigation as being important, the

emphasis being on writing rather than reading the test software. If an external bus is provided, such as the IEEE 488 bus, this is not supported by the software in a consistent high-level fashion. Although this type of system is potentially the most cost effective, and also has certain technical advantages, the technology is not yet mature. There is a lack of standards and thus flexibility is difficult to achieve.

Certain vendors are starting to use standard microprocessor busses for this purpose. A typical system, using the VME bus for the instrumentation (although it uses an IBM PC as the user interface), has been described by Elder and Potts (1984). The two most common busses for this purpose are the Multibus and the VME bus. At present these busses are expensive relative to other instrumentation, and the variety of modules available is limited to such things as parallel digital I/O modules, Analog to Digital converter modules, and other relatively low level functions.

The closed, general purpose ATE architecture is typically found in top-of-the-range systems from suppliers such as Membrane and Hewlett-Packard. These systems consist of a central mini- or micro-computer controlling a fixed set of general purpose instrumentation in a console or desk. In order to cover a wide range of testing problems they will have a large number of sophisticated instruments built in. Their major advantage is that all these instruments are

supported at a high level by the operating system software. However, these systems are by their nature extremely expensive. This is incompatible with the requirement, inherent in sub-assembly testing, for a limited number of special to purpose instruments, preferably of low cost.

The architecture chosen was the uncommitted open architecture. This consists of a controller housed in an instrument console or desk, tools for supporting the testing function, and an operating system specifically designed to allow a user selected range of instruments to be added as required. The various instruments are interfaced to the controller via an instrumentation bus (usually the IEEE-488 bus). A library of software modules to support various instruments may be available, and will be extended as new instruments are chosen by various users. This minimal system will typically start out as simply a computer together with the operating system software and the library of instrument drivers. The user will then select the instruments he needs and add them in. The overall system packaging (eg in a desk or console) is up to the user.

When this study was done, no commercially available systems were available that fitted into this category. Recently, certain companies, notably Hewlett-Packard and Tektronix, have introduced systems that are in some respects similar to this concept; these are primarily aimed at laboratory testing rather than production

testing. A description of a system that partially fits this concept has recently been given by Adams (1985); he emphasises that flexibility of hardware selection can save costs. He does not, however, satisfactorily address the software aspects, or the needs of all the users.

The open architecture has a cost advantage over the closed architecture since the minimum number of resources can be included. When implemented correctly, the software specifically make provision for the addition of new devices in a consistent fashion. The hardware is however more expensive than the self contained bus system since there is a duplication of certain functions, such as the front panels of the instruments (the front panels are not used in the ATE environment). It has the advantage however in the range of instruments available; virtually any testing situation can be catered for by the correct choice of IEEE-488 bus compatible instrumentation.

This architecture is clearly superior to the others in this case, since sub-assembly testing typically requires the use of a small number of special instruments which must be integrated with the system.

It is possible to integrate this approach with the self-contained bus approach by making use of an instrument that is generically known as a data acquisition and control unit or DACU. This type of instrument typically has a microcomputer controlling an

internal bus. Various modules, such as multiplexer, voltmeters, and so on can be plugged into the DACU. The system controller treats this combination as a single instrument on the IEEE-488 bus. Hewlett-Packard is an example of a company manufacturing these devices. They are in general superior to the microprocessor-bus based systems described earlier, being comparable in functionality to general purpose instruments such as digital voltmeters.

3.4 Choice of Computer and Programming Language

The choice of the computer and systems level programming language were interdependent. The final decision was dictated by two other decisions that were made. These were that the operating system should allow multi-tasking, and that a special English-like test language should be designed.

The computers considered were the Hewlett-Packard HP9826, the Fluke 1720A, and the Apple II. The IBM PC was not considered, since there was no locally known IEEE 488 bus interface available for it at the time that the decision was made.

The languages considered were BASIC, C, Forth, Pascal, and Assembler.

BASIC was eliminated because the versions available on the computers considered did not support multi-tasking.

In addition it was considered inappropriate as a language to write a compiler with, being too slow and lacking systems level programming capabilities. This decision effectively eliminated the Fluke computer from consideration, since it supported no other language.

The language C seemed an ideal choice, but was not yet available on any of the machines considered, although several C compilers seemed to be on the verge of being released. It was considered too risky to wait for them. The same argument had already eliminated a language that from many points of view would probably have been ideal, namely Ada.

Pascal was also eliminated because it was not available in a multi-tasking environment. In addition, Pascal is weak in its primitive I/O capabilities, ie its ability to interface with hardware, and this is a serious drawback in the ATE environment.

Forth was the only remaining high level language available for both the remaining computer alternatives. Although Forth has some draw-backs it is an excellent system level programming language. Forth code is fast and very compact, supports hardware interaction as effectively as Assembler, and can support multitasking without assistance from an operating system.

Its draw-backs are chiefly that it is difficult to learn, requires an expert to maintain, and has no built in protection features such as are found in Pascal. It

can also be difficult to read, but this often depends more on programming style than on the language itself. There are several high level languages (such as Ada) which directly address the problem of readability; however none was available for the computers in question.

The only viable alternative to Forth was Assembler. It is significant that most operating systems and compilers are written in Assembler. Assembler suffers from all the drawbacks of Forth; in fact it is more difficult to write good Assembler code than Forth code, and Assembler code is definitely more difficult to read, mainly due to the sheer length of the listings and the lack of inherent structure of Assembler programs.

While conventional Assembler code is potentially faster than Forth code, it less compact. It was expected that parts of the system would have to be written in Assembler no matter what high level language was chosen. However, the system cost constraints made the compactness of Forth very desirable. (It should be noted that this compactness is less a function of Forth itself than of the indirect threading method used commonly to implement it. However, very few languages other than Forth are indirect threaded. Implementing indirect threaded code directly in Assembler is extremely tedious and prone to error, and was not considered to be a viable alternative).

METHOD OF SOLUTION

It was expected that programming in Assembler would take far longer and be more risky than programming in Forth. Forth provides high level constructs and is inherently structured. Forth is also interactive in nature which makes for extremely fast program generation and debugging. Even when comparing Forth to other high level languages, Forth code is usually faster to implement. However, this must normally be weighed against its disadvantages.....poor readability, lack of strong typing, and lack of robustness. These disadvantages are, however, not relevant when Forth is compared to Assembler, which suffers from them to perhaps an even greater degree.

The final decision was to implement the system in Forth, and then recode the time critical portions of the system in Assembler. This approach would allow swift system development, and a near optimum combination of speed and compactness.

The choice of computer was then made in favour of the Apple. The major reason was cost. At that time the HP9826 cost around R15000, in comparison with approximately R5000 for the Apple. Since then the Apple has become cheaper, while the Hewlett Packard machine has become more expensive. The HP9826 is a far more powerful machine, but previous experience with Forth in the Apple led to the expectation that the Apple would be sufficiently powerful. The relatively small memory of the Apple would be counteracted by the extremely compact

code that Forth generates. These assumptions were tested in the feasability study phase discussed in the next section. Finally, the Apple was a well proven IEEE-488 bus controller (Mandelzweig, 1980).

3.5 ATE Packaging

The physical packaging of the system was considered to be sufficiently important to warrant some investigation. The alternatives considered were

- the 19 inch vertical rack system
- an instrument console with the computer built in
- an instrument desk with the computer free standing on it

These systems are illustrated in figure 3.5.

The 19" rack approach is the most widely used for in-house ATE systems. It does not result in a particularly aesthetically pleasing system, and as a result the other systems are preferred by the ATE operators questioned. There is evidence that elegant and good looking systems are less prone to operator abuse than unattractive systems. For this reason the other two alternatives were investigated. The relative costs were not found to differ significantly, typical ATE systems costing approximately R8000 (manpower costs included) to package. The console system provides an attractive and neat solution, but the built in computer was thought to be a disadvantage ergonomically, since most desktop

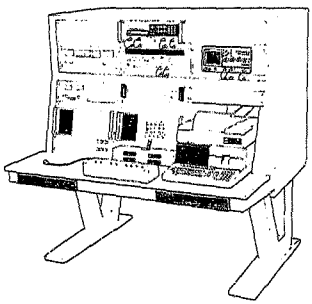
computers are designed to be free standing. In addition, printers and plotters are not easily housed in a console.

The third alternative was therefore selected. This has the computer and its peripherals free standing on a desk. The instrumentation is housed in the pedestals of the desk. Instrumentation that needs to be visible to the operator (such as an oscilloscope) is mounted in a rotatable console on top of the desk.

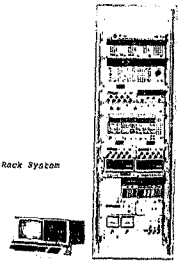
This phase resulted in a system specification which can be summarised as follows:

- the system controller was to be an Apple II desktop micro computer
- the system programming language was to be Forth
- the operating environment would be a multitasking system based on a desktop paradigm, using menus and a pointing device, and supporting a modeless method of operation
- the test language would be English like in both vocabulary and syntax and would be hardware oriented and user extendible
- the software would support a library of instrument drivers, to allow easy and flexible configuration
- the architecture would be open-ended and general-purpose, and would support the IEEE 488 bus
- the system would be housed in an instrument desk, with the computer free-standing on it, and the

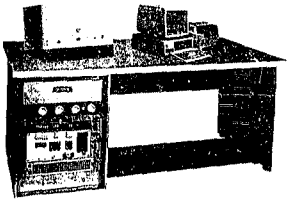
instruments housed in the desk pedestals and in a rotatable console.



A Console



A 19" Rack System



A Desk

Figure 3.5: ATT Packaging Methods

3.6 Feasibility Study

The specification described in the previous section called for a number of technical obstacles to be overcome. The most obvious of these was the English-like test language. In addition, the FORTH that was available on the Apple did not support multitasking, and had to be modified to do so. The operating system specification called for a relatively sophisticated display involving movable overlapping windows and menus. Whether these could be generated and changed sufficiently quickly was not certain.

A set of feasability studies was undertaken to see if these requirements could be met within the other system constraints. The approach taken was to create disposable prototypes, rather than to generate pieces of software that could be used later. This approach generally results in the feasability of a concept being demonstrated in the most efficient fashion. The one exception to this rule was the multitasking FORTH system. This was completely designed and implemented during this phase. The reason for this was that MultiForth (as the system was named) could be considered a product on its own, and is now in use as a general purpose software system.

The question of the English-like test language was approached from a theoretical point of view first. This resulted in the concept of context dependancy. It is

reasonably obvious that the words in an ordinary English sentence are not independent in their meaning from the other words in the sentence, ie they are context dependant. Thus the test language would need to allow context dependant meanings to the words in a sentence. Further analysis showed that the simple imperative sentence was the only type of sentence required. Finally it was shown that a compiler could be designed that would parse such a sentence, and that it could be a single pass, single direction compiler. A simple prototype of such a compiler was written and tested. The practical implementation turned out to be straightforward.

As has been mentioned, the multitasking software was generated as a complete language system, termed MultiForth. The source code for a normal single-tasking Forth system was purchased in machine readable form (from Forthwith Computers), and modified. Since a Forth system is usually written in Forth, this modification could be done in the same way as any other Forth programming task.

During this period too, the ergonomics of the desktop paradigm were investigated, with particular attention being paid to the choice of pointing device. The desktop paradigm presents an imaginary desktop to the user. The screen represents the field of view of the user, who is apparently sitting at the desk. There are pieces of paper on the desk, some of which are menus, and some of

which are pages used to write on. The function of the pointing device is to allow the user to select items on the desktop by pointing to them. In this way it is possible to select items from a menu, select words or letters on a page, or select an object (such as a page) on the desktop and move it around.

No final decision was made at this stage, but it was decided to use a joystick as the initial pointing device, and experiment with other devices when the operating system was working.

The "pieces of paper" on the desk required software to generate overlapping "windows" on the screen. The feasability of doing this was investigated by writing a piece of software that could generate two windows that overlapped. Various methods of generating the display were investigated, the major problem being in the methods used to determine which parts of the display were hidden at any one time. A simple system was chosen whereby the windows were kept in a queue and displayed sequentially, in a similar manner to a person putting several pieces of paper onto a desk. If a piece of paper overlaps another, the portion overlapped is hidden. If the data is written to the display in the correct sequence, this happens automatically, the topmost "piece of paper" being displayed last, and thus over-writing any data that is logically obscured. This approach uses more memory than one in which the hidden portions are determined before display, but turned out to be far more

acceptable in terms of speed. The simple correspondence between the logical appearance of the display and its actual generation also made for a more reliable and maintainable implementation.

3.7 Revised Specification and System Implementation

Once the feasability of the system had been proved, the specification was finalised and the system implemented.

An object oriented design philosophy was used for the system design, separating the objects in the system and the actions that would be performed on the objects. These actions were then allocated to tasks (since many of the actions would occur logically independently from the others). A complete top-down design and coding exercise was then undertaken. The system was compiled bottom up, one routine at a time. Since each routine had a well defined specification, it could be debugged completely before it was used by a higher level routine. Everything was written in high level Forth at this stage.

The working operating system was termed DeskForth, and at this stage could be considered as a general purpose language system. The facilities specific to the ATE were separated completely from the implementation of the operating system. This allowed the operating system to be debugged in a simple and straightforward fashion. This is taken further in chapter 4.

Once the operating system was working, the facilities required for a working ATE were written and incorporated in two tasks, one concerned with writing tests, and the other concerned with running them. A third task was also created which provides the operator with a continuous system status reporting function. These facilities are described in chapters 5 and 6.

Finally, the test language was implemented. The test language must by its nature be tailored to suit a particular UUT, and so a target application was chosen at this time. Suitable instruments were selected and the requisite software was written. These were then tried out by the engineers involved in actually developing the test equipment for that UUT. Their feedback was used to implement system improvements, and to evaluate both the system specification and its implementation.

3.8 Choice of a Target System

The choice of a suitable UUT was governed by two considerations. Firstly the UUT had to be available. Secondly it should exercise those aspects of the design that had been identified as [illegible]tial, viz.;

- The UUT chosen should in [illegible] measuring more than simple electronic characteristics.
- The parameters to be measured should be relatively few, but of a specialised nature.

- Control of a mechanical jig should be involved.
- A limited number of devices should be integrated using the IEEE 488 bus.
- The UUT should still be in development, since the system was designed to cope with a changing requirement.

A UUT meeting the above criteria was identified. It was an electro-optical device requiring measurement of such things as polar sensitivity to infra-red radiation. However, the UUT chosen was not yet at a suitably advanced stage of development, and this led to inevitable delays in this investigation. The extra time available was utilised in an investigation into the ergonomics of the desktop operating system in keyboard intensive applications, such as word-processing. These are described in chapter 4.

3.9 Application to the Target System

The process of applying the system to testing the target UUT proceeded as follows. An Acceptance Test Instruction was written which specified in detail what tests needed to be performed on the UUT. From this document the hardware requirements were deduced. The ATE and associated mechanical jigs were purchased, built and tested. The switching matrix was specified and wired. Software drivers for the various parts of the ATE were written and tested. These included drivers for the

switching matrix used, a digital voltmeter, a two pen plotter, and stepper motor drive units. In addition special requirements regarding the format of the result sheets were addressed. The software was integrated with the ATE. During this period the Acceptance Test Instruction was translated into test software using the writing aid. However, as the UUT was still under development, the test instruction was subject to continuous change. Rather than continuously rewrite the test software, it was decided to leave this task until as late as possible. This procedure was made possible by the writing aid, which enabled the tests to be written at short notice. Thus the development process could be separated into hardware development, ATE software development, and finally test implementation. The feedback obtained from the system users during this phase was used to improve the system. No fundamental changes were made to the original concepts, however. This phase is described in chapter 7.

4 OPERATING ENVIRONMENT

The operating environment is a software generated interface between the user and the hardware. A distinction must be drawn between this concept and the concept of a classical operating system. The classical operating system is a set of tools that allow the user to access the various system utilities, such as the disk drives, printer, and so on. The operating environment described here provides a unified approach to editing software, running software, and using peripherals.

4.1 Functions of an Operating Environment

This section describes the primary functions of a classical operating system, together with those functions and characteristics that were found to be desirable in integrating them into an operating environment

The operating environment must provide the following basic functions to the ATE user

- The ability to edit text to form test programs and procedures
- the ability to compile test programs from such text
- The ability to run a test program
- The ability to write and compile software other than test programs, such as system level software and instrument drivers

- The ability to store tests, result sheets, and other data in mass storage
- The ability to print result sheets, source listings, and so on
- The ability to extend the operating environment to encompass other peripherals and instruments as required
- The ability to partition instrument drivers and other utilities as seperately compiled modules in a library

In order to address the needs of all the users of the operating environment effectively, several characteristics were found to be desirable. These can be categorised as those characteristics leading to a requirement for multitasking, those characteristics leading to a requirement for a sophisticated menu syster, and those characteristics which were satisfied by using a desktop paradigm. An overall characteristic was also identified that had to do with the very fact that the system would have many users, rather than the individual requirements of any one class of user. This characteristic led to the basic requirement that the operating environment should accommodate both the sophisticated and the novice user. Each of these characteristics is discussed in detail in the following sections.

4.2 Multitasking

The requirement for multitasking arose from the following requirements

- Status information should be available to the operator while a test is running. This can be provided explicitly by the test writer in the form of messages, or it could be made available by means of some sort of interupt driven routine, perhaps generated from the keyboard. It is attractive to divorce the basic status generation facility from the test writing function for reasons of consistency, and also to reduce the burden on the test writer. This implies that the status be generated by the operating system rather than the test program, and implies at least a limited degree of multitasking capability.

- Printing result sheets is time consuming due to the slowness of typical printers. The testing time should not be limited by the printer speed. This implies that the result sheets should either be saved in mass storage and printed at a later stage, or that a multi-tasking system be used that would allow testing and printing to proceed in parallel. It is often advantageous to be able to see results as they are generated, rather than after a long test has been completed, especially if a particular test result may require some intervention by the

operator. The use of a multitasker would greatly increase the overall system efficiency, since in a typical test the computer spends a significant amount of time waiting for an instrument or for the UUT. This time can be used to print results.

- The operator will often have a choice of several modes of ATE operation that he can adopt under different circumstances. For instance, he may choose to abort a test if any result is outside limits, or alternatively he may wish to get a printout of such results. It is attractive to be able to change these options during a test. For example the operator may choose to allow a complete test sequence to run to completion before requesting a printout. While a particular test is running however, he may notice that the UUT is not behaving quite as it usually does in this particular test. At this point he could abort operation by hitting some sort of panic button. A more attractive option is to request a printout of any failed results. If it appears that the UUT is indeed failing many tests, he could then abort testing. Alternatively, if the UUT has not yet failed a test, he could request that in the event that it does fail, the system should then stop. This once again implies at the very least a live keyboard facility during test running.

- The complexity of the pro_ ed operating system made multitasking an attractive tool in the

implementation of the operating system itself. Many functions of an operating system are sequence independent from each other, ie they do not have to occur either before or after the other functions in the system. It is an attractive technique to allocate these functions to separate tasks in a multitasking environment. This allows the physical structure of the software to be implemented as a mirror image of the logical structure. This has several advantages.

- In general, the system was required to be mode free in its operation. This concept is discussed later, but in essence it means that all the system facilities should be available to the user at all times. Allocating these facilities to several tasks which can run independently of each other greatly simplifies the implementation of this concept.

As has been described, the requirement for multitasking was satisfied by means of a modification to the standard Forth model. These modifications are described in the form of a supplement to the installation manual, which is included as Appendix B. A closer look at the development of MultiForth, with greater emphasis on the software aspects is provided in Appendix C.

4.3 Menus

The requirement that the novice should be able to use

the system implies some sort of facility to assist him in doing so. In addition, the requirement for a software writing aid (discussed in chapter 6) implies the same thing. The method chosen was the use of menus. There are several alternatives and these are discussed in detail in chapter 6.

The primary requirements of the menu system were identified as the following

- They should be unenforced. This means that the operator should be able to bypass the menus when it is expedient for him to do so.

- The menus must all be a e at once. Nonetheless, a hierarchy of menus will exist, and the operator must be able to choose to follow it.

- There will be a large number of menus (say fifty). Some method of arranging them so that they are all available, but nonetheless accessible must be devised.

- Since there is a large number of menus, a suitable method of selecting menu items must be chosen. Choosing the first letter of the selected menu item would be ambiguous; the other common method, moving a cursor to point at the menu item is therefore indicated. This has implications on the method used to move the cursor around. It also implies that rearranging the menus to suit the user's current

the system implies some sort of facility to assist him in doing so. In addition, the requirement for a software writing aid (discussed in chapter 6) implies the same thing. The method chosen was the use of menus. There are several alternatives and these are discussed in detail in chapter 6.

The primary requirements of the menu system were identified as the following

- They should be unenforced. This means that the operator should be able to bypass the menus when it is expedient for him to do so.

- The menus must all be available at once. Nonetheless, a hierarchy of menus will exist, and the operator must be able to choose to follow it.

- There will be a large number of menus (say fifty). Some method of arranging them so that they are all available, but nonetheless accessible must be devised.

- Since there is a large number of menus, a suitable method of selecting menu items must be chosen. Choosing the first letter of the selected menu item would be ambiguous; the other common method, moving a cursor to point at the menu item is therefore indicated. This has implications on the method used to move the cursor around. It also implies that rearranging the menus to suit the user's current

needs should be possible.

- The menus must behave in a consistent fashion. This means that the *method used for selecting items* in the menu should always be the same. The method used for selecting data for the selected command to act upon *should always* be the same. In addition the selection of a menu item should not cause side effects, such as the disappearance or appearance of *other menus*, unless that command is explicitly to cause that effect to occur.

- It should in general be possible to browse through the menus, trying them out to see what they do without disastrous consequences. This allows the novice user to become au fait with the system quickly and easily.

- The selection of a menu item must never be followed by a question of the "are you sure?" kind. Wherever possible it must be possible to undo the results of a particular selection through the selection of another menu item, either its complement, or an explicit undo command. Thus deleting a word in a piece of text can be undone by selecting the "undo" option. Similarly, a command to force all text to uppercase characters on the display can be reversed by selecting the complementary *command* to allow lower case letters. Some cases cannot be easily undone; for example, erasing a disk. Instead,

potentially hazardous situations must be avoided by enforcing a two step process, the first step of which is reversible. For example, erasing all data on a disk is preceded by marking the disk as no longer containing useful data. Only disks so marked can be erased. To be consistent, disks marked as "trash" cannot be used to store data. The step taken to mark a disk as "trash" is reversible. The first step (marking the disk as "trash") is intuitively obvious as a hazardous step, and forms a psychological barrier through which the user must proceed before doing any real damage. Selecting the "trash" command by accident is also catered for, since it can be undone.

- A particular result must be achieved by a minimum of menu selections. The impression of wading through a whole hierarchy of menus in order to get to a conceptually trivial end result must be avoided.

The desktop paradigm described in the next section assisted in satisfying those requirements. A more detailed look at the software supporting the menu system is provided in Appendix C.

4.4 The Desktop Paradigm

The idea of using common office objects to symbolise the interface between the user and the computer was developed primarily by researchers working at Rank Xerox

(Tesler, 1981). The idea was to replace the more threatening concepts and computer jargon with paradigms that the user is familiar with. Thus the editing of text is symbolised by typing on a piece of paper. The piece of paper can be placed in a file, as opposed to storing it on disk. The common editing functions are replaced by more familiar commands analagous to working with a real piece of paper. For example text can be moved from one place to another on the page by "cutting it out" and "pasting it in". This type of editor has become known as a "cut and paste" editor. In addition no formatting commands are required.....the text appears on the screen in its final format, a concept commonly known as "what you see is what you get" or WYSIWYG. The advantages of these editors have been described by many authors, including Tesler (1981).

A source of operator frustration that was identified was the perception that the computer is the boss and the operator the slave. This perception arises from the operating system presenting the operator with a limited set of options and demanding that he should choose one ("CHOOSE ONE OF THE ABOVE:"). This is particularly to be avoided in the case of the test equipment operator in production. He is required to spend a good deal of his time using the ATE. As time goes by he gets to know the system very well and to resent being "ordered around" by the computer.

The approach adopted here is to allow the operator to

tell the computer what to do if he knows what he wants it to do, while allowing him the security of being able to choose from a menu. Thus the operator is still presented with the options; the psychologically important difference is that he is no longer told to choose one.

This allows him, for instance, to change the order in which things are done (eg to enter the date second and his name first), or to omit meaningless steps (eg if his name hasn't changed between this UUT and the previous one). In effect the computer is no longer demanding action from the operator ("tell me your name"), but the operator is telling the computer something ("my name is.."). An argument raised against this is that the operator is free to enter false data if his responses are not rigidly controlled by the computer. This argument is considered fallacious, since the operator has little or no motive to deliberately falsify information. The chance that he will unwittingly make an error is reduced by presenting the data he has entered for his inspection , and by allowing him to change or correct it at any time. The onus is placed on the operator to do the right thing, and the system merely makes it easy for him to do this, through its consistent interface and ability to undo errors.

It is the task of the test operator's supervisor to ensure that he is conscientious and trustworthy, not the task of the computer. Acceptance of any type of

automation in the factory environment is greatly reduced if the human dignity and self esteem of the human workers is assaulted by the equipment.

A further concept that has been seen to be important is that of modality. It was shown by the researchers at Xerox PARC that modes, and especially exclusive modes should be avoided. This implies that performing one set of operations (such as editing) should not preclude one from performing another (such as printing or running a test). This concept, also known as "the dilemma of preemption" (Swinehart, quoted by Tesler, 1981: 90), is more fully explained in Appendix C. The implications of non-modality are the following

- All commands must be available at all times. This means that all menus must be available at all times.

- Functions that would preclude the use of another command due to the length of time that they take to complete must be performed by dedicated tasks that can run in parallel with any other function. Thus running a test is allocated to a different task from the one used to edit text or to print listings.

- Commands that produce an output on the screen must be usable in parallel with other commands that also produce an output to the screen. This implies that the screen must be divisible into partitions, and that it must be possible to specify that the output

of a particular task should go to a particular partition.

Each partition on the screen is readily transformed into a "piece of paper". Thus there may be several pieces of paper lying in the screen area, which can readily be conceived of as the top of a desk. The large number of menus can also be seen as smaller pieces of paper, also lying on the desk. The screen area is not large enough to contain all these pieces of paper and so the desktop can be made much larger than the screen. Selecting commands from various menus causes separate tasks to perform the commands, "behind the scenes", and the desired result to appear on the logically correct piece of paper.

The screen itself is a window onto the desktop, and corresponds in the analogy to the field of view of the person sitting at the desk. In order that this analogy should work, it is necessary that the screen should move around the desk freely as the user turns his attention to various parts of the desktop. This places great demands on the pointing device used; this is discussed more fully later.

The question as to how the pieces of paper should appear on the screen permits of two basic solutions. Either the papers are allowed to overlap each other, or they may present a "tiled" appearance. The analogy suggests that overlapping the papers should be allowed. In addition,

more papers can be put onto a given desktop using this solution. The major disadvantage is that information can be hidden at the bottom of a pile of papers and that the desktop can become untidy. The tiled solution is also the easier solution to implement. In the end it was decided to use the overlapping solution, for the two reasons given, ie the greater correspondence with the analogy, and the larger amount of information that can be made available using this method. The disadvantages were addressed by making it easy to move the papers around the desktop; using the pointing device, one can "pick up" a piece of paper and bring it to the top of a pile or move it to another place on the desk. Thus it is possible to organise the desktop into manageable groups (or piles) of papers and yet have all the information readily available if needed. It is also possible to remove pieces of paper from the desk entirely, and return them when needed, by means of simple menu selections. The "tiled" solution was found to be clumsy by comparison. If there is a large number of papers on the desktop, moving them around without overlapping them resembles a Chinese puzzle.

The means of selecting items from a menu, text on a page or a piece of paper to be moved was the subject of experiment. The pointing device used to select "objects" on the desktop would also define what portion of the desktop was in the field of view of the user, ie which portion of the desktop was on the computer screen. Thus

it was important that the device chosen should allow an intuitive transfer of the operators attention from place to place on the desktop. At first the system was implemented using a joystick to drive a cursor around the screen, but this was found to be unsatisfactory. Various methods were tried, and finally the mouse was selected. These experiments are described in chapter 9.

