



# The Generalised Instrument

Dissertation for the degree of Master of Engineering Science

George Vokalek

January 27, 1991

# Contents

Abstract	x
Acknowledgements	xi
Statement of Originality	xii
<b>1 Introduction</b>	<b>1</b>
<b>2 System Overview</b>	<b>4</b>
2.1 Specifications . . . . .	4
2.2 Overview . . . . .	9
2.2.1 Software Overview : SPaM . . . . .	10
2.2.2 Hardware Overview . . . . .	12
<b>3 Hardware System Detail</b>	<b>13</b>
3.1 Operations required . . . . .	13
3.2 Major Design Decisions . . . . .	13
3.2.1 Selection of Basic Hardware Format . . . . .	13
3.2.2 Selection of PCB Format . . . . .	14
3.2.3 Selection of Signal Processor . . . . .	15
3.2.4 Justification for a Control Processor . . . . .	16
3.2.5 Analogue Front-end Performance . . . . .	17
3.3 Implementation . . . . .	25
3.3.1 Control Processor . . . . .	27
3.3.2 Digital Signal Processor . . . . .	29
3.3.3 Analogue front end . . . . .	31

<b>4 Software Architecture</b>	<b>33</b>
4.1 Requirements . . . . .	33
4.2 Major Design Decisions . . . . .	34
4.3 Host Software . . . . .	36
4.3.1 The Lexical Analyser . . . . .	37
4.3.2 The Parser . . . . .	38
4.3.3 The Program Execution Unit . . . . .	39
4.3.4 The Math Function, and Primitive Math Operation Libraries . . . . .	42
4.3.5 The Graphics Library . . . . .	43
4.3.6 The Input Event Filter . . . . .	46
4.3.7 The GI Hardware Control Library . . . . .	47
4.3.8 Online Help . . . . .	47
4.3.9 SPaM as an Object-Oriented Environment . . . . .	47
4.4 Control Processor Software . . . . .	48
4.5 DSP Software . . . . .	50
4.6 Application Level Software . . . . .	52
<b>5 SPaM Reference Information</b>	<b>57</b>
5.1 SPaM : Philosophy of Design . . . . .	57
5.1.1 The User Interface . . . . .	58
5.2 SPaM Programming Language . . . . .	61
5.2.1 General Rules . . . . .	64
5.2.2 SPaM Entering Program Text : Script Files . . . . .	65
5.2.3 SPaM Variables and Numbers . . . . .	65
5.2.4 SPaM Control Statements . . . . .	71
5.2.5 User Defined Functions . . . . .	73
5.2.6 User Defined Handlers . . . . .	74
5.2.7 Shelling to MS-DOS . . . . .	75
5.3 The Graphical User Interface . . . . .	75
5.3.1 Mouse & Keyboard . . . . .	76
5.3.2 The Console Window . . . . .	76
5.3.3 The Backdrop Window . . . . .	78

5.3.4	Creating Screen Objects With Menus . . . . .	78
5.3.5	Interactions with Screen Objects . . . . .	80
5.3.6	Programmed Generation of Screen Objects . . . . .	89
5.3.7	Poster Mode . . . . .	94
5.3.8	Printing the Screen Contents to a Printer . . . . .	95
5.4	Loading and Saving Variables . . . . .	96
5.4.1	SPaM disk file format . . . . .	96
5.5	Very Large Matrices . . . . .	98
5.5.1	Creating a VLM . . . . .	98
5.5.2	Using a VLM . . . . .	98
5.5.3	Caching a VLM . . . . .	98
5.6	SPaM Language Reference . . . . .	100
5.6.1	SPaM Reserved Words and Symbols . . . . .	100
5.6.2	Mathematical Functions . . . . .	104
5.6.3	Generalised Instrument Control Functions . . . . .	110
5.6.4	Graphics Management Functions . . . . .	113
5.6.5	Program Flow Control . . . . .	118
5.6.6	Disk Access Functions . . . . .	120
5.6.7	Miscellaneous Functions . . . . .	121
5.7	Online Help . . . . .	125
<b>6</b>	<b>Integration Issues</b> . . . . .	<b>126</b>
6.1	Aspects of the RS232 link between host and GI . . . . .	126
6.1.1	Link Error Management . . . . .	127
6.1.2	Host to GI Synchronisation . . . . .	127
6.2	Control Processor to Signal Processor Synchronisation . . . . .	130
6.3	Integrating Software and Hardware . . . . .	130
<b>7</b>	<b>System Evaluation</b> . . . . .	<b>133</b>
7.1	DSP Throughput . . . . .	133
7.1.1	A/D System Performance . . . . .	134
7.1.2	Link Throughput . . . . .	134
7.2	Experience with the GI in a Teaching Laboratory . . . . .	134

7.3	Areas for Future Progress	135
<b>8</b>	<b>Conclusion</b>	<b>138</b>
	<b>Bibliography</b>	<b>140</b>
<b>A</b>	<b>The Generalised Instrument Hardware Design</b>	<b>141</b>
A.1	The 68000 Control Processor Module	141
A.1.1	Control Processor Module Specifications	143
A.1.2	CON-0000	144
A.1.3	CON-0001	145
A.1.4	CON-0002	150
A.1.5	CON-0003	152
A.1.6	CON-0004	156
A.1.7	CON-0005	159
A.1.8	CON-0006	161
A.1.9	CON-0008	165
A.1.10	CON-0009	168
A.1.11	CON-0010	172
A.1.12	CON-0012	173
A.1.13	CON-0013	175
A.1.14	CON-0014	177
A.2	The TMS320C25 Digital Signal Processor Module	178
A.2.1	DSP Module Specifications	179
A.2.2	DSP-0000	180
A.2.3	DSP-0001	181
A.2.4	DSP-0002	184
A.2.5	DSP-0003	185
A.2.6	DSP-0004	191
A.2.7	DSP-0005	194
A.2.8	DSP-0006	196
A.2.9	DSP-0007	199
A.2.10	DSP-0008	202

A.2.11 DSP-0009 . . . . .	203
A.2.12 DSP-0010 . . . . .	204
A.2.13 DSP-0011 . . . . .	207
A.2.14 DSP-0012 . . . . .	208
A.3 A Prototype Analogue Interface Module . . . . .	210
A.4 Guidelines for Designing New Modules . . . . .	214
<b>B Control Processor Onboard Software</b>	<b>215</b>
B.1 Interactive Monitor Commands . . . . .	215
B.1.1 Memory Displaying Commands . . . . .	216
B.1.2 Memory Modifying Commands . . . . .	217
B.1.3 DSP Module Control Commands . . . . .	219
B.1.4 Commands for Data Transfer . . . . .	220
B.1.5 System Control Commands . . . . .	220
B.2 Command Mode . . . . .	221
B.2.1 Available Commands . . . . .	222
<b>C Creating a Virtual Instrument Using SPaM</b>	<b>229</b>
C.1 Example TMS320C25 Code . . . . .	233

# List of Figures

2.1	The Generalised Instrument System . . . . .	5
2.2	SPaM Graphics Screen . . . . .	11
3.1	Typical PC-card DSP System . . . . .	17
3.2	DSP stand-alone system . . . . .	18
3.3	Analogue Interface block diagram . . . . .	18
3.4	Loss of A/D resolution in terms of $\frac{\omega}{\omega_C}$ . . . . .	21
3.5	Loss of A/D resolution in terms of $\frac{\omega_C}{\omega_S}$ . . . . .	22
3.6	FIR Filter and decimator . . . . .	23
3.7	Commutated FIR filter and decimator . . . . .	24
3.8	Loss of A/D resolution in terms of $\omega/\omega_C$ . . . . .	25
3.9	Major GI hardware modules . . . . .	26
3.10	Control Processor Module block diagram . . . . .	27
3.11	Digital Signal Processor module block diagram . . . . .	29
3.12	DSP and Control processor interprocessor port . . . . .	30
4.1	A sample Sigproc script . . . . .	34
4.2	Sigproc command self prompting . . . . .	34
4.3	Sample Sigproc code . . . . .	35
4.4	SPaM block diagram . . . . .	37
4.5	Sample yacc rule . . . . .	39
4.6	SPaM execution model . . . . .	40
4.7	CRT Window . . . . .	44
4.8	Graph Window . . . . .	45
4.9	Argand Window . . . . .	45

4.10 Buttons & Numerics . . . . .	46
4.11 Control Processor Software . . . . .	49
4.12 DSP code execution sequence . . . . .	51
4.13 Application generated graphics screen . . . . .	53
4.14 Application script example . . . . .	54
4.15 Application script flow chart . . . . .	55
5.1 SPaM execution flow . . . . .	59
5.2 Execution of commands from scripts . . . . .	60
5.3 Nesting of script files . . . . .	61
5.4 Chaining multiple input scripts . . . . .	66
5.5 Mouse buttons . . . . .	77
5.6 Console window on graphic screen . . . . .	78
5.7 Creating a display window with the mouse . . . . .	82
5.8 Moving/resizing graphics windows . . . . .	84
5.9 SPaM <i>button</i> activation . . . . .	85
5.10 Setting the value of a <i>numeric</i> . . . . .	87
5.11 Making measurements in <i>CRT</i> window . . . . .	88
5.12 Making measurements on <i>argand</i> window . . . . .	88
5.13 Graph before scaling . . . . .	92
5.14 Graph following scaling . . . . .	92
5.15 Graph showing waveform clipping . . . . .	93
5.16 Annotated poster display . . . . .	95
6.1 SPaM software layers . . . . .	126
6.2 Host $\Rightarrow$ GI Data Transfer . . . . .	128
6.3 GI $\Rightarrow$ Host Data Transfer . . . . .	129
6.4 Control Processor IDLE State . . . . .	130
6.5 SPaM GI control script . . . . .	132
A.1 Prototype AIM block diagram . . . . .	211
C.1 A Digitiser / FFT Analyser Implemented with SPaM . . . . .	232



# List of Tables

2.1	Generalised Instrument Specifications . . . . .	6
3.1	Prototype AIM Specifications . . . . .	32
4.1	Sample pseudoprogram . . . . .	41
4.2	SPaM numeric object attributes . . . . .	43
4.3	Example of type-dependent addition . . . . .	49
5.1	SPaM graphical screen objects . . . . .	62
5.2	SPaM predefined variables . . . . .	71
5.3	Mouse Button Functions . . . . .	76
5.4	Keypad key functions . . . . .	77
5.5	Menu for the Screen Backdrop . . . . .	79
5.6	CRT Window Menu Options . . . . .	89
5.7	Graph window menu options . . . . .	90
5.8	Poster mode menu . . . . .	94
5.9	Various ways of printing graphics . . . . .	96
5.10	SPaM file format . . . . .	97
5.11	SPaM tokens . . . . .	100
5.12	SPaM special symbols (operators) . . . . .	102
5.13	SPaM special symbols (s & logicals) . . . . .	103
A.1	Drawing index for Control Processor Module . . . . .	142
A.2	Control processor module specifications . . . . .	143
A.3	Setting jumpers for EPROM capacity . . . . .	145
A.4	Setting jumpers for RAM capacity . . . . .	145

A.5 EPROM & SRAM Address Map . . . . .	145
A.6 Jumpers for wait state control . . . . .	156
A.7 System time intervals for various master clock oscillator frequencies. . . . .	159
A.8 Control Processor Memory Map . . . . .	168
A.9 Control Processor control register bit assignments . . . . .	173
A.10 Control Processor board status register . . . . .	175
A.11 Drawings of the DSP Module . . . . .	178
A.12 TMS320C25 DSP Module Specifications . . . . .	179
A.13 DSP wait state jumper . . . . .	181
A.14 DSP Module Backplane Memory Map . . . . .	185
A.15 Board Control Area Memory Map . . . . .	186
A.16 IO space address correspondence . . . . .	186
A.17 DSP RAM type jumper settings . . . . .	194
A.18 TMS320C25 BIO' signal sources . . . . .	200
A.19 DSP-accessible control register bit assignments . . . . .	200
A.20 TMS320C25 onboard interrupt sources . . . . .	203
A.21 DSP Module Control Register . . . . .	207
A.22 DSP Module Status Register . . . . .	208
A.23 Prototype AIM Specifications . . . . .	211
A.24 Production IOM Registers . . . . .	212
A.25 Production IOM PGA Gain Settings . . . . .	213
B.1 Command Packet Structure . . . . .	221
B.2 Valid Command Byte values . . . . .	222
B.3 Command Mode, Reset Modifier values . . . . .	223
B.4 Command Mode, Hold Modifier Values . . . . .	223
B.5 Command Mode, Download Modifiers . . . . .	224
B.6 Command Mode, Download Command Field Assignments . . . . .	224
B.7 Set-Baud-Rate Command Field Allocations . . . . .	227

## Abstract

This thesis sets out and discusses the architecture of a signal processing system known as The Generalised Instrument (GI). Composed of both hardware and software, the GI system is a viable solution to the need for versatile measuring instruments which are not bound to any specific task.

By providing an integrated platform of hardware and software which lays the foundation of the machine's performance and interaction with the user, applications which fall within the bounds set by this performance can be developed as high-level software applications, without the deep system knowledge required for traditional programming. By using one hardware system, and a library of application specific software, a wide variety of measuring instruments can be emulated.

## Acknowledgements

The Generalised Instrument Project involved many people during its progress. The following technical staff contributed not only labour, but practical insights: Norman Blockley, Carmen Constantini, Garry Cox, Keith Ford, Clive Fuller, Peter Hunter, Ian Linke, and Geoff Pook.

Gang Yun Yuan served as the first 'test-pilot' for the GI system, and has my appreciation for his suggestions and criticisms.

My supervisor, Professor R.E. Bogner, has my gratitude for supporting the project through all of its phases.

### Statement of Originality

To the best of my knowledge, this thesis is an original work, and contains no material accepted for any other degree. Except where references are provided, no material is included herein which has been previously published.

I grant permission for this thesis to be made available for loan and duplication if accepted for the degree of Master of Engineering Science.

 George Vokalek



# Chapter 1

## Introduction

As single-chip digital signal processors have become established, custom DSP hardware has become cheaper to build, and a variety of programmable DSP hardware has appeared in the marketplace. Software packages for signal processing have also become available at all levels, from the high-end workstations to the low-end personal computers found on most engineers' desks.

The systems of interest here are those in which software packages are designed to work in conjunction with a hardware signal processing system. Many of the recent software packages are designed to support the common DSP peripherals available for personal computers, aiming to provide a high-level control interface to that hardware.

The Generalised Instrument consists of both a hardware DSP system and a software package to support it on a host machine. There are important differences between the GI and other commercial systems, not the least of which is the open architecture of the GI. Whereas commercial hardware and software tend to come from different sources, the GI has been designed as an integrated system where the connection between hardware and software can be made invisible to the user.

A programmable hardware acquisition and processing system, controlled by powerful yet intuitive software, allows new instrumentation functions to be created with a minimum of engineering, and with very low resulting reproduction cost.

### **The limitations of traditional, stand-alone instrumentation**

- Modern laboratory techniques require sophisticated instrumentation. Some examples are frequency domain analysers, storage oscilloscopes, transfer function analysers, and so on. The cost of such equipment is high.

Much of this cost is replicated in a laboratory, since most instruments have similar front-end (analogue and A/D) circuitry, and display circuitry. The major differences exist in the processing of the signals which occurs between acquisition and display.

There is also the physical problem of having many boxes, with each having its own operating procedures.

- Laboratory measurements must often be logged on a computer for storage or further analysis. In many cases, a computer is required to support the instruments.
- Interfacing instruments to a computer often means writing support software to implement that interface. This software, if internally written in the organization, is usually limited in features, since it will have been written to fill a specific need. Should other features be required at a future time, the software engineering is non-trivial.

By combining many test instruments into one 'box', the Generalised Instrument adds new features which were either difficult to obtain, or simply unobtainable, in traditional stand-alone instrumentation.

### **Advantages of programmable instrumentation systems such as the Generalised Instrument**

- reduced cost since only a single hardware platform is required to provide the signal acquisition and processing system. The processing is programmable in nature. The cost of the hardware is reduced to that of one 'box'.

Provided that the required signal acquisition performance falls within the range of the GI, the same hardware can be used for many instrumentation applications. Therefore, manufacturing is reduced to one set of system components, reducing cost.

- integration of software into a single suite of PC software which acts as both the control software for the hardware 'box', and as a data analysis and display package.

The software provides not only the flexibility of a large number of built-in signal processing functions, which can be used in an algebraic language, but also a graphical shell which provides a simple interface to control emulated instruments. The front panel of an instrument can be emulated on the PC screen.

- customisable behaviour, since the behaviour is controlled by software which can be easily modified at relatively low cost.
- low replication cost. Once an instrumentation application has been developed for the GI, the cost of manufacture of that instrument is virtually negligible, since the application consists of software. Therefore, to existing users of the GI, new applications can be supplied at low cost. The cost of development of software has lower capital cost and risk than hardware development, making development more attractive.
- The hardware of the GI can be used for multiple applications simultaneously. The software could simultaneously provide the facilities of multiple test instruments.
- If the hardware of the GI does not meet the performance requirements, a new module can be designed to meet that performance, and be incorporated into the GI without interfering with the GIs performance.

For instance, many different acquisition cards built for specific applications could coexist within a single GI. Since the same base core GI hardware can be used, as well as the same host software, there is no replication, and costs are lower, for this type of development compared to building a completely new system for every application.

- As well as being useful for measurement-only tasks, the GI's signal processor has the ability to process signals in real time. By providing it with suitable signal output hardware, the GI becomes a development system for DSP work, or indeed it could be embedded into a larger (presumably analogue) system as a programmable component.

One application which comes to mind is that of control systems, where the GI's DSP could digitally implement control strategies, or in telecommunications research (where modem algorithms could be tested.)

Since the GI is a stand-alone unit, the host PC need not be connected at all times. Once GI is set-up, the PC could be removed. Ultimately, the GI could be programmed to implement only the user's program from power-on. In this way, the same hardware would serve for both fixed- embedded applications, and programmable ones.

The system described in this text is functional, and has been used successfully in undergraduate laboratories for teaching.

It is important to note that the GI is a *system*, and not simply a signal processor with some attached software. Importantly, the user can install the system and begin signal processing operations immediately, since many of the required functions are already built in. The host software (SPaM, an acronym for *Signal Processing & Matrices*) features a built-in algebraic language which technical users can use with less practice than traditional programming languages.

Since single-chip DSP devices have become readily available, the emphasis has been on using them for embedded applications where a complete hardware system is designed for each new application. For research and teaching applications, such an approach is not acceptable since it implies high cost and a delay in having functional hardware.

The alternative of purchasing PC peripheral boards offers a better solution, but still has two disadvantages of high cost (DSP products are still regarded as 'exotic', with exotic prices) and limited flexibility. The user is limited to whatever hardware facilities are provided by the manufacturer, and whatever software is available to control that hardware.

At the high-end of the DSP marketplace, there are products with great flexibility. For instance, based systems are available which allow control processors, signal processors, and acquisition systems to be mixed and matched in unlimited configurations. However, attendant capital cost of such hardware is high.

The GI hardware falls into a category midway between the expensive but flexible VME based systems, and the cheaper but more restricted PC based systems. Being a card-based system, the GI offers the flexibility of future expansion of the hardware capabilities, and using the PC as the host gives the benefit of low cost to the user.

Chapter 2 of this work gives an overview of the requirements which shaped the design of the GI system as a whole. The following chapters 3 and 4 discuss actual implementation issues of the hardware and software separately.

A comprehensive guide to the PC software suite (called SPaM) can be found in chapter 5. Though developed as part of the GI system, SPaM is a self-contained numeric processing package which can be used without the GI hardware.

The issue of integrating the hardware and software into a manageable system is treated in chapter 6.

The electrical drawings from which the hardware was created are given in Appendix A. Appendix B contains the detailed information about the onboard software of the Generalised Instrument's control processor, and Appendix C gives an example of a virtual instrument created with the Generalised Instrument.



# Chapter 2

## System Overview

A diagram of a typical system using the GI is shown in figure 2.1. Those parts of the system which are dealt with in this document are the GI hardware system, and the software which runs on the host PC. The remainder of the system consists of standard commercially available computer and peripheral equipment.

The specifications for the GI hardware and software follow. Unfortunately, the budget of this project did not allow grand specifications to be written at the outset, and met in any way possible. The requirement of keeping the cost of the final system low meant that goals had to be adjusted as the design proceeded, tailoring it to what was affordable and available within reasonable time.

The question of availability is an important one. One of the principal goals of this project was to design a system which would be manufacturable. In the A/D-D/A area especially, there are many exotic components available which suffer not only from the disadvantage of high price, but from poor availability (This is intolerable: if the delivery time is too long, a potential customer will reject the affected product and go elsewhere). Such considerations are not technically interesting, but do have a great bearing on the evolution of any system designed to be built in numbers greater than one.

### 2.1 Specifications

The requirements set at the inception of the GI project were:

- I It is required to build a system which will acquire, process, and display signals in a manner which can be made to emulate common laboratory instruments, such as CROs, spectrum analysers and other common instruments.
- II It should be possible to create new applications without extensive study of the system.
- III The signal inputs of the device are to allow signals in the range 0-500kHz, voltage levels in the  $\pm 10V$  range, and sampling rates of the order of 1MHz.
- IV The system will be based around a low cost personal computer which will provide the keyboard and display which the user will use to control the instrument, and view the results.
- V Though it is to be used in conjunction with a host computer clone, the hardware of the GI is not to be dependent on the hardware of the host computer.

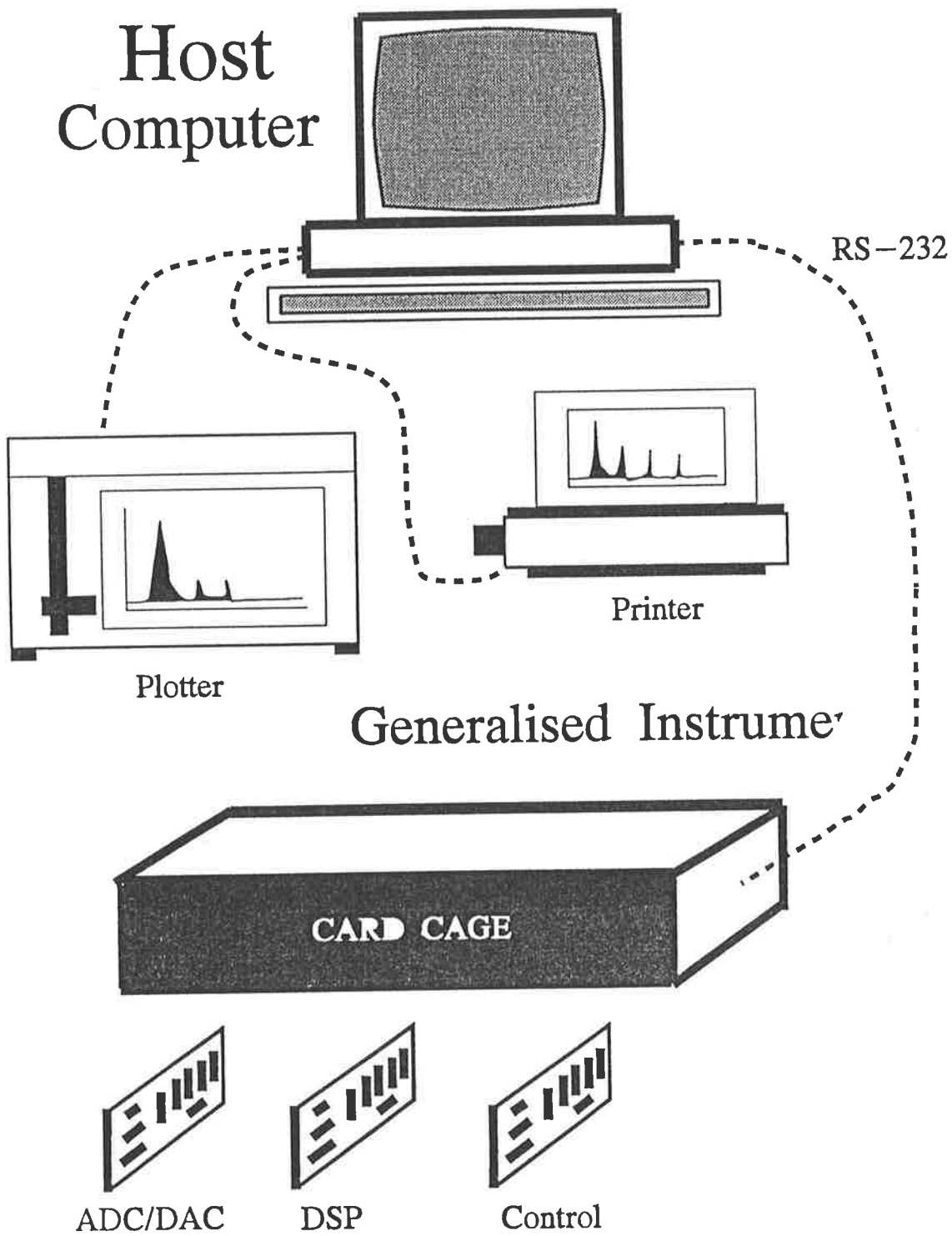


Figure 2.1: The Generalised Instrument System

Requirement	Achieved
Fast Numeric Processing	10MIPS Using a dedicated signal processor (TMS320C25) capable of executing 10 million-instructions per second. Refer to section 3.3.2.
Program and Data Memory	64k-words each. The full complement of 64k-words of local program and data memory is implemented in the TMS320C25 module. Refer to section 3.3.2.
Host Interface	RS232. The link has a throughput of 57600kbps, equivalent to about 2800 samples (16-bit) per second. Refer to section 3.2 and 7.1.2.
Control Processor	MC68000. One control processor per GI is responsible for managing communication between the host and the GI, and for controlling the system on behalf of the host. The control processor can manage up to 15 additional modules. Refer to section 3.3.1
Analogue inputs in the range $\pm 15V$ , sampling up to 1MHz	a 3dB bandwidth of 350kHz, and a maximum sampling rate of 1MHz. Refer to section 3.3.3 and A.3.
Host Software	A suite of software called SPaM has been written. SPaM controls the GI hardware, and features a built-in programming language. Refer to section 2.2.1.
Functionality	Determined by software. All functions are software controlled. Any application which falls within the performance limits of the hardware can be implemented as a software algorithm. Refer to section C.
Hardware Architecture	Modular. The various system modules are implemented on separate circuit boards in a backplane based system, allowing systems to be customised to particular needs.
Expandability	New functional modules can be added. As a modular system, the GI allows new hardware modules to be added to existing systems, or multiple existing modules to be added to one system. Refer to section A.4. Software expandability is implicit, since new DSP algorithms can be written, and new SPaM scripts created to provide the interface between the user and the signal processor. An example of the necessary programming is given in appendix C.