In summary, the operating environment is in the form of a desktop paradigm. The operator uses a mouse to point at objects on the desk to select them for use. For example he can point at a page, select it by pressing the button on the mouse and then move the page around the desk by moving the mouse. Commands are selected by pointing at a menu item written on a piece of paper and pushing the select button. To delete text from a page, the text to be deleted is selected (by pointing at it and pressing the button), and then "delete" is selected. If the result is not satisfactory, selecting "undo" restores the deleted text. See figure 4.4.

The other functions common to operating systems, such as file handling, disking, etc. are provided in menus as well. The various commands available in the desktop environment are described in Appendix D.

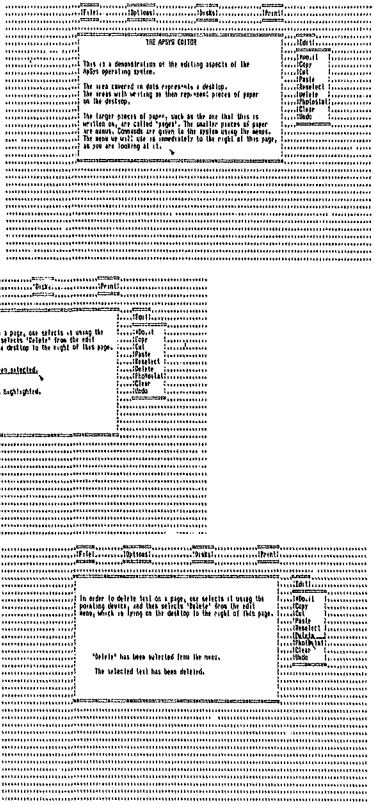


Figure 4.4: Using the Editor

4.5 Accommodating the Specialist and the Non-specialist

It was important that the tools provided to allow the novice user to use the system easily should not hamper the activities of the specialist. Thus the tools provided for editing text, saving it on disk, compiling the resultant software and debugging it were designed to allow the specialist to use them effectively. The editing features are powerful enough to allow Forth software to be written easily. The compiling commands will compile Forth source code as readily as code written in the test language. In addition, access to the normal interactive Forth system is facilitated; any Forth command can be typed onto one of the sheets of paper and executed by selecting the command "do.it" from a menu. In fact, all the software written for the system at levels higher than the basic operating system was written under the desktop environment. The system was configured as a Forth system using the desktop paradigm but without any features specific to ATE. This system was dubbed DeskForth. DeskForth is a complete software development system, with certain definite advantages over traditional systems. Many of the tools created to assist the novice user remain useful to the specialist. In particular the non-directive nature of the operating system, its multitasking features, and its lack of exclusive modes are appreciated at least as much by the specialist as by the novice.

5 ENGLISH-LIKE SOFTWARE

5.1 Requirement for English-Like Software

As has been seen (in Chapter 2), the majority of the users of an ATE system are specialists in a field other than software. In addition, it has been shown that the majority of these users are primarily interested in reading rather than writing software. The requirement for a test programming language that is readable by non-specialists is therefore apparent.

It must be borne in mind that the users of ATE are neither illiterate nor technically incompetent. Thus the language used to write the tests does not have to be "simple" in the sense that it should be read and understood by a non-technical audience. The user needs to be able to understand, in technical detail, what it is that the test program represents. This chapter will describe a method of meeting this requirement.

It is clear that the test software will need to be English-like in order to be understood by those who do not understand computer languages. It is possible at this stage to specify somewhat more exactly the "understandability" requirement of the language: the test programs should be understandable by anyone who can read and understand a laid down acceptance test instruction. However, it is necessary to address the

requirements of the more important potential readers of the software more clearly before a more exact specification can be achieved.

The UUT designer needs to be sure that the test as represented by the software accurately represents his intentions when he specified that that test be done. Thus although his requirement is oriented to the UUT, he needs to know the details of how the ATE is interacting with the UUT. Thus reading the software must lead him to the ATE hardware, since it is the effect of the ATE hardware on the UUT which is of interest to him.

The Quality Assurance inspectors have a requirement similar to that of the UUT designer. However, their need for detailed knowledge is even more exacting, since they must question the accuracy of the system, and investigate possible sources of error. What they might typically do is examine the tolerance statistics of a particular production process, their aim being to ensure that the process is under control, and that the products being delivered are of acceptable quality (ie that the customer's risk is as contracted and that the producer's risk is as low as possible). A major input into this equation, especially when the process yields products that have a large statistical probability of being near or outside the acceptable limits in some important characteristic, is the accuracy of the acceptance test apparatus.

The required accuracy of any ATE is inseperably linked to the statistical properties of the process that is producing the UUT. Thus changes in the process may set more or less stringent requirements on the ATE. The production engineer interested in finding out these statistics will want to know more than that a voltage measurement is taken. He will need to know the accuracy of that measurement.

That particular measurement may have been analysed and documented as sufficiently accurate by the ATE engineer; however, this will have been done in terms of the ATE engineer's perception of the production process statistics. Since this was in all probability done before production of the UUT started, the chances are good that some degree of error is involved. In addition, the production engineer may want to use the ATE to actually generate the production statistics. His accuracy requirements may be quite different in this case.

In short, production personnel will often want to know exactly what instrument is being used, on what range, and on what function, and will generally want to know the capabilities of that instrument beyond the simple accuracy specification documented in the acceptance test instruction of the UUT.

The maintenance engineer will need to know precisely what the ATE was supposed to be doing when it failed. He

will need to know which relays were closed, what instruments were in use, how the power supplies were being used, and so on. As before, we see that is insufficient for him to know that a voltage measurement was being taken; he must know exactly what ATE resources were being used at the time.

The previous discussion leads to the conclusion that the readers of the test program need to know the details of how the ATE resources are being used, and will often want to use this information to address more general problems than the specific test they are reading. This can be stated in another way: they need information relating to the ATE and UUT hardware, rather than abstract concepts such as "voltage". They do not want to know that the UUT's voltage is being measured; they want to know that the digital voltmeter is being used to measure the voltage at some particular point on the UUT, via some particular set of relays in the switching matrix. They will also need to know exactly which digital voltmeter is being used (many ATEs have more than one means of measuring a particular parameter), and may want to refer to the user manual of that instrument for information that would not normally be given in the test software.

5.2 Other Solutions

The single most common language in use in in-house

developed ATE's is BASIC. This and other programming languages, such as Pascal and Fortran, do not meet the requirement of being readable by any person who can read and understand a laid down acceptance test instruction. However, there does exist a test programming language which is reasonably English like in vocabulary, if not in syntax. This language, ATLAS (IEEE-STD-416, 1981), deserves attention if only because it is a standard. It does not wholly meet the requirement of being readable by a non-specialist, but a case could be made for all users of ATE to learn the language, since it is a standard.

ATLAS is so enormous that an approved subset, C/ATLAS, is available. Even this is very large. The stated goal of this language is (IEEE-STD 716, 1982: 2-1)

> "to define a high order language....which is to be used for the writing of test programs for Units Under Test (UUTs), so that these programs can operate on various makes and models of Automatic Test Equipments (ATE)."

The goals, and indeed the actuality, of the ATLAS language are thus incompatible with the requirements developed in this study. Two important aspects will be dealt with here; these are that ATLAS is writer rather than reader oriented, and it's ATE independent nature.

These aspects are dealt with apart from the practical fact that very few ATE manufacturers support ATLAS in

its standard form. The concept of a standard language in the form proposed by the IEEE appears to be impossible to implement satisfactorily; there are simply too many types of parameters, UUTs and measuring instruments. Each implementation will therefore be different, in order to accommodate the special parameters etc. The language definition is thus open ended, and thus not truly *standard*. *It is also impossible to expect that all* users of ATE become familiar with such a large test language.

The writer orientation of ATLAS can be derived from the statement of goals quoted earlier. Further evidence can be *derived from the following example.* ATLAS defines single and multiple action verbs. A typical single action verb would cause a particular relay to close in order to connect a point on the UUT to a resource of the ATE. As was shown in the previous section, this is the level of detail required by the users of the ATE. The standard however discourages the use of these single-action verbs, preferring the use of multiple action verbs such as MEASURE. This verb will typically close the relevant relays, setup the relevant instruments, take the required readings, and return, *say, a voltage.* The *standard* has this to say on the subject (1982:11-4):

> "The (single action) verbs were incorporated in the language primarily as a means of properly defining multiple-action verbs. These multiple action verbs

will be used for most test statements because they are easier to write....."

The second objection to ATLAS is it's ATE independent nature. In practice, this means that all measurements are made in terms of abstracts such as "voltage", and all reference to the actual ATE hardware is deliberately hidden. The disadvantages of this approach have already been discussed. However, the reason behind this concept is also open to question.

The idea is to make the test programs themselves portable. A typical sub-assembly has an acceptance test instruction typed on from three to ten pages. The number of test results may vary from one to at most a hundred. The effort involved in typing in a test suite for such a UUT is roughly equivalent to that involved in typing the original test instruction, if one postulates a special purpose test language rather than, for example, BASIC. This clearly represents a very small portion of the cost of developing ATE. (This statement is not true for systems where the tests are written in a general purpose language such as BASIC or FORTRAN).

By far the greatest investment in software writing is in the operating system and compiler to support the ATE and the test language. Thus it may be more economical to write a specific compiler for a particular ATE, rather than to attempt to implement a standard ATLAS subset on that ATE. Thus, at least in the case of sub-assemblies,

there is no justification for using ATLAS for its portability.

A case might still be made for ATLAS if a particular ATE were to be used for a very large number of different UUT's. The cost involved in implementing an ATLAS compiler to support this ATE could then be spread over all the UUT's. However, we have already seen that such general purpose ATEs are uneconomical when used for sub-assembly testing. In addition there is no reason why the ATLAS compiler should be cheaper to implement than any other, and so this argument could be used to justify any compiler.

From the foregoing, it is clear by using a special to purpose test language it should be possible to reduce the cost of actually writing test software to a negligible amount; the portability of test programs is not therefore a primary requirement.

5.3 English-like Vocabulary and Syntax

A common argument for the use of ATLAS is that the meaning of any ATLAS statement is "precise". What is actually meant by this is that any ATLAS statement has a meaning which is precisely defined. A meaning which is precisely understood is another thing entirely.

The definitions of meaning that any programming language requires must of necessity be given at some point in

English or some other natural language. That English is capable of being precisely understood is open to argument. It is however clear that it is the closest thing to a universally and precisely understood language that exists among English speaking people. The precisely defined ATLAS statement is only understood as far as its definition (in English) is understood. Thus it can be postulated that the ideal language for our purposes is English itself, if precision of meaning and understandability are the criteria.

Efficiency is another criterion that needs to be addressed. A very precise meaning might only be obtainable from the use of English after a lengthy discourse on each function. This could conceivably preclude the use of English on the grounds that it is grossly inefficient. This argument is, however, not very compelling since the test instruction, on which the test program is based, is usually written concisely and understandably in English.

The meaning of an English sentence derives from three sources; the words in the sentence, their ordering (syntax), and the context of the sentence. It is the contextual information that allows English to be concise even when used in a test instruction. It is also possible to more clearly define the meaning of the words in the sentence by means of a special purpose dictionary (this can be seen as adding to the contextual basis of the sentence).

Thus it is established that a test program could be written using English words with their meaning precisely defined and understandable in the context in which the test program is used.

The final contributor to meaning is syntax. Even those programming languages that do attempt to use a very English-like vocabulary tend to enforce a rigid but unnatural syntax. It is postulated that in order to make the sentence easy to understand, an English-like syntax should be used. This, when combined with an English vocabulary, should permit writing a test program that is as understandable as is possible.

This extends to words that are commonly called "noise words" (originally from COBOL)...the, a, to, from, etc. The statement "connect DVM" is understandable. However, it can be contrasted with the statements "connect the DVM", and "connect a DVM". A system with several DVMs, and with the capability of dynamically allocating them to various tasks, would need the precision that can be achieved by the more complete syntax of the latter two statements.

It seems sensible therefore to require the test language to be English-like both in vocabulary and in syntax.

5.4 Context Sensitivity

As has been discussed in the previous section, English

uses context to define the meaning of words precisely. This context dependancy is not confined to the context of a sentence as a whole. Contextual information can be contained within the sentence itself....."kick the ball" vs "just for kicks" or "kick the bucket". This at first seems to be a serious practical problem. Is it possible to define a compiler that will parse correctly a statement containing context dependant words? This problem was the subject of a program of experiments in itself.

An empirical approach was used; a large number of typical test programs was analysed and the following conclusions drawn.

- the simple imperative sentence is the only sentence type required for this application
- the context of any word in the sentence can be satisfactorily determined from the words preceding it
- a clear indication of sentence termination is necessary in order that the first word of the next sentence should not be interpreted within the context of the previous sentence

compiler can be constructed that will parse a sentence conforming to the above restrictions. These restrictions are not NECESSARY conditions for a compiler to be possible. They do however greatly ease the task of

writing the compiler. In particular the compiler need only parse the sentence once, from left to right. Further details on the compiler can be found in Appendix C.

5.5 Application to the ATE Environment

Each sentence in the language is a simple imperative that starts with a verb such as SETUP, CONNECT, or MOVE. This word defines the overall context of the sentence and is referred to as defining the CLASS of the sentence. These classes are very similar to the STATEMENT TYPES found in ATLAS, and generally use the same word as the equivalent ATLAS statement. A typical sentence would be

CONNECT THE DVM TO THE AMPLIFIER INPUT :-

CONNECT sets the overall class of the sentence. AMPLIFIER sets the context for INPUT within the overall context of CONNECT. The result is that the compiler recognises a UUT point that is multiplexed. The relevant multiplexer is defined by the word DVM. The following variation is allowed:

CONNECT THE AMPLIFIER INPUT TO THE DVM :-

Note that this flexibility is only of service to the writer of a test.

This sentence tells the reader exactly what instrument

is being used, and what point on the UUT is being connected to the instrument. However, it is not possible to tell directly from the sentence exactly which relay in the switching matrix is to close to make the connection. This could be achieved in several ways. The use of UUT and instrument names could be avoided:

CLOSE MULTIPLEXER 5-1 :-

This however errs on the other side. The information as to what this multiplexer achieves is hidden. A comment could be added:

CONNECT THE DVM TO THE AMPLIFIER INPUT (MUX 5-1) :-

This solution is acceptable when the switching is simple. The comments can be inserted automatically by the writing aid described in the next chapter.

However, the best solution in practice turns out to be a drawing. The drawing shows the UUT points by name, and by connector number, identifies the looming involved, the multiplexer used, and the instrument. Thus more information is presented in a clearer manner in the drawing than could be easily achieved in the test program.

The drawing also contains information that logically does not belong in the test program, such as the colour of the relevant wire in the loom. The important point is that the test program allows the reader to use the drawing effectively by providing him with the correct

references. Armed with the drawing and the test program listing the maintenance engineer can find any fault.

Other examples of test language syntax can be found in Appendix E, which contains a step by step example of the generation of a test program.

5.6 Problems Inherent in Natural Language Programming

Programming in a natural language such as English poses problems for the writer. Among the problems are the large vocabulary, the loose syntax of the sentence, and the sometimes lengthy constructions that are used for clarity. Thus test programs written in a natural language can be tedious to write. This problem is addressed in the next chapter.

6 THE SOFTWARE WRITING AID

6.1 Need for a Writing Aid

As has been discussed in the previous chapter, a natural language program can be tedious and difficult to write. This is not because the overall program length is great, but because the individual words and constructs are longer than is usual in a programming language. As has been shown in the previous chapter, a special purpose programming language is likely to be much shorter than the equivalent program written in, for example, BASIC. However the length of the individual words used tends to irritate the writer, and lead to a greater frequency of typing errors. For example, the BASIC construct

IF A>B GOTO 10

can be contrasted with

DO THE FOLLOWING IF THE VALUE OF A IS GREATER THAN THE VALUE OF B :-

This problem is compounded when one considers that the likely writers of test programs are relatively unskilled as software writers. It is one of the requirements of the system that it be usable by non-specialists. Thus an aid to writing the software is indicated.

In any software development cycle, significantly more time is spent in debugging the software than in writing

it. Errors can be broadly divided into two areas, namely conceptual errors (the wrong program was written) and coding errors (the program was written incorrectly).

The very fact that the software is written in English can be expected to reduce the number of conceptual errors, since the software will mirror the acceptance test instruction very closely. Finding and correcting conceptual errors will also be facilitated, since the UUT designer will be able to read the software and identify problems. However the number of coding errors can be expected to increase.

Bailey (1984) has stated that 80% of coding errors are procedural (use of the wrong code, or code in the wrong place) and 20 % are clerical in nature (spelling, typing errors). Use of a writing aid would have to address both these problems.

6.2 Methods of Providing Aid to the Software Writer

Other languages with long words have traditionally provided abbreviations for use by the writer. The best known examples of this are COBOL and ATLAS. This relieves the todium of writing software, at least for the specialist. It does nothing, however, to make it *easier for the non specialist* to write software. If anything, it makes his job harder by extending an already large vocabulary. Typing errors might be reduced fractionally (since less typing is required) but the

procedural errors would be unaffected.

Other methods of aiding the writer of software can be conveniently divided into the "question and answer" method, the menu method, and the "fill in the form" method. The question and answer method involves asking the user questions, usually requiring a yes/no response, (do you want to use the DVM? (Y/N)) followed by a further question (do you want to use the DC Volts function?), and so on. This is very tedious, and has been largely superseded by the menu approach.

Menus are widely used to provide guidance to the users of application programs. A list of options is usually provided, the user being required to choose one. There are two common sources of frustration in this scheme. The first is caused by clumsy or inconsistent methods of selection. The second occurs when the user does not want any of the options.

Typical selection methods require the user to type the first letter of a menu item, type a number associated with the menu item, or move a cursor to the menu item and hit a key (often the space bar or carriage return). Each of these methods has its problems.

Selecting the first letter of the menu item is usually the fastest once the user knows the system well, and has more mnemonic value than selecting a number. The disadvantage is that menu items are limited in number and variety since their first letters must be unique.

This is often obviated by organising the menus into hierarchies. This has the disadvantage that several menu selections may be required to obtain one conceptually simple result. Some systems require the user to first type the letter and then hit another key, such as carriage return, before the command is selected. This is safer than acting upon the command immediately the letter is pressed. However, some users find this double action irritating, especially when a deeply nested hierarchy of menus must be negotiated before the final selection is made. In addition, the hierarchical structure tends to hide information from the user. Thus it is necessary that he be able to browse through the menus without initiating any potentially dangerous actions.

Moving the cursor to the menu item and then selecting it has certain advantages. There is no requirement for the first letter of each menu item to be unique and so the nesting of menus can be greatly reduced, with a resulting speed increase. However, moving a cursor to a menu item is a visual activity, and thus slower for the expert user, who is able to remember the sequence of commands required to obtain a desired result and will type them without reference to the menus.

Using cursor control keys to move the cursor is also extremely laborious if the menus are large. This is due to the slowness of movement of the cursor, which is limited by the keyboard repetition rate.

In order to reduce frustration, the number of menu selections must be reduced by ensuring that each selection is as powerful as possible. A single conceptual command should require only one menu selection.

The third system in common use is the forms based system. This presents a standard form or set of forms and requires the user to fill in the blanks. This type of system is commonly used in data base systems, but is also used in certain ATE applications, often in combination with a menu based system. In order for instance to setup the parameters of a certain instrument, one would call up the form for that instrument and fill in the options required.

A problem that occurs with this type of system is that reading the resulting software can be laborious. Typically, each instrument setup is contained in a form that takes up a page. Seeing the program flow is difficult. This problem is usually got around in one of two ways. Overviews can be created which show less detail (for example simply the form headings) or the operating system can use the forms to generate a software listing which can be read more easily.

Forms based systems generally require more typing than menu based systems. The amount of typing can be reduced by presenting options to select at each space, and then this is effectively a menu based system.

The forms based system is conceptually "step structured". The most common form of control structure in a forms based system is the form "GOTO STEP 20". Menu based systems tend to allow more powerful controls, mainly due to the broader picture which is available in a menu based system. The "IF...ELSE...ENDIF" type of structure is difficult to use when only one step is visible at a time.

Several software generation packages were investigated., particularly "The Last One" from D.J. "AI" Systems Ltd, and "Quic-N-Easy" from Standard Microsystems limited.

"The Last One" is a menu based system that displays many of the faults mentioned earlier. It is slow and clumsy, and has a tortuous menu structure that makes it very difficult to change from one activity to another.

Quic-N-Easy on the other hand is a "super high level" language. It does not relieve the user from having to learn a language; its main achievement is that it is specifically tailored to a particular type of program (in this case data base management), and so is more effective than general purpose languages such as BASIC or Pascal.

Johnson (1982) gives a description of various software generation packages produced by thirty different software houses. Most of these use menus, some in combination with form filling. The software produced by

these systems varies from completely unreadable (ie maintenance requires rewriting portions of the software), to almost readable (English like vocabulary but application peculiar syntax). One of these systems, "Escape" from Software Management Systems Inc., features a language that has a user extendible vocabulary.

A feature of all the packages investigated was that they were targeted to specific applications, such as data base management, report generation, and so on. This allows automation to be simplified since the number of options is reduced compared to a general purpose system, and the options that are presented can be made more precise. An ATE application will benefit from this effect as well. However, none of the program generators investigated generated a program that was English like in both vocabulary and syntax.

This English syntax of the host language greatly simplifies the writing aid, since the user of a writing aid that is interactively producing an English output needs far less information from the writing aid as to what he should do next. The user does not need to learn a new programming language; the writing aid, on the other hand, can assume that the user is familiar with the programming language.

The lessons learned from the investigation can be summarised as follows

- complex and deeply nested menu hierarchies are

undesireable

- users need to be able to browse through the system to find out what it does

- the method of selection from the menus needs careful attention if one is to avoid annoying experienced users while at the same time protecting the novice

- the result of any menu selection should be obvious; this is greatly aided if the test language is English-like both in vocabulary and in syntax

- there are times when the use of menus is *inefficient...the user must* be able to use the menus or ignore them as he pleases .

6.4 Effects of the Structure of the Test Language

The structure of the test language lends itself to the use of a writing aid. This is due to the left to right context refining process that each sentence uses. First, all the classes of sentence are displayed as a menu. Once a class has been selected all the options within that class are displayed, and so on; the options displayed at any time are dictated by the context that has been created by previous selections. Thus the structure of the menu hierarchy mirrors the structure of the contextual hierarchy of the test language.

This logical parallel is of value to the system

configurer. Since the writing aid can easily follow the same path as the compiler, this naturally assists in reducing or eliminating procedural errors. The problems of hierarchical menu structures that were highlighted in the previous section are eliminated since:

- each selection achieves something concrete, rather than simply leading to another menu
- the hierarchy is not enforced; the user can bypass it of he wishes.

6.4 Using the Writing Aid

In order to use the writing aid the user selects the WRITE menu. All the menus used while writing will then appear on the desktop, piled on top of each other. The top menu contains the primary program structures. The user will select BEGIN.A.TEST, and the writing aid will write on the page:

Begin Test ?

The question mark indicates that the name of the test must be typed in by the user. This is done by the normal editing functions of the desktop environment. In this case, the user would simply point at the question mark using the mouse and type in the test name. It is important to emphasise that the writing aid does no more than put text on a page. The text on the page can be edited by the user as he would any text produced by any

means.

Pressing the cueing key will now take the user to the next menu, which displays all the classes of test operation available. He could for example select CONNECT. The cueing key will now take him to a menu containing the various instruments. He could then select DVM (Digital VoltMeter). The cueing key would then take him to a menu of all the things that the DVM can be connected to. The final sentence might read

Connect the DVM to the Amplifier Output :-

A complete program can be written in this way. Procedural errors are eliminated by the natural progression of the menus. Clerical errors are eliminated since virtually no typing is required.

A complete example of how a test is written is contained in Appendix E.

7 AN EXPERIMENTAL SYSTEM

7.1 Initial experiments

The initial experiments required only a computer. They were primarily aimed at verifying the software theories that had been developed. Thus prototypes were developed to ensure that the operating system, menus, windowing, etc. were practical and useful. The concept of a test language that is English like in both vocabulary and syntax was tested, and it was proven that a compiler could be designed that would implement the language.

The first hardware experimentation involved the interface to the IEEE 488 bus. At first the interface software was designed in the traditional fashion. This interface was used in the early stages of the project. Later, a HP 7470A plotter was interfaced, and this led to a revision of the IEEE 488 bus driver. Since the plotter was extremely slow on the bus, a buffered interface was designed and implemented using the multitasking tools developed earlier. This is similar in concept to the printer interface already described.

Once a UUT had been identified, the software was further refined by using it to write tests for the UUT. These could not at that stage be run, since neither a UUT nor an ATE existed then; but valuable information was obtained regarding the user interface, the test language, and methods of specifying language

extensions.

Over the same period, the system was used for general programming; in fact the great majority of the software was written using the desktop operating system. Thus writing the software became part of the experiment.

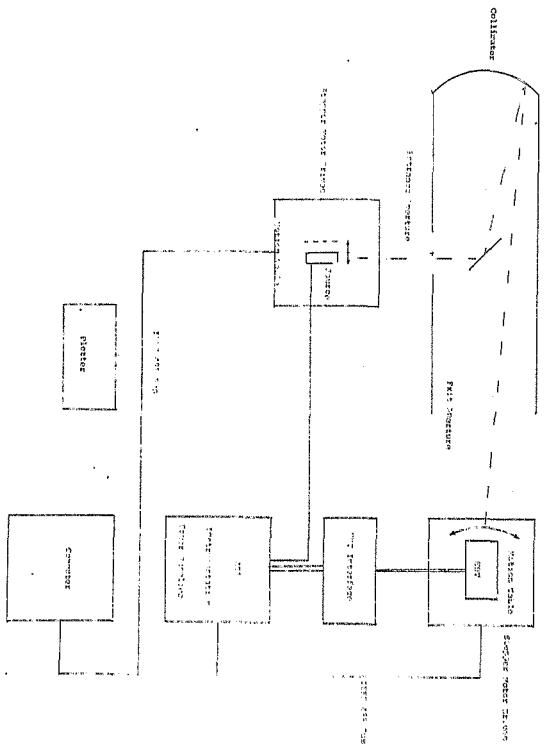


Figure 7.2: Experimental Setup

7.2 Description of the Unit to be Tested

The chosen UUT was chosen to exercise the major experimental ideas that had been developed, particularly those regarding the test language, and its requirements for flexibility and extensibility.

The UUT is an electro-optical device. The primary characteristic that needs to be tested is angular sensit[illegible]o light radiation. The tests involve moving a po[illegible] source in the focal plane of the entrance aperture o[illegible] a collimator, and measuring resultant outp[illegible] from the UUT, which is placed in the exit aperture of the collimator. This is shown in figure 7.2. The effect of this is to simulate the angular motion of a point source at infinity, relative to the UUT's optical axis. Optically, the UUT itself has a very long focal length, hence the use of the collimator. The UUT has two fields of view, electrically selectible, one narrow and the other wide.

In addition to the angular sensitivity of the UUT, its alignment relative to a mechanical reference needed to be verified, as well as the internal noise of the UUT electronics and detector circuitry. Certain other measurements also needed to be made, but those given are sufficient to describe the experiments.

7.3 Test System Hardware Description

The test system consists of the computer and instrumentation mounted in a desk (the ATE), and a jig consisting of a granite table, on which is mounted a collimator and two motion systems. The light source is mounted on one of the motion systems, in the entrance aperture of the collimator, and the UUT is mounted on the other motion system, in the exit aperture. The set up is shown in figure 7.3.

Two motion systems are required to obtain the required range of movement, together with the required resolution. The light source can be moved within the entrance aperture of the collimator, but not over a wide range. It is a property of the collimator that movement over small distances in the focal plane of the entrance aperture translate to angular movements in the exit aperture. Thus moving the source in the entrance aperture either up, down, left or right will result in light falling on the UUT in the exit aperture that simulates light from a point source at infinity at any angle from the axis of the collimator, over a rather small range. Larger angles of incidence on the UUT are obtained by rotating the UUT, using the second motion system, either in pitch or in yaw.