Table 2.1: Generalised Instrument Specifications

VI The system is to have a high processing throughput, especially on numerically intensive algorithms, such as those found in digital signal processing.

Items IV and V above define the basic environment in which the device will be working, and also the way in which the device will be built. Some further consideration is required here, for instance:

1. As the GI is to be used as a peripheral to a host computer, which host should be used? Clearly, it should be a machine which is readily available, inexpensive, and common. The machines which meet these requirements are

- The IBM-PC and clones.
- The Apple MacIntosh.
- Workstations (eg Suns).
- Mainframes (eg VAX).

In numbers sold, the IBM-PC, its derivatives and clones, is the most common computer in use at the time of writing. The availability of most types of engineering software for this machine makes it, in most cases, the machine of choice for engineers. For the software developer, the PC marketplace offers low cost and high performance software development tools.

The Apple MacIntosh series provides a platform which in many ways is superior for the intended purpose, since it supports a built-in graphical user interface which is essential for the GI. However, it is less common in engineering establishments, and is more expensive, than the PC.

Workstations and Mainframes offer important features such as superior computing power, multi-user and multi-tasking operating systems. However, the variety of machines and software in this area requires greater effort to ensure that the GI is compatible with all possible configurations. In the area of operating systems and graphical user interfaces, there has been little standardisation until recently. The widespread acceptance of standard operating systems (such as Unix) and graphical user interfaces (such as X-windows) will make software development on these systems more attractive in the future.

There are many other personal computers which have not been mentioned here. While some are impressive in price (but low in performance), and others are impressive in performance (but not in price), none offer the advantages of the mass-market computers listed above. A widely used standard computer architecture, (such as the IBM-PC and its clones) has the advantage that through competition (between various manufacturers and authors of software), it forces prices to decrease, performance to increase, and software quality to improve as time passes.

In considering the economics of these various potential hosts for the GI, it is necessary to include not only the initial cost of the hardware which the end user must purchase, but also:

- The cost of authoring software for the platform. Such tools as compilers and debuggers are required. The quality of these tools is not automatically high. It has taken several years (up to 5) for the current high quality tools to evolve for the IBM-PC.
- The cost of manpower to author and maintain software. By using a well established industry standard architecture, information and expertise become more commonly available and therefore less expensive.
- The performance of the machine. If the host is lacking in computing throughput for the desired operations, what are the options to increase the throughput?

In the case the machines listed above, most run in families which are fully compatible with one another, but with different performance figures.

As an example, the IBM-PC clones currently on the market span approximately 2-orders of magnitude in computing power, with a corresponding 1-order of magnitude range in price.

- What hardware the user already owns. Most potential users of the GI would wish to maintain compatibility with their current computing equipment.

New and alien computing hardware requires time to be allocated to training in its use, and may result in increased delays if problems occur and need to be fixed.

The IBM-PC was chosen as the host for the GI. The next choice to be made is which operating system to support. The PC market offers a variety of operating systems, including MSDOS (the most common), OS/2 (still under development), and Unix (a highly fragmented market). The reasons for choosing the PC, such as low cost and compatibility with available software packages and hardware, suggest that MSDOS be the operating system used.

The rate of evolution in the PC market is very high. However, the magnitude of the existing software market indicate that MSDOS based software will be supported and viable for many years to come.

2. Given that the host will be a computer running the MSDOS operating system, most of the hardware requirements could be met by existing, commercially available, plug in PC cards.

It was decided at an early stage to avoid building an internally mounted PC peripheral, since such a device would be completely dependent on its host. Such dependence is not desirable for several reasons:

- (a) PCs are available which have few or no slots for peripheral cards. While the GI was in the conceptual stage, one such PC was a likely host computer.
- (b) The inside of PC represents an electrically noisy environment.
- (c) Applications are envisioned where the GI would be used for an extended time as an embedded part of a system. In such cases the host PC would be little more than a case and power supply.
- (d) Many users will need to install more cards within their PC than there are slots available.
- (e) The rate of evolution of PC technology is so rapid that the user will probably change hosts a number of times during the lifetime of the GI. Similarly, the large number of PC variants available now and in the future means that hardware compatibility is always in question.

A recent example is the introduction of the EISA bus, in which manufacturing tolerances in PC plug-in-cards determines whether they will function correctly.

3. There *will* be users of the GI who wish to run the system with a host other than an MSDOS PC. The most likely hosts would be unix-based workstations and PCs running OS/2. The design of the GI as a stand-alone system means that costly redesign is avoided, and the customer does not have to purchase entirely new hardware when she upgrades her host computer.
4. As a stand-alone box, the GI can be placed closer to the source of the signals it is to process, without the associated host computer needing to be placed there also. The connecting cable can then be run to the PC.

To meet specifications I and II, it is necessary to design the host software in such a manner as to allow users (who are technically competent, but not necessarily expert in the details of the GI) to construct new applications. To fulfill the dual (and seemingly contrary) requirements of flexibility and simplicity, the host software will need to incorporate some sort of 'macro language' which allows the user to string together the high-level, easily understood, commands to obtain a desired result.

It is not sufficient to generate a library of, say, C-callable code which the user could use to produce her own, standalone, programs. The reasons for this are multiple:

1. The user should not have to be proficient in any particular programming language. Programming languages generally require a greater level of detailed understanding which is not directly relevant to carrying out signal processing computations.
2. The user would be required to possess a particular type of (commercial) language compiler, which implies added expense. The great number of compilers available on the market means that either the SPaM library would have to be compatible with all (possibly implying multiple libraries), or the user would be forced to buy a particular one.

Considerations of these requirements has led to a stand-alone system which incorporates a TMS320C25 digital signal processor, as well as a 68000 processor to control the system and communicate with the host. A sophisticated software package has been developed as the interface between the user and the signal processor. This package, called SPaM, features full programmability to allow the generation of new and varied user interfaces without extensive programming.

Item VI of the requirements states that some form of specialised numeric processing circuitry be used. It is not acceptable to use most microprocessors such as the 68000 for this task, since their throughput in numeric processing is so low that only low frequency operation would be possible.

Acceptable alternatives are the use of semi-custom hardware, for instance bit-slice processors, and the use of single chip digital signal processors, which are now available from a variety of manufacturers. The single-chip DSP was chosen for the following reasons:

1. Single-chip DSP devices are now available which will reach performance levels of the same order of magnitude bit-slice processors in the same silicon-technology.  
Single-cycle instruction execution, memory caching, and other techniques are now becoming standard features in DSP devices.
2. To achieve similar programmability and versatility, a bit-slice system would consist of much more circuitry than a single-chip DSP.
3. The widespread use of single-chip DSP devices is pushing their cost down dramatically. Bit-slice technology is more expensive, due to less widespread use and more complex designs.

For several years, modern microprocessors and digital signal processors have been converging in respect to performance and architecture. Microprocessor designers have been implementing single cycle instruction execution (the so called 'RISC' paradigm), and separate instruction and data busses (eg the AMD29000 series), while DSP designers have been providing richer instruction sets, larger address spaces, and so on. The result will be that standard microprocessors will be suitable for numerically intensive real-time tasks in the near future.

## 2.2 Overview

A block diagram of the GI system is shown in figure 3.9. The GI consists of hardware and software components, which together with a host computer create the instrument. The computer supported so far is the ubiquitous IBM PC clone, since this is probably the most widely used class of computer used in engineering and research.

Running on the PC is an environment called SPaM, which provides the user with the following facilities.

- An interpreter for performing algorithmic calculations.

- A graphical shell for displaying results, and interacting with the instrumentation hardware.
- Storage and retrieval of data to and from fixed media (eg hard disk.)
- Generation of hardcopy of the results on a printer.

The hardware of the GI consists of sets of circuit boards, each of which is a self-contained functional module. Specific details are given in section 3, and in appendix A. A typical configuration includes a control processor, a signal processor, and an analogue sampling board, as shown in figure 3.9.

## 2.2.1 Software Overview : SPaM

The suite of software which runs on the host is called **SPaM**, which is an acronym for Signal Processing & Matrices. The software is not unlike the package MatLab, in that it provides the user with an algebraic programming language which implicitly understands real and imaginary numbers, vectors and matrices. These are the data types most commonly encountered in DSP work.

**SPaM** is built around a central compiler for its embedded language. The compiler can accept input either from the user via the host's keyboard, or from script files stored on disk. This allows prepackaged applications to be stored on disk simply as scripts, allowing the system to perform actions automatically without the user's intervention.

The user may interact with the system through a text only interface, where algebraic expressions are entered from the keyboard, processed by **SPaM**, and the results displayed to the screen. These expressions may include builtin functions, user defined functions, and various standard mathematical operators. In this mode of operation, **SPaM** is able to carry out manipulation of data vectors and matrices as directed by the user. The text-only display is sometimes preferable to a graphic display, the main advantage being more rapid text printing.

An alternative way of using **SPaM** is via the graphical shell. When operating in this mode, **SPaM** presents the user with a graphical screen on which the user may open windows, and create objects, to display various pieces of information. Arrays of data may be displayed in windows, and 'buttons' may be created to call user defined programs when the user clicks on them. The graphical shell is designed to work with both a mouse and keyboard.

Using the flexibility of the embedded language, and the uncluttered presentation offered by the graphical shell, complex screen displays may be created to simulate familiar instruments, as shown in figure 2.2.

As well as interacting with the user, **SPaM** handles interactions with the GI hardware. The GI hardware interfaces to the host via an RS232 interface. The protocol used in the communication is a packet based one. Packets (of variable size) containing commands, or data, are transmitted between the GI and the host. An error detection and correction protocol is implemented by associating a CRC<sup>1</sup> with each packet. If a packet is corrupted during transmission, the sender is informed and it is resent.

Such a protocol raises the possibility of the GI and its host being remote from one another, connected through some sort of data link (for instance, via modem over the dial-up telephone network.) The error detecting and correcting features of the protocol provides the necessary immunity to transmission errors over such a link.

---

<sup>1</sup>CRC is an acronym for Cyclic Redundancy Check, which is a validity value akin to, but more robust, than a checksum.

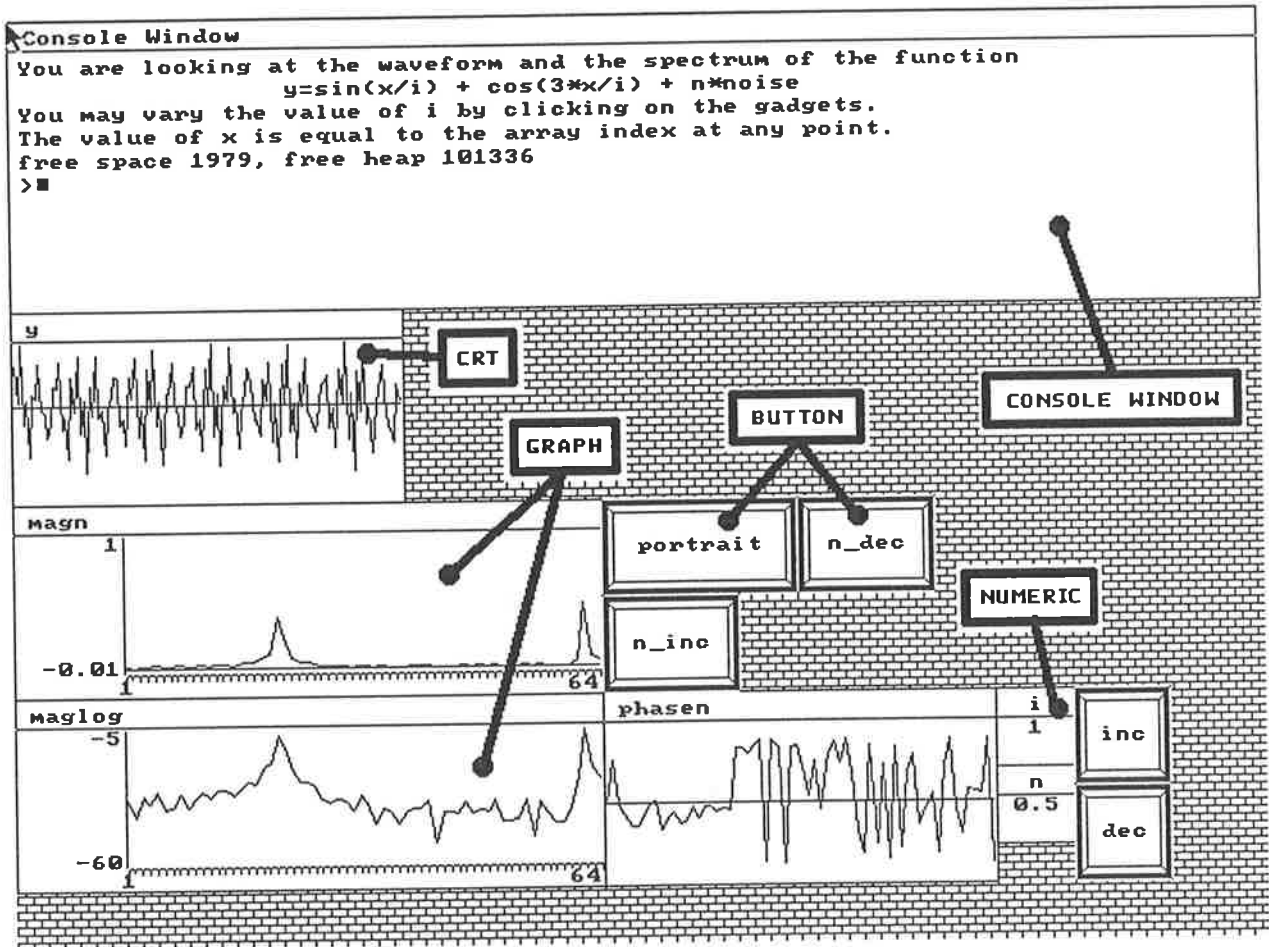


Figure 2.2: SPaM graphics display screen, showing waveforms and control buttons.



### 2.2.2 Hardware Overview

The GI hardware is a 'black box' which attaches to the host computer via an RS232 connection. The GI box contains hardware modules which perform specific duties within the system:

1. The control processor module. This is responsible for overall system control, and communication with the host.
2. The signal processor module. This module contains a single chip digital signal processor, which performs the mathematically intensive operations.
3. Signal acquisition module. This is the A/D and D/A interface board. The current module connects to the DSP board to provide the analogue front end for the signal processor. The design of this module is critical to the overall performance of the instrument, but unlike the digital circuitry it is difficult to design with certainty.

A block diagram of the GI hardware is shown in figure 3.9.

Providing that the signal parameters fall within the performance limits of the GI hardware, the operation of the hardware is transparent to the user. In cases where extended performance is required, additional hardware modules can be mated to the GI, and appropriate software modifications made to support them.

## Chapter 3

# Hardware System Detail

This chapter looks in more detail at the architectural aspects of the hardware component of the GI.

### 3.1 Operations required

The basic operations and features required of the GI hardware are summarised below:

1. To provide an interface method to the host computer.
2. To provide a protocol and command set over that interface to allow the host to command the GI to perform specific actions.
3. To provide a means of transferring data to and from the GI over the interface.
4. To provide a connection to analogue signals, both as inputs to the GI, and outputs from the GI.
5. To build the GI around a fast DSP device which allows the rapid processing of signals. The results of the processing are to be sent to the host for inspection by the user, or used to generate an output from the GI in real time.

### 3.2 Major Design Decisions

In this section some of the major design decisions are discussed and clarified. In a large system such as the GI, there are often alternative ways of doing things. Making the correct choice is often difficult, since bad choices are invisible until the system is either in manufacture or active use.

#### 3.2.1 Selection of Basic Hardware Format

**Should the GI be built as a PC peripheral card, or a stand alone unit?**

This question was addressed in section 2.1, and the main points are reiterated here. It is desired that the GI be physically independent of its host so that it can be used with *any* host.

The only host computer currently supported is a IBM-PC compatible running the MSDOS operating system. The majority of such computers are equipped with multiple internal slots, which could be used for a plug in card. However, there are classes of PCs which have either limited or no slots. At the time the GI was being designed, a likely host seemed to be the locally produced Microbyte PC230, which is limited to one or two internal short cards, these being insufficient to hold all of the GI circuitry.

Another class of potential host is the laptop or portable PC, which is becoming more common as costs fall. Such PCs provide PC compatibility in a portable briefcase sized unit, and would be suitable portable hosts for a portable GI.

There are two common interfaces provided by virtually all PC compatible computers, even those mentioned above. They are the Centronics compatible 8-bit parallel interface, and the RS232 serial interface.

It was decided to use the RS232 interface to communicate with the GI after considering the most common uses for these ports. On the majority of PC computers machines, the parallel port is used to drive a printer. Since the majority of PCs feature only one parallel Centronics port, this port is unavailable in many cases.

The RS232 interface is universally available in the commercial computer industry. By being separate from the PC host, connecting to it via a standard interface, the GI is better able to resist obsolescence. When the host is superseded by the next generation of PC, the user can be assured that (for the foreseeable future) an RS232 connection will be available to facilitate connection to the GI.

The majority of computers have, or are capable of having, two RS232 ports. One of these is required for the GI.

The majority of RS232 ports on PC-compatibles use an Intel 8250 compatible UART, which is capable of transmission at 57600bps and higher. This speed corresponds to about 5.8k-bytes per second.

In a typical application of the GI, spectral analysis is performed on a signal using a 1024-point FFT algorithm. The results of the calculation will be 512 16-bit integers representing the magnitudes of the signal in 512 frequency bins. Thus 1024 bytes must be uploaded to the host PC for display, corresponding to a delay of 0.2 seconds.

In practice the delay between screen refreshes will be longer, due to host-processor intensive operations such as screen updating, and the transfer time will lose its dominance in determining the response time of the system.

Even so, the serial port communication does represent a bottleneck for the transfer of large amounts of data. To overcome this, a parallel port will be designed into future versions of the GI to speed the transfer process.

The universality of the RS232 interface means that computers running the Unix operating system could become hosts for the GI. Such hosts would allow a major leap forward in the host software, due to the more advanced nature of their operating systems (eg multi-tasking), and graphical user interfaces.

### 3.2.2 Selection of PCB Format

Should the GI be designed as a one-circuit-board or multi-board system?

This is an extremely important question, since it plays a large part in determining the economics and manufacturability of the the hardware.

The prototype of the GI was designed as one, large printed-circuit board. The rationale was to minimise the interconnections required, and the cost of a cage to hold multiple cards.

A constraint imposed by the limited budget was that only 2 layer PCBs could be used (due to the prohibitive<sup>1</sup> cost of multi-layer ones). The side-effect of using 2 layer boards in processor circuits is that the component density on the PCB becomes low, since much area is required for the routing of data, address, and control busses. For a circuit with a large number of components, this results in a large PCB.

An obvious means of reducing board space is to use multiple boards, connected via a backplane bus. This adds cost by increasing the cost of connections, and card cage, etc. However, since the busses have been moved offboard, each circuit board can be more densely populated, even using 2-layer PCB technology.

The added benefits of multiple circuit boards, as compared with a single board, include:

- Easier assembly.
- Easier testing, since there is less circuitry per board to test, and a test-jig can be constructed in which all boards which are present are known to work correctly, allowing one new board to be plugged in and tested in an otherwise fully verified system.
- Expandability. The user may design new cards to fit into an existing system, so that new functions can be added to the system. For instance, since the digital signal processor is isolated on its own board, other boards could be designed for other signal processors, and yet would work within the same system. Alternatively, multiple DSP boards could be fitted in the same system, allowing an MIMD (Multiple-Instruction, Multiple-Data) multiple-processor to be built.
- Selective evolution of modules. Modules can be individually redesigned without affecting the rest of the system. This feature allows the GI to keep pace with technology by minimizing the amount of redesign which must be done.
- Standard enclosures may be used, if an industry standard card format is adopted (as has been the case.)
- Higher component density on each circuit board means a lower overall volume, resulting in a more compact device. For 2-layer PCB technology, the multi-board system allows higher component density than a single board because the wide system signal busses are moved onto the backplane. Areas occupied by busses cannot generally be used for other circuitry, resulting in low overall density.

### 3.2.3 Selection of Signal Processor

#### Which Digital Signal Processor should be used?

When choosing a DSP device for any particular application, the designer must weigh up a number of factors.

1. Performance of the DSP device in the intended role.
2. Cost of the device itself.

---

<sup>1</sup>To illustrate the point, the 2-layer prototype boards which were produced cost \$70 each. A corresponding 4-layer board would have cost \$500, and 6-layer board \$700 each.

3. Cost of placing the device in circuit, ie are any special mounting techniques or hardware required.
4. Cost of software tools (compilers, assemblers, simulators, etc).
5. Availability of the devices.

During the early design stages of the GI, the TMS320C25 was one of the few readily available and well supported signal processors available, and it was consequently chosen for the GI.

Since then, other signal processors have become available which have comparable architecture, and higher computational performance. Experience with the prototype GI, however, demonstrated that when the host computer was involved in the data-flow process (as in transferring data to the host and having that data displayed on the screen), the computational performance of the DSP in the GI was not critical, since the main bottlenecks were elsewhere in the system.

The unit cost of signal processors, like other semiconductors, decreases with time as volume production and distribution take effect. During the course of this project the TMS320C25 unit cost has dropped from over \$200 to about \$70. A major contribution to cost is the package in which the device is sold. A pin grid array is generally the first package type released, but is expensive to manufacture. Once demand for the device exists, and the silicon has been finalised, less expensive plastic packaging (such as PLCC) is released. The recent release of new versions of the TMS320C25, featuring higher clock speeds and more internal program and data memory mean that the performance of the GI will be suitably increased. These new processor versions are pin compatible with their ancestors, so PCB redesign is unnecessary.

While other manufactures had devices whose performance exceeded that of the TMS320C25 in many areas, other considerations weighed against them. The lack of available production silicon, was (and continues to be) a major problem.

### 3.2.4 Justification for a Control Processor

**Should a separate control processor be used, or should the Digital Signal Processor be the sole processor in the system?**

The purpose of the control processor is to be a fixed servant of the host computer, allowing the host computer to exercise complete control over the signal processor(s), and any other system components, even if those components go out of control.

For instance, if the user's DSP code is somewhat 'experimental' and unstable, it would be very difficult to regain control of the DSP without being able to perform basic tasks such as asserting the RESET signal to the signal processor. A plug-in PC card has access to the necessary signals from the host, but a stand-alone unit requires some reliable way of converting the command-stream from the host into hardware actions.

Were the GI to be built as a PC plug-in card, there would be no need for a control processor as such, since the PC itself would fill that role, as shown in figure 3.1. The arguments against building the GI as a peripheral card for one type of computer have already been presented, and so the problem of reliable overall control of the GI system remains.

Without a control processor, basic tasks such as the transfer of data from host to GI would have to be performed by the DSP itself. While it could perform such tasks with little difficulty, this would require that the DSP code for such operations be kept in ROM in the GI. The code in this ROM would not be trivially small, and would decrease the (already small) memory available (64k) to the signal processor for signal processing code. This could be cured by using various memory mapping

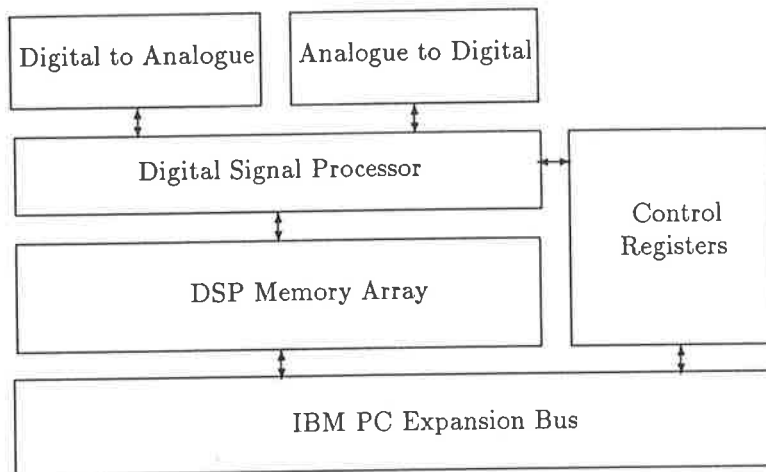


Figure 3.1: A typical PC-card DSP System. The PC host performs all system management functions.

schemes, but one is still left with the possibility of the system entering a state from which it cannot be remotely removed.

Communication with the host computer is a task accomplished under software control by a processor (hypothetically, the signal processor itself), as in the system shown in figure 3.2. The processor must receive command and data packets from the host, and transmit data packets to the host. Actions specified in the command packet must be carried out. All of these operations require processing time, which is then unavailable for the execution of signal processing algorithms.