While providing a greater range of movement, the resolution of movement obtained by moving the UUT is not as great as when moving the source. Except at the centre

position, the position of the UUT movement system is not known to the same accuracy as the sight movement system. Full potential accuracy is therefore only realised when the source is moved, and the UUT is positioned at its "on-axis", or centre, position.

The motion system required to move the UUT is very much larger than that required to move the source, due to a large difference in their masses.

The ATF hardware consists of power supplies for the UUT, a data acquisition and control unit (containing a DVM and a switching matrix), a monitor oscilloscope, the drive units for the motion tables, and the computer. The angular sensitivity plots are presented on a small two pen plotter.

In addition there is an interface panel, with some signal conditioning electronics, which interfaces the UUT to the ATE. Test points are brought out on this panel so that any signal can be monitored on the oscilloscope, as well as switches that can be used to manually test the UUT if necessary. This facility is particularly useful while the UUT is still in development, and can be used later for diagnosing faults in a failed UUT.

7.4 Test System Software Configuration

Configuring the ATF required software to drive the data

acquisition and control unit (DACU), the motion systems, and the plotter. The resource drivers were written as separately compilable packages. An example of one of these, the DACU driver configuration file, can be found in Appendix F.

The extraction of the optical axis of the UUT from the angular sensitivity data required a special routine to be written. This was written in Forth, in accordance with the principle that complex software should be written by specialists using specialist tools.

The movement systems were written so that the user could move either the source or the UUT a specified number of milli-radians (or micro-radians) in pitch or yaw, relative to the UUT. The data for the plotter is stored in an array, and plotted using a high level command paragraph. Switching the UUT between narrow and wide field is accomplished by means of relays in the DACU, as is connecting power to the UUT, switching it on and off, and so on.

It was discovered that it was impractical to write the writing aid software as a set of separately compilable packages. Instead a single configurable writing aid package is used. The effort required to add a new instrument to the writing aid package is very small. All that is required is that the relevant menus be extended to include the new options, and that software be written (using high level tools) to print the correct words on

the page. This is not as convenient as having the writing aid software in separate packages, but it works reasonably well. The total effort for a typical ATE should occupy less than a day, if the instrument drivers are available. The source listing for a configured writing aid package is given in Appendix G.

The major reason for the difficulty is that each specific ATE requires a different combination of menu items; the software required to control this basically trivial (but practically complex) operation was considered an unnecessary luxury.

Configuring an ATE therefore requires choosing instrument drivers from a library, and configuring the writing aid.

The tests were not written by the author, but by the test engineer, in accordance with system's design goals. An example of a test routine using the software on this UUT can be found in Appendix H.

8 RESULTS OF THE EXPERIMENTS

Once the specification for the system had been worked out, it was implemented and tried out. The experience gained during the trial period confirmed much of the theory developed during the early phases of the research. The practical aspects of the implementation were highlighted during this phase and several revisions were gone through before the present configuration reached. Certain aspects of ergonomics were emphasised in practice, and these also gave rise to changes. No experience has been gained as yet in the production environment, but certain extrapolations can be made.

8.1 Ergonomics of the Desktop Environment

The user interface is the first unusual thing that the user of the system experiences. It is significantly different from the normal operating system, although certain computer companies are now bringing machines with this type of interface onto the market, and the novelty is beginning to wear off.

The desktop environment was found to be very useful. It was particularly useful during compilation to be able to have the software listing on one page, and compiler messages on another, visible at the same time (as the compiler moves through a file it continually shows the page it is working on on the desktop). The ability to move menus and pages on the desktop was also very

useful; the arrangement of objects could be manipulated to suit the user and the application. In general, the operating environment was successful because it was flexible, non-modal, and easy to use.

8.1.1 Menus

It soon became obvious that handling a large number of menus on the desktop was a problem. The desktop became untidy, and moving fifty menus, one by one, around the desktop was laborious. The idea of grouping the menus together under headings solved this problem. Menus not in use can be "turned off" using this method. Whole groups can be moved by turning the group off, moving the heading, and turning the group on again. The group of menus then appears in a neat pile below the heading.

It soon became apparent that the most satisfactory method of keeping a group of menus tidy was to keep them in a pile, one on top of the other. This immediately prevented the full use of the mouse in going to menus at the bottom of the pile. The cueing key became the primary method of going from menu to menu. This led to the development of the "previous" key. Holding the cueing key down allows the cursor to "page through" a pile of menus at between 5 and 10 menus a second. It is just possible to follow what is going on at this speed, but it is very easy to go too far. The previous key allows one the option of backing up if a menu is

overshot.

A logical extension of the heading idea was also implemented: pressing the control key in conjunction with the first letter of a heading immediately takes one to that menu. This utility was developed as an aid to keyboard intensive applications, which are discussed later.

8.1.2 Speed of Response

One of the complaints that surfaced quite early concerned speed of response during test writing. Benchmarks done at the time showed that the system was capable of writing tests as fast as the average engineer could type them. However, the perception was that the system was slow. Specifically, selecting a piece of text on the page took about a second, cueing from menu to menu took roughly the same. Turning over a page in a file, on the other hand, took up to ten seconds. It seemed strange at first that complaints were made concerning the cueing speed, but not the slowness of the filing system. It then became apparent that people were basing their complaints to a large extent on the perceived complexity of the operation. Selecting a piece of text on a page was seen as trivial, while getting a new page from the disk was seen as complex. In fact the two are of almost equivalent complexity, and the slowness of the filing system was due to an

extremely inefficient algorithm. Improving the response speed was at first aimed at speeding up the response of conceptually simple commands. This was achieved by rewriting a few key routines in Assembler (the original system was almost entirely written in high level Forth). It was only when these improvements had been made that complaints about the filing system became common. This was solved by an algorithm improvement. The system now has acceptably fast response times in the opinion of those who have tried it.

An interesting corroboration of the above was observed when the system was modified to include virtual memory. Although the system became slower when this change was made, no complaints were forthcoming. This was because the delays were grouped into occasional waits of two or three seconds. These occur very infrequently and the operator accepts the wait quite happily, being placated by the whirring disk drive.

8.1.2 Aesthetics and Personal Preferences

One of the most powerful emotional responses to the system was to the decision that had been made to make the pieces of paper light coloured with the text on them dark. Some of the objection was justified by practical considerations: poor quality monitors flicker unacceptably if there are large highlighted areas displayed. However, some users found the realism of the

black on white display preferable when a good quality monitor was used. Similarly, people were continuously asking about the "surface" of the desktop: why wasn't it blank, why not use vertical lines, why not use horizontal lines and so on. It soon became apparent that these decisions were governed by personal preference. Options were included to allow the user to customise the appearance of the screen. It is signifigant that the several users each have their systems configured differently.

8.1.3 Overstrike vs Insert Mode

Another of the decisions which was changed in the light of experience concerned the the use of overstrike versus insert mode. In overstrike mode, the key struck causes the letter it represents to be written at the cursor, overwriting whatever was there before. In insert mode, the letter is placed at the cursor as before but the letter underneath, and all text to the right on the same line, is shifted to make room for the new letter.

Initially the decision was made in favour of overstrike mode due to its simplicity and resemblance to the working of a normal typewriter. In addition, overstrike mode is ideal for the two most common typographical errors. These are striking the wrong key, and transposition (striking the correct two keys in the wrong order). In the first case, a backspace and retype

is all that is needed. In the second case, two backspaces and retyping the letters is required If the error is discovered later, using the mouse to move the cursor to the incorrect letter and typing over it solves the problem. However, the third common fault, leaving out a letter, is impossible to correct simply in overstrike mode; one has to either use CUT and PASTE, or retype everything to the end of the line. Insert mode on the other hand works very easily in this case, although in the first two cases it requires deleting the wrong letters before replacing them with the correct ones.

Overstrike mode was quite accepted for a long time, but once real work began to be done on the system it became apparent that it wasn't the right choice, and the change was made to insert mode.

A dilemma now arose concerning the delete or destructive backspace key. In most editors and word processors the delete key causes the character to the left of the cursor to be deleted, and the rest of the line to move one space left. This works well if one is deleting the character just typed. However if one moves the cursor with the mouse to point at a character, one expects the character pointed at to be the one deleted, rather than the character immediately to the left. After trying out both systems for a while, the more intuitive one was chosen as preferable, ie the character that the cursor is on is deleted. This was chosen despite the single

most common typing error being typing the wrong letter, the correcting of which now requires backspacing and deleting as two keystrokes. Intuitive consistency was found to be preferable even if it meant slightly increasing the number of keystrokes in a commonly used function.

8.1.4 Pointing Devices

Conventional systems move the cursor using keyboard cursor control keys. In this application, however, they are too slow. Moving the cursor across the page using cursor keys takes between 8 and 10 seconds for an 80-column display. This is due to the typical keyboard repetition rate being around 10 hertz. Continuously striking the key allows faster movement (the debounce time constant being shorter than the repetition time), but this is tiring, and still not very fast. As has been mentioned earlier, the screen is analagous to the field of view of the user on the desktop. In order for the analogy to work successfully, the cursor control should be fast enough that it can follow the user as he transfers his attention from place to place on the desktop. This demands virtually instantaneous movement from one end of the desktop to the other (much more than eighty columns). At the same time, great precision of movement is necessary to be able to point without effort to single characters on the page.

The initial system was implemented with a joystick as the main cursor control device, supplemented by the four keyboard cursor control keys. This was found to be totally unsatisfactory. Joysticks do not have sufficient precision of movement (their arc of travel being too small) to allow the average person to use them as positioning devices. They are more successful at controlling the rate and direction of movement of the cursor, but precision of positioning is difficult to achieve without sacrificing overall speed. An exponential transfer functions was necessary to get any real usefulness out of the concept, but fast, accurate, positioning of the cursor was never achieved using a joystick.

One of the basic reasons for this is that using a rate device for cursor positioning is too visual an activity. No kinaesthetic feedback is possible. The human nervous system works better with kinaesthetic feedback than with visual feedback.

Light pens are relatively expensive and have reputation for unreliability. A light pen on a UNIVAC based CAD (computer aided design) system was tried out and this experience was sufficient to eliminate light pens from further consideration. The problems perceived were that the pen had to be picked up before use, oriented in the right direction, and held in front of the face during use. Discussion with the CAD operator revealed that hand and arm tiredness prevented the use of the light pen for

any extended period, and that graphics tablets were pref rred. However, they did allow potentially very fast and accurate movement.

Touch screens do not require one to pick anything up before use. Their biggest problem however is their lack of precision. A finger is simply too big to pick out single characters with ease.

Graphics tablets are expensive, and suffer from the problem of the pen having to be picked up and oriented in space before use. Use of this type of device in a keyboard intensive activity would soon become intolerable.

Other devices were tried, such as track-balls (too visual, difficult to move the cursor long distances in one movement) and touch pads (insufficient resolution), but none was as successful as the mouse. The mouse overcomes virtually all the disadvantages mentioned. It is a position device. Kinaesthetic feedback allows the positioning of the cursor on the desktop with intuitive accuracy. The range of movement is large enough that great precision is combined with high possible speed of movement. It does not need to be picked up; it is sufficient to put ones hand on top of it. Although a hand must be removed from the keyboard to use the mouse, the presence of the [illegible] the mouse makes up for the hand being off the keyb[illegible] in virtually all instances. A particular advantage of the mouse is its large size,

which allows it to be located with the peripheral vision; laying a hand on the mouse can be achieved without removing one's attention from the screen.

Certain kinds of graphics tablets do not use a pen, but a mouse-like device. These are much more expensive than mice, but would probably work as well in this application. They are considered as special cases of mice for the purposes of this study.

8.1.5 Keyboard Intensive Applications

The use of a pointing device and menus in keyboard intensive applications is controversial. A study was made of this aspect by using the DeskForth system for some time as a pure word processor and editor for memos, letters and similar things. It became apparent that the pointing devices mentioned in the previous section were unsatisfactory, with the sole exception of the mouse. For the reasons explained there, the mouse is surprisingly usable in the keyboard intensive application. However, the whole concept of using menus in a strictly word processing environment was thrown into doubt.

Typing is such an automatic process that the use of menus is disruptive unless large scale changes are to be done, such as moving paragraphs around. The mouse and editing menu was found to be very successful in the type of situation where the old fashioned writer would have

used a pair of scissors and a bottle of glue. In hindsight, this is not very surprising. In response to the need for small scale editing features to be available at the keyboard, several changes were incorporated, such as the change to insert mode, the inclusion of a delete key, and the inclusion of a pair of functions that allow the insertion or closing up of blank lines. These few changes, when combined with the mouse and menu system resulted in a very successful editor. Appendix D contains a list of the keyboard functions available.

8.1.6 Non-Directive Operation

The non-directive nature of the system proved to be a problem for a certain group of users. This was the experienced software "expert". No other user found it to be problem, in fact the novice user found it to be quite natural. While the novice user was quite happy to try out different ways of doing things, the more experienced person felt insecure without the familiar rigid structure and "are you sure?" messages. This problem generally resolved itself within a few days.

8.2 Practical Problems

As is always the case, hardware limitations and practical restrictions resulted in many modifications to the initial system before it could be called

successful.

8.2.1 Memory Size

An unforeseen characteristic of test software caused a fundamental change to have to be undertaken at a rather late stage of development. This is that the test suite of a UUT grows in size from the original test specification proposed by the designer, to include a whole series of tests designed to cover grey areas in the design. When the UUT goes into production, the suite grows some more, to cover the grey areas in the production process. After a period, the UUT design and the production process mature, and at this point the test suite begins to shrink again until it is once more relatively small. The initial estimates as to the required test suite size had been based on several units in production, and the acceptance test specifications of several new designs (ie both populations were at phases of development where the test suite is relatively small).

It became obvious during the practical trials that the system would need to be able to accommodate significantly larger test suites than had originally been planned for. In addition, the changes and improvements that had been instituted under pressure from the users had all used up memory. This problem was solved by mapping a large proportion of the system data

to disk based virtual memory.

8.2.2 Speed

The problem of response time has been dealt with before. Some of the techniques used to speed up the system were to write key routines in assembler (something which Forth accommodates very well), to pay close attention to the algorithms used in certain key areas, and to remove error checking from parts of the system that were running reliably. A simple solution to the speed problem became available about halfway through this process. This was an "accelerator" card which replaced the Apple's 1 MHz CPU with a new version running at 3.5 MHz. Although the system works acceptably without the accelerator card, testing applications requiring high speeds have this option open to them.

8.2.3 Varieties of Apples

Apple released the Apple //e soon after the development of ApSys began. The primary differences concern the keyboard and the display. The system was modified to make use of the improved features of the new machine, but care was taken to ensure that older Apple II machines could be used. The section describing the options menu in Appendix C covers the most important differences. Certain more subtle differences, especially with respect to the reset cycle and interupt handling

also had to be addressed.

8.3 The Test Language

The design and implementation of an English like test language proved to be straightforward once the requirements had been *finalised*. The concept of explicitly defining the context of the words made the implementation relatively problem free.

Use of such a language to write actual application software has proven very successful. The ability to tailor the *language* to suit the requirement results in software that is elegant and straightforward to read and to write. However, the fact the language was compiled, rather than interpreted, has proven to have certain disadvantages.

8.3.1 Context Dependancy

While the idea of context dependant meanings to words grew out of a purely theoretical analysis, the practical application has proven to be both easier and more useful than was anticipated.

The first by-product discovered was that the implementation of software in this fashion often leads to code that is faster at run time and also more compact than is the case with more classical methods. Since the nuances of run time behaviour can be sorted out at

compile time by a context sensitive compiler, only the code that is necessary need be compiled. In addition, no time is wasted at run time in deciding what needs to be done. Naturally, many compilers do this as a matter of course. Making the context sensitivity explicit, however, seems to be a particularly simple and elegant method.

Another by-product is that the compiler can check to make sure that words are being used correctly, by ensuring that the words used are in context. Further reflection shows that this is in fact a generalisation of the concept of strong typing.

Another benefit was that the classical system of parameter passing to procedures could be assisted. Parameters passed to procedures can be divided for our purposes into two types. Firstly there is data; two numbers passed to an "addition" procedure to be added together fall into this category. The second kind of parameter is information intended to modify the behaviour of the procedure; an adding procedure might require, apart from the data, a parameter saying whether the things to be added are numbers or arrays. The second type of parameter can be seen as contextual information. Having the concept of explicit contexts available leads easily to the use of overloaded operators, after the fashion of the language Ada. An overloaded operator is one which always looks the same to the programmer, but which the compiler recognises is

one of a family of similar operators; the compiler chooses the correct one by referring to the type of the operands.

Finally, this system allows the systems level programmer to write his software in a variety of ways, depending on the merits of the case; parameters can be passed via data or contextual information, and the compiler can resolve questions either at compile time or run time depending on the efficiency needed, without the test writer needing to know any of this. Once the syntax has been finalised, the system programmer can change his implementation without affecting the test writer at all. The primary reason for this flexibility is the large amount of redundant information that is available in standard English syntax.

In summary, the concept of context sensitivity embraces the fields of typing, overloaded operators, and visibility. It is of great assistance in generating understandable software, and also assists the compiler in generating efficient run-time code.

8.3.2 The Simple Imperative Sentence

Restricting the sentence structure to the simple imperative has not proven to be a problem. No case has yet been found where this structure has not served. Defining the result sheet information was the most difficult problem as it turned out, and this was solved

by breaking the definition into several sentences. An example can be seen in Appendix C.

8.3.3 Supporting the Test Writer

The goal of dividing the software writing between specialist and non-specialist was one of the cornerstones of this project. The idea that the straightforward software should be written by the user, and the specialist reserved for the complex problems, made sound sense in theory. Initially however, problems were experienced.

The problem lay in how to spe e requirements for the special UUT functions. E configuration software, being structured around hardware, was no problem. The difficulty lay in specifying special, UUT specific functions. After a period it became clear that the problem was due to the user being concerned about the implementability of his specification. This was compounded by the specialist whose first reaction to seeing the specification was to change it to ease implementation, or increase overall consistency. At this stage the method of specification was a list of functions that needed to be generated, supplied by the test writer. With both parties thinking in terms of implementation rather than requirements, problems were inevitable.

A method of working that concentrated on the test

requirements, rather than on software, had to be found. A solution that worked was to ban the use of all software related terms, such as procedure, parameter, subroutine, etc. from the specification. This caused a subtle change in orientation. Instead of specifying the software that he wished to write, the user would now specify actions that the test equipment, or operator, or UUT would need to perform.

Thereafter, the specialist creates English sentences to describe the required actions. (It was found that writing suitable sentences requires a certain knack, and that this requires practice; expecting the novice user to write these sentences had been one of the problems of the earlier system). The sentences are then discussed with the test writer, and once agreement has been reached that the sentences do indeed describe the actions that are needed, they are included in the writing aid. At this point it was found to be sensible to wait a while before proceeding to implement the run-time verbs. As the test suite of a UUT is developed, there are invariably changes that suggest themselves.

8.3.3 Compilation versus Interpretation

The decision to implement the test language as a compiler was found to limit the usefulness of the system in certain significant ways. It is probably true that an interpretive system would have been too slow on the Apple; however, as technology improves, this should

cease to be a problem.

Interpreters allow short programs (test programs fall into this category) to be easily debugged, since they are generally more interactive in nature than compiled systems. This is one of the reasons why BASIC is so popular. The edit/compile/run cycle is not only time-consuming, but also less easy to use for the non-specialist user.

During debugging of the ATE, and later when faulty UUTs needed to be debugged, it became apparent that the ability to single-step through a program would have been very useful. The problem in implementing a single stepping function using a compiler is concerned with indicating to the user which step he is on. An interpreter can show the exact line in the source to the user as it is interpreting it; this information is often not easily recovered from the compiled object code.

Many very short programs were written to fault-find either the ATE hardware or the UUT. These were often five or fewer lines long. Having to compile and recompile such short programs is inefficient. It would be ideal to be able to write and execute such programs on a line by line basis. Each successive line might in fact be determined by what happened when the previous line was executed.

Thus two needs were identified that had not been addressed at the start of the study; the need to be able

to single step through programs, and the need to be able to interactively write and run test software on a line by line basis. Implementing such a system is easier to do with an interpreter, or an incremental compiler. A fully compiled system is also possible, but would require more effort to develop.

8.4 The Writing Aid

The use of a writing aid resulted in a subtle change in the development strategies followed by the ATE developers using it.

In the past, the debugging of the test software had proven to be a major part of the development effort. Timescales had usually dictated that the test software and the ATE hardware be debugged in parallel. This had in itself caused problems. In addition, since the UUT was usually undergoing development in parallel with the ATE, the test requirements were usually changing, resulting in continuous delays and rewriting of software.

The use of the writing aid reduced the test generation time from typically months to the order of hours or days. This allowed the ATE developer to concentrate on the hardware for a longer period secure in the knowledge that the tests would be written quickly when the hardware was working.

The writing of ATE specific instrument drivers was separated conceptually from the writing of tests, and this resulted automatically in cleaner, more clearly specified packages being generated. The ATE developer concentrated on producing a working ATE system rather than on testing the UUT. As a result, the effects of changes in the UUT were reduced considerably. The actual test software was typically not generated until a few days before the scheduled delivery of the ATE.

Once the ATE had been delivered, a further effect was noticed. The ATE users began generating all sorts of diagnostic tests. In addition, once it became clear that changing the test software was a relatively simple operation, the acceptance test procedures underwent radical changes. Thus the normal process of change that acceptance test procedures undergo was greatly speeded up, since experimentation was now economically possible.

A problem became evident, however; configuration management of the test software became difficult. In the past, changing the software had been the domain of one or two "experts", and they had kept control of issue raises in a reasonably informal manner. This system had to be formalised. Configuration audits were now simplified, since the test software could be easily read and compared with the Acceptance Test Instruction.

It is normal for test software to undergo revision after

it has been in use for a period. Three techniques were found to be useful in this process.

- Where relatively short tests needed major revisions, the writing aid made it easy to simply rewrite the whole test.

- Small changes to tests (for example changing the acceptance limits, which is a very common change) are most easily achieved using the normal editing facilities of the system, rather than the writing aid.

- Larger changes or additions to relatively long tests were most easily achieved by using two pages on the desktop. Apart from the file containing the original test, a second page was used to write the revision on. This was achieved using the writing aid. The revised portion was then inserted into the old software using the cut and paste editor. This technique also allows the revised portion to be tested independently.

The techniques used for changes demonstrate the flexibility of the system, and emphasise the usefulness of being able to bypass the writing aid menus when appropriate. The fact that the editor is always available is an instance of non-modality; the updating techniques described depend on this characteristic.

9 CONCLUSIONS

9.1 Validity of the Goals

The specific goals of the investigation, described in section 2.4, were found to be valid. The major goals can be summarised as follows

- develop a test language that is hardware oriented, and English like in both vocabulary and syntax
- develop an operating system that is non-directive, non-modal, multitasking, and easy for both specialist and non-specialist to use
- develop a test writing aid that allows non-specialists to write tests

The validity of the concept of an English-like test language was proven unequivocally. The hardware orientation of the test language was greatly preferred by the users (typical comments include "you know where you are using ApSys, unlike system X"'. The writing of the test programs was effectively reduced to a conceptually and practically trivial exercise, due to the understandability of the software. This result emphasised the importance of readability over easy of writing.

The operating system concepts were proven to be workable and very useful. Whether or not they are the best

solution to the problem is not as clear. Certainly, the multitasking proved invaluable, and the editor and windowing part the system was very easy to work with. The non-modality of the system is probably its greatest advantage; using other systems is extremely frustrating after using this one.

The method used to access mass storage (the filer), while easy to use, has limitations as regards future growth; there are better techniques that can be used.It would be desirable to allow the system to interface more easily with other applications such as spreadsheets and database systems. This was not a goal of the investigation, but in hindsight it should perhaps have been.

In summary, while the user interface was proven, the interface with more standard operating systems could be improved. Nonetheless, the primary goal of the operating system, that it should be easy for both the specialist and non specialist to use was achieved, and proved to be valid and useful; both specialists and non-specialists can, and do, use the system.

The test writing aid proved both useful and necessary. The writing of tests with it could be reduced to the last, and easiest, item on the agenda. No prior training was needed in order to use it, apart from a five minute demonstration. Since it manipulates text only, and does not have to function as a procedure generator, it was

easy to implement and configure. Since the test language was English, there was no need for a complex procedure generator in the writing aid. The synergy achieved between the operating environment and the English *like* test language made the test *writing aid* both the easiest and most *successful part* of the project. The English *syntax* was as important here as the vocabulary, since the order of words was quite natural, making prompts to the operator explaining how to choose items from menus totally unnecessary.

Finally the cost goal was proven totally valid; this is hardly surprising, since low cost is always a desirable attribute.

One subsidiary goal that deserves attention is the decision that was made to place the onus of doing the correct thing on the user, rather than having the computer check everything rigorously. Although the system has not yet been used on the production line, there is some evidence to support the validity of the *reasoning leading* to this decision. Unfortunately it will be some time before the final answer to this question is known.

9.2 Usefulness of the Techniques Used

The following concepts were *found* to be definitely useful:

CONCLUSIONS

- The test software should be hardware oriented; this is a necessary attribute from the point of view of all system users

- The test language should be English like in both vocabulary and syntax; this is necessary if a writing aid is to be used, or non-specialist personnel is to write the tests; it also aids the task of all other system users

- The system should allow a user selected range of instruments to be slotted in as required; this has substantial cost advantages (the system described cost approximately one fifth the cost of comparable bought in systems; the cost was not signifigantly higher than for an equivalent manual test station.)

- Splitting the effort between specialist areas is a highly desirable goal, which not only reduces costs, but also shortens timescales, since specialists are always in demand

- Non-modality is highly desirable, although not absolutely necessary

- Multitasking is very useful, although not absolutely necessary; it saves time on the production floor, and eases the task of the software specialist, both of which save money

- The use of the desktop paradigm was successful in that it was easy and enjoyable to use; other

operating environments may be as successful, however. No final conclusion can be drawn; all that can be said is that it works well.

- The use of a mouse as a pointing device is very desirable, if the chosen operating system requires cursor movement; no other device presently available was found to be as successful.

- The use of a software writing aid is definitely indicated; this should be done in conjunction with a test language that has an English like syntax; this greatly simplifies the implementation of the writing aid.

- The ability to bypass the menu system is invaluable; small scale changes can be made to the tests far more simply by the use of the editor, rather than the writing aid. This is probably generally true.

- The compile step between writing and running the software should be eliminated; test programs are small and the disadvantages of compilers outweigh the advantages in this case; interpreters allow greater flexibility in interactive testing of the UUT, especially when diagnosing UUT faults.

9.3 Possible Future Directions

It would be ideal to be able to run a spreadsheet program on one page on the desk, the test system on

another, and a *different* application (perhaps a communications program) on yet another, and allow data to be shared between these applications. The Apple computer is too limited a machine to allow this easily, but more modern computers, with larger memory addressing ranges, could achieve this easily. The Apple Macintosh, while not multitasking, achieves data sharing. The success of the windowed, mouse driven user interface is emphasised by the enormous number of commercial packages now (beginning of 1985) becoming available, that use this type of interface.

Integration of ATE systems with the rest of the production system is another area that this project did not touch on. The fields of Computer Aided Engineering (CAE) and data/software integration are presently widely separated, and much benefit could be obtained by integrating the recent work that is being done in these separate fields.

Faster computers make the use of an interpreted or incrementally compiled test software feasible. This type of system allows easy diagnosis of faults, both in the test program, and in the UUT. More work is definitely indicated in the area of UUT diagnostics after failure.

Artificial intelligence has a clear application in the conversion of test instructions into English like test programs. The flexibility required of sub-assembly testers makes them prime candidates for the application

of flexible manufacturing systems (robotics) especially in the area of mechanical jigs, which this investigation only touched on. The cost of the mechanical jigs in the test case presented here was was great as the cost of the ATE. Many sub-assemblies need to be oriented in space and moved around as part of their test procedure, an area where robotics is already active. Integration of robotic assembly with test systems appears well within the capability of current technology.

ATE systems in all areas except that of sub-assemblies have matured to the stage where it is possible to purchase systems more cheaply than to design them in-house. These cost and time savings are not yet realisable in the sub-assembly ATE area. Nonetheless, the concept of user configurable ATE, together with a vendor supplied software library, brings this within reach. The system described here could probably be used as the basis for such a system.

APPENDIX A

ATE QUESTIONNAIRE

ATE QUESTIONNAIRE

This Appendix contains an example of a questionnaire used to establish the perceived needs of various users of ATE. The results of the questionnaire are summarised in the following table.

The operator must feel that he is in control
The operator must be prevented from damaging the system
The test software must be readable
The test software must be understandable by the operator
The system must be configurable to do unusual tests
The system must be reliable
The system must be design ergonomically
A system self-test is required
The techniques used by the test writer must be obvious
Operators must be low skill personnel
The latest technology instruments must be used
Standard instruments must be used
Standardisation should not be aimed at, since it is inflexible
The tests must be well documented
Tests must be easily changed
Tests must be readable
Tests must be repeatable
Costs must be kept low to allow cheap system duplication
The system must be usable to track down problems manually
The system should be interesting to work with
The actual working of the ATE should be obvious
Interfaces between the ATE and the UUT must be robust
Support for the controller and instruments must be available
The operator must know whether the UUT or ATE is faulty
The test language should suit the type of testing being done
"The machine must work for the operator, not vice versa"
Maintenance time must be kept to a minimum
Instruments must be easily replaceable
Testing should procede quickly
The operator must like the equipment
The ATE should be testable
The ATE designer must get satisfaction from using the system
New tests must be easily added
The operator must be able to choose which tests to do

ATE QUESTIONNAIRE

→ *PJ DUGGAN (Office N32, ENGINEERING BLOCK)*

TEST EQUIPMENT QUESTIONNAIRE
TOETS UITRUSTING VRAELYS

1. *There are two identical questionnaires enclosed, one in English and one in Afrikaans*
   - *fill in only one*

   Daar is twee identiese vraelyste aangeheg, een in Afrikaans en een in Engels. Vul net een in.

2. *Please post, or return by hand*
   Stuur asseblief terug, per pos of per hand

ATE QUESTIONNAIRE

A. PROPOSAL DETAILS

Please fill in the following details. None of the questions in this questionaire are sensitive. No names will be published or distributed in any way.

NAME .....

DEPARTMENT .....

WHAT IS YOUR MAIN INVOLVEMENT WITH TEST EQUIPMENT

1) *Operator*
2) *Maintainance*
3) *Test designer/ writer*
4) *Test system designer*
5) *System user (eg production engineer)*
6) *Other (specify)*

WHAT EXISTING TEST SYSTEMS DO YOU, OR HAVE YOU HAD CONTACT WITH?

ATE QUESTIONNAIRE

B. *HARDWARE*

1) *IS THE CHOICE OF INSTRUMENT IMPORTANT TO YOU?*

YES ☐ *NO* ☐

WHY?

_____

_____

_____

2) *IS THE CHOICE OF COMPUTER IMPORTANT TO YOU?*

YES ☐ *NO* ☐

WHY?

_____

_____

_____

3) *IS THE CHOICE OF CONFIGURATION, EG INSTRUMENT DESKS INSTRUMENT RACKS, HEIGHT OF INSTRUMENTS, SIZE ETC IMPORTANT TO YOU?*

YES ☐ *NO* ☐

WHY?

_____

_____

_____

_____

4) *DO YOU FEEL YOU SHOULD HAVE A MAJOR INFLUENCE ON ANY OF THE ABOVE FACTORS?*

YES ☐ *NO* ☐

IF SO, WHICH ONES?

_____

_____

_____

_____

5) *WHAT OTHER ASPECTS OF TEST EQUIPMENT HARDWARE AFFECT YOU? IN WHAT WAYS?*

_____

_____

_____

_____

_____

ATE QUESTIONNAIRE

C. *SOFTWARE*

1) *IS THE LANGUAGE THE SOFTWARE IS WRITTEN IN (EG BASIC FORTRAN, ASSEMBLY; FORTH; POSCAL ETC) IMPORTANT TO YOU?*

YES ☐ *NO* ☐

WHY?

_____

_____

_____

_____

2) *IS A SELFTEST IMPORTANT TO YOU?*

YES ☐ *NO* ☐

WHY?

_____

_____

_____

3) *IS READABILITY OF THE SOFTWARE IMPORTANT TO YOU?*

YES ☐ *NO* ☐

WHY?

_____

_____

_____

4) *IS THE WAY THE SOFTWARE BEHAVES IMPORTANT TO YOU?*

YES ☐ *NO* ☐

_____

5) ARE THE TECHNIQUES USED BY THE TEST WRITER IMPORTANT TO YOU?

YES ☐ NO ☐

WHY?

_____

_____

_____

6) ARE THERE OTHER ASPECTS OF TEST EQUIPMENT SOFTWARE WHICH AFFECT YOU?

YES ☐ NO ☐

IN WHAT WAYS?

_____

_____

7) DO YOU FEEL YOU SHOULD HAVE A MAJOR INPUT INTO ANY OF THE ABOVE DECISIONS?

YES ☐ NO ☐

IF SO, WHICH ONES

_____

_____

_____

_____

_____

 Page A8 omitted in pagination

D: SYSTEM

IN ORDER FOR YOU TO OPERATE BETTER

1) SHOULD TEST SYSTEMS BE TAILORED TO THE LATEST TECHNOLOGY, OR STANDARDISED FOR A NUMBER OF YEARS?

LATEST TECHNOLOGY

STANDARDISED

EXPLAIN:

_____

_____

_____

_____

_____

2) SHOULD THE TEST EQUIPMENT BE IN CHARGE OF WHAT IS HAPPENING OR SHOULD YOU CONTROL THE SYSTEM?

IF YOU ARE NOT A TEST EQUIPMENT OPERATOR, DO YOU THINK THE OPERATOR SHOULD CONTROL THE TESTER, OR THE OTHER WAY AROUND?

DISCUSS:

_____

_____

_____

_____

_____

3) SHOULD EACH UUT (UNIT UNDER TEST) HAVE ITS OWN SPECIAL TC PURPOSE, DESIGN OF TESTER OR SHOULD A COMMON SYSTEM BE USED FOR EVERY UUT IN

a) A PRODUCT RANGE SPECIAL ☐ COMMON ☐

b) OVER ALL PRODUCTS SPECIAL ☐ COMMON ☐

DISCUSS:

_____

_____

_____

4) IS ONE BIG, FAST, TESTER BETTER, OR SEVERAL SMALLER TESTERS?

BIG, FAST ☐ LOTS OF SMALLER ☐

WHY?

_____

_____

_____

_____

_____

5) SHOULD TEST SYSTEMS BE DESIGNED INSIDE KENTRON, OR BY OUTSIDE FIRMS?

INSIDE ☐ OUTSIDE ☐

WHY?

_____

_____

_____

_____

E. *YOUR VIEWS*

PLEASE WRITE DOWN THE FIVE MOST IMPORTANT ASPECTS OF TEST EQUIPMENT AS THEY AFFECT YOU; AND INDICATE THINGS THAT SHOULD BE DONE TO IMPROVE THESE ASPECTS

APPENDIX B

MULTIFORTH SUPPLEMENT

MULTIFORTH

This Appendix contains a copy of a supplement to the Fig-Forth Reference Manual. This supplement contains a description of the differences between the standard Forth described in the Reference Manual and the *multitasking* Forth system developed by the author to support the Desktop Operating System described in this dissertation.

Since MultiForth is a general purpose software system, and is not confined in application to the field of ATE, there is a need for documentation to support the general user. The MultiForth Supplement is supplied to users of MultiForth. It is expected that they will already have obtained a copy of the Reference Manual from the Forth Interest Group, or their representatives in South Africa (FIGSA).

MULTIFORTH

SUPPLEMENT TO THE FIG-FORTH REFERENCE MANUAL

P J DUGGAN

The FIG-Forth Reference Manual is obtainable courtesy of the Forth Interest Group, PO Box 1105, San Carlos, CA, USA.

1 INTRODUCTION

1.1 Scope

This supplement describes those features of MultiForth which are distinct from the *FIG standard*. In the main, this consists of the verbs required to define and control tasks. Some *modifications of the kernel were* also made and these are described. This document is a supplement to the *FIG-Forth User Manual, and the reader* will gain most benefit from it if he reads the User Manual first.

1.2 General Description

MultiForth is a version of *FIG Forth with the addition* of multitasking features. In brief, this means that several processes can proceed in parallel, *sharing the* CPU between them.The applications of multitasking vary from multi-user systems, where several terminals are connected to the same computer, to multi-process systems, where it is desirable to have several processes running or available at the same time. Examples of the latter can be found in some of the integrated software packages which are becoming available on personal computers, where a spreadsheet, a word-processor and perhaps a graphics program are available simultaneously to a user in a multi-window environment. Often, several processes which need to be completed for a given end

result are conceptually parallel in nature, and writing software for these cases is greatly simplified using a multitasker.

2 TYPES OF TASKS

There are several types of tasks. These are Terminal tasks, Background tasks, and Shadow tasks. They differ in the amount of memory they take up, the facilities which they offer and system overhead they entail.

2.1 Terminal Tasks

These are the most complete tasks. The normal Forth system is a terminal task. They have their own dictionary, own stacks, own User Variables, own Terminal Input Buffer (TIB), and own pad. The main Forth system is a terminal task called OPERATOR. All the other tasks in the system are usually contained in OPERATOR's dictionary space. A terminal task can have it's own terminal device, such as a VDU. Another example of a terminal task is PRINTER. This is a complete terminal task except that it has no keyboard.

2.2 Background Tasks

These tasks are more limited than terminal tasks since they have no dictionary, no TIB, and no pad. They take up much less memory than terminal tasks. They are used

for system functions or functions which do not need direct operator I/O.

2.3 Shadow Tasks

These tasks are very limited in size and facilities. They do not have their own stacks, user variables etc. They cannot run high level FORTH verbs. Their advantage is that they take up very little memory space, and the time to activate them is very small. They therefore impose very little overhead on the system. They are commonly used for simple system functions such as polling for a given status. They will often activate a background or terminal task do the more complicated functions a particular status requires.

3 CREATING A TASK

Creating an active task requires typically three steps. The first step creates the task "header" and allocates memory to the task. The second step sets up the task to be ready to run. The third step activates the task and tells it what to do. A shadow task is slightly different, but this will be described later.

3.1 Task Construction

In order to create a task the following verbs are used:

```
a b c TERMINAL name
```

will create a terminal task where
a is the size of the dictionary, in bytes
b is the size of the stacks, in bytes
c is the number of user variables
name is the name of the task

This must be followed by:

```
name CONSTRUCT
```

Creation of a background task is similar:

```
b c BACKGROUND name
name BUILD
```

where b and c are as in the case of a terminal task. Since a background task has no dictionary, it's size need not be specified.

The user variables in each case are initialised to the values current in the task that created the new task (usually OPERATOR).

A shadow task takes only one parameter, the address of the machine code subroutine it will execute:

```
adr SHADOW name
```

The shadow task performs the routine at adr by using a subroutine call (JSR). The routine returns control to the system by executing a return from subroutine (RTS).

3.2 Activating a Task

A task is created with it's status byte set to 1. This effectively disables it until the status byte is set to 0. Before a terminal or background task is activated, it's interpretive pointer (IP) must be set to point at the verb(s) it must execute. This is done in the following way. A verb is defined which, when executed, will set the task's status byte to 0, and will point the task's IP at the verbs to be executed :

```
: GO taskname ACTIVATE some-verbs 1 STOP ;
```

This will activate the task called "taskname", and set it to execute "some-verbs". The construction 1 STOP deactivates the task when some-verbs is finished, by placing a 1 in the task's STATUS byte. Any non-zero value in the task's status byte will deactivate that task.

A shadow task is activated by simply setting it's status byte to zero:

```
0 name SHADOW.STATUS
```

WARNING: A background or terminal task must never be allowed to execute the ";" at the end of the verb which activated it. A STOP must always be placed before the end of the activation verb to prevent this.

4 HOW THE MULTITASKER WORKS

The multitasker is a simple round-robin. Each task is activated in turn, when the previous task releases the CPU.

4.1 Pausing a Task

Any t.sk that is active, but that is not present does not have the CPU, is said to be PAUSED. Naturally, at any instant all but one of the tasks in the system will be paused. Each task does a few things, and then pauses; the next task in the round robin is then activated, until it in turn pauses. The verb which a task uses to release the CPU is, as might be expected, named PAUSE. A task can PAUSE at any time, but the most common place is during I/O operations. It is good practice to put PAUSEs inside loops that will take a long time to execute, to avoid causing delays in other tasks in the system.

4.2 Task Registers

The FORTH system has several system registers. These are IP, the interpretive pointer; UP, a pointer to the base of the user variable table; W, the current working pointer which is derived from IP just before a verb is executed; SP, the stack pointer (in the 6502 model this is synonymous with the N-register); RP, the return stack pointer (in the 6502 model this is the CPU machine

stack pointer); and ME, the pointer to the currently active task's header. All these registers, except W, are saved in a tasks buffer area when it PAUSES. W need not be saved, since it is pointing at PAUSE whenever PAUSE executes, and it's next value will be derived from IP when the task is reactivated.

4.3 The System Stacks

The Forth system has two stacks, the Parameter stack and the Return stack. In many models, these two stacks are in main memory, and each task has it's own stacks. However in the 6502 model, the parameter stack is always in page zero, to allow the X-register to be used to address it, and the return stack is the machine stack. Therefore, whenever a task pauses both these stacks must be saved to a task buffer area. This is naturally slower than simply saving pointers to the stacks, as can be done if every task has it's own stack area. The advantage is that actual stack operations are very much faster in the 6502 model. The return stack especially is used heavily by the inner interpreter, NEXT, and so efficient operation of this stack is important. Normally the stacks are very small in size when PAUSE is executed, and so the overhead involved in saving them is small.

4.4 User Variables

Each task has a group of variables that it alone can access. They are arranged in a table, the base of which is pointed at by UP, the User Pointer. When a task is active, UP points to that task's User Variables. Among the uses of the user variables are to store task specific system values such as OFFSET, S0,,R0, CURRENT, and CONTEXT. In addition, it is convenient to vector the primitive I/O functions such as KEY, EXPECT, TYPE, EMIT, CR, etc via user variables. In this way, each terminal task can have it's own specific I/O primitives, which are transparently used by other verbs such as LIST. Executing LIST in OPERATOR will automatically cause the TYPEd output to appear on the VDU, while the same verb executed from PRINTER will cause the output to appear on the printer. This is because the user variables 'TYPE and 'EMIT are set up specifically for each task. Executing TYPE or EMIT from any task will vector via that task's user variable table to the I/O verbs defined specifically for that task.

VECTORING CONVENTIONS : The following conventions are used in naming vectored verbs :

Primary verb	: TYPE, EMIT
Variable containing the PFA of the verb vectored to	: 'TYPE, 'EMIT
Actual verb executed	: (TYPE), (EMIT)

For example, if a new version of TYPE is to be used in a new terminal task, this would be achieved by the following means:

```
: (NEW.TYPE) NEW VERBS ; ( defines the new verb )
' (NEW.TYPE) 'TYPE ! ( changes the vector )
```

5 MODIFICATIONS TO THE FIG MODEL

The following modifications (ie not merely extensions) to the FIG model were made.

5.1 Insertion of Pauses

Pauses were embedded in all I/O verbs, ie terminal, VDU, printer, and disk related verbs.

5.2 Resource Sharing

Since several tasks may want to use a particular resource at the same time, it is necessary to have an arbitration rule. This is described later. Useage of the printer and the disk drives is governed using this rule, and the relevant verbs were modified accordingly.

5.3 Vectoring

The following system verbs are vectored: RESTART, QUIT, ABORT, ERROR. They are the same for all tasks, and are

merely vectored for convenience. eg

```
: ERROR (ERROR) ;
```

A new version of ERROR can be defined and the vector changed thus :

```
' (NEW.ERFOR) CFA ' ERROR !
```

In addition the following user variables are used to vector task specific verbs : 'TYPE, 'EMIT, 'CR, 'EXPECT, 'KEY, and 'PAGE.

5.4 Forget

The memory management of a multitasking system is more complex than a single tasking system, and FORGET has been modified to reflect this. Essentially, since the memory is broken up into discrete portions by the various tasks, and each task's dictionary is inside it's own memory portion, new verbs are not always at higher locations in memory than older verbs. In addition, each task's dictionary links into the dictionary of the task which created it, all the way back to OPERATOR. The new FORGET, therefore, must avoid forgetting verbs in other task's dictionaries. More subtly, a verb may be preceded in the linked dictionary list by a verb in another task, and succeeded by a verb in yet another task. Thus it is insufficient to merely reset DP and LATEST when forgetting... those verbs created later than the one being forgotten, but which exist in other tasks'

memory areas must remain in the dictionary.

The new FORGET therefore will only forget words in a task's own dictionary, and will leave a correctly linked dictionary when it is finished. Any words above the forgotten word which are in that task's dictionary will also be forgotten as is normal. Words in other tasks' dictionary areas which are above the forgotten word either physically, because they are higher in memory, or logically, because they were defined later, will remain. A side-effect of the new FORGET is that words which are in a task's dictionary area, and are logically later in the dictionary than the forgotten word, but which are lower physically in memory due to some non-standard manipulation, will be delinked, and thus forgotten, although that memory area will not be automatically recovered.

Note that if a vocabulary is forgotten, all the verbs in that vocabulary will be forgotten, no matter which task they are in. Potential problems can also occur when a task defines a verb in its own dictionary which uses a verb from another task's dictionary which is subsequently forgotten by the other task. The verb using the forgotten verb will remain, and behave erroneously, and probably catastrophically. It is therefore good practice to define all verbs which will be used by more than one task as part of OPERATOR's dictionary. It is also a good idea for each terminal task to have its own VOCABULARY, which will seperate the verbs in its

dictionary from the rest of the system conceptually as well as physically. In keeping with the FIG model, but in contrast to most actual implementations, FORGET will only work on the CURRENT or "definitions" vocabulary. This is logically sensible and analagous to the defining case. If each task has its own vocabulary, as described, it is explicitly necessary to invoke another task's vocabulary before verbs in its dictionary area can be used, with the consequences this entails.

Due to the complexity of the new FORGET, a debug tool has been included. This lists all the verbs considered during a FORGET, and indicates which of them have been delinked, and thus forgotten. The utility is controlled by a system variable, FWARNING. When FWARNING is 1, the tool is operational, and when it is 0, it is disabled. If the conventions described earlier have been followed, however, FORGET behaves consistently, and this tool is not required.

5.5 Error handling and Restart

This aspect has been improved so that most errors in tasks are isolated to that task and do not cause the rest of the system to crash. Background tasks however are created with WARNING set to -1, since they have no means of error notification; thus if an error occurs in a background task, the system will ABORT. ABORT has been modified to indicate which task was aborted. The

verb WHY? may be executed from the keyboard to get the error message. A second error occurring during the process of handling an error also causes the system to ABORT. Since the error messages are contained on the disk, a terminal task trying to indicate an error might not be able to if another task has the disk. This case is treated as though WARNING was 0, ie an error number is shown.

The system startup is via the normal COLD or WARM vectors. The verb executed is RESTART which is a vectored. This is especially useful to allow additional tasks to be restarted apart from OPERATOR. For example, if a task activation verb was called TASK1.GO, and that task was to be activated at startup, the following procedure would be followed:

```
: (restart) (restart) task1.go ;
' (restart) cfa ' restart !
```

If this procedure is followed, any addition to the startup routine will follow automatically, and no knowledge of the previous (restart) is necessary.

6 USING THE MULTITASKER

The use of a multitasking system is different in a few subtle ways from a single tasking system. The biggest differences are discussed here.

6.1 Resource Sharing

Common resources such as the disk must be shared between tasks in a suitable fashion, and conflicts prevented. The way this is done in MultiForth is by means of the functions *GET and RELEASE*. Before a task can use the disk drive, for example, it must first obtain the use of it by executing:

```
DISK GET
```

where DISK is actually a variable which is defined in the system. GET first checks that no-one else is using the disk; this is the case if DISK contains 0. If someone else is using the disk, DISK will contain an address in that task's header; in this case, GET will PAUSE until DISK becomes 0. GET then places the address contained in ME in DISK. When the disk operation is complete a task releases the facility by executing:

```
DISK RELEASE
```

which puts 0 in DISK. GET and RELEASE can be used at any time, ie you can GET or RELEASE a facility, whether you presently have it or not, without harm.

GETting and RELEASEing the disk is done automatically in the disking primitives in MultiForth, and so it is not necessary to explicitly GET the DISK in most cases. It is worth while remembering that a verb such as "BLOCK" RELEASEs the disk *immediately after it is executed*. Another task could then use the disk drive and

over-write the buffer at the address left by BLOCK. The contents of a block should be immediately moved to a safe place before editing, for example. When the time comes to replace that block on the disk, the original address will, in general, no longer be correct, and BLOCK should be used again, the data moved to that address, and the block UPDATEd.

6.2 Printing

During a printing operation, the computer spends a large proportion of it's time waiting for the printer. One of the advantages of multitasking is that the time spent waiting need not be wasted, but may be allocated to another task to do something in. MultiForth contains a terminal task called PRINTER which has all the facilities of a normal terminal, with the exception of a keyboard. PRINTER obtains it's input in the form of commands from other tasks such as OPERATOR. These commands are exactly like the commands that would be typed at the keyboard of a normal terminal task. The normal commands that would be sent to PRINTER are printing commands such as LIST, but PRINTER can do other things as well. The format of commands is as follows:

```
PRINT # XXXXXXX #
```

where # is any character (often a quote ") and XXXXXX represents any FORTH verbs that can be executed interactively. Note that the delimiting character is the

first non-blank character after PRINT, and that all characters from that character until its next occurrance will be sent to PRINTER. A practical example could be:

```
PRINT / ." PROGRAM LISTING " DR1 20 LIST /
```

A more complex example :

```
PRINT / : zap 30 2C do i list loop ; /
PRINT / zap forget zap /
```

The last example will list screens 20 to 29 on the printer, and then forget the verb used to do the printing. In both cases, while the printer is busy OPERATOR is active and can be doing other things, such as editing. An important point is that ZAP is derined in PRINTER's dictionary, and OPERATOR therefore could compile new verbs into its own dictionary while ZAP is being executed, and these would not be affected when ZAP is forgotten by PRINTER.

PRINT can be used inside definitions:

```
print " : (disk.dump) 141 1 do i list loop ; "
print " : disk.dump print / (disk.dump) / ; "
```

Note that (DISK>DUMP) and DISK.DUMP will be contained in PRINTER's dictionary (although they could just as easily have been put in OPERATOR's dictionary). Since PRINTER has its own vocabulary, PRINTING, using DISK.DUMP from OPERATOR requires that the vocabulary PRINTING be invoked first. If it had been put into OPERATOR's

dictionary, then this would not be necessary.

6.3 Interupt Handling

In most systems the handling of interrupts requires the use of long assembly language programs. A multitasking system allows the interupt handler to be written in high level FORTH, and placed in a task. The task remains dormant (ie its status byte is non-zero) until an interupt occurs. A small assembly language program then simply sets the status byte of the interupt handler task to zero. When the task is finished, it places a 1 in its status byte again. The following trivial example illustrates the point.

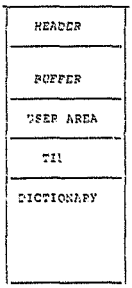
```
1C0 1C2 5C terminal interrupter
interrupter construct

 : go.interrupting  interrupter activate
   begin 1 status c! ( switch off until interupt occurs )
       pause  ( will not be reactivated until status = 0 )
       ." interupt occurred   ( when reactivated )
   again
 ;

 code handle.interrupts
   sei, (mask interrupts)
   pha, ( and  save accumulator )
   ' ( lda,
   interrupter 1+ sta, ( restart interrupter task )
   pla, ( restore acc )
   cli, ( unmask )
   rti, ( carry on )
 end-code

 ' handle.interrupts  hex FFFE !
 ( set 6502 interupt vector to this routine )
 go.interrupting ( start task operating )
```

Testing the interupt handler is easy since HANDLE.INTERRUPTS is executable from the keyboard if the

RTI, is replaced with the usual NEXT JMP, construction.

7 ANATOMY OF A TASK

Any task has a header. In a shadow task, this is where it ends. Background and terminal tasks have buffer and user areas as well, and terminal tasks have a dictionary area.



7.1 The Header

A dis-assembled listing of PRINTER's header follows:

```
6F5C- AC 01     LDY #$01        ( CHECK STATUS )
6F5E- F0 03     BEQ $6F64       ( 0 MEANS TASK IS ACTIVE )
6F60- 4C F0 F5  JMP FOLLOWER    ( NOT ACTIVE, SO GOTO HEADER OF
                                  NEXT TASK IN QUEUE )
6F64- A9 6F     LDA #$6F        ( PUT ADDRESS OF JMP TO
6F66- 85 11     STA W+1           FOLLOWER IN W FOR USE DURING
6F68- A9 61     LDA #$61          TASK SWITCH OVER )
6F6A- 85 10     STA W
```

6F6C- 4C E4 4E JMP (PAUSE) (GOTO ROUTINE TO SWITCH TASKS)
ME contains the address of the jump to the next task's header; in PRINTER's case ME would contain 6F61. When PAUSE is executed, it does a jump to the address in ME, which causes a jump to the header of the next task. The previous task to PRINTER in the queue would have ME pointing to a jump to 6F5D. Therefore when the task prior to PRINTER PAUSEs, a jump occurs to PRINTER's header. The first thing that happens then is that PRINTER's status byte is checked. In the example shown, PRINTER's status is not zero, therefore a jump is performed to the header of the next task...PRINTER effectively PAUSES before it even starts. If however 6F5E contains a zero then a jump is performed to the multitasker, which is called (PAUSE). Before this happens PRINTER's ME is stored in zero page, so that (PAUSE) knows who to activate.

A SHADOW task behaves slightly differently. The header of this kind of task does not contain a jump to (PAUSE). Rather, it does a JSR to the machine code routine that the SHADOW task is to execute, and then after the RTS it jumps to the header of the next task.

7.2 The Buffer Area

The first thing (PAUSE) does is save the old task's stacks and system registers in it's buffer area. The buffer area contains the following data:

SP+RP,UP,IP,SP,RETURN STACK,PARM STACK.

After saving the old task's buffer, (PAUSE) fetches the contents of the new task's buffer and fixes the system registers and stacks for the new task. It then fetches the new value of ME from where it was saved in zero page, and puts it in ME. It then jumps to the inner interpreter via ?STACK. ?STACK is a code routine to check for stack overflows. The new task is then active until it executes PAUSE, which will jump via ME to the jump contained in the tasks header to the next task.

7.3 The Dictionary

A terminal task has it's own memory area, consisting of the User Variable table, the TIB, and the dictionary. A background task has only a User Variable table. FENCE is set to the beginning of a terminal task's dictionary when the task is CONSTRUCTed, and another user variable, DTOP, points to the end of the task's memory area.

8 GLOSSARY

(PAUSE) (__..__)

The multitasker. Jumped to from the header of a task that is about to be activated.

(WAIT) (n..__)

The normal routine vectored to by WAIT. Causes the task executing it to PAUSE 10 * N times.

ACTIVATE (n..._)

Used inside a colon definition to activate a task. Eg

: GO taskname ACTIVATE some.verbs ;

will cause the task named taskname to become active and start executing some.verbs. The task executing GO does not execute past ACTIVATE.

BACKGROUND (n1 n2..__)

Used to create the header and allocate memory for a background task with n2 User Variables, and a combined return and parameter stack size of n1 words. Used in the form

n1 n2 BACKGROUND name

When executed, name will leave the address of the start of the task's header. Name 1+ will leave the address of the task's status byte. See BUILD .

BUILD (n..__)

Used to initialise a newly created background task. Must be executed before ACTIVATE. Used in the form

name BUILD

where name is the name of the task.

CONSTRUCT (n..__) Used to initialise a newly created terminal task. Must be executed before ACTIVATE. Used in the form

name *CONSTRUCT*

where name is the name of the task.

FREE? (n..f) Used to determine whether another task is using a shared facility. Used in the form

DISK FREE?

Leaves a true flag if the *facility is available, else* leaves a false flag.