Conceptually, it may be possible to integrate tightly the system management and signal processing software to achieve the required performance. However, this defeats a design goal of the GI which was to design a system where new applications could be prototyped very quickly.

The presence of a control processor (figure 3.9) in the system gives a great deal of flexibility in the control of the system. The DSP need not be affected at all by communication between host and GI. Indeed, as is described in the later sections, the communications between the DSP and control processor are such as to impose as little delay as possible on the DSP when transferring data to the host.

A control processor is essential for a multi-DSP system, which can be produced with the current GI. Signal processors can then be individually accessed and controlled without any effect on the others.

### 3.2.5 Analogue Front-end Performance

What should be the performance of the analogue front end?

Prototypes of the GI revealed that the analogue front end of the DSP systems is really the most important part of the system (on a par with the user interface). The analogue front end determines what magnitudes and bandwidths of signals can be acquired, and therefore almost entirely defines the performance of the system in these areas.

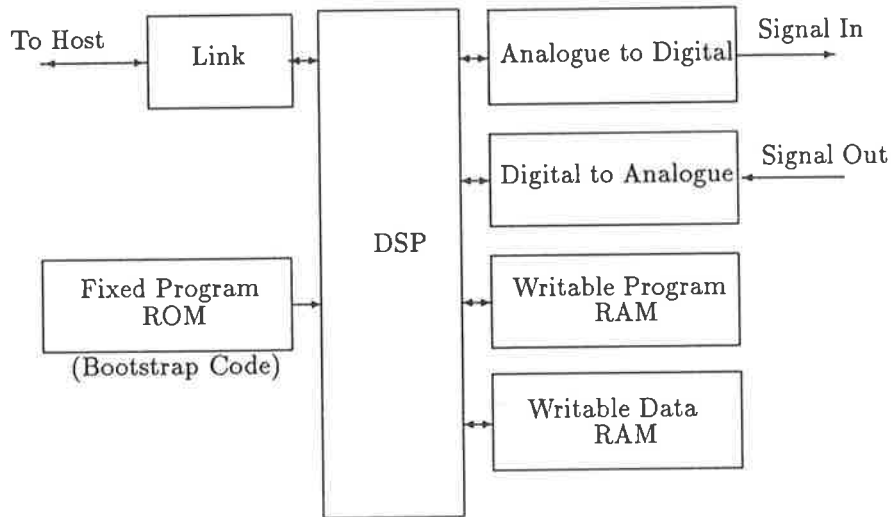


Figure 3.2: A stand-alone system could be built around the signal processor alone, but there the requirement to perform signal processing and system-administrative tasks would limit the capabilities of such a system.

There are several important issues which must be addressed in the design of a analogue to digital interface system. These are

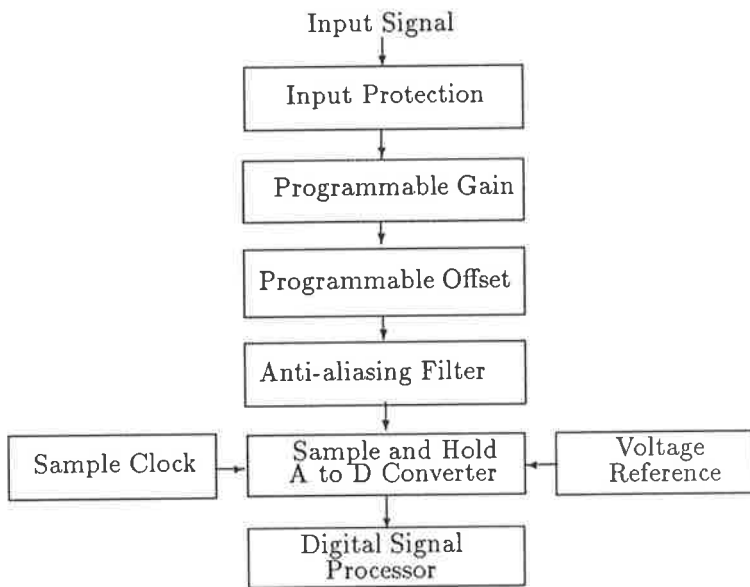


Figure 3.3: Analogue Interface block diagram

### 1. Acceptable input signal amplitudes.

The restrictions of analogue circuitry limit the internal signals of the GI to a range of  $\pm 12V$ . There will be further restrictions imposed by specific parts of the system, such as the A/D.

A greater input range can be accommodated by using resistive attenuators. at the input. On a CRO or similar instrument, such attenuators are controlled by a multi- position switch. Since the GI is to operate as an autonomous device, the switching must be done with relays.

Solid state switches should not be used at this front, since they will either be susceptible to overload, or will detrimentally affect the bandwidth of the system (due to internal on-resistance (and capacitance)).

There are several points flowing on from using a resistive divider at the input:

- The resistor chain must not significantly load down the circuit under test. A high input impedance can be achieved by allowing the input attenuator to be switched out of circuit completely when not needed.
- The resistor chain resistance must not be so high as to combine with parasitic capacitance to form a low-pass filter.
- The buffer amplifier following the divider must be of high input impedance, and low input capacitance.

The poles created in the transfer function of the input by resistance (from the attenuator) and capacitance (parasitic) will cause low- pass behaviour. This effect can be alleviated somewhat by using feed-forward capacitors.

Compensating values must be determined from circuits once built, since the parasitic capacitance will be largely distributed, requiring a test and set procedure during manufacture.

### 2. Input overload protection.

While it is not difficult to protect against slight overloads by clamping the input voltage to the power rail using diodes, it is less easy to protect against order-of- magnitude overloads (such as connecting 240V to the input.)

The aforementioned attenuators would provide such protection if used properly (eg start with the highest attenuation when measuring an unknown voltage.)

Probably the most cost effective means of protection is to minimise possible damage rather try to prevent it altogether. This can be achieved by having attenuators, in which case it is only resistors which are destroyed by an overload, and by placing a buffer amplifier between the input and expensive converter ICs. In this way, cheaper components are sacrificed before the overload can reach the expensive components.

### 3. Programmable gain control

In order to obtain the highest possible signal-to-noise ratio in the conversion process, it is necessary that the signal occupy as much of the A/D converter's input range as possible. To effect this, programmable gain is required.

- (a) For practical reasons, 1LSB of the A/D converter should not be less than 1mV. Below this level, induced noise will be expensive to eliminate.
- (b) A programmable gain amplifier will amplify not only the signal voltage at its input, but also the input offset voltage. For this reason an offset subtraction mechanism must be provided.
- (c) For a converter with a full-scale input range of 3V (typical, eg AD7870), a gain of 100 will cause a full-scale reading for a 30mV signal. Gains in the range 1 to 200 will provide resolution comparable to standard oscilloscopes (eg 1mV/div). Input attenuators will provide the sub-unity gain needed for signals of amplitude larger than the input range of the converter.



#### 4. Input offset adjustment

Rarely does a signal have zero DC component. To allow the full use of an A/D converter's input range, it is necessary to remove the DC component so that the AC component can be acquired. This is done on an analog CRO by use of a capacitor coupled input, or vertical trace position controls.

A capacitively coupled input has the disadvantage that it adds a zero to the input transfer function, resulting in high-pass behaviour. One of the great advantages of *digital* signal processing over processing in the analogue domain is its suitability for very low frequency work. Capacitive coupling at the input would destroy this ability.

Instead, the DC component can be removed by generating an offset voltage using a D/A converter, and subtracting that offset from the input signal.

Such a scheme has the added advantage of being able to accurately zero the instrument for any configuration. For instance, in the prototype it was found that programmable amplifiers amplify their input offset voltages, as well as the input voltage. Unless this amplified offset is removed, the useable range of the A/D converter is reduced.

#### 5. Signal bandwidth, Sampling rate.

When designing A/D interfaces, it is not sufficient to simply ensure that the A/D samples at the Nyquist frequency implied by the signal bandwidth.

There are two bandwidth limitations imposed by any acquisition system. The first and most obvious is the maximum frequency of the input signal as dictated by the Nyquist sampling theorem. This imposes the limit that the input signal may contain no frequencies higher than one-half of the sampling frequency.

The second bandwidth limitation is that imposed by the analogue preprocessing circuitry which precedes the A/D converter, and the analogue bandwidth of the converter's input stages<sup>2</sup>.

The input circuitry to the sampling system is typically a combination of passive and active components: resistors, inductors, capacitors, and op-amps. The passive properties may be lumped in physical components, or distributed, as in the PCB tracks forming the connections. The result of these components is multiple poles in the input transfer function, leading to low-pass behaviour. Note that this low-pass behaviour is distinct from the anti-aliasing filter.

Now to ensure that the analogue circuitry does not interfere with conversion, any cutoff frequencies must lie far above the Nyquist frequency. The relationship of the input circuit bandwidth to signal bandwidth will be explored in the following paragraphs. The intention is to determine what effect a realistic input circuit will have on the measured signals.

Consider the case of an  $M$ -bit A/D converter. Suppose that the converter is preceded by a  $N$ -th order low-pass system, whose transfer function is simplistically modeled as

$$H(j\omega) = \frac{1}{(1 + j\frac{\omega}{\omega_c})^N} \quad (3.1)$$

Let us further assume that the anti-aliasing filter has a very steep cutoff, so that the A/D converter can convert signals all the way to its Nyquist limit, given by  $\frac{F_s}{2}$ .

Now, let us inject a sinusoid of amplitude  $2^M$  units into the system, which would, if unattenuated, fully exercise the converter's dynamic range. The presence of the analogue filter will, however, produce attenuation so that the amplitude the converter sees is  $2^m$ , for some  $m$  (where  $0 \leq 2^m \leq 2^M$ ).

$$\frac{2^m}{2^M} = 2^{m-M} = \left( \frac{1}{1 + (\frac{\omega}{\omega_c})^2} \right)^{N/2} \quad (3.2)$$

---

<sup>2</sup>Note, this is distinct from the Nyquist frequency, and is determined by the analogue properties of the A/D converter

Since the original signal was of amplitude  $2^M$ , and the resulting measured signal has an amplitude of  $2^m$ , we can consider the quantity  $(M - m)$  to be the number of bits of resolution which has been lost by the system, due to the action of the analogue preprocessing. For instance, if the input circuitry halves the signal amplitude at the working frequency, only one half of the A/D dynamic range will be exercised, and one bit of resolution will be lost. Let us define

$$\begin{aligned} M_{LOST} &= M - m \\ &= \frac{N}{2} \log_2 \left( 1 + \left( \frac{\omega}{\omega_C} \right)^2 \right) \end{aligned} \quad (3.3)$$

The graph in figure 3.4 shows the nature of this function. The above expression is not yet meaningful because it does not relate to the sampling rate of the system.

It is worth noting that figure 3.4 is somewhat deceptive. The loss of resolution is from the most significant bit of the converter. Thus, a loss of 1-bit of resolution means a halving of input signal amplitude.

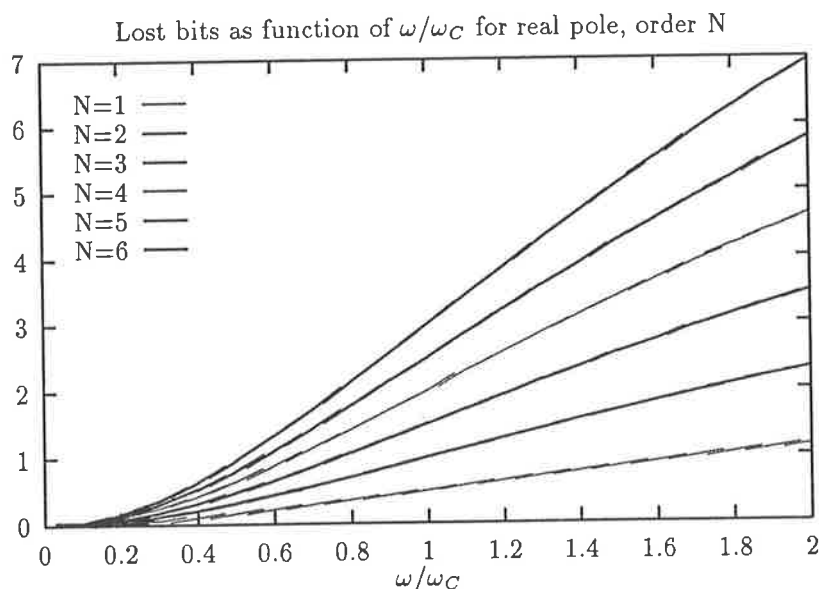


Figure 3.4: Loss of A/D resolution in terms of  $\frac{\omega}{\omega_C}$ , for the real-pole system of equation 3.1.

We can incorporate the sampling rate by setting  $\omega = \omega_S/2$ , where  $\omega_S$  is the angular sampling frequency. In effect, this tells us the number of bits lost at the Nyquist rate, which represents the maximum possible signal frequency which the system is permitted to sample.