GET (n..__) Used to obtain the use of a shared facility. If the facility is being used by another task, the requesting task will be paused until the facility becomes free.

GRAB (n..__) *Unconditionally takes control* of a shared facility. Used by GET.

MAKE.OPERATOR (__..__) Used during system startup to create OPERATOR.

ME (__..n) A system register in zero page containing the address of a jump to the header of the next *task in the queue*. PAUSE jumps indirectly via this location. The address in ME is a fixed offset from the current task's status byte and this fact is

used by STATUS to return the address of the current task's status byte.

PAUSE (__..__) Releases control of the CPU to the next task in the queue.

PBRANCH (__..__) The primitive used by REBEGIN. Takes the address compiled in-line and puts it in IP.

REBEGIN (__..__) Analogous to AGAIN, and used in the same way, with the difference that REBEGIN causes the task to PAUSE before returning to BEGIN. Used in the form

```
BEGIN some.verbs REBEGIN
```

to create an endless loop executing some.verbs.

RELEASE (n..__) Used to release control of a shared facility. Used in the form

```
DISK RELEASE
```

REMOVE (n..__) taskname REMOVE will remove the task called taskname from the round-robin. It is essential to do this before FORGETting a task.

RENDEZVOUS (n..__) Used to synchronise two tasks. Task-A executing

```
TASK-B RENDEZVOUS
```

will then PAUSE until TASK-B executes

```
TASK-A RENDEZVOUS
```

RESTART (__..__)

The last verb executed by the cold and warm start routines. Executes the latest version of (RESTART), which can be modified by defining a new (RESTART) and putting the CFA of the new verb into the PFA of RESTART. Usually, the new (RESTART) will execute the old one before the additions.

SHADOW (n..__)

Used to create a shadow task.

address SHADOW name

will create a task, name, which will execute the machine-code subroutine at address each time the multitasker round-robin activates name. The subroutine must end in an RTS,

SHADOW.CODE (__..__)

An ASSEMBLER macro used by SHADOW to create the header code of a shadow task.

STOP (n..__)

Puts the value on the stack into STATUS, and PAUSES. A non-zero value will cause the task to stop.

TASK.CODE (__..__)

An ASSEMBLER macro to create the header code of a background or terminal task. Used by TASK.CORE.

TASK.CORE (n1 n2...a)

Used by BACKGROUND and TERMINAL to create a task header. Leaves the address, A, of the buffer area. Takes the number of user variables, N2, and the size of the stacks, N1.

TERMINAL (n1 n2 n3...__) Creates a terminal task with a dictionary of n1 bytes, a stack buffer of n2 bytes, and n3 user variables. Used in the form:

n1 n2 n3 TERMINAL name

Executing name will return the address of the start of the task's header. The task's status byte is the second byte in the header (ie name 1+). Must be followed by CONSTRUCT.

WAIT (n .._) Waits a time determined by n (often n milliseconds). In systems without a realtime clock will pause 10*n times.

APPENDIX C

SOFTWARE DETAILS

1 INTRODUCTION

This appendix describes the details of the primary software modules that make up the system. The overall structure of the system is described and the methods used to overcome the major design problems is set forth. The emphasis is on description of key aspects of the system software rather than on the actual source code. No attempt is made to exhaustively describe the inner workings of FORTH itself; the methods used are not language dependant. Nonetheless, some of the examples are necessarily fragments of FORTH code. These have been extensively commented. Where special extensions or modifications to the FIG Forth model have been made, these are described briefly.

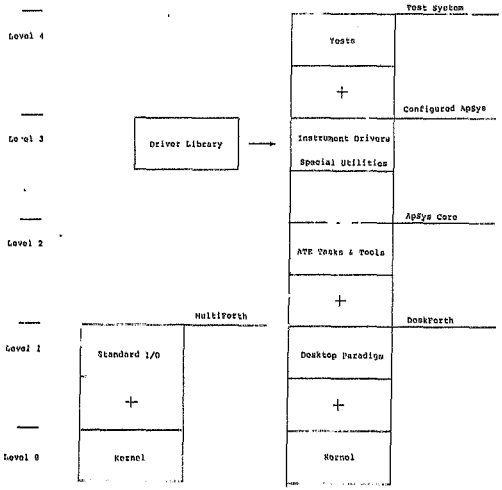


Figure C2: Software Structure

2 SOFTWARE STRUCTURE

The overall system was built using a step by step, or *layered approach. This allowed the debugging* of the software to proceed in manageably small increments, while also ensuring that the effects of changes could be easily gauged from their effect on the interface between each layer. The layers are shown diagramatically in figure C2. It is convenient to use names for the various layers. These are listed here.

- MultiForth...a multitasking version of FIG Forth, used in level-C.

- The DeskForth Kernel... a package of the most primitive FORTH verbs and utilities. In combination with an I/O package and other tools, such as an editor, it can be used to form a bootable FORTH *system. MultiForth consists of the* Kernel together with an editor and a standard set of I/O primitives.

- DeskForth ... a multitasking version of FIG Forth which has a desktop paradigm and menu driven operating system in place of the usual user interface. It consists of the Kernel plus the desktop user interface. Synonymous with level 1.

- ApSys .... an acronym derived from "Apple" and "Test System" which has evolved to describe the entire test software system. Level 2 is called the ApSys

Core, while level 3 is referred to as a Configured ApSys. The ApSys Core consists of DeskForth together with the ATT support tasks. The configuration process, which results in a Configured ApSys, consists of adding utilities to the core to allow the control of the facilities of a particular ATE.

- UUT Test System....this is the final configuration and consists of the Configured ApSys plus the suite of tests required to test a particular UUT.

3 THE KERNEL DEVELOPMENT SYSTEM

3.1 MultiForth

One of the first requirements that was addressed was the development of a multitasker. This culminated in a complete Forth system, based on the FIG model, with multitasking enhancements. The differences between MultiForth and FIG Forth are described in Appendix B. MultiForth was used to develop DeskForth. The source code was generated using the MultiForth editor, and each verb tested under MultiForth before the entire system was compiled.

The choice of Forth as the system development language was made in an earlier phase of the development. The reasons for this choice are repeated briefly here:

- Forth is very compact, an important consideration in keeping the system cost down.
- Despite its compact nature, Forth is fast.
- The availability of source listings and meta-compilers meant that the Forth kernel could be modified to efficiently suit the system's requirements.
- It was available on the Apple II computer.
- Forth lends itself to context sensitive software development, a requirement which had been identified

3 THE KERNEL DEVELOPMENT SYSTEM

3.1 MultiForth

One of the first requirements that was addressed was the development of a multitasker. This culminated in a complete Forth system, based on the FIG model, with multitasking enhancements. The differences between MultiForth and FIG Forth are described in Appendix B. MultiForth was used to develop DeskForth. The source code was generated using the MultiForth editor, and each verb tested under MultiForth before the entire system was compiled.

The choice of Forth as the system development language was made in an earlier phase of the development. The reasons for this choice are repeated briefly here:

- Forth is very compact, an important consideration in keeping the system cost down.
- Despite its compact nature, Forth is fast.
- The availability of source listings and meta-compilers meant that the Forth kernel could be modified to efficiently suit the system's requirements.
- It was available on the Apple II computer.
- Forth lends itself to context sensitive software development, a requirement which had been identified

as crucial in the design of an English-like test language.

- The risks involved in developing a multitasker for Forth were considered less than for other contenders, such as Pascal.

3.2 The Meta-Compiler

The Forth kernel source is traditionally written in Forth. This means that existing Forth system must be used to compile it. A proprietary meta-compiler was purchased for this purpose. The meta-compiler can compile itself into any location in memory, and can be used to compile the ApSys kernel into another place in memory. In fact, only the bare minimum for a working Forth system was compiled this way, the Kernel itself being used to compile the rest.

3.3 Debugging the Kernel

It was anticipated that the Kernel would be the most difficult part of the system to debug. This was because the operating system itself was being debugged; errors would invariably cause the entire system to crash catastrophically. For this reason it was decided to debug the Kernel in as minimal a configuration as possible, and to get it working reliably before the comparatively unknown and risky desktop operating system

was compiled. This approach was supplemented by testing all the kernel primitives individually before the Kernel was compiled as a system. As a result of this approach the Kernel ran the first time it was compiled by the meta-compiler. Since the behaviour of the Kernel was designed to be similar to a standard Forth, bugs were easily identified, and the errors found were usually trivial mistakes, rather than conceptual blunders.

4 THE KERNEL RUN SYSTEM

This consists of the minimal bootable system compiled using the meta-compiler, together with tools and utilities used by the later layers, all running under a temporary user interface.

4.1 Temporary I/O

In order that the kernel could be debugged seperately from the desktop operating system, a set of standard I/O primitives was included temporarily in the Kernel. This was facilitated by making use of one of the characteristics of MultiForth; all I/O primitives are called via vectors contained in each task's User Variable table. The temporary I/O verbs were compiled physically seperate in memory from the main dictionary, and when they were no longer required they were simply discarded, the system I/O being vectored to the new desktop operating system.

4.2 Virtual Memory

Much of the ApSys system is not needed in main memory at any particular time. An example of this is the dozens of menus associated with the writing aid, which are probably not being used at run time. Another example is the Assembler, which is very rarely used at any level higher than level-C. These parts of the system are kept

on the disk until needed. A set of tools was developed to make use of the disk as virtual memory. These include virtual arrays, byte fetch and store verbs, and block moves between main and virtual memory. The virtual memory is configured as 64 Kilobytes of byte addressable memory, plus a further 10 blocks of page addressable memory. These latter are used as buffers for the pages on the desktop, and similar applications. They are 1024 bytes in size. The entire virtual memory space is paged as required into 8 Kilobytes of main memory.

4.3 Headerless Definitions

The Forth dictionary consists of a series of verbs. Each entry in the dictionary consists of the verb header and the verb body. The header contains information used by the compiler when the verb is used by a higher level verb, and by the outer interpreter when a verb is used interactively from the keyboard. The body is simply a list of addresses of the verbs that comprise that particular verb.

Naturally, in a complete system the only headers that are required are those of the verbs that are going to be called interactively. The vast majority of verbs are called by other verbs, and not interactively from the keyboard. The name of the verb is therefore only used when the system is being compiled. In many turn-key systems, no verbs will be called interactively; the

system simply boots and performs its task. Many of the headers are therefore of use only for a short time and are simply a waste of space after that.

Another version of Forth, PolyForth (a *registered* trademark of Forth Inc), has a facility which enables the bodies and headers of such verbs to be compiled into seperate memory blocks. When the headers are no longer required, they can be discarded. Although the PolyForth *dictionary* structure is signifigantly different from that of FIG Forth, the concept remains valid and implementable. Tools supporting *headerless verbs were* therefore developed and included in the kernel.

4.4 Arrays

The desktop operating system makes use of arrays in several places. The array facility was developed as a generalised facility for creating and accessing arrays of bytes, words or other arrays. Boundary and type checking is done. Arrays of two or more dimensions are addressable as arrays of arrays, and this can have signifigant speed advantages.

4.5 The Printer

One of the most obvious advantages of having a multitasking operating system is that the printer need not hold up the rest of the system. Normally a printing

task spends a large proportion of its time simply waiting for the printer. In a multitasking system, the computer can be doing something useful during that time. The printer task is implemented as a full terminal task, except that it has no keyboard. The printer task can do everything that a normal terminal task can, and thus one can send it high level commands and it will perform all the disk access, calculating or whatever else, that is required.

5 THE DESKTOP OPERATING SYSTEM

5.1 Modeless Systems

One of the design goals of ApSys was that it should avoid frustrating the user as much as possible. One common cause of operator frustration and resentment is the phenomenon known as modality.

A simple example of a mode, found in editing applications, is the insert vs overstrike mode. In insert mode, striking a key causes all text starting from under the cursor to the end of the line to move one space to the right, the key struck is placed into the resulting blank space, and the cursor moves one space to the right. In overstrike mode, the struck key simply replaces the character under the curgor, and the cursor moves one space to the right.

Each mode has advantages in certain circumstances. In practice however, most people choose one mode and stick to it for most of their work, since the continuous frustration of typing a key and seeing the wrong thing happen, due to being in the wrong mode, has greater weight than the extra keystroke required at times when only one mode is used.

Even a simple modal effect like the one described causes resentment when it interferes with an automatic process, such as typing.

Modality is even more frustrating when it appears under the guise of the "exclusive mode". An exclusive mode of operation is one which prevents the user from accessing a system facility while that mode is in effect.

Imagine a person editing a file on a word-processor. He wishes to append another file, say an appendix, or even a series of appendices, to the present document. However, he has forgotten what the filenames of those appendices are. He must leave edit mode, and enter *command interpreter mode, in order* to get a catalog of the files on disk. There is a whole list of appendices, so he creates a file and lists all the appendices into that file. Then he goes to print and *format mode and* prints the file containing all the appendix filenames. Then he returns to the command interpreter mode and ...can't remember the filename of the original document. He gets a catalog of files on disk and loads the file and enters edit mode...to discover that he forgot to save the latest changes before finding out the appendix names. Luckily, the system always saves a temporary copy of the last file used, so he exits edit mode, and returns to command interpreter mode and loads the temporary copy.He then goes back to edit mode, *and...foiled again...*it kept a copy of the last file he worked on, and he has before him a list of appendices!

The problem described above is caused by the exclusiveness of each system mode. One cannot for

example get a disk catalog while in edit mode.

This is circumvented in DeskForth by having all the commands available at once. Certain commands are apparently mutually exclusive but this is got around by one of two methods. Either they are executed by seperate tasks, or else their output is sent to seperate places, or both.

Each of the menus in the operating system represents a group of commands that are often modal in classical operating systems. For example, if one is editing a Pascal file under the UCSD p-system, which is a very common operating system, it is not possible to get a catalog of the disk files without leaving the editor. The editor is an application that has exclusive use of the system while it is running. Thus modality is required because only one application can run at one time.

Even if the editor were able to get a catalog of disk files, a second problem would arise. Either the catalog would have to be written over the program that is being edited, or the editing would have to be suspended while the catalog was being displayed. This would make it difficult to use the data in the program without actually retyping it in by hand. The data would be available, but not as useful as it might be.

The analogous situation in this system is editing either a Forth source program, or a test program. This is done

by writing the software on one of the pages on the desktop, and using the editor menu as required. Another application, the text writing aid, is available, if it has been compiled into the system, simply by accessing its menus. Even if a facility that is wanted has not been compiled into the system, this can be done at any time, simply by inserting the relevant disk, and compiling it. If a file index is needed at any time, this can be obtained by using the filer menu, and another page on the desk. Data obtained by any means can be "cut" from one page and "pasted" into another.

5.2 Non-Directive Systems

A directive system is one which tells the user what he must do, and then asks him if he's sure that that is what he wants. Many people feel secure in this kind of environment, but it can be a source of great resentment to others. DeskForth is a non-directive system that uses an UNDO command to provide security. In most cases selecting UNDO will reverse the effect of the last command. For example, while editing one might wish to delete a paragraph. A directive system might say

SELECT COMMAND : M)OVE, D)ELETE, K)ILL.....etc

one selects DELETE and the computer responds:

DELETE WHAT? ENTER LINE NUMBER(S):

After the line numbers have been entered, the computer

asks:

THIS COMMAND DELETES TEXT: ARE YOU SURE? (Y/N):

DeskForth avoids this, while providing a level of security that is at least as comprehensive. In order to achieve the same result as the above, in DeskForth one would point to the beginning and end of the text to be deleted (using the mouse), move to the edit menu and select DELETE (again using the mouse). Immediately the text to be deleted is selected, it is highlighted. As soon as DELETE is selected, the highlighted text is removed. However if one does have second thoughts, selecting UNDO replaces the text.

The actual effort involved in the two cases is about equal. In the first example, one types D 5..7 Y; in the second, one selects the beginning, then the end, then DELETE.

Apart from the resentment that the first method causes, the second method is also quicker to learn, since one can try things out without fear of the result being irreversible. The UNDO method allows you to change your mind after you have seen the result of your actions, and this is very useful.

Certain things are difficult to UNDO. An example of this is formatting a disk. Short of saving the entire contents of a disk in memory before reformatting, it is difficult to undo a format command. Most systems provide

a two step command to safeguard the user, usually via the FORMAT, ARE YOU SURE?, YES route. DeskForth also provides a two step command formula, for the same reason. However, the psychology is entirely different.

When reformatting a disk one first selects TRASH from the Disking menu. This command "trashes" the disk, ie makes it unusable. A message is given that the data disk has been trashed. Any attempt to use a trashed disk will result in an error message. This command is reversible (UNTRASH). Once a disk is trashed, it can be reformatted via the INITIALISE command. If one tries to initialise a good disk, a message is given to the effect that the disk is good, and should be trashed if it is to be reformatted. However, initialising a brand new disk does not require the trashing step, since the system accepts an unformatted disk as not containing good data.

So the user who is perfectly aware of what he is doing will select TRASH then INITIALISE, without having to answer impertinent questions. The unsure user will be able to select TRASH or INITIALISE secure in the knowledge that nothing too terrible can happen. Finally, if INITIALISE is selected in error, the system will not damage a disk with good data on it. If TRASH is selected in error, the error is immediately reversible, by simply selecting UNTRASH.

5.3 The Desktop Paradigm

The user interface is represented by a desktop paradigm. Pieces of paper can be put on the desktop, and they can written on, moved around, overlapped, etc. The system menus are also represented as little pieces of paper with lists of commands on them. They can also be moved around, arranged and overlapped on the desktop. The total desktop is bigger than the screen. A mouse pointing device is used to move a cursor around the desktop, and the screen moves to follow the cursor if it is about to move out of the viewing area. The mouse has a button on top, and this is used to select whatever it is that the cursor is pointing at. This may be text, a page, a menu, or an item in a menu.

5.4 Menus and Pages

Menus have become a well accepted way of presenting the available options to the user. They are particularly helpful while learning a system, and to the occasional user.

The normal way of inputting data to the system is by writing it onto pages that are lying on the desktop. Most data that the system outputs to the screen is also written on a page on the desktop. Thus each page can be likened to the screen of a more classical computer system. The advantage here is that there are several screens available without the hardware cost of several

terminals.

The page that the user is writing on at any particular time is always the topmost of all the pages on the desk. Any page can be brought to the top of the pile by selecting it with the pointing device.

The pages are physically mapped to disk based virtual memory. Data concerning where they are on the desk, whether they are currently displayed or not, where they are kept in virtual memory, and so on are kept in an array. This array is indexed to a smaller array known as the page queue which specifies the order in which they are to be put onto the desk.

Moving a page around the desk involves changing its position data. If a page is moved out from under another page and placed on top of it, this is reflected in the page queue.

Creating the desktop image involves placing all the pages on the desk. First the area of memory that corresponds to the desk image is filled with the character that represents the surface of the desk. The pages are then put onto the desk by moving the data from the "real" (disk based) page to the desktop memory. The order in which this is done is determined by the queue. The position of each page on the desk is obtained from the main page array. The topmost of any overlapping page is written last, thus automatically overwriting any obscured data. This task is known as "building" the

desktop.

Building the desktop is a reasonably time consuming process (taking typically several hundred milliseconds to achieve), and so it is only undertaken when necessary. The task that rebuilds the desktop can be prompted to do so by using the command PROMPT. This command is embedded in those procedures that cause changes to the desktop.

The area of the desktop that is displayed on the screen is determined by a coordinate variable. The screen image is moved from the desktop image to the video memory (the Apple video output is a character mapped memory system) whenever any change to the screen image occurs. This is very fast (a few milliseconds).

It is not necessary to rebuild the desktop unless something happens that changes the desktop radically. For instance, if the operator move the cursor around the desktop, the desktop itself does not change. All that is necessary is that the screen memory be refreshed to display the correct portion of the desktop. Thus moving the screen "window" around the desktop can be accomplished swiftly and smoothly. Similarly, typing onto a page does not require the desktop to be rebuilt. The character is placed onto the correct place on the desktop (as well as onto the "real" page) and the screen refreshed. However, if a page is moved then the desktop must be rebuilt.

The menus are represented by small pieces of paper with writing on them. They are always "on top" of any of the other pages on the desk. They are also kept in virtual memory, and the ways in which they are displayed and manipulated are very much the same as for the other pages in the system.

Although the pages and menus are kept physically on disk, the disk buffer is large enough to accommodate all the desktop data in main memory at once. Rebuilding the desktop therefore does not usually involve actual disk access.

The method described for creating and displaying the desktop paradigm results in a very fast system response. This is achieved at some cost in memory efficiency. Data corresponding to a page on the desktop may be duplicated in several places....the disk itself, the disk buffer, the desktop image buffer, and the screen memory. However, the success of the paradigm depends on a virtually instantaneous response to the pointing device, and so optimising for speed rather than memory efficiency was necessary.

8.5 The Editor

The commands in the editor menu act on text selected from the a page on the desktop. One will typically select a piece of text, and then select a command from

the editor menu.

- CUT This command removes the selected text from the page, leaving a blank space. The text is kept in a buffer. The analogy is with a physical piece of paper and a pair of scissors.

- PASTE Will paste a previously CUT piece of text at the selected position on the page.

- COPY Similar to CUT, but does not replace the text with blanks, leaving it on the page.

- DELETE Replaces the selected text with blanks.

- PHOTOSTAT Shorthand command to copy an entire page.

- CLEAR Shorthand command to delete an entire page.

- UNDO Restores the page to the condition it was in prior to the last editing command.

- *DO.IT Takes the selected text and interprets it as a command. Any Forth command may be executed in this manner. The verb is interpreted and executed by the OPERATOR task. The asterisk in front of the name indicates that the command requires data that it will interpret.

5.6 The Filer

The filing system is analogous to a simple loose-leaf box-file. Pages can be inserted and removed, and text

written on them. The first page in the file is a title page. Th following ten pages comprise an index, which is generated automatically from the first line of each page in the document. See Appendices F and G for examples. The disk in drive-2 is the file. Drive-1 is the system disk containing the virtual memory.

- OPEN.FILE This opens the file by accessing the disk in drive-2 and replacing the current page on the desk with the title page of the file. The normal editing commands work as usual on the file pages. There is no need to save changes. Like a real piece of paper, once a change has been made, it has been made. The normal editing UNDO, and a filing command, LAST.VERSION, provide security.

- CLOSE.FILE This closes the file, saving any pages in memory to disk, and replacing the contents of the current page with whatever was there before the file was opened..

- TURN.OVEP Analogous to turning over a page in a book, this command displays the next page in the file.

- TURN.BACK Similar to TURN.OVER, except displays the previous page.

- *TURN.TO Displays the page whose number has been selected.

- LAST.PAGE Turns to the last page in the file.

- INDEX Turns to page 1, which is the first page of the ten index pages.

- INSERT.PAGE will insert a page into the file, causing the current page and all subsequent pages in the file to be renumbered accordingly. The index will also be updated. The inserted page will, in general, not be blank (see REMOVE.PAGE).

- REMOVE.PAGE Removes the current page from the file, and renumbers all pages accordingly. The page displayed is the next one in the file, and will now have the page number the discarded page had. The removed page is not lost but can be recovered by selecting INSERT.PAGE. A series of pages can be removed and then inserted in another part of the file using this command.

- LAST.VERSION This command replaces the contents of the current page with whatever it had on it when it was first turned to, ie all editing done since the page was first displayed will be discarded.

- FILE.PAGE This command forces the pages in memory to be written to disk. It is never absolutely necessary to use this command, but it is a safeguard against loss of data in the event of a power failure during an editing session.

- COMPILE.PAGE This command causes the source on the page of the file currently turned to to be compiled.

Compilation can be continued from page to page by putting a compiler directive at the end of the page. This directive is implemented in the verb "FTO".

5.7 The Disker

The commands provided by the disker utility allow the user to *copy and format disks.*

- INITIALISE This command formats the disk in the data drive (drive-2). It will only initialise a disk that is *brand* new, or that has been explicitly trashed, using the TRASH command.

- BACKUP This command copies the disk in the system drive (drive-1) to the disk in the data drive (drive-2). This is the only case where the system drive contains a disk that is not the system disk. The standard nightmare of copying in the wrong direction is addressed by the same method as is used for INITIALISE; the system will only copy from a good disk to a trash disk.

- TRASH This command marks an initialised disk in an unambiguous fashion so that further attempts to use it as a data disk will raise an error. Unformatted disks are automatically considered as trash disks and so an attempt to trash an unformatted disk results in no change.

- UNTRASH Reverses the action of the TRASH command, ie

makes the disk usable again. Attempting to UNTRASH an unformatted disk will result in an error being raised.

5.8 Options

These commands allow the user to configure the system to his preferences, and to conform with the Apple configuration he is using. They have no effect on the data in the system or on the pages, and merely modify the display.

- 40.COLUMNS AND 80.COLUMNS These two commands change the display between 40 and 80 columns. Attempting to use 80 columns on an Apple II+, or on an Apple //e without an 80 column card, will result in every second character of the display being lost. The system is still fully functional under this condition and selecting a 40 column display will restore everything to normal.

- LIGHT.PAPER AND DARK.PAPER Certain people prefer white on black displays, and others prefer black on white displays. A black on white display requires a good quality monitor with a long persistence phosphor if excessive flicker is to be avoided. These two commands allow the user to choose the display he prefers.

- UPPER.ONLY AND LOWER.CASE The Apple II+ does not

allow the use of lower case letters, while the Apple //e does. The display can be forced to display everything as upper case by using the UPPER.ONLY command. Using LOWER.CASE will allow the display of lower case letters again.

- *DESK, *SIDES, AND *TOPS The aesthetics of the screen are determined to a large extent by the characters used to denote the sides of pages (*SIDES), the tops and bottoms of pages (*TOPS), and the surface of the desktop (*DESK). These can be altered by the user. These commands have an asterisk in front, denoting that the system will interpret the selected text to get the data it needs. To change the desktop surface to blanks, one could type 20 (the ASCII value for a blank) onto a page, select it and select *DESK. The desktop surface would then be a uniform blank. Alternatively, one could type ASCII +, select this and select *DESK. The desktop surface would now be covered in plus signs. Perhaps the best way however would be to type KEY, select that and then select *DESK. The Forth verb KEY waits for a keystroke and returns the ASCII value of the key struck. So selecting *DESK with KEY selected will cause KEY to be interpreted and whatever value it returns will be used as the desktop surface. The OPERATOR task will therefore wait for the next keystroke, and surface the desktop with the character chosen. All the verbs in the system which

are prefaced by an asterisk behave in this powerful fashion.

- PAGE.ONE, PAGE.TWO, AND PAGE.THREE The system can accommodate up to ten pages on the desktop at once, although this would be rather crowded. Normally three pages is sufficient. Three pages are provided as standard and they are named PAGE.ONE, PAGE.TWO, and PAGE.THREE. Selecting their names in the options menu causes them to be placed on or removed from the desktop. These commands toggle. The pages are always in existence, whether they are on the desktop or not. In the test system, a fourth page is available, the status page. This page contains information as to the system status and is continuously updated by the MONITOR task. However, most of the time the operator is not interested in the status, and he can choose not to have that page on the desktop, unless he needs it.

5.9 The Printer

As has been described before, the printer is controlled by a terminal task called PRINTER. Selecting items from the Printer menu causes the printer task to execute those commands.

- TOP.PAGE This will cause the currently selected page, which is always the topmost one on the desktop, to be printed.

- FILE This will cause the entire contents of the disk in the data drive to be printed.

- *PAGES This will cause the selected pages from the file to be printed. The parameters required are the first and last page numbers in the range to be printed.

- DESKTOP This will cause an image of the entire desktop to be printed.

- *DO.IT Behaves exactly the same as the *DO.IT in the Editor menu, but the commands are interpreted by PRINTER, not OPERATOR.

- COMPILE.PAGE Will compile source code from the current page in the file. The code is compiled into the Printer task's dictionary. This facility is usually used to compile drivers for different kinds of printer.

- *SPACING Sets the line to line spacing (single, double, etc)

- *LINES Sets the page length, in lines

- *COLUMNS Sets the page width.

- TOGGLE.LF Sets the option of a linefeed with every carriage return (toggles).

5.10 Operating System Tasks

The Operating System makes extensive use of the multitasker. The system tasks are listed here, and their functions are described briefly.

The Terminal Input task is a shadow task that monitors the keyboard. If a key is struck, this task gets the character and passes it to the Desktop Control task. It also receives data from the Mouse Control task.

The Desktop Control task is a background task that performs the small scale editing and cursor control functions, and also updates the [illegible] and menu arrays and queues. It is responsible for all the housekeeping functions involving the desktop. Any character that must be placed on a page is put there by this task, pages are moved around by this task and so on. It does not actually build the desktop, but alerts the Terminal Output task to do so when necessary.

Menu selections are also coordinated by the Desktop Control task. This task gets the menu item selected and sends it to the task whose function it is to act upon the selected command.

The Terminal Output task is a background task that performs the screen refresh. Whenever necessary it also rebuilds the desktop.

The Cursor task is a shadow task that maintains the cursor on the screen. It monitors the position of the

cursor and places the cursor arrow at the correct place in the video memory. If the cursor has moved off the screen, this task updates the screen coordinate variable and alerts the Terminal Output task to refresh the screen.

The Mouse Control task is a shadow task that is activated by an interrupt routine. It services the mouse interrupt, and updates the cursor position if necessary. If the select button has been pressed it alerts the Terminal Input task, which will in turn pass this information to the Desktop Control task.

The Operator task is a terminal task that acts upon commands from the EDIT, OPTIONS, FILE, and DISK menus. In addition it is the overall system master, and controls the startup of all the other tasks at system startup.

The Printer is a terminal task that receives its commands from the PRINT menu, and performs all printing actions.

In addition to the tasks listed above there are several more tasks that are concerned with the testing function, rather than the operating system. These will be described in the next chapter.

6 THE APSYS CORE

This level of the system contains the tools and tasks that are required to turn DeskForth into an ATE system. It provides the infrastructure around which the configured system is built.

6.1 Writing Tests

The test language itself was expected to reduce the number of conceptual errors in the test programs. However, the problem of coding errors (especially typing errors) was expected to become worse. For this reason a writing aid was required that would assist non-specialist personnel to use the right word, in the right place, and to spell it correctly.

In order that the writing aid should also be of use to experienced test writers and test engineers, it was set as a design goal that writing software using the aid should be at least as fast as writing it without the aid. Another factor that emerged during evaluation of the aid was that using the aid should also be perceived by the user to be easier than writing the test by hand.

6.1.1 The Writer Task

In order that the writing aid should not impose exclusive modes on the user, it was implemented as a seperate terminal task. The user can specify to the task

which page on the desktop to use. This may or may not be a page in a file. The user can write directly into a file, or he can write onto a page on the desk and later place it in the file.

6.1.2 The General Method

The user will start by selecting the *place on a page* where he wants to write the test. This he does by simply pointing at that place with the mouse and pressing the select button. He will then select the menu item BEGIN.A.TEST. The writer task will write

Begin Test ?

on the page, starting at the point selected. The question mark indicates that user supplied text is required, in this case the test name. The user can go immediately to that place on the page by using the mouse or the "text cue" key, and type it in, or he can leave it till later.

The user then starts writing the text body. The body consists of sentences. To write a sentence, he must first select NEW.SENTENCE. The cueing key will then take the user to the menu containing the classes of sentence that can be written, such as CONNECT, SETUP, CALCULATE and so on.

Selecting a class will cause the beginning of the sentence to be written to the page, usually simply the

name of the class, for example CONNECT.

```
Begin Test Example :-
  Connect
```

Note the indentation. This is done automatically by the writer task. Pressing the cue key will take the user to a menu containing the names of the things that can be connected, say for example a digital voltmeter (commonly called a DVM), a counter-timer, a data-acquisition and control unit (usually called a DACU), and perhaps a function generator. Selecting one of these will write the correct software on the page.

```
Begin Test Example :-
  Connect DVM
```

If at this stage the user decides that he has made a mistake and he really wants to use the DACU, he could delete "DVM" (using the normal editing facilities), select where he wants the writing aid to continue writing, and select DACU from the menu. (Note that the menu containing the instruments didn't go away to be replaced by the next menu when DVM was selected; the cursor only moves to the next menu when the user asks it to, by using the cue key).

Of course, since DVM was the last selection made, the same effect could be achieved by selecting UNDO and then DACU. Undoing only works however on the last menu selection, and so more major changes require the user to

use the editor. If for example he decides after finishing the connect sentence that he would rather SETUP the DVM first, he could use the editing facilities to move the connect statement down the page, select where the writing aid must write, and write the setup statement using the writing aid as usual.

Continuing with the original example, after the instrument to be connected has been chosen, pressing the cue key will take the user to a menu containing the UUT points that can be connected to the DVM, and one of these can be selected.

Begin Test Example :-
  Connect DVM to Amplifier Output

Note that the word "to" is inserted automatically. This kind of word is often used to make the software read easily. The final stage in writing the sentence is to again select NEW.SENTENCE which will put the sentence terminator ":-" on the page; one can then continue to write the next sentence in exactly the same way.

Begin Test Example :-
  Connect DVM to Amplifier Output :-

6.1.3 Operator Entered Text and Numbers

At times it is necessary for the operator to enter text or numbers by hand. The system indicates this by means of question marks in the case of text, and the "#"

symbol in the case of numbers. A special "text cue" key will take the cursor to the place on the page requiring text input. Alternatively, the user can elect to leave this until the whole test has been written.

6.1.4 A Sample Session

Appendix E contains a series of snapshots of the screen during a sample test writing session. With the exception of the test name, the comment-line on the result sheet, and the result number, the keyboard was not used to enter any text. Everything was done using the cue key and mouse. A benchmark was done to compare the time taken to write this sample test by hand (ie without using the menus), with that taken to write it using the menus. The results were

- By Hand - 120 *seconds*
- By Menu - 90 seconds

Both benchmarks were done by the *same person, who was* experienced in using the system. The results shown are the average of three runs. The typing speed achieved in the "by hand" test was in excess of 20 words per minute, which is probably faster than most occasional computer users will achieve. However, the speed obtained using the menus is attainable even by novice typists.

6.2 Running Tests

Once a test has been written, it must be compiled before it can be run. Once a suite of tests has been compiled, the operator may wish to run them individually, or as a suite. He may wish to have print-outs after every test, or after the suite is complete. He may wish to stop testing if a failure occurs, or simply get a printout and continue. The Runner task provides the tools the operator needs to test a UUT using a suite of tests written using the Writer task.

6.2.1 The Runner Task

Especially during debugging, the system user will want to be able to run tests while having a listing of the test he is running available to him. To prevent problems of the "exclusive mode" nature the runner task is a seperate terminal task. The user can tell the runner task to send its output, such as messages to the operator, to a particular page on the desktop, and keep a listing of the test being run on a seperate page.

6.2.2 Compiling Tests

Tests can be compiled from a page on the desktop, or from a file. When compiling from a file, the page being compiled is displayed on the desktop. Usually, the runner task is used to compile tests, but other tasks,

such as OPERATOR, can be used as well.

A test can be written over more than one page, as long as the compiler directive P.T.O is put at the end of each page except the last. If the compiler comes across a P.T.O., it "turns over" the page in the currently open file.

A test is not compiled into the main dictionary of Forth verbs, but is stored on the system disk in virtual memory. It is brought into the runner task's dictionary area when needed and linked into the RUNNING vocabulary.

6.2.3 Running Tests

The operator runs tests by selecting RUN.ALL.TESTS from the runner vocabulary, or by typing the names of the specific tests he wishes to run, selecting them, and selecting *RUN.SELECTED.TESTS. A tool that will assist him in this is the menu command CATALOG. If the operator is going to run certain tests or groups of tests on a regular basis, he can list the test names to a page on the desk, edit them into suitable groupings using COPY and PASTE, and then select groupings as required. Having several pages available on the desk, he can have the runner task output on one page, the system status on another, and the list of test names on a third.

6.3 System Status

The system status page shows things such as the date, the UUT number, the Operator's name, the name of the test currently being run, and the stop and print options curre y in force. Often, the user will want to change some of these options while a test is running, so the status display is controlled by a seperate terminal task.

6.3.1 The Monitor Task

The monitor task is a terminal task whose output is placed on the status page. MONITOR maintains a menu of options which can be changed by the operator at any time. MONITOR also keeps track of what RUNNER is doing and displays the name of the test currently being run.

6.3.2 The Status Menu

The status menu is the normal way of communicating with the Monitor task. Commands include SHOW.STATUS, which displays the status page; STATUS.OFF, which turns the MONITOR task off; *CHANGE.DATE, *CHANGE.UUT, and *CHANGE.OPERATOR which are used to change the displayed date, UUT number, or operator's name; PRINT.ON.FAIL, PRINT.ALL, PRINT.OFF, STOP.ON.FAIL, and STOP.OFF, which control conditions under which the system will stop or print a result sheet.

6.4 *Test Result Sheets*

These are printouts of all the test results in a suite. They contain a list of high limits, low limits, actual results and a comment line for each result. The high limits, low limits and comment lines are stored on the system disk when a test is compiled. The date, result, UUT number and how many times the test has been run on that UUT is stored when the test is run.

6.5 The IEEE 488 Bus

The IEEE 488 bus is an industry standard instrumentation bus. In the same way as the printer can hold up the rest of the system, the 488 Bus can hold the system up while it is waiting for an instrument. To avoid this, the 488 bus driver has been implemented as a shadow task. Since interaction on the bus is more complex than with a printer, a slightly different approach has been used here. Data to be sent over the bus in placed in a buffer by the runner task, while the 488 Bus task takes the data from the buffer and puts it on the bus. Incoming data is treated in a similar fashion, except that the roles of the tasks are reversed. Bus control is performed at all times by the runner task, who sets up the talkers and listeners explicitly before letting the 488 bus task talk or listen. None of this is visible to the test writer however, although it is important to the ATE configurer.

7 CONFIGURATION FOR A SPECIFIC ATE

Before ApSys can be used to test a UUT it must be configured for use with a specific set of instruments and a switching matrix, and any special verbs required must be added to the system.

7.1 *The Configuration Process*

Configuration takes place in phases. Usually an acceptance test instruction is written first. The ATE Engineer then decides on the instruments he needs and designs an ATE to suit the UUT. Sometimes several UUTs will be tested on the same ATE and the instrumentation will be chosen accordingly. At this point the instrument drivers can be written, or selected from a library. The interfaces and looms can now be designed and the switching configuration for each UUT decided on. This information is used to configure the software that controls the switching matrix. Finally, special and complex procedures required, if any, can be written by the ATE designer and added to the ApSys system. At this point, the specialist work is finished, and the process of turning the acceptance test instruction into test software is straightforward.

Later changes to the software will be easily accomplished in most cases. It is only when changes are required that need specialist attention by virtue of their complexity that an ATE specialist need be called

in.

7.2 Defining the Vocabulary

First of all the instruments are named. The classes of operation that must be allowed for are selected. The special verbs that are needed are named, with close attention being paid at all times to making the result look like English. The noise words that are required are chosen. The names of the DUT connection points, modes of operation and so on are chosen. Only when every word in the vocabulary is known, and sample sentences using them have been tried out for readability is actual implementation begun. Implementability is ensured by following a few simple rules. These are described later in the section dealing with the test language.

7.3 Configuring the Writing Aid

This consists of writing the verbs that will be used to write the tests, and creating menus to allow the user to access them. The first step is to create the menus. High level Forth tools are provided to help with this. For example, creating a menu with two instrument names in it would be done like this:

```
18 2 menu instruments
instruments menu.is Function-Generator Counter-Timer
```

The first statement creates a menu of 2 items, each item

being 18 characters or less in size. The menu is named "instruments". The second statement puts the text into the menu. The menu items will later be defined as verbs in the writing vocabulary, and must obey the rules governing the naming of Forth verbs.

Once all the menus have been defined, the verbs required to write the tests must be defined. These must in general do two things. Firstly they must write the appropriate thing on the page, and secondly they must set the cueing system to go to the next logical menu when the cue key is pressed. An example follows.

```
: Counter-Timer (Define a verb "Counter-timer")
  who.called? (finds out the class of sentence )
  ' setup  = if   c/t-setups
              else UUT.points
              endif menu.is.next (set cueing system )
  ." Counter-Timer " and.post.word  (write the software)
;
```

The construction ": Counter-Timer" is the way that Forth defines the start of a procedure named "Counter-Timer". When "Counter-Timer" is selected from a menu, this procedure will be executed.

The procedure "Counter-timer" first finds out whether it is being used to write a "setup" or a "connect" sentence. The verb "who.called?" finds out which class of sentence the word "Counter-Timer" is being used in. If it is "setup", then the next menu to be used will be

"c/t-setups", while if it is "connect" then the menu containing the things that the counter timer can be connected to will be called next, viz "UUT-points". Who.called? leaves a value on the stack. This is compared with the value associated with the setup menu.." ' setup = ". Depending on the result of this comparison a value corresponding to either "c/t setups" or "uut-points" is left on the stack. The verb "menu.is.next" takes this value and sets up the cueing system so that the correct menu will be displayed when the user presses the cueing key.

The second function of the verb counter-timer is naturally to write the correct software on the page. This is done by printing "Counter-timer" on the page, together with any noise words. The noise word "to" will be printed by "and.post.word" if the class of sentence is "connect".

A complete set of tools to configure the writing aid has been created. The example given is about the most complex situation that occurs, so generating the writing aid is reasonably trivial, as long as the programmer is familiar with Forth.

An example of a complete writing aid configured for an ATE with several instruments is given in Appendix G.

7.4 The Switching Matrix

In virtually all cases, the switching matrix is implemented in the form

Connect Instrument to UUT-point :-

The writing aid consists of a menu of instrument names, a menu of UUT points, and definitions for each instrument and UUT point. The instrument definitions are all similar to the one given in the previous section. The UUT points are defined even more simply:

: Amplifier.Output Primary.writer menu.ii ct ." Amplifier O
;

Implementation of the run-time verbs to drive the switching matrix is done next. The actual switching matrix driver is the verb ":-", as is usually the case in ApSys. The two parameters taken by ":-" will be the UUT point and the instrument to be connected. The actual implementation will vary from system to system. A common way of doing things is to use an array, one dimension of which is referred to the instruments, while the other is referred to the UUT points. The data in the array would then be the characters to be sent over the 488 Bus to close the right relays in the switching matrix. The instrument names and UUT points would be defined as constants in the vocabulary "connect". At run time, the instrument name would leave a number on the stack, as would the UUT point. The verb ":-" would pick up these numbers, use them as coordinates in the switching array, get the relevant data and send it over the 488 Bus.

An example of a switching matrix driver can be found in the DACU driver in Appendix F.

7.5 The Instrument Drivers

The instrument drivers are defined in a similar way to the switching matrix driver. Menus containing all the options for each instrument are created first, followed by write time verbs, and then the run time implementation is decided on. As has been shown, the writing aid is relatively trivial: each instrument is accommodated in a few lines of code. The run-time code is however much longer. Each instrument driver is written as seperately compilable module. These are kept as a library, so that configuring an ATE to use an instrument for which a driver has already been written is relatively simple. An example of such a driver is to be found in Appendix F.

7.6 Special Purpose Procedures

Special purpose procedures can be written either in the high level test language, or in Forth. They can be included in the writing aid by adding the relevant items to the menus, and creating the relevant write time verbs.

8 THE TEST LANGUAGE

In the earlier phases of the project, a specification for the test language was drawn up. This called for the language to be English like in vocabulary and syntax. It was required to be hardware oriented, ie it should refer explicitly to the UUT and to the instruments being used, rather than abstracts such as "Voltage". The language was further specified as being extendible and flexible; in fact each particular ATE system would have the language specially tailored for it, and extensions to suit each UUT would be supported.

The analysis then showed that such a language was possible if the words in the sentence could be context sensitive. In addition it was shown that the simple imperative sentence was the only type required, and that restricting the implementation to this type of sentence resulted in certain advantages.

8.1 CONTEXT DEPENDANCY

One of the characteristics of human speech which is not shared by classical computer languages is that the words in a sentence do not have single, simple meanings. In order to design a test language that looks like English it is necessary that the meanings of the words in a sentence be context dependant. In fact, it became apparent that the majority of the words in a sentence could be considered as having no other purpose than more

closely defining the context of the sentence.

8.2 THE SIMPLE IMPERATIVE SENTENCE Restricting the sentence structure to the simple imperative simplifies the implementation of the language. Since the sentence structure is defined, the words in the sentence can be used in the implementation to do things that do not correspond exactly with standard English grammar. For example, the first word in the sentence is always a verb. In ApSys, this word is used to provide an an overall context for the sentence.

8.3 SENTENCE STRUCTURE AND PARTS

The basic sentence structure is described here in a simplified form. The sentence consists of "contexts", "nouns", "modifiers" and a "verb". There are some variants on these, but they need not be discussed here. The words "noun" and "verb" as used here are only superficially similar in meaning to their counterparts in English grammar.

The contexts provide the global meaning of the sentence. They can be nested, each new context deriving meaning from the previous context and contributing meaning to the words which come after. The effect of contexts is restricted to the sentence they are used in.

The nouns in a sentence contain data, and can be considered as parameters that are passed to the verb. The meaning of a noun (ie what data it represents) depends on the context in force at that point in the sentence.

Modifiers are used to alter the effect or meaning of a word in the sentence. Modifiers differ from contexts in that they affect only one word in the sentence.

The verb in most sentences is the construction ":-" appearing at the end of the sentence. The meaning of the verb depends on the context in force at the end of the sentence. The verb performs the action that the meaning of the sentence requires.

For example the sentence

Switch the power-supply on :-

begins with a context, namely "switch". There will probably be several things that can be switched. This is defined by the noun "power-supply". The context of switching can be further refined to either be "switch on" or "switch off". The word "on" in the example is used as a nested context within the context of "switch". Thus the compiler is able to compile a specific procedure call to a routine that switches the power supply on. Note that the choice over whether or not the word "power-supply" should be implemented as a context or as a noun is reasonably arbitrary in this

sentence. "Power-supply" could as well be considered as a nested context refining the context from "switch" to "switch the power-supply". The decision is based on whether or not the data is best obtained at compile time or at run time.

8.4 COMPILE TIME BEHAVIOUR

At compile time the sentence is scanned from left to right. Contexts have effect at compile time, while the other words have effect at run-time. Thus, in the example given above, the compiler decides on which procedure to compile a call to depending on the context, which in this case is "switch on". The parameters that are passed to the procedure (in this case the "power-supply") lead to the decision as to what should be switched being made at run time.

If the word "power-supply" had been implemented as a context, then a specific routine to "switch the power supply on" would need to be available. Implementing the word "power-supply" as a noun allows a more general purpose routine to be used, one that can switch many things on.

Similarly, if the word "on" was implemented as a noun, then a very general purpose routine, one that could switch many things either on or off could be used. In general it can be said that using many nouns means using fewer, more general purpose, routines; using many

contexts implies a special to purpose routine for each context. Thus passing parameters in the form of nouns to general purpose routines leads to *space efficiency* (fewer routines need to be used). However, compiling specialised single purpose routines by using contexts at compile time means greater speed at run time. In the example given the compromise could be arrived at using the following logic;

- there are many things that can be switched, and so creating a seperate routine for each of these would be very inefficient; thus the thing that needs to be switched should be implemented as a noun;

- there are only two cases of how the the thing should be switched (ie on or off) and thus two simple routines can be written without a great space penalty, the on/off *decision can thus be made* at compile time, and contexts used for this decision.

Other factors could influence the choice. For instance if switching the power-supply on was in fact conceptually very different from switching something else, then it might make sense to implement the two conceptual types of switching as contexts.

At compile time the contexts act as directives to the compiler. They are in *fact implemented as* Forth Vocabularies. Invoking a context tells the compiler which vocabulary to use. A second context in the sentence must be in the vocabulary of the previous one.

Each successive context instructs the compiler to look at a particular vocabulary in preference to all others. The run-time nouns and the verb which are compiled are thus chosen from the vocabulary of the current context.

The construction ":-" at the end of the sentence is considered to be the sentence verb. Upon striking this construction, the compiler immediately compiles the procedure that is indicated by the current context; thus the context in force when the ":-" is encountered determines the action that sentence will produce.

8.5 RUN TIME BEHAVIOUR

At run time, the sentence behaves in the following fashion; nouns leave data on the parameter stack, modifiers change the data, and the verbs act on the data. In the example given above, the noun "power-supply" could be implemented as a constant with value equal to the number in the switching matrix of the relay used to switch the power-supply. This value is left on the stack at run time. The procedure that is compiled as a result of the contextual information in the sentence will then take that number and send the required command to the switching matrix over the IEEE 488 bus. If the word "on" had been implemented as a noun, then the procedure would receive two parameters on the stack, viz. the number of the power-supply relay and a flag telling it to switch the power-supply on rather

than off.

The action of a modifier can be seen in the following example.

```
SETUP THE POWER-SUPPLY TO 5 VOLTS :-
SETUP THE POWER-SUPPLY TO 5000 MILLIVOLTS :-
```

The words "5" and "5000" are nouns. Thus at run time the value 5 or 5000 will be left on the parameter stack. The modifier "volts" multiplies the value on the stack by 1000, while "millivolts" does nothing in this case. Thus the value is modified to always be in millivolts at run time. The two sentences are thus seen to be equivalent.

8.6 RULES WHEN CONFIGURING THE VOCABULARY

There are a few simple rules and guidelines to follow when configuring the vocabulary.

The sentences must be simple imperatives.

The final verb must be unique in the context in which it will be used.

Use contexts instead of nouns whenever the data is fixed at compile time, except where the number of optional parameters is very large.

Use complete sentences to specify the vocabulary.

Always insert the noise words (the, to, and, etc) that an English sentence would have. Since there is no way of

telling from simply reading the software which words are functional and which not, the rule should be to put them in at all times.

APPENDIX D

DESKFORTH COMMANDS

APPENDIX D

DESKFORTH COMMANDS

1 THE EDITOR

The commands in the editor menu act on text selected from the a page on the desktop.

- CUT This command removes the selected text from the page, leaving a blank space. The text is kept in a buffer.

- *PASTE Will paste a previously CUT piece of* text at the selected position on the page.

- COPY Similar to CUT, but does not replace the text with blanks, leaving it on the page.

- DELETE Replaces the selected text with blanks.

- PHOTOSTAT Shorthand command to copy an entire page.

- CLEAR Shorthand command to delete an entire page. with blanks.

- UNDO Restores the page to the condition it was in prior to the last editing command.

- *DO.IT Takes the selected text and sends it to the Operator task as a command. Any Forth verb may be executed in this manner.

2 THE FILER

- OPEN.FILE This opens the file by accessing the disk in drive-2 and replacing the current page on the

desk with the title page of the file.

- CLOSE.FILE This closes the file, saving any pages in memory to disk, and replacing the contents of the current page with whatever was there before the file was opened..

- TURN.OVER Analogous to turning over a page in a book, this command displays the next page in the file.

- TURN.BACK Similar to TURN.OVER, except displays the previous page.

- *TURN.TO Displays the page whose number has been *selected*.

- LAST.PAGE Turns to the last page in the file.

- INDEX Turns to page 1, which is the first page of the ten index pages.

- INSERT.PAGE Will insert a page into the file, causing the current page and all subsequent pages in the file to be renumbered.

- REMOVE.PAGE Removes the current page from the file, and renumbers all pages accordingly. The removed page is not lost but can be recovered by selecting INSERT.PAGE.

- LAST.VERSION This command replaces the contents of

the current page with whatever it had on it when it was first turned to, ie all editing done since the page was first displayed will be discarded.

- FILE.PAGE This command forces the pages in memory to be written to disk.

3. THE DISKER

- INITIALISE This command formats the disk in the data drive (drive-2). It will only initialise a disk that is brand new, or that has been explicitly trashed, using the TRASH command.

- BACKUP This command copies the disk in the system drive (drive-1) to the disk in the data drive (drive-2). The system will only copy from a good disk to a trash disk.

- TRASH This command marks an initialised disk in an unambiguous fashion so that further attempts to use it as a data disk will raise an error. Unformatted disks are automatically considered as trash disks.

- UNTRASH Reverses the action of the TRASH command, ie makes the disk usable again. Attempting to UNTRASH an unformatted disk will result in an error being raised.

4 OPTIONS

These commands allow the user to configure the system to his preferences, and to conform with the Apple configuration he is using.

- 40.COLUMNS AND 80.COLUMNS These two commands change the display between 40 and 80 columns.

- LIGHT.PAPER AND DARK.PAPER Selects a white on black display, or a black on white display.

- UPPER.ONLY AND LOWER.CASE The display can be forced to display everything as upper case by using the UPPER.ONLY command. Using LOWER.CASE will allow the display of lower case letters again.

- *DESK, *SIDES, AND *TOPS Used to change the characters used to denote the sides of pages (*SIDES), the tops and bottoms of pages (*TOPS), and the surface of the desktop (*DESK). The data required is the ASCII value of the character to be used in the display.

- PAGE.ONE, PAGE.TWO, AND PAGE.THREE Causes the selected page to appear on or disappear from the desktop (toggles).

5 THE PRINTER

Selecting items from the Printer menu causes the printer task to execute those commands.

- TOP.PAGE This will cause the topmost page on the desk to be printed.

- FILE This will cause the entire contents of the currently open file to be printed.

- *PAGES This will cause the selected pages from the file to be printed. The parameters required are the first and last page numbers in the range to be printed.

- DESKTOP This will cause an image of the entire desktop to be printed.

- *DO.IT Behaves exact same as the *DO.IT in the Editor menu, but the commands are interpreted by PRINTER, not OPERATOR.

- COMPILE.PAGE Compiles the source on the current page of the currently open file. The verbs will be compiled into the printer task's dictionary. This facility is used to compile drivers for different printers.

- SETUP Will send a pre-defined setup string to a printer to set the desired printer options.

- *LINES Sets the number of lines per page.

- *COLUMNS Sets the number of columns per page.

- *SPACING Sets the number of line feeds per carriage return.

- TOGGLE.LF Certain printers automatically do a linefeed on receipt of a carriage return, while others do not. This commands sets the printer task to send a linefeed character after every carriage return, or not.

9 CONTROL KEYS

The following control keys are used to perform small scale editing functions, and to move the cursor around the page or desktop.