We now rearrange equation 3.3 to be dependent on a new variable,  $k = \omega_C/\omega_S$ , to give equation 3.4. The graph of the function is shown in figure 3.5. This graph shows, for a given ratio of system cutoff frequency to sampling frequency, the number of bits lost by the system.

$$\begin{aligned} M_{NYQUIST} &= \frac{N}{2} \log_2 \left( 1 + \left( \frac{2\omega_C}{\omega_S} \right)^{-2} \right) \\ &= \frac{N}{2} \log_2 \left( 1 + (2k)^{-2} \right) \end{aligned} \quad (3.4)$$

Equation 3.4 is valid for all values of  $\frac{\omega_C}{\omega_S} > 0$ . For values of  $k = \omega_C/\omega_S$  close to zero, the argument of the logarithm will be large and the resulting loss in bits will also be large. The

reason for this is clear: if the analogue circuitry has a cutoff frequency much lower than the sampling rate (so that  $\omega_C/\omega_S = 0^+$ ), then the attenuation at the Nyquist frequency will be high, and  $M_{NYQUIST}$  will also be high.

Conversely, for a value of  $k = \omega_C/\omega_S$  which is very large, the argument of the logarithm will be marginally greater than one, and the number of lost bits will approach zero. Again, this is reasonable, since analogue circuitry will not greatly affect the sampled signal if the 3dB bandwidth is many times greater than the sampling rate.

It should be noted here that the number of bits lost, eg  $M_{LOST}$  or  $M_{NYQUIST}$  are subtracted from the resolution available. For instance, if the system uses a 10-bit A/D converter, and is preceded by a 2nd order analogue circuit whose cutoff frequency is one half of the sampling rate ( $k = 0.5$  in equation 3.4), then  $M_{NYQUIST} = 1$ . Therefore, at the Nyquist frequency, our 10-bit converter becomes a 9-bit converter.

Since a negative resolution is meaningless, the number of bits lost in practice cannot exceed the number of bits which are physically available.

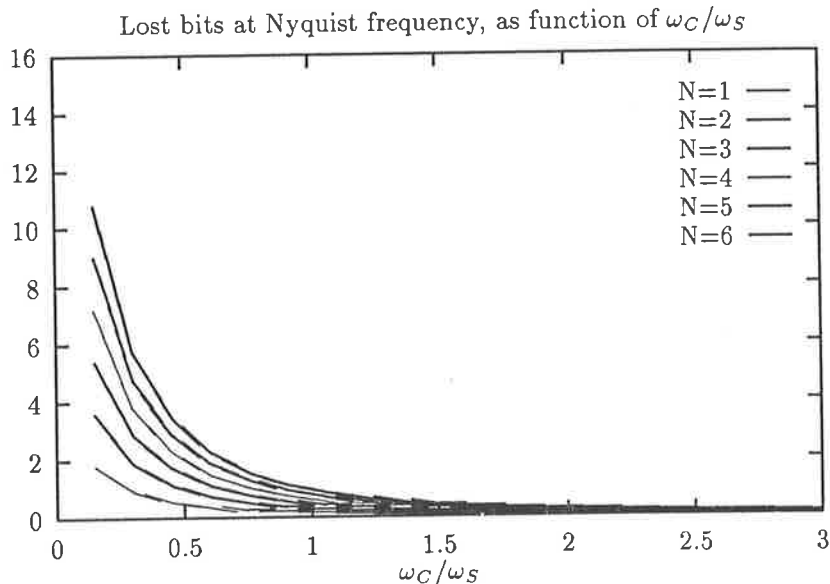


Figure 3.5: Loss of A/D resolution as a function of  $\frac{\omega_C}{\omega_S}$ , for real-pole system described in equation 3.1.

Clearly, the higher the cutoff frequency  $\omega_C$ , the less will be the effect of the analogue system at low frequencies below  $F_S$ .

The essence of this information is that the analogue system must generally have a bandwidth far greater than the maximum sampling rate in order to maximise the effectiveness of the converter. The situation is complicated further by the presence of the anti-aliasing filter (discussed below), which must have a cutoff frequency of  $F_S/2$ .

While the analogue circuitry determines the system bandwidth, it is the digital circuitry which determines the maximum achievable sampling rate.

A choice must be made whether the sampling is to be performed as an action of software (executed by the DSP), or is to be implemented as a hardware feature. The former option has the advantage of low cost, and is limited by the speed of the DSP device. The latter has the advantage of much higher sampling rates, at the expense of circuit complexity and cost.

The software sampling option is the one chosen for the initial analogue module for the GI. With the TMS320C25 clocked at 40MHz (thereby achieving 10MIPs), a maximum sampling

rate of approximately 1MHz is achievable. At such a high speed, the DSP is doing nothing but moving data from A/D to memory. At lower sampling rates, the intersample time may be sufficiently long to carry out processing.

### 6. Anti-aliasing filtering

For a system such as the GI, the anti-aliasing filtering provides particular challenges.

As outlined in the previous section, the sampling rate of the system may vary over six or more orders of magnitude. To devise an antialiasing filter capable of variation over the same range would be very difficult, and not economical.

For low frequencies (eg 0-100kHz), a viable solution is provided by microprocessor programmable switched capacitor filters. Such devices are available commercially, and allow programmed filter response over that range.

For higher frequencies, the only viable alternative is the use of switchable filter banks. Such filter banks would be tuned for particular cutoff frequencies, and would be switched into the signal path as appropriate. The economics of such filters would limit the choice of cutoff frequencies to a small number.

Following a different route, antialiasing filtering could be performed digitally by oversampling the signal, using a digital filter process to band-limit it, and decimating the filter output to the desired (lower) sampling rate. The only antialiasing required in this case would be at the high (oversampled) sampling rate. If the high speed sampling were always to be performed at the one, high rate, then only one analogue filter would be required.

There are some difficulties with this technique. In order to preserve desirable linear phase characteristics, the digital anti-aliasing filter of choice is an FIR filter. However, to achieve a rapid cutoff, the filter needs many taps. Using brute-force, the FIR will need to be processed once per input sample. ie many taps at a high rate. Using current signal processing devices, the primary sampling rate would be limited to tens of kilohertz. For this frequency range solutions are available using switched capacitor techniques, so nothing is gained.

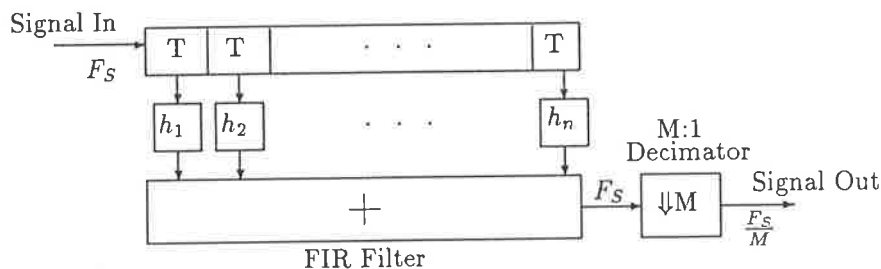


Figure 3.6: Cascaded FIR Filter and decimator. This implementation is not efficient, since for every one output sample,  $M$  FIR outputs are generated, and  $M - 1$  outputs of the FIR filter are discarded.

However, by applying a commutation operation to the combination filter and decimator, an alternative representation of the system is produced.

By commutating the coefficient weighting operations of the traditional FIR structure with the decimation operation which would follow it, we arrive at a modified FIR in which the decimation operation precedes the coefficient weighting operations (see figure 3.7). The important result of this is that the FIR computations (multiply-add) need only be performed at the decimated (ie lower) output rate of the filter[7].

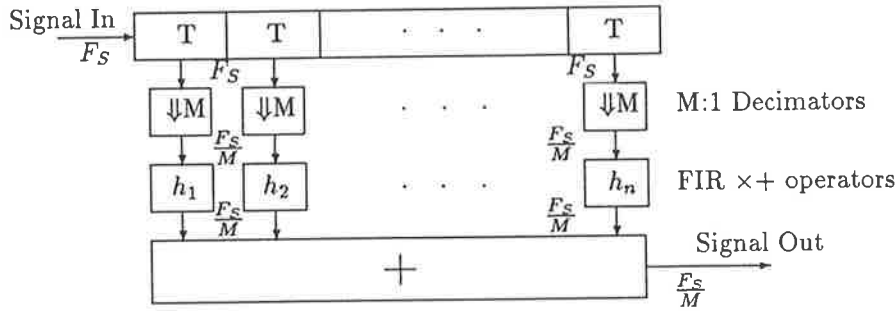


Figure 3.7: The FIR filter and decimator in figure 3.6 has been modified by commutating the decimation and multiply-add operations. This yields a much more efficient system in which the time consuming operations (multiply-add) are performed at the output rate, which is  $M$  times lower than the input rate.

For a decimation rate of  $M$ ,  $M$  less computations are required for this new structure than for the unmodified one. A further reduction in the number of multiplication operations can be achieved by exploiting the symmetry of the FIR impulse response.

The resulting reduction in computation allows much higher primary rates to be achieved. The benefit of a higher primary sampling rate is that the primary anti-aliasing filter can be of low order, since it has an extended frequency range (between the highest frequency of the signal which one wishes to retain, and the primary sampling frequency.)

Whatever the technique used to achieve anti-aliasing at the final sampling frequency, the system will need one primary analogue antialiasing filter. The discussion of system bandwidth in the previous item suggests that the slow roll off of the all-real-pole filter makes it unsuitable for this role.

A better filter is the Butterworth[9] filter. An  $N$ -th order butterworth filter has the desirable properties of maximal flatness in the passband, and relatively low variation in group delay in the transition region. A near-constant group delay (analogous to a linear phase characteristic) ensures that the filter has an acceptable transient response. This is necessary to ensure that accurate time domain representations of waveforms can be recorded by the GI.

The characteristics of the Butterworth filter are shown in figure 3.8. Even for a 6-th order filter, considerable aliasing will occur unless the cutoff frequency is moved to  $F_s/3$  (for an 8-bit converter.) This will allow the use of 66% of the converters bandwidth for valid signal measurement, with the remaining one-third being the transition region of the antialiasing filter.

## 7. Sampling precision

Being a sampled data system, this module will have a finite sampling precision, usually either 12 bits or 8 bits. Converters of 8-bit precision are available with very high conversion rates (eg 200MSPS in the case of the AD770). Such rates are achieved by employing flash conversion techniques, which use one voltage comparator for each of the possible output codes of the converter (256 in the case of 8-bit), and a priority encoder to generate the resulting binary code.

Flash 8-bit converters with conversion rates of 20MSPS were used in the prototype analogue interface of the GI. The high speed of the flash converters is achieved at the cost of silicon area on the die: a single extra bit of precision requires twice the components. An additional penalty

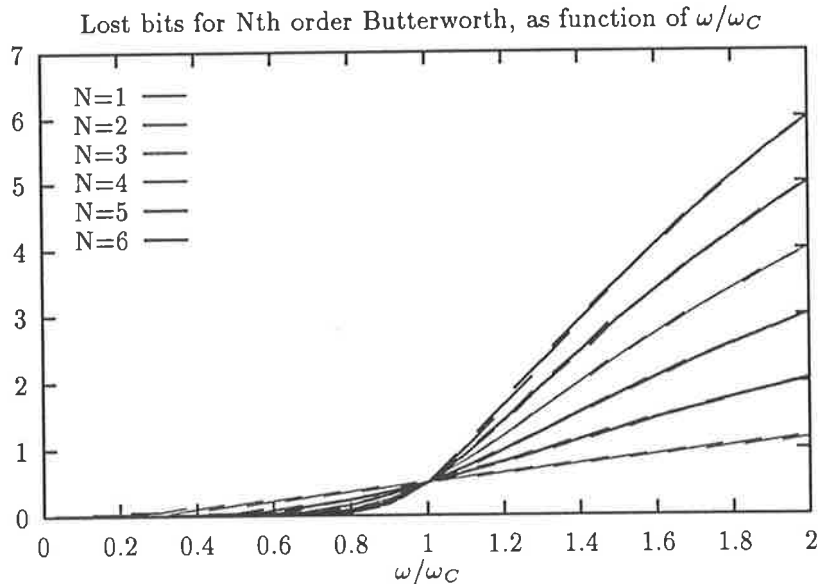


Figure 3.8: Loss of A/D resolution as a function of  $\omega/\omega_C$ , for N-th order Butterworth filter.

is the need for accuracy in the reference voltages supplied to the voltage comparators within the flash converter, to ensure the linearity of the device. These two factors have prevented flash techniques from being scaled to arbitrary precision.

While 20MSPS flash converters are relatively common, higher precision flash converters are not. Devices of 12-bit precision are now appearing on the market which employ a two step flash conversion process, where the first step provides the six most significant bits, and the second step the least significant 6-bits.

As well as converters, associated circuitry such as voltage references and sample-hold amplifiers are required. Fortunately, there now exist integrated sampling converters which combined all of these features.

At the time of design, fast (>1MHz rate) 12-bit converters are still prohibitively expensive for use in the GI. However, lower sampling rate (100kHz) 12-bit devices are expected to be used in future analogue interface designs.

### 3.3 Implementation

This section discusses the detailed implementation issues of the hardware. An architectural overview of the whole system is shown in figure 3.3. The system is card-based, which provides a considerable degree of flexibility in that new boards can be incorporated into the system without difficulty. Multiple boards of the same type (for instance multiple DSP boards) can coexist in the one system, allowing the system to be easily configured for more specialised applications.

A block diagram of the major GI system modules is shown in figure 3.3. The backplane bus does not conform to any industry standard, since considerable engineering effort (ie cost) and circuitry is required to conform to present 16-bit bus standards. The bus is an asynchronous one, largely based on the 68000's external timing. It is a single master bus, capable of addressing 16 megabytes of address space. The data paths in the system are 16-bit.

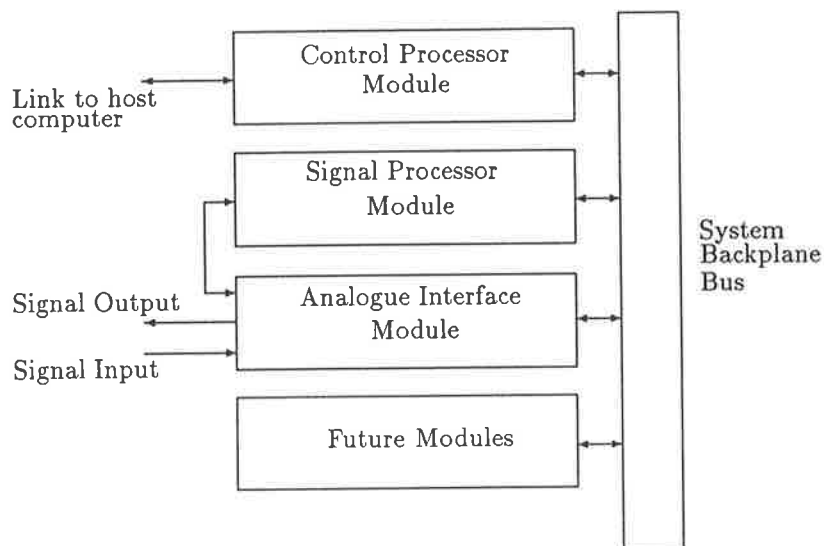


Figure 3.9: Major GI hardware modules

The control processor acts as the slave of the host computer, and controls the GI according to commands transmitted from the host. In normal operation, the processes are invisible to the user, who simply sees that the appropriate data has been displayed on the screen.

In some cases the user will need to see more deeply into the system, as in the process of software development for the signal processor. The control processor aids in software development by allowing the user to inspect in detail the program and data memory areas of the TMS320C25. The signal processor's support for multiprocessing (specifically, its ability to hand control of its busses over to another processor) allows the user to examine program and data memory before, during, and after DSP program execution.

As well as being useful for software debugging, this feature is useful for the following reasons.

1. The ability to load program code into the TMS320C25's program memory without intervention from the DSP allows the whole 64k words of external program memory to be implemented.

If the program memory could not be accessed from outside the DSP module, the signal processor would need to be supplied with bootstrap code in ROM. This bootstrap code would then communicate with the control processor to move downloaded program code into the DSP's program memory.

The presence of such bootstrap code would reduce the address space available for downloaded code. It would also mean that 3 distinct operating systems are involved in the GI (the host, the control processor, and the DSP), adding undesirable complexity.

2. Since the input/output ports of the TMS320C25 share the same address and data busses as the program/data memory, the control processor can directly access real world interfaces (A/D, D/A, digital ports) without the DSP.

This feature is essential when the user needs to verify the correct operation (or calibration) of interface hardware without having to write DSP software first.

3. Test during manufacture. A test-jig can be created with a control processor and backplane assembly. The control processor can be programmed to fully test all circuitry on the DSP board, allowing any fault to be localised.

### 3.3.1 Control Processor

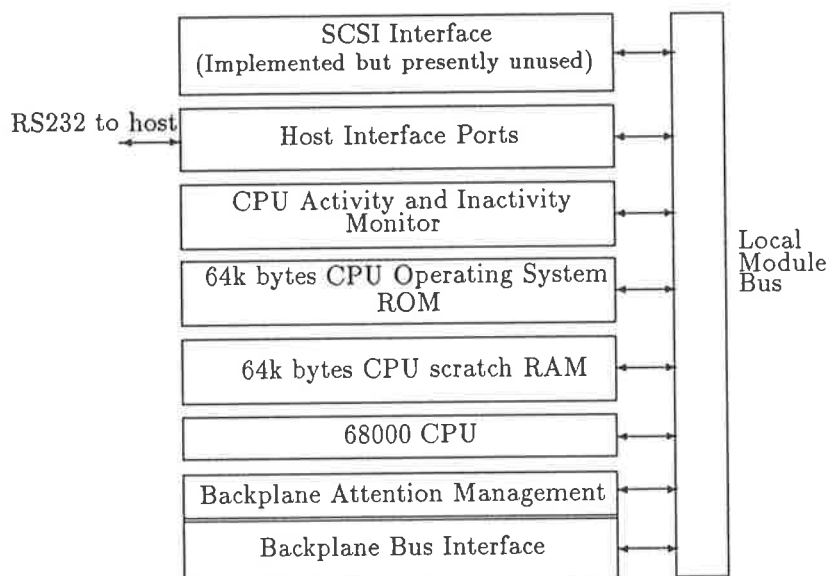


Figure 3.10: Control Processor Module block diagram

As can be seen in figure 3.10, the control processor module is composed of the following major subsections.

- **CPU, with associated ROM, and local RAM.**

The CPU, a 68000 family processor, was chosen because of its low cost, simple circuit implementation, and 16-bit bus structure (which makes it compatible with the 16-bit structure of the digital signal processor.)

The card is equipped with 64kbyte ROM which contains the operating system for the control processor, allowing it to communicate with the host as soon as power is applied.

There is a local RAM area of 64kbytes. This RAM is for local use by the control processor, and is used to store temporary variables, and data in transit between the host and the signal processor. It is not intended as a sample storage area. Future expansion for the GI will include large memory arrays for storage of sample data.

- **Serial port to host.**

The serial port is RS232 compatible, allowing the GI to be connected directly to any host equipped with such a port. This allows virtually all PC-clones to be used as host without any modification.

Communication over the serial port occurs at a rate of up to 57600bps, giving a maximum throughput of 5.76k bytes per second.

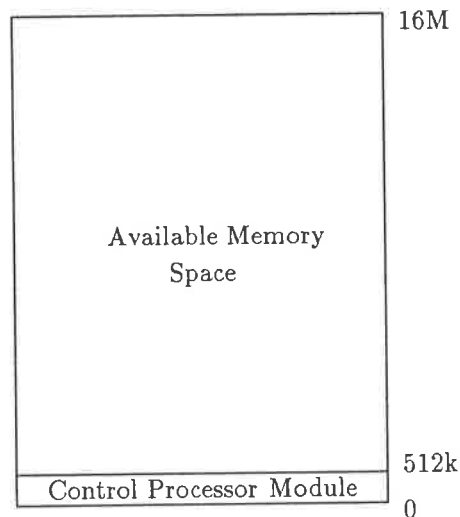
Communication over the serial link uses a packet-based protocol featuring error detection and automatic retransmission. Such a scheme is needed to prevent corrupted transmissions from causing incorrect system behaviour.



- **Backplane interface.**

The control processor acts as the master of the backplane bus. On that bus it can address a 16 megabyte address space (minus the 512k reserved for its own peripherals and local memory), which can be fully populated with peripherals.

All slave modules (eg the DSP module) feature configurable decoding, allowing multiple identical modules to exist within the 16 megabyte address space.



- **CPU activity and inactivity monitor**

The control processor has fixed ROM code for the reason that it should operate in a predictable manner from powerup, and it should operate reliably under all conditions, so that the host may maintain control of the GI at all times.

There are two circuit conditions which could cause the control processor to deviate from normal operation.

1. Attempting to access a device in an unused area of the control processor memory map will cause an indefinitely extended bus cycle.  
To detect this condition, a bus activity timer has been implemented which causes the extended cycle to be aborted after some tens of microseconds. The 68000 at this time begins bus-error exception handling, which culminates in a return to the IDLE COMMAND state, in which it awaits further host commands.
2. Execution of a corrupted instruction by the 68000, or a software flaw, may cause the 68000 to enter an inactive state (eg after executing the STOP opcode, or experiencing a double bus fault). In this case, the 68000 would normally need to be physically RESET. An inactivity timer detects this condition by looking for an extended period during which the 68000 initiates no bus cycles. Should this happen, the timer automatically resets the 68000. If the hardware is still functional, the control processor will enter the IDLE COMMAND state, and wait for host commands.

- **parallel port to host, SCSI port**

These ports have been implemented to allow future evolution of the GI. They are not used in the current system.

The parallel port, in conjunction with a plug-in-card for the PC host, would provide a much faster communication link between the GI and the host (although the RS232 link will always be available.)

The SCSI port is intended to allow the GI to control an SCSI hard disk, with the goal of storing samples directly to hard disk. This will allow sampling at medium rates for extended periods. The data will then be uploaded to the host, in packets of manageable size, for processing and display.

The SCSI interface provides a high bandwidth (up to 1.5 megabyte/second) interface which is available on many PCs and workstations, and therefore is a possible successor to the GI's RS232 link. Unfortunately, most PCs equipped with SCSI can only use the bus for mass storage devices, and not for intelligent peripherals in general.

### 3.3.2 Digital Signal Processor

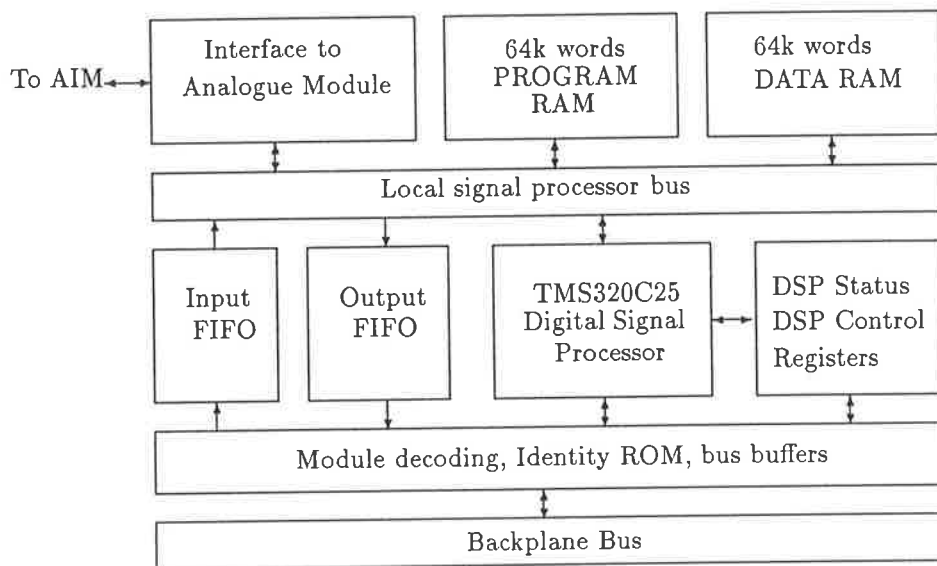


Figure 3.11: Digital Signal Processor module block diagram

A block diagram of the DSP module is shown in figure 3.11. The main features of that module are:

- **The Digital Signal Processor, a TMS320C25**

The version used in the design is capable of execution at 10MIPS (1 MIP = 1 Million Instructions Per Second), though new versions of that device are now available with greater and lesser performance.

- **Program RAM**

The TMS320C25 is capable of directly addressing 64k *words* of program memory, and 64k words have been implemented on the board. Note that the TMS320C25 supports only 16-bit words, whereas the 68000 supports 8, 16, and 32-bit words.

- **Data RAM**

The DSP can address 64k words of data memory, and the full 64k has been implemented on the board.

The TMS320C25 processor is structured around 'Harvard Architecture', which is an over-used term used to state that it has separate internal data and program busses. Unlike a standard

microprocessor, in which data coexists in one degenerate RAM area with program instructions, the TMS320C25 separates them into distinct partitions.

While this architecture has benefits within the silicon of the TMS320C25 itself (for instance, instruction fetches can occur simultaneously with data moves), it is of no benefit outside of the device, since program RAM and data RAM must share a common data and address bus.

- **Bus interface**

The address, data, and control busses of the TMS320C25 may be disabled, allowing another processor to take over its operation. This feature has been exploited to allow the control processor (via the backplane bus) to take complete control of the DSP's busses.

While this involves considerable circuitry, this facility is necessary for several reasons, as described in section 3.3.