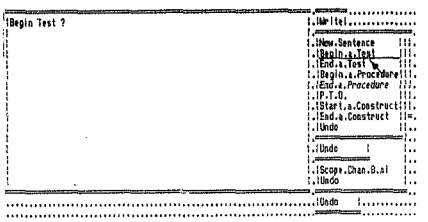
ESC - Equivalent to the button the mouse; the "select" key
DEL - Deletes the character pointed at by the cursor
TAB - Used to set tab positions
^A - Cues the cursor to the next menu
^W - Takes the cursor to the WRITE menu
^E - Takes the cursor to the EDIT menu
^R - Undefined
^T - Takes the cursor to the next text entry point
^Y - Undefined
^U - Equivalent to the right arrow key
^I - Equivalent to the TAB key
^O - Undefined
^P - Takes the cursor to the PRINT menu
^A - Closes a blank line of text. See ^Z.
^S - Takes the cursor to the STATUS menu
^D - Takes the cursor to the DISK menu
^F - Takes the cursor to the FILE menu
^G - Bell
^H - Equivalent to the left arrow key
^J - Equivalent to the down arrow key
^K - Equivalent to the up arrow key
^L - Undefined
^Z - Opens a blank line on the page. See ^A.
^X - *Undefined*
^C - Undefined
^V - Undefined
^B - Takes the cursor back to the previous menu
^N - Line feed
^M - Carriage Return
^Q - Undefined

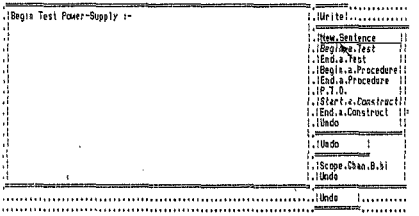
ARROW KEYS - move the cursor one space up, down, left, or right

APPENDIX E

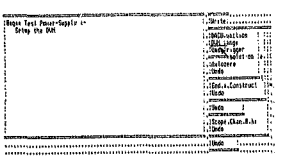
GENERATION OF A SAMPLE TEST

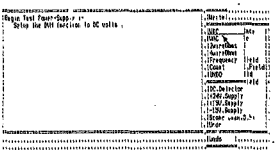
GENERATION OF A TEST

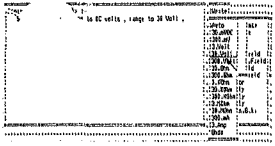


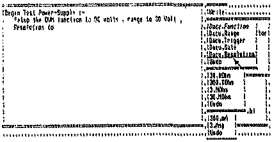


GENERATION OF A TEST

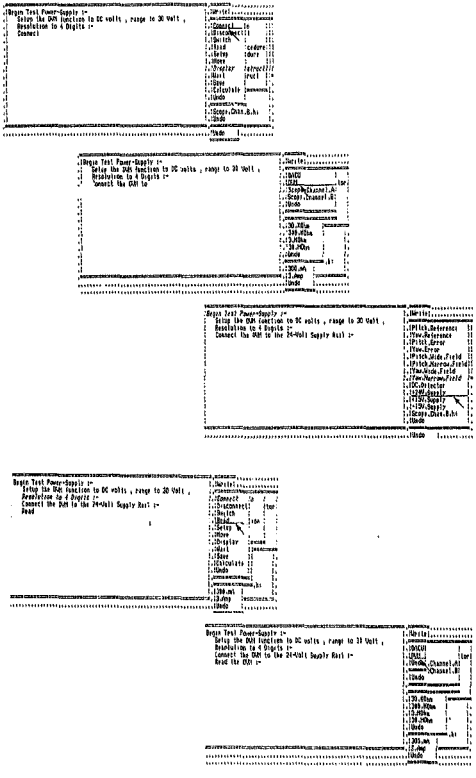


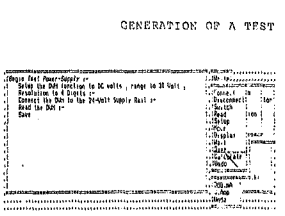


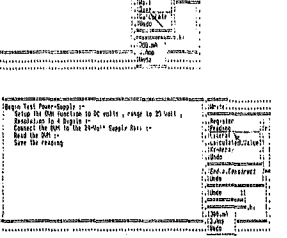


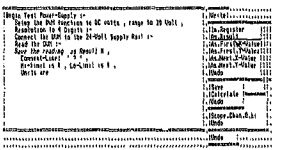


GENERATION OF A TEST




Begin Test Power-Supply :=
Setup the DVM function to DC volts , range to 30 Volt ,
Resolution to 4 Digits :=
Connect the DVM to the 24-Volt Supply Rail :=
Read the DVM :=
Save


Begin Test Power-Supply :=
Setup the DVM function to DC volts , range to 25 Volt ,
Resolution to 4 Digits :=
Connect the DVM to the 24-Volt Supply Rail :=
Read the DVM :=
Save the reading


Begin Test Power-Supply :=
Setup the DVM function to DC volts , range to 30 Volt ,
Resolution to 4 Digits :=
Connect the DVM to the 24-Volt Supply Rail :=
Read the DVM :=
Save the reading as Result N ,
Comment=Limit ' ' :=
min=limit is 0 , Lo=Limit is 0 ,
Units are


Begin Test Power-Supply :=
Setup the DVM function to DC volts , range to 30 Volt ,
Resolution to 4 Digits :=
Connect the DVM to the 24-Volt Supply Rail :=
Read the DVM :=
Save the reading as Reqs 1 1 ,
Comment=Limit ' 24 Volt Power Supply ' ,
Hi-Limit is 24.5 , Lo-Limit is 23.5 ,
Units are V :=
End Test :=

APPENDIX F

EXAMPLE OF AN INSTRUMENT DRIVER : HP 3421A DACU

HP-3421A DATA ACQUISITION AND CONTROL UNIT
LIBRARY SOURCE
Rev 8 : July 1985

Configured by : F J Duggan
Latest Change : P J Duggan

THIS FILE MUST BE COMPILED BY THE RUNNER TASK

(Text starts on page 11)

8

INDEX

1

INDEX

2

INDEX

3

```
( Memory Management          PJD 1/8/85 )

Running Definitions
Drunner
6 ' UN? !
hex
here fence !
Dtop @ 280 - dup dtop @ headermark 3421A-Heads
dtop !


decimal

pto
```

11

```
( Main Vocabularies    PJD 1/8/85 )

RUNNING SETUP DEFINITIONS
| VOCABULARY 3421A
RUNNING READ DEFINITIONS
| VOCABULARY 3421A
pto

There are no sub-vocabularies for SWITCH and CONNECT.
This utility is efficiently defined for systems using
a single DACU for switching.
There may be more than one DACU in the system, as long as
only one is used for switching...the DVM features are
extendible to several DACUS.
Use the Multiple-3421A  library file for systems with
more than one DACU used for switching.
```

12

```
( 488bus address  PJD 22/1/85                              *.* )
decimal
running 488bus definitions
( THE NAME OF THE FOLLOWING CONSTANT MAY BE MODIFIED; IF
  THERE IS MORE THAN ONE DACU IN THE SYSTEM, EACH MUST
  HAVE ITS OWN ADRESS; eg DACU).ADR, DACU2.ADR, ETC
  The actual adress is system dependant; set it here.      *** )


9  constant dacu.adr


pto
```

13

```
( Connect  PJD 28/2/84 )
RUNNING CONNECT DEFINITIONS

( NB This switching matrix is only usable for systems  with one
  DACU. Use the Multiple-3421A  library file for systems with
  more than one DACU USED FOR SWITCHING  )

2 BASE !
01111111 constant from immediate  ( disconnect from )
11111111 constant to   immediate  ( connect to )
pto
note: 9 bit number defines connection:
      6 lsbs define out point,
      bits 7,8 define instrument,
      msb defines connect/disconnect
```

14

```
( Connect Instruments  PJD 28/2/84                              *** )

( A reference must be included for each instrument              *** )
10011111 CONSTANT DACU IMMEDIATE
10111111 constant DVM immediate

: (channel : immediate
vocabulary scope immediate scope definitions
11011111 constant A)
11111111 constant B)
( eg ..connect scope [channel A)..... )

running definitions
pto
```

15

```
( connections matrix   PJD 18/1/85 )

connect definitions    decimal

VP 9  | CONSTANT MATRIX  512 2* VALLOT
( matrix contains the virtual address of the beginning of each
  string that must be sent for that channel to be closed )

: ; CHANNEL AND and 2* MATRIX + ;
( converts instrument, UUT point and to/from to matrix adress )
pto


pto
```

16

```
( connect  :-  PJD 18/1/85 )
connect definitions     decimal

| : (CONNECT) dup vc9 ( count )
    1+ v8  pad rot v)move  ( data string to pad )
    488bus dacu.adr
    listen.to.me pad count  send.string  ALL.SENT ;
( VADR.....  sends string )

( vadress from matrix, moves the string and its length
  to pad, and then sends it out from pad)

: :- 3 ?S# CHANNEL V2
  [COMPILE] LITERAL COMPILE (CONNECT) ; IMMEDIATE   pto

pto
```

17

```
( connect uut  PJD 21/5/85                                      *** )
connect definitions
( A reference must be  created for each UUT point               *** )

vocabulary pitch immediate
vocabulary yaw immediate
vocabulary detector immediate
: field ; immediate       ( noise )
: signal ; immediate      ( noise )
pitch definitions
11!000000 CONSTANT reference IMMEDIATE
11!000001 con+  ni error IMMEDIATE
11!000010 CONSI.NT wide IMMEDIATE
11!000011 CONSTANT narrow IMMEDIATE

pto
```

18

```
( connect uut  PJD 21/5/85                                        *** )

yaw definitions
111000100 CONSTANT reference IMMEDIATE
11100010! constant error IMMEDIATE
111000'18 CONSTANT wide IMMEDIATE
1110·    1 CONSTANT narrow IMMEDIATE
detec    definitions
111001300 CONSTANT ac IMMEDIATE
1110010w1 constant dc IMMEDIATE
connect definitions
: filter 1000 or ; immediate
( pitch nf  filter : 111001010  pitch wf filter :  111001011 )
( yaw nf filter    : 111001110  yaw wf filter   :  111001111 )

pto

                                                                    19

( connect uut  PJD 21/5/85                                        *** )
decimal
connect definitions
: volt ; immediate
vocabulary supply  immediate   supply definitions
: rail dup [ decimal ]  24 = if [ 2 base ! ]  11010000 else
       dup [ decimal ]  15 = if [ 2 base ! ]  11010001 else
       dup [ decimal ] -15 = if [ 2 base ! ]  11.010010 else
           [ decimal ]  5 error then then then  swap drop
; immediate
connect definitions
11101001  constant dvm-hi
vocabulary scope-hi immediate scope-hi definitions
11101010 constant B)

pto

                                                                    20

( connect ==)   PJD 28/2/84 )

connect definitions
: ==) channel up @ swap v! ( put vadress in matrix )
  () dup c@ 8 min 1+ up @ v@ rot dup vallot v!move
; ( channel-no ==) "string".... puts string in virtual
    memory, and points  matrix[channel-no] at the string )
  ( any character can be used for parentheses )


pto

                                                                    21

( connect data strings   PJD 26/6/85                              *** )
connect definitions
connect dvm to pitch reference ==)        "CLS02"
connect dvm to pitch error ==)            "CLS03"
connect dvm to yaw error ==)              "CLS04"
connect dvm to yaw  reference ==)         "CLS05"
connect dvm to pitch wide field ==)       "CLS06"
connect dvm to pitch narrow field ==)     "CLS07"
connect dvm to yaw wide field ==)         "CLS08"
connect dvm to yaw narrow field ==)       "CLS12CLS89"
connect dvm to detector DC signal ==)     "CLS13CLS89"
connect dvm to +24 supply rail    ==)     "CLS14CLS89"
connect dvm to +15 supply rail    ==)     "CLS15CLS89"
connect dvm to -15 supply rail    ==)     "CLS16CLS89"
connect dvm to scope-hi (channel b) ==)   "CLS17CLS89"
PTO

                                                                    22
```

```
( connect data strings   PJD 26/6/85                    *** )

connect scope (channel B) to detector AC signal ==>    "CLS02"
connect scope (channel B) to yaw reference      ==>    "CLS03"
connect scope  B) to pitch narrow field filter  ==>    "CLS04"
connect scope  B) to pitch wide field filter    ==>    "CLS05"
connect scope  B) to yaw narrow field filter    ==>    "CLS06"
connect scope  B) to yaw wide field filter      ==>    "CLS07"

connect scope (channel A) to dvm-hi             ==>    "CLS20"



PTO
                                                              23

( disconnect data strings   PJD 26/6/85                 *** )
disconnect definitions
disconnect dvm from pitch reference ==>        "OPN02"
disconnect dvm from pitch error ==>            "OPN03"
disconnect dvm from yaw error ==>              "OPN04"
disconnect dvm from yaw  reference ==>         "OPN05"
disconnect dvm from pitch wide field ==>       "OPN06"
disconnect dvm from pitch narrow field ==>     "OPN07"
disconnect dvm from yaw wide field ==>         "OPN08"
disconnect dvm from yaw narrow field ==>       "OPN12OPN09"
disconnect dvm from detector DC signal ==>     "OPN13OPN09"
disconnect dvm from +24 supply rail    ==>     "OPN14OPN09"
disconnect dvm from +15 supply rail    ==>     "OPN15OPN09"
disconnect dvm from -15 supply rail    ==>     "OPN16OPN09"
disconnect dvm from scope-hi (channel b) ==>   "OPN19OPN09"
PTO
                                                              24

( disconnect data strings   PJD 26/6/85                 *** )

disconnect scope  B) from detector AC signal          ==> "OPN02"
disconnect scope  B) from yaw reference               ==> "OPN03"
disconnect scope  B) from pitch narrow field filter   ==> "OPN04"
disconnect scope  B) from pitch wide field filter     ==> "OPN05"
disconnect scope  B) from yaw narrow field filter     ==> "OPN06"
disconnect scope  B) from yaw wide field filter       ==> "OPN07"

disconnect scope (channel A) from dvm-hi              ==> "OPN20"

connect definitions forget ==>


PTO
                                                              25

( switches  PJD 28/3/85  )
( NB This switching matrix is only usable for systems  with one
  DACU. Use the Multiple-3421A  library file for systems with
  more than one DACU USED FOR SWITCHING )

RUNNING switch DEFINITIONS

2 BASE !
1111 CONSTANT ON  IMMEDIATE
0111 CONSTANT OFF IMMEDIATE
pto

( 4 bit number.... hi bit determines on/off          )
(                  lo bits determine which switch    )


                                                              26
```

```
( switch names PJD 28/3/85                                  *** )
( each switch must be named to suit the application         *** )

1001 constant goniometer immediate
1010 constant modulator immediate
1011 constant light-source   immediate
1100 constant narrow-field
1101 constant power-supplies
: boresight ; immediate
vocabulary to immediate to definitions
: narrow [compile] narrow-field [compile] on ;  immediate
: wide [compile] narrow-field [compile] off ;  immediate
: field [compile] switch ; immediate pto
```

27

```
( switch on/off special names PJD 28/3/85                   *** )
(
  If words other than on or off are to be used to define
  switch position, then these must be defined here           *** )

: boresight ; immediate  ( noise )

vocabulary to immediate
to definitions   ( context for special names )

: narrow [compile] narrow-field [compile] on ;  immediate
: wide [compile] narrow-field [compile] off ;  immediate

: field [compile] switch ; immediate  ( restore context )
( eg switch to narrow field :- = switch narrow-field on :- )
pto
```

28

```
( switches matrix and :- PJD 28/3/85 )

decimal
switch definitions
VP 9  | CONSTANT MATRIX 16  2* VALLOT
: : CHANNEL AND 2* MATRIX + ;
: :- 2 ?56 CHANNEL VP
  [COMPILE] LITERAL connect COMPILE (CONNECT) ; IMMEDIATE
( matrix contains the virtual adress of the beginning of each
  string that must be sent for that switch to be thrown )
( see connect on previous pages )

pto
```

29

```
( COMPILER CONT switch ==>   PJD 28/3/85 )

switch definitions

: ==> channel vp 9 swap v! ( put vadress in matrix )
  () dup c@ 8 min 1+ vp 9 v@ rot dup vallot vmove
; ( channel-no ==> 'string'.... puts string in virtual
    memory, and points  matrix[channel-no] at the string )



pto
```

30

```
( switch data strings  PJD 28/3/85                          *** )

switch goniometer on              ==)   "CLS00"
switch goniometer off             ==)   "OPN00"
switch boresight modulator on     ==)   "CLS01"
switch boresight modulator off    ==)   "OPN01"
switch light-source on            ==)   "CLS11"
switch light-source off           ==)   "OPN11"
switch to narrow field            ==)   "CLS10"
switch to wide field              ==)   "OPN10"
switch power-supplies on          ==)   "CLS21"
switch power-supplies off         ==)   "OPN21"
switch definitions
forget ==)
pto
```

31

```
( setup  PJD 29/2/84                                        *** )

SETUP DEFINITIONS

( the following verb can be renamed; if there is more than
  one 3421A in the system, the verb must be duplicated;
  each dacu having its own name, eg dacu1, dacu2 etc        *** )

: dacu 488bus dacu.adr init.pad HP3421A ;  IMMEDIATE




pto
```

32

```
( setup sub-contexts   PJD 29/2/84 )

SETUP HP3421A DEFINITIONS

VOCABULARY FUNCTION IMMEDIATE
VOCABULARY RANGE IMMEDIATE
VOCABULARY TRIGGER IMMEDIATE
VOCABULARY RESOLUTION IMMEDIATE



PTO
```

33

```
( setup function   PJD 29/2/84 )


setup HP3421A FUNCTION DEFINITIONS
: to ; immediate   ( noise )
VOCABULARY DC IMMEDIATE
VOCABULARY AC IMMEDIATE
VOCABULARY 2-WIRE IMMEDIATE
VOCABULARY 4-WIRE IMMEDIATE

PTO
```

34

```
( setup function data strings  PJD 22/1/85 )

DC DEFINITIONS
F$ 2+  2 2constant volts immediate
F$ 10 +  2  2constant amps immediate

AC DEFINITIONS
F$ 4 +  2 2constant volts immediate
F$ 12 + 2 2constant amps  immediate

2-WIRE DEFINITIONS
 F$ 6 +  2 2constant ohms immediate
4-WIRE DEFINITIONS
F$ 8 +  2 2constant ohms  immediate

 pto
```



```
( setup range PJD 22/1/85 )

setup HP3421A range definitions
vocabulary 3 immediate
vocabulary 30 immediate
vocabulary 300 immediate

r$ 20 + 2 2constant auto immediate

3 definitions                immediate
r$ 2 2constant Volts         immediate
r$ 2 2constant Amps          immediate
r$ 4 +2 2constant Kohm       immediate
r$ 12 + 2 2constant MOhm immediate

 pto
```



```
( setup range data strings   PJD 22/1/85 )

30 definitions
r$ 24 + 2 2constant mVDC  immediate
r$ 2 + 2 2constant Volts      immediate
R$ 2 + 2 2constant Ohm    immediate
r$ 8 + 2 2constant KOhm      immediate
r$ 14 + 2 2constant MOhm     immediate

300  definitions
r$ 2+ 2 2constant mV  immediate
r$ 2+ 2 2constant mA  immediate
r$ 4 + 2 2constant Volts immediate
r$ 4 + 2 2constant Ohm  immediate
r$ 18 +  2 2constant KOhm immediate
pto
```



```
( setup trigger, resolution  PJD 22/1/85 )

setup HP3421A trigger definitions
t$ 2+ 2 2constant internal
t$ 4 + 2 2constant external

setup HP3421A resolution  definitions
:  digits ;  immediate
n$ 6 + 2 2constant 3 immediate
n$ 8 + 2 2constant 4 immediate
n$ 10 + 2 2constant 5 immediate

pto
```

```
( read   PJD 22/1/85                                          *** )
read definitions

( the following verb can be renamed; if there is more than
  one 3421A in the system, the verb must be duplicated,
  each dacu having its own name, eg dacu1, dacu2 etc
  and its own 488 bus adr eg dacu1.adr, dacu2.adr etc         *** )

: dacu 488bus dacu.adr  HP3421A ; immediate



pto
```

39

```
( read :-   PJD 22/1/85  )


read 3421A definitions


: :-  [compile] running definitions compile read.instrument ;
immediate
pto
```

40

```
( status/errors     PJD 04/02/85                              *** )
decimal
running 488bus definitions
( If there is more than one dacu in the system, each must
  have its own status variable                                *** )
0  variable dacu.status

( If there is more than one dacu in the system, the
  following verb may be duplicated to raise a different
  error for each dacu                                         *** )
: : dacu.error runner.task activate 53 error quit ;


pto
```

41

```
( srq serial poll PJD 04/02/85                                ***  )

( If there is more than one dacu in the system, this
  page must be duplicated for each dacu; the correct
  versions  of dacu.adr, dacu.status and dacu.error
  must be substituted                                         *** )
: : dacu.srq  dup if dacu.ADR spoll dup 64 and
                     if dup dacu.status c! 32 and
                        if dacu.error then  drop 0
                     else drop then
                  then ;

: (srq) (srq) dacu.srq ;

' (srq) cfa ' srq !
 pto
```

42

```
( activate dacu srq  PJD 04/02/85                              *** )
decimal
( If there is more than one DACU in the system, the
  following verb must be extended to send "M32" to
  each dacu; extend the verb, do not duplicate it.             *** )

running 488bus definitions
| : activate.DACUs
    dacu.adr listen.to.me send "M32" all.sent ;

: (activate.bus) (activate.bus) activate.dacus ;

' (activate.bus) cfa ' activate.bus !

pto
```

43

```
( last compiled page  PJD 22/1/85 )
hex

cr ." 3421A driver extends to " here u.
running definitions
here fence !
Killhead 3421A-heads
hex 208 dtop +!
here fence !
cr ." dtop restored to " dtop 2 u.
cr dtop 2 here - u. ." left in runner's dictionary " cr
RUNNING DEFINITIONS
decimal
! ' UN? !
```

44

```
( Syntax - READ DACU  PJD 1/8/85 )


 Read DACU :-
```

45

```
( Syntax - SETUP DACU  PJD 1/8/85 )
Setup DACU  Function to DC volts, Range to  3  \   Volt
                        AC Volts            30  -   mV
                                            300/

                        DC Amps, Range to  3 mA
                        AC Amps            30 mA
                                           300 mA

                        2-wire Ohms, Range to 3  \   Ohm
                        4-wire Ohms,          30  -  KOhm
                                              300/  MOhm ,

    Trigger to Internal , Resolution to 3 digits :-
               External                 4
                                        5
```

46

(Syntax - CONNECT PJD 1/8/85)

```
Connect Instrument-name to  UUT-point-name ;-
        UUT-point-name  to Instrument-name ;-

Disconnect Instrument-name from  UUT-point-name ;-
           UUT-point-name  from Instrument-name ;-
```

47

(Syntax - SWITCH PJD 1/8/85)

```
Switch   switch-name   on                          ;-
                       off
                       to user-defined-condition

         to user-defined-condition
```

48

APPENDIX G

EXAMPLE OF A CONFIGURED WRITING AID PACKAGE

Optical Sight Configuration
Write-time Source

Rev 3.1 : AUGUST 1985

Configured by : P J Duggan
Latest Change : P J Duggan

THIS FILE MUST BE COMPILED FROM THE OPERATOR

(text starts on page 11)

0

INDEX

1

INDEX

2

INDEX

3

INDEX

4

INDEX

5

```
( APSYS WRITER CONFIGURATION PJD 01/3/84 )

ASSEMBLER forth definitions ( to ensure assembler is in )


HEX 9000 dtop !
( 9000 ....B900 overlay area )
( B100.....B800 Writeheads )

pto
```

11

```
( Writer Task Overlay      PJD 28/1/85 )
hex
forth definitions
0 VARIABLE WRITER.DONE

: writer.overlay writer.task activate overlay?
  [ r# @ ] literal r# ! ( SET WRITER'S PAGE TO CURRENT TOP PAGE)
  400 OUT ! ( START WRITING ON BOTTOM OF PAGE )
  ( scr @ 2+ ) literal load.page
  operator writer.done rendezvous quit ;
( THIS VERB ACTIVATES THE WRITER TASK, WHICH THEN LOADS THE REST
  OF THE FILE FROM PAGE 14 ON )
pto
```

12

```
. Finalise Configuration      PJD 28/1/85 )

writer.overlay  ( START COMPILATION IN WRITER TASK )
writer.task writer.done rendezvous  ( WAIT UNTIL FINISHED )
forget writer.done ( CLEAN UP OPERATOR'S DICTIONARY )

here fence !
cr hex ." Operator dictionary extends to " here u.

decimal
forth definitions
cr ." writer configuration complete "
```

```
( WRITER MEMORY MANAGEMENT        PJD 28/9/84 )

writing DEFINITIONS

hex B100 B800 headermark writeheads
( NB top 100 bytes not recovered at boot time....)
B0FF dtop !
```

pto

14

```
(WRITER.MENU        PJD 28/9/84 )

writing DEFINITIONS

: WRITER.MENU
  WRITER.TASK i+ ( TASK STATUS BYTE )
  WRITER.TASK 13 + 0 A + 0 ( Terminal Input Buffer address )
  C/P 2+ 15  + ( initial horizontal position of menu )
  L/P 5 + ( initial vertical position of menu )   ;
( SETS UP PARAMETERS  FOR A MENU THAT IS ASSOCIATED WITH THE
  WRITER TASK )
```

pto

15

```
( PRIMARY WRITER MENU      PJD 28/9/84 )
writing DEFINITIONS



DECIMAL  17 9 ! MENU.STRING PRIMARY.WRITER

PRIMARY.WRITER menu.IS New.Sentence
 Begin.a.Test End.a.Test Begin.a.Procedure End.a.Procedure
 P.T.O.  Start.a.Construct End.a.Construct  Undo

WRITER.MENU PRIMARY.WRITER MENU
pto
```

16

```
( DVM CONNECTIONS MENU     PJD 28/9/84 )

18 14 ! MENU.STRING DVM.CONNECTIONS

dvm.connections menu.is   Pitch.Reference  Yaw.Reference
  Pitch.Error Yaw.Error Pitch.Wide.Field Pitch.Narrow.Field
  Yaw.Wide.Field Yaw.Narrow.Field DC.Detector
  +24V.Supply +15V.Supply -15V.Supply Scope.Chan.B.hi  Undo

WRITER.MENU  dvm.CONNECTIONS MENU



pto
```

17

```
( SCOPE CONNECTIONS MENU    PJD 28/9/84 )

15 7 ! menu.string scope.b.connections

scope.b.connections menu.is  AC.Detector Yaw.Reference
Pitch.NF.Filter Pitch.WF.Filter Yaw.NF.Filter Yaw.WF.Filter
undo

writer.menu scope.b.connections menu

6 2 menu.string scope.a.connections

scope.a.connections menu.is DVM.HI  undo

writer.menu scope.a.connections  menu
pto
```



```
( CLASSES MENU    PJD 25/7/85 )

10 11 ! MENU.STRING CLASSES

CLASSES menu.IS  Connect  Disconnect  Switch Read Setup
 Move Display Wait Save Calculate  Undo

WRITER.MENU CLASSES MENU


pto
```



```
( INSTRUMENTS  MENU   PJD 25/7/85 )

15 5 ! MENU.STRING INSTRUMENTATION

INSTRUMENTATION menu.IS  DACU DVM  Scope.Channel.A
Scope.Channel.B Undo

WRITER.MENU INSTRUMENTATION MENU

4 9 ! MENU.STRING bus.INSTRUMENTATION

bus.INSTRUMENTATION menu.IS  DACU DVM  Undo

WRITER.MENU bus.INSTRUMENTATION MENU

pto
```



```
( UNITS  MENU    PJD 25/7/85 )


4 6 ! MENU.STRING UNITS

UNITS menu.IS  V mV mRad Min Sec Undo

WRITER.MENU UNITS MENU



PTO
```

```
( CALCULATE MENU      PJD 28/9/84 )

11 7 ! MENU.STRING CALCULATE.WHAT

CALCULATE.WHAT menu.is Add Subtract Multiply Divide
Slope Y-intercept Undo

WRITER.MENU CALCULATE.WHAT MENU
pto
```

22

```
( SWITCH MENU     PJD 25/6/85 )
19 07 ! MENU.STRING SWITCH.WHAT

SWITCH.WHAT menu.is Goniometer  Boresight-Modulator
 Xenon-Source  To.Narrow.Field  To.Wide.Field Power Undo

WRITER.MENU SWITCH.WHAT MENU

04 03 ! MENU.STRING ON/OFF

ON/OFF menu.IS On Off Undo

WRITER.MENU ON/OFF MENU

pto
```

23

```
( WAIT MENU     PJD 28/9/84 )


08 03 ! MENU.STRING WAIT.FOR.WHAT

WAIT.FOR.WHAT menu.IS Time Operator Undo

WRITER.MENU WAIT.FOR.WHAT MENU


pto
```

24

```
( SAVE MENU     PJD 28/9/84 )

16 06 ! MENU.STRING SAVE.WHAT
SAVE.WHAT menu.IS Register Reading
   Literal Calculated.Value XY-Array  Undo
WRITER.MENU SAVE.WHAT MENU

16 07 ! MENU.STRING IN.WHAT
IN.WHAT menu.IS In.Register As.Result  As.First.X-Value
  As.First.Y-Value  As.Next.X-Value As.Next.Y-Value  Undo
WRITER.MENU IN.WHAT MENU
```

25

(CONSTRUCTS MENU PJD 28/9/84)

07 04 ! MENU.STRING DO..HAT

DO.WHAT menu.IS If Until N.Times Undo

Writer.Menu Do.What Menu

5 9 ! MENU.STRING IF.WHAT

IF.WHAT menu.IS > < = <> >= <= True False Undo

WRITER.MENU IF.WHAT MENU
pto

26

(MOVE MENUS PJD 4/10/84)
6 3 ! menu.string move.what

move.what menu.is Source Sight Undo

writer.menu move.what menu

13 5 ! menu.string move.where

move.where menu.is To.XY By.Hand To.Zero Incrementally Undo

writer.menu move.where menu

pto

27

(DACU SETUP MENU PJD 28/9/84)

15 06 ! MENU.STRING DACU.SETUPS

DACU.SETUPS menu.IS Dacu.Function Dacu.Range Dacu.Trigger
Dacu.Gate Dacu.Resolution Undo

WRITER.MENU DACU.SETUPS MENU

pto

28

(DACU RANGE MENU PJD 28/9/84)

11 12 ! MENU.STRING WHAT.DACU.RANGE

WHAT.DACU.RANGE menu.IS Auto 300.mVDC 3.Volts 30.Volts
300.VoltsDC 300.Ohm 3.KOhm 30.KOhm 300.KOhm 3.MOhm 30.MOhm
Undo

WRITER.MENU WHAT.DACU.RANGE MENU

pto

29

```
( WRITE MENU HEADER      PJD 9/8/84 )

writer.menu WRITE.bar menu
: e.x compile WRITE.bar [compile] [compile]
  compile do.bar ; immediate
: (Write) e.x dvm.connections e.x scope.a.connections
  e.x scope.b.connections e.x what.channel  e.x move.what
  e.x move.where  e.x instrumentation e.x bus.instrumentation
  e.x classes
  e.x units  e.x calculate.what e.x wait.for.what e.x save.what
  e.x in.what e.x on/off e.x do.what e.x if.what
  e.x dacu.setups e.x what.dacu.range  e.x what.dacu.function
  e.x digits e.x what.dacu.gate  e.x what.dacu.trigger
  e.x dvm.setups e.x what.dvm.range e.x what.dvm.function
  e.x what.dvm.trigger  e.x switch.what e.x primary.writer ;
pto
```



```
( WRITE DESKTOP UTILITIES      PJD 9/8/84 )
decimal
: write forth write writing (write) ;
( TURNS MENUS ON AND OFF, AND ENSURES OVERLAY IS IN )

: ^w write.bar goto.menu ;
23 ' ^w make.control
( SET CONTROL W TO GO TO WRITER MENUS )
hex
write

cr ." Menus extend to " here u. cr
decimal
pto
```



```
( WRITIN'margin+!' margin +! ; 0/84 )
WRITING DEFINITIONS
hex
: 'margin+! ' margin +! ;
( N.....MOVES MARGIN IN BY N SPACES..-VE MOVES LEFT )

0 ! variable ended?
( FLAG TO CONTROL INDENTING IN A TEST )
: end? ended? @
  if ."-" -3 'margin+! cr 0 ended? ! then write.here ;
( RESTORE MARGIN AT END OF A TEST )
: start end? ! ended? ! 3 'margin+! ;
( INDENT AFTER TEST HEADING )


pto
```



```
( COMMON STRINGS    PJD 9/8/84 )
: "." ." . " ;
: "-" ." - " ;
: "test" ." Test " ;
: "procedure" ." Procedure " ;
: "?" ." ? " ;
: "to" ." to " ;
: "DC" ." DC " ;
: "AC" ." AC " ;
: "the" ." the " ;
: "mSec" ." mSec " ;
: "function" ." function " ;
: "range" ." Range " ;

 pto
```

```
( DACU FUNCTION, GATE, TRIGGER MENUS      PJD 28/9/84 )

11 07 ! MENU.STRING WHAT.DACU.FUNCTION
WHAT.DACU.FUNCTION menu.IS  VDC VAC 2wireOhms 4wireOhms
 Frequency Count    UNDO
WRITER.MENU WHAT.DACU.FUNCTION MENU

08 04 ! MENU.STRING WHAT.DACU.GATE
WHAT.DACU.GATE menu.IS  100.mSec 1.Sec  10.Sec Undo
WRITER.MENU WHAT.DACU.GATE MENU

08 03 ! MENU.STRING WHAT.DACU.TRIGGER
WHAT.DACU.TRIGGER menu.IS Internal    External Undo
WRITER.MENU WHAT.DACU.TRIGGER MENU

pto
                                        30

( DVM RANGE, SETUP MENUS      PJD 15/1/85 )

3 16 ! MENU.STRING WHAT.DVM.RANGE
WHAT.DVM.RANGE  menu.IS  Auto 30.mVDC 300.mV 3.Volt 30.Volt
 300.Volt  30.Ohm 300.Ohm  3.KOhm 30.KOhm 300.KOhm
 3.MOhm 30.MOhm 300.mA 3.Amp Undo
WRITER.MENU WHAT.DVM.RANGE MENU

15 06 ! MENU.STRING Dvm.SETUPS
DVM.SETUPS menu.IS  DVM.Function  DVM.Range DVM.Trigger
 DVM.Resolution Autozero Undo
WRITER.MENU DVM.SETUPS  MENU

pto
                                        31

( DVM FUNCTION, RESOLUTION, TRIGGER MENUS      PJD 28/9/84 )

11 05 ! MENU.STRING WHAT.DVM.FUNCTION
WHAT.DVM.FU..CTION menu.IS  VDC VAC 2wireOhms 4wireOhms UNDO
WRITER.MENU WHAT.DVM.FUNCTION MENU

08 04 ! MENU.STRING DIGITS
DIGITS menu.IS  3.Digits 4.Digits  5.Digits Undo
WRITER.MENU DIGITS MENU

08 03 ! MENU.STRING WHAT.DVM.TRIGGER
WHAT.DVM.TRIGGER menu.IS  Internal External Undo
WRITER.MENU WHAT.DVM.TRIGGER MENU

pto
                                        32

( SCOPE CHANNEL MENU      PJD 28/9/84 )

13 03 ! MENU.STRING WHAT.channel

WHAT.channel menu.IS  channel.A  channel.B UNDO

WRITER.MENU WHAT.channel MENU

pto
                                        33
```

```
( MORE COMMON STRINGS )
: "volts" ." volts " ;
: "mvdc" ." mVDC " ;
: "ohm" ." ohm " ; : "Kohm" ." Kohm " ;
: "Mohm" ." Mohm " ;
: "3" decimal ." 3 " ;
: "30" decimal ." 30 " ;
: "300" decimal ." 300 " ;
: "digits" ." digits " ;
: "resolution" ." Resolution " ;
: "trigger" ." trigger " ;
```

38

```
( BEGIN AND END OF BLOCK   PJD 9/8/84 )

: BEGIN.A.BLOCK set.caller PRIMARY.WRITER MENU.is.next
  end? out @ c/p mod 3 + ' margin !
  start ." Begin " ;

: END.OF.BLOCK PRIMARY.WRITER MENU.is.next -3 'margin+!
  start ." End " ;

pto
```

39

```
( PRIMARY.WRITER   PJD 9/8/84 )


: P.T.O. primary.writer menu.is.next CH/P 6 - OUT ! ." PTO "
;
: BEGIN.A.TEST BEGIN.A.BLOCK "Test" "?"  ;
: BEGIN.A.PROCEDURE BEGIN.A.BLOCK "Procedure" "?" ;
: END.A.TEST end.of.block "Test" end? ;
: END.A.Procedure end.of.block "Procedure" end? ;
: new.SENTENCE CLASSES MENU.is.next start
  SET.CALLER ;

pto
```

40

```
( CALCULATE, SWITCH   PJD 9/8/84 )


: CALCULATE CALCULATE.WHAT MENU.is.next  SET.CALLER
  ." Calculate "  ;

: Switch SWITCH.WHAT MENU.is.next SET.CALLER ." Switch " ;




pto
```

41

```
( CONNECT, READ, SETUP   PJD 28/9/84 )

: CONNECT INSTRUMENTATION MENU.is.next   SET.CALLER
  ." Connect " POST.WORD IS "to " ;

: DISCONNECT  INSTRUMENTATION MENU.is.next SET.CALLER
  ." Disconnect " POST.WORD IS "from " ;

: READ bus.INSTRUMENTATION  menu.is.next SET.CALLER ." Read " ;

: SETUP bus.INSTRUMENTATION MENU.is.next SET.CALLER
  ." Setup " SET.COMMA     ;


pto
                                        42

( MESSAGES, WAIT    PJD 5/10/84 )

: DISPLAY  primary.writer menu.is.next ." Display Message "
  string.required ;

: wait wait.for.what menu.is.next set.caller ." Wait " ;

: time primary.writer menu.is.next ." for "
  value.required "mSec" ;
: srq primary.writer menu.is.next  ." for SRQ " ;
: Operator primary.writer menu.is.next
  ." until the operator responds " ;


          pto
                                        43

( SAVE    PJD 5/10/84 )
( : ,cr "," cr ;
: Save Save.what menu.is.next  SET.CALLER ." Save " ;
( : "hi-limit" ." Hi-limit " ;
( : "lo-limit" ." Lo-Limit " ;
( : "is" " is " ;
: AS.RESULT  UNITS menu.is.next  ." As Result " VALUE.REQUIRED
  ,cr  ." Comment-Line: " string.required  ,cr
  "Hi-Limit" "is"  VALUE.REQUIRED  ,cr
  "Lo-Limit" "is" VALUE.REQUIRED ,cr ." Units are "
  SET.CALLER ;
: IN.REGISTER PRIMARY.WRITER MENU.is.next ." in "
  REGISTER.REQUIRED ;
( : REG.MENU WHO.CALLED?  dup ' SAVE =
  IF in.what MENU.is.next
  Else primary.writer menu.is.next THEN set.caller ;   PTO
                                        44

( SAVE IN ARRAY   PJD 1/10/84)

( : "as.first" " as the First "  ;
( : "as.next" " as the Next " ;
( : "X-value" " X-value " ;
( : "Y-value" " Y-value " ;
: as.first.x-value  primary.writer menu  .next
  "as.First" "X-value" ;
: as.first.y-value  primary.writer menu.is.next
  "as.First" "Y-value" ;
: as.next.x-value primary.writer menu.is.next
  "as.Next" "X-value" ;
: as.next.y-value  primary.writer menu.is.next
  "as.Next" "Y-value" ;

                                  pto
                                        45
```

```
( PLOT ARRAY  PJD 11/1/85 )

| : "profile" v" Profile  ( X1 Y1 , X2 Y2 ,....Xn Yn ):"
  cr value.required space value.required ;cr ;
: XY-Array Primary.writer menu.is.next ." XY-Array as Result "
  Value.required 3 'margin+! ;cr
  ." Heading: " string.required  ;cr
  ." X-axis name: " string.required  ;cr
  ." Y-axis name: " string.required  ;cr
  ." Origin ( X,Y): " value.required "," value.required ","
  ." Maxima ( X,Y): " value.required "," value.required ;cr
  "Hi-limit" "Profile"
  "Lo-limit" "Profile"
  -3 'margin+! ;
pto
```



```
( REGISTERS AND LITERAL   PJD 15/10/84 )

: REGISTER REG.MENU REGISTER.REQUIRED AND.POST.WORD ;

: READING REG.MENU ." Reading " AND.POST.WORD ;

: LITERAL REG.MENU ." the value " VALUE.REQUIRED
  AND.POST.WORD ;

: CALCULATED VALUE REG.MENU ." Calculated Value "
  AND.POST.WORD ;


pto
```



```
( CONSTRUCTS   PJD 15/10/84 )

: START.A.CONSTRUCT DO.WHAT MENU.is.next  start
  SET.CALLER 3 ' margin +! ." Do the Following " ;

: END.A.CONSTRUCT PRIMARY.WRITER  MENU.is.next  -3 'margin+!
  start  ." That's all " ;

: IF IF.WHAT MENU.is.next ." If " ;

: UNTIL IF.WHAT MENU.is.next ." Until " ;

: N.TIMES PRIMARY.WRITER MENU.is.next
  VALUE.REQUIRED ." Times " ;
pto
```



```
( LOGICALS & COMPARISONS   PJD 11/10/84 )

: > save.v `st menu.is.next Set.caller POST.WORD is ">" ;
: < > POST.WORD IS "< " ;
: >= > POST.WORD IS ">= " ;
: <= > POST.WORD IS "<= " ;
: = > POST.WORD IS "= " ;
: <> > POST.WORD IS "<> " ;
: TRUE > POST.WORD IS "true " ;
: FALSE > POST.WORD IS "false " ;



pto
```

```
( ON, OFF  PJD 1/10/84 )

: ON PREVIOUS.MENU.next  ." on " ;
: OFF PREVIOUS.MENU.next  ." off " ;


pto
```

50

```
( UNITS  PJD 1/10/84 )
WRITING DEFINITIONS

! : (unit) forth who.called? ' as.result  =
  if primary.writer menu.is.next else previous.menu.next then ;

: V (unit)  ." V " ;
: mV (unit) ." mV " ;
: MIN (unit) ." Min " ;
! : "Sec"  ." Sec " ;
: SEC (unit) "Sec" ;
: mRad (unit) ." mRad " ;
```

PTO

51

```
( DACU FUNCTION & RANGE   PJD 28/9/84 )



: DACU.FUNCTION WHAT.DACU.FUNCTION MENU.is.next COMMA?
  "Function" "to" SET.CALLER ;

: DACU.RANGE WHAT.DACU.RANGE MENU.is.next
  COMMA? "Range" "to" SET.CALLER ;
pto
```

52

```
( DVM FUNCTIONS AND RANGES   PJD 28/9/84 )

: VDC PREVIOUS.MENU.next "DC" "volts" ;
: VAC PREVIOUS.MENU.next  "AC" "volts" ;
: 2wireOHM PREVIOUS.MENU.next  ." 2-wire " "Ohms" ;
: 4wireOHM PREVIOUS.MENU.next  ." 4-wire " "Ohms" ;
: 80.mVDC PREVIOUS.MENU.next "80" "mVDC" ;
: 300.mVDC PREVIOUS.MENU.next "300" "mVDC" ;
: 300.mV  300.mVDC ;
: 3.VOLT PREVIOUS.MENU.next "3" "Volts" ;
: 30.VOLT PREVIOUS.MENU.next "30" "Volts" ;
: 300.VOLTDC PREVIOUS.MENU.next "300" "Volts"   ;
: 300.Volt 300.VoltDC ;
pto
```

53

```
( DVM RANGES CONT   PJD 15/1/85 )
: 3.OHM PREVIOUS.MENU.next  "3" "Ohm" ;
: 30.OHM PREVIOUS.MENU.next  "3v" "Ohm" ;
: 300.OHM PREVIOUS.MENU.next "300" "Ohm" ;
: 3.KOHM PREVIOUS.MENU.ne..t "3" "KOhm" ;
: 30.KOHM PREVIOUS.MENU.next "30" "KOhm" ;
: 300.KOHM PREVIOUS.MENU.next "300" "KOhm" ;
: 3.MOHM PREVIOUS.MENU.next "3" "MOhm" ;
: 30.MOHM PREVIOUS.MENU.next "30" "MOhm" ;
: 300.MOHM PREVIOUS.MENU.next "300" "MOhm" ;
: 3.amp previous.menu.next "3" ." Amps " ;
: 300.mA previous.menu.next "300" ." mA " ;
pto
```



```
( AUTO, GATE TIMES, RATES   PJD 28/9/84 )

: AUTO PREVIOUS.MENU.next  ." AUTO " ;
: 100.mSEC PREVIOUS.MENU.next decimal ." 100 "  "mSec" ;
: 1.SEC PREVIOUS.MENU.next ." 1 " "Sec" ;
: 10.SEC PREVIOUS.MENU.next decimal ." 10 " "Sec" ;
: /SEC PREVIOUS.MENU.next VALUE.REQUIRED
  AND.POST.WORD  ascii / emit "Sec" ;
: /MIN PREVIOUS.MENU.next VALUE.REQUIRED AND.POST.WORD
  ." /Min " ;
: /HOUR PREVIOUS.MENU.next VALUE.REQUIRED AND.POST.WORD
  ." /Hour " ;

pto
```



```
( DACU RESOLUTION   PJD 28/9/84 )



: DACU.RESOLUTION  DIGITS MENU.is.next comma?
  "Resolution" "to" ;


: 3.DIGITS PREVIOUS.MENU.next 3 , "Digits" ;
: 4.DIGITS PREVIOUS.MENU.next 4 , "Digits" ;
: 5.DIGITS PREVIOUS.MENU.next 5 , "Digits" ;

 pto
```



```
( DACU FREQUENCY, COUNT, TRIGGER, GATE   PJD 28/9/84 )

: FREQUENCY PREVIOUS.MENU.next ." Frequency " ;
: COUNT PREVIOUS.MENU.next ." Count " ;

: DACU.TRIGGER WHAT.DACU.TRIGGER MENU.is.next comma?
  "Trigger" "to" ;
: INTERNAL PREVIOUS.MENU.next ." Internal " ;
: EXTERNAL PREVIOUS.MENU.next ." External " ;
: DACU.GATE WHAT.DACU.GATE MENU.is.next  comma?
  ." Gate-time "  "to" ;
pto
```

```
( MOVE STEPPERS   PJD 4/10/84)

: move move.what menu.is.next ." Move " ;
: Source  move.where menu.is.next "the" ." xenon source " ;
: Sight move.where menu.is.next  "the" ." sight " ;
: To.XY primary.writer menu.is.next ." to (X,Y) ("
  value.required ." ," value.required ." ) mRads " ;
: By.hand primary.writer menu.is.next
   ." by hand " string.required ;
: To.zero primary.writer menu.is.next  ." to zero position " ;
: Incrementally  primary.writer menu.is.next
  value.required ." mRads in X , " value.required
   ." mRads in Y " ;
pto
```



```
( ARITHMETIC   PJD 28/9 /84)

: ADD > POST.WORD IS "+ " ;
: SUBTRACT > POST.WORD IS "- " ;
: MULTIPLY > POST.WORD IS "* " ;
: DIVIDE > POST.WORD IS "/ " ;

| : "of.array" ." of the XY-Array " ;
: slope  primary.writer menu.is.next ." slope " "of.Array" ;
: Y-intercept primary.writer  menu.is.next
  ." Y-intercept " "of.array"  ;

pto
```



```
( CONNECT STRINGS  PJD  1/10/84)
| : "Pitch" ." Pitch " ;
| : "Yaw"  ." Yaw " ;
| : "Reference" ." Reference " ;
| : "Signal" ." Signal " ;
| : "Field" ." Field " ;
| : "Wide"  ." Wide " ;
| : "narrow" ." Narrow " ;
| : "error" ." error " ;
| : "detector" ." detector " ;
| : "filter" ." filter " ;
| : "v.supply" ." Volt supply rail " ;
| : pnx compile primary.writer compile menu.is.next ;
immediate
pto
```



```
( UUT CONNECTIONS   PJD  1/10/84)   ( uut )
: pitch.reference pnx "Pitch" "Reference" and.post.word ;
: Yaw.reference pnx "Yaw" "Reference" and.post.word ;
: pitch.error pnx "Pitch" "Error" and.post.word ;
: yaw.error pnx "Yaw" "Error" and.post.word ;
: AC.Detector pnx "the" "Detector" "AC" "Signal"
  and.post.word  ;
: DC.detector pnx "the" "Detector" "DC" "Signal"
  and.post.word  ;
: Pitch.wide.field pnx "Pitch" "Wide" "Field" and.post.word ;
: Pitch.Narrow.Field pnx "Pitch" "Narrow" "Field"
  and.post.word ;
: YAw.wide.field  pnx "Yaw" "Wide" "Field" and.post.word ;
: yaw.narrow.field pnx "Yaw" "Narrow" "Field" and.post.word ;
pto
```

```
( UUT CONNECTIONS CONT  PJD 25/6/85)    ( uut cont )
: pitch.ni.filter pnx "Pitch" "Narrow" "Field" "Filter"
  and.post.word ;
: pitch.wf.filter pnx "Pitch" "Wide" "Field" "Filter"
  and.post.word ;
: yaw.ni.filter pnx "Yaw" "Narrow" "Field" "Filter"
  and.post.word ;
: yaw.wf.filter pnx "Yaw" "Wide" "Field" "Filter"
  and.post.word ;
: +24v.supply  ." +24 " "v.supply" and.post.word ;
: +15v.supply  ." +15 " "v.supply" and.post.word ;
: -15v.supply  ." -15 " "v.supply" and.post.word ;
: dvm.hi pnx ." DVM-HI " and.post.word ;
: scope.chan.b.hi pnx ." Scope-HI (channel B) " and.post.word ;
pto
```

62

```
(INSTRUMENTATION   PJD 25/6/85)
writing definitions
| : drop.menu drop 2drop ;
| ; 2drop.menu  2drop 2drop 2drop ;
| : inst set.undo WHO.CALLED? dup )r ' read forth =
  if 2drop.menu
  else r ' SETUP =
    IF  drop.menu )R )R )R drop.menu r> R> R>
    else  r ' DISCONNECT = R ' CONNECT = OR
      IF )R )R )R 2drop.menu  r> R> R>
      else 2drop.menu drop.menu PRIMARY.WRITER
      THEN
    then
  THEN is.next rDROP primary.writer is.after ;
'( read.menu setup.menu connect.menu )  pto
```

63

```
(INSTRUMENTATION CONT   PJD 5/10/84)
writing definitions
| : inst" COMPILE inst [COMPILE] ." COMPILE and.post.word
; IMMEDIATE
( read.menu setup.menu connect.menu inst" name"
  will cause name to be printed
  and next  menu to be selected from read, setup or connections,
  or, if none of these called, then primary writer )


  pto
```

64

```
(INSTRUMENTS CONT   PJD 25/6/85)

: dacu primary.writer dacu.setups dvm.connections
  inst" the DACU " ;
: DVM primary.writer dvm.setups dvm.connections
  inst" the DVM " ;
: scope.channel.A primary.writer primary.writer
  scope.a.connections  inst" the scope (channel A) " ;
: scope.channel.B primary.writer primary.writer
  scope.b.connections inst" the scope (channel B) " ;



  pto
```

65

```
( UJT SWITCHES  PJD 5/10/84     ( Switching )

: Switch" compile on/off compile menu.is.next
  (compile) ." compile primary.writer compile is.after ;
IMMEDIATE

: Goniometer switch" the goniometer " ;
: Boresight-modulator switch" the boresight modulator " ;
: Light-source switch" the light source " ;
: To.narrow.field primary.writer menu.is.next
  "to" "Narrow" "Field" ;
: To.wide.field primary.writer menu.is.next
  "to" "Wide" "Field" ;
: power switch" the Power-Supplies " ;
pto
```



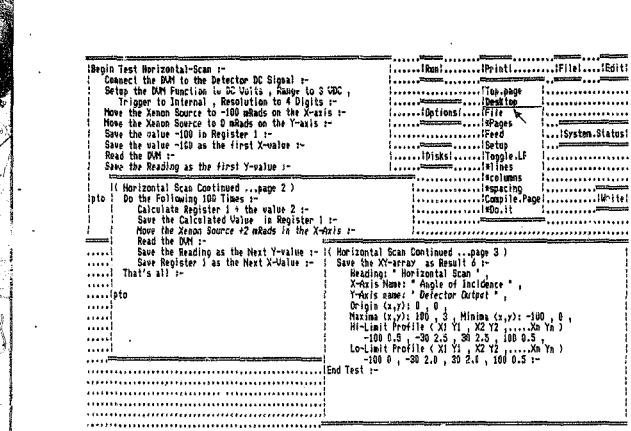
```
( DVM FUNCTION, RANGE, TRIGGER   PJD 5/10/84)

: Dvm.FUNCTION WHAT.Dvm.FUNCTION MENU.is.next COMMA?
  "Function" "to" SET.CALLER ;
: DVM.RANGE WHAT.DVM.RANGE MENU.is.next
  COMMA? "Range" "to" SET.CALLER ;
: DVM.trigger WHAT.DVM.trigger menu.is.next comma?
  "trigger" "to" set.caller ;
: DVm.resolution digits menu.is.next comma?
  "resolution" "to" set.caller ;
: Autozero on/off menu.is.next comma? ." Autozero " set.caller
;
pto
```



```
( writer definitions last page  PJD 12/7/85 )
hex
cr ." writer overlay extends to " here u.

writing definitions
killhead writeheads
hex b800 dtop !
here fence !
cr dtop @ here - . ." left in writer's dictionary area "
writing definitions
decimal
cr ." writer all done " cr
```

APPENDIX H

EXAMPLE OF A COMPLETE TEST


!Begin Test Horizontal-Scan :-
Connect the DVM to the Detector DC Signal :-
Setup the DVM Function to DC Volts , Range to 3 VDC ,
Trigger to Internal , Resolution to 4 Digits :-
Move the Xenon Source to -100 mRads on the X-axis :-
Move the Xenon Source to 0 mRads on the Y-axis :-
Save the value -100 in Register 1 :-
Save the value -100 as the first X-value :-
Read the DVM :-
Save the Reading as the first Y-value :-
!(Horizontal Scan Continued ...page 2)
!pto ! Do the Following 100 Times :-
Calculate Register 1 + the value 2 :-
Save the Calculated Value in Register 1 :-
Move the Xenon Source +2 mRads in the X-Axis :-
Read the DVM :-
Save the Reading as the Next Y-value :-
Save Register 1 as the Next X-Value :-
That's all :-
!pto
!Run!.....!Print!.....!File!.....!Edit!
!Top.page
!Desktop
!Options!.....!File
!Pages
!Feed
!Setup
!Disks!.....!Toggle.LF
!Mrites
!#columns
!spacing
!Compile.Page
!eDo.it
!System.Status!
!(Horizontal Scan Continued ...page 3)
Save the XY-array as Result 6 :-
Heading: ' Horizontal Scan ' ,
X-Axis Name: ' Angle of Incidence ' ,
Y-Axis name: ' Detector Output ' ,
Origin (x,y): 0 , 0
Maxima (x,y): 100 , 3 ; Minima (x,y): -100 , 0 ,
Hi-Limit Profile (Xl Y1 , X2 Y2 ,.....Xn Yn)
-100 0.5 , -30 2.5 , 30 2.5 , 100 0.5 ,
Lo-Limit Profile (Xl Y1 , X2 Y2 ,.....Xn Yn)
-100 0 , -30 2.0 , 30 2.0 , 100 0.5 :-
!End Test :-

REFERENCES

Adams, P. (1985) Modules cut ATE costs, ElectronicsWeek, April 15, 1985, pp. 49-51.

Bailey, R.W. (1984) Ergonomics in computer systems, Seminar Presentation, International Computers and Communications Forum, 1984

Elder, W.L. and Potts, G.R. (1984) Personal instrumentation system blends two worlds in testing and measuring, Electronic Design, October 31, 1984, pp. 166-178

Forss, A.K. and Lundin, R. (1984) An approach to long term maintenance of ATE software, IEEE, 1984

Institute of Electrical and Electronic Engineers (1981) Abbreviated Test Language for all Systems (ATLAS), New York: IEEE (IEEE Std 416-1981)

Institute of Electrical and Electronic Engineers (1982) Common ATLAS, a Subset of the ATLAS Test Language, New York: IEEE (IEEE Std 716-1982)

Johnson, C.R. (1982) Automated software development eliminates application programming, Electronics, June 2, 1982, pp. 129-140

Mandelzweig, D.M. (1982) Fully automatic production *testing of rate gyroscopes*, MSc Dissertation, University of the Witwatersrand, Johannesburg, 1982

Ohran, R. (1984) Lilith and Modula-2, Byte, August 1984, pp. 181-192

REFERENCES

Sharma, B. and Avila, J.F. (1983) Automated test program generation, Proceedings of the Automated Testing for Manufacturing Conference (ATE Central), Rosemont Illinois, October 1983, pp. IIIA.13-IIIA.19

Stewart, G. (1982) Program generators, Byte, August 1982, pp. 38-56

Tesler, L. (1981) The Smalltalk environment, Byte, August 1981, pp. 90-146

Turino, J. Subsystem testing with machine vision and robotics, Proceedings of the Automated Testing for Manufacturing Conference (ATE West), Anaheim California, January 1983, pp V.39-V.49

Wirth, N. (1981) The Personal Computer Lilith, Report 40, Zurich: Swiss Federal Institute of Technology, 1981.

PICTURE CREDITS

FIGURE 1.4a: (From top) Membrane, Kentron, Rhode & Schwartz
FIGURE 1.4b: Author
FIGURE 2.2 : Author
FIGURE 3.3 : (From Top) Rhode & Schwartz, Rhode & Schwartz, IST
FIGURE 3.5 : Kentron, Rhode & Schwartz, Author
FIGURE 4.4 : Author
FIGURE 7.2 : Author
FIGURE C3 : Author