- **Interprocessor Communication**

While the bus interface allows the control processor to invade the address spaces of the TMS320C25 at any time, it may not be desirable to do this (except possibly when debugging DSP code) while the DSP is executing its programs. If the DSP is executing real-time code, such intrusions will interfere with correct operation, for instance by delaying sampling instants.

To allow interprocessor communication without the problems of bus contention, a bidirectional port mechanism has been provided which allows data to be transferred between the control processor and the DSP without the DSP delegating control of its busses.

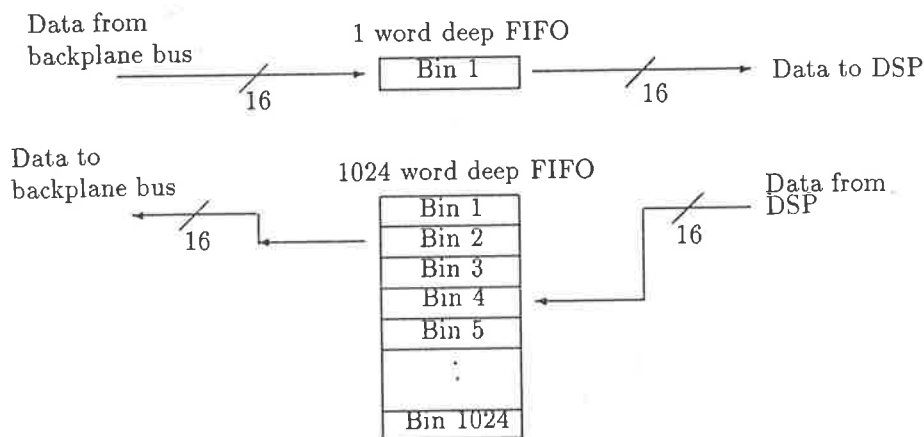


Figure 3.12: DSP and Control processor interprocessor port

The interprocessor port is bidirectional, but not symmetrical, as explained below.

- **Control Processor to DSP**

The port from the backplane bus (ie from the control processor) to the DSP is one word deep. That means that the control processor may write one 16-bit word into that port, and must wait for the DSP to read that word out before the next word can be written to the port.

- **DSP to Control Processor**

The port from the TMS320C25 to the backplane bus is a 1024 word deep FIFO. This means that (assuming the FIFO is initially empty) the DSP can write 1024 words to it without waiting for the control processor to read any out.

In practice, the DSP will be executing signal processing code, possibly in real-time. The control processor will be communicating with the host, and may have a poor response time for servicing the DSP.

Therefore it is important to move any communication bottlenecks away from the DSP, so that it does not waste time waiting for communication channels to clear before resuming computation.

The FIFO devices used in the DSP module are industry standard parts, allowing 4k and 8k words of FIFO to be implemented by choosing suitable (pin compatible) devices, though at increased cost.

Both the DSP and the control processor may access the ports asynchronously. Hardware flags may be tested to determine whether the ports are full, or available for data transfer. If a port is full, the processor must wait until it is cleared (by the processor on the other side of the port reading data out of it) before more data can be transferred.

The FIFO system has not been built symmetrically. The intended uses of the GI at the current time will see the GI returning digital data to the host. The larger FIFO was placed into that data path to allow the signal processor to resume computation with minimum delay.

- **Attention Request Mechanism**

Since the system is designed to support multiple boards (which may or may not be DSP boards) on the backplane bus, a mechanism has been provided to allow boards to request service from the control processor without the need for the control processor to continually poll the boards.

The attention mechanism consists of 8 attention request signals. Different client boards can signal their request for servicing on different signals. These signals generate vectored interrupts to the control processor, allowing rapid response.

### 3.3.3 Analogue front end

The analogue front end described herein is a prototype unit which has been used for testing, and also for teaching in the laboratories. While it is functional, it provides only a bare minimum of facilities, and should only be used as a simple example by those building more complete analogue interfaces.

A block diagram of the module is shown in figure 3.3 while the important features are shown in table 3.1.

Antialiasing filters were omitted in the initial design. The difficulty associated with making variable bandwidth filters has already been discussed, and the use of external filters was considered reasonable.

Flash A/D converters have the desirable property of fast conversion, in this case 50ns. This is less than the instruction execution time of the DSP, allowing triggering to be immediately followed by reading of the resulting value. Two input channels are provided, and are triggered simultaneously to begin conversion.

Two channels of D/A are provided, which are also updated simultaneously. Both input and output channels have variable gain and offset, which are set using trim resistors. Control of these circuit parameters must be made digitally programmable in future versions of the module, since thermal drift and component variations cause the loss of accuracy in the A/D and D/A process.

The GI, designed as it is for measurement purposes, must be constructed to yield a precise relationship between the numbers being viewed in the machine (ie the samples), and the voltages which they represent.

Input Channels	2×8-bit.
Input conversion time	50ns
Input sampling	Simultaneous
Input Gain	Digitally programmable $gain = [1, 2, 4, 8, 16] \times [1, 2, 4, 8, 16]$ Only one value is chosen out of each set in [ ]
Anti-aliasing filter	None
Input Bandwidth	approx 350kHz
Output Channels	2×8-bit.
Output updating	Simultaneous

Table 3.1: Prototype AIM Specifications

Modules such as the prototype AIM (Analogue Interface Module), which rely on calibration by a user (via trimpots) are prone to drift and incorrect calibration. For this reason, future versions of the analogue interface module will be based on precise voltage references and tight-tolerance components, which will provide the necessary precision without user intervention.

More information about the this AIM is found in appendix A.

## Chapter 4

# Software Architecture

As the major point of interaction with the Generalised Instrument, the software plays a crucial role in determining the usefulness of the system. As well as providing a control environment for the GI hardware, the software must provide a means of displaying the data generated by the GI on the PC screen, and enable the user to interact with that data in useful way.

Since the goal of the Generalised Instrument was to design a system which could emulate different instruments, the host software must also be flexible enough to allow such emulation. This may mean presenting a different screen display, causing different DSP algorithms to be used, and performing different data manipulations within the host itself.

### 4.1 Requirements

Listed below is a basic set of features which the host software must exhibit.

1. Provide communication to the GI hardware, and control over its various modules. This involves:
  - Transfer of data to and from the digital signal processor within the GI.
  - Transfer of program to the DSP.
  - Provide control over the execution status of the DSP.
  - Debugging facilities for DSP code.
2. Management of arrays of data, as may be uploaded from the GI.
3. A graphical user interface which allows the user with minimal training to interact with the GI. This interface must support both display and interaction.
4. A means of storing data on the PC's disk for future analysis.
5. A means of creating a preprogrammed application environment, which the user can invoke easily. Such environments would typically emulate different test instruments, such as a CRO or spectrum analyser.
6. Management of a library of DSP code modules which will be used in conjunction with the preprogrammed applications to process data in a required manner.
7. Generation of hardcopy of data displays.

## 4.2 Major Design Decisions

In this section major design decisions which affected the construction of the software are discussed. The engineering of the GI project was split approximately in equal amounts between the hardware and the software.

It is often difficult to appreciate the complexity of a software system, since it is largely invisible. Unlike hardware, where an experienced engineer can gauge the performance of a system just by inspecting the printed circuit board, the depth of a software system is hidden until one begins to use it.

### To compile or interpret? — That is the question!

Two main approaches to the host software were explored. Their main difference was in the way that the user's input (ie textual commands) was processed. Early work with an interactive signal processing package, Sigproc[5], showed it to be an effective tool, especially for teaching. Sigproc is a command line driven signal processing package which has a repertoire of signal processing commands.

The commands are effectively invoked as soon as the user types the command's name. The command is usually followed by arguments, which are processed differently by each command's code. Therefore, the same argument could mean different things to different command handlers.

```
chirp signal1 0 200           generate a chirp 0-200Hz, place values in array 'signal1'
window;hamming signal1 signal2 apply a window function to 'signal1', place result in 'signal2'
fft signal2 signal3          'signal3' now holds fft of 'signal2'
plot signal3                 graph the values of 'signal3'
```

Figure 4.1: A sample Sigproc script. The first word on each line is the command name, the words following it are arguments (usually either numbers or array names.)

A merit of such a command line interpreter exists in the ability, of program code for each command, to check the user's input at each stage. The user may be prompted for further information (or corrected information) if required:

```
chirp                               (incomplete command line)
What is the name of the destination array? signal1
What is the starting frequency? 0
What is the ending frequency? 200
```

Figure 4.2: Sigproc commands may prompt for missing arguments. This reduces the mind-load on the user, since each command presents a standard form for the entry of vital information.

This is a useful way to build an interactive system, since it is highly tolerant of errors in the user's commands. If the user supplies incorrect or ambiguous information, the code can take corrective action, or simply assume default values for parameters which the user omitted.

There are three main reasons why Sigproc was deemed unsuitable as a model for the host software.

1. Sigproc lacked the ability to understand expressions.

Rather than allowing a form such as  $a = b \times c$ , Sigproc required a specific multiplication command which was invoked as 'mult a b c'. The latter is less natural for mathematically inclined users. While some form of expression handling could have been built into Sigproc, it would not have been a pleasant marriage, since its use would have been restricted by Sigproc's general syntax.

2. Sigproc lacks control structures, such as IF-THEN, WHILE, and others which programmers are familiar with. While Sigproc is useful for interactive processing, where a user is present to type in commands, it is not able to execute algorithms which are not hard-coded into the Sigproc package. Sigproc did not provide any of the execution flow-control structures needed to write effective algorithms.

The most important result of the inability to control program flow is that the Sigproc language is unsuitable for writing signal processing algorithms. If the user must implement an operation which is not in Sigproc's dictionary, the only way of performing the operation is to either a) write a new command into Sigproc to perform the desired operation, or b) write an external program to implement the user's algorithm.

In both cases, the user must be a proficient programmer (or have access to such), and be familiar with Sigproc's internal construction. Neither is a suitable solution.

Sigproc's lack of support for control structures follows from the way in which Sigproc executes lists of commands. Sigproc may be given a list of commands to execute from a file, but in reality this is no different from the user typing them at the keyboard. Each command is interpreted, executed, and discarded before the next is processed. Because each command is discarded after execution, all sense of algorithm structure is lost. No branches can be taken, since the code which would be branched to has either not yet been processed, or has been discarded.

It would not be impossible to add primitive control statements to interpreters such as Sigproc, but the result would be a language which would more closely resemble an *assembly language* than a high-level language (see figure 4.1.)

3. As an interpreter, Sigproc parses its input as text. Were it to be given some form of looping ability (as was done in an experimental version of Sigproc), at each pass through the loop it would be parsing the same textual program code. Now, since Sigproc cannot modify the text program which it executes, it would be parsing identical code on each pass through the loop. The command statements would be identical, as would the text names of the arguments to those commands. The values of the arguments may be different on each pass through the loop, but the textual name of the argument is not.

```
start:                                % label
  upload foo                          % get a packet of samples from GI
  fft;polar foo foomag foophase      % fft it to get the magnitude
  display foomag                      % display the result
  goto start
```

Figure 4.3: Sample Sigproc code

For instance, consider the script in figure 4.3. The data values contained within the array called 'foo' may change each time the loop is executed, but the code being executed within the loop is always the same. Therefore, interpreting the text of the program on each pass of the loop is wasteful of time. The first pass through the interpreter caused all of the necessary actions to be performed.



There exists a way to extract the user's intended algorithm from the text of her program: it is the process of program compilation. This is the technique used by SPaM, and is described in detail the section 4.3.

### What style the User Interface?

Several choices were available for the user interface:

1. Command line driven (example: Sigproc).

Here the user would type commands, which would be executed and the results displayed.

The commands would either be taken directly from the keyboard or from prepared script files.

2. Text menu driven.

In this scenario, the user would be presented with text menus. She would choose an option from a menu, resulting either in an action being taken by the system, or another menu being presented.

3. Graphical.

A full graphical interface with interactive screen displays is ideal for the GI. It will allow the PC to simulate on screen the controls and display one would expect to see on the front panel of a physical test instrument.

By providing an air of familiarity, the display will allow the user to begin work immediately, without needing to study a user manual.

Like the GI as a whole, the host software must be sufficiently flexible to allow it to be moulded for new applications without extensive low level programming. For this reason a combination of command line and graphical display was chosen.

A command line interface was considered essential as it provides the most expressive way for the user to state her requirements. When processed by a well-designed parser, the command line is capable of processing full arithmetic expressions, and allows advanced loop constructs to be used to build new algorithms.

SPaM's command line interaction can be carried out on the graphical display screen. The commands and their responses are displayed in a window reserved for that purpose (called the 'console window'), and share the screen with other display window. This eliminates the need to constantly switch between text and graphics displays (as in Sigproc), which can be distracting.

## 4.3 Host Software

The software package which runs on the host PC is called SPaM. This section will deal with the architectural aspects of SPaM. A detailed reference for the SPaM language can be found in chapter 5.

A block diagram of the SPaM package is shown in figure 4.4.

The essential parts of SPaM are described below.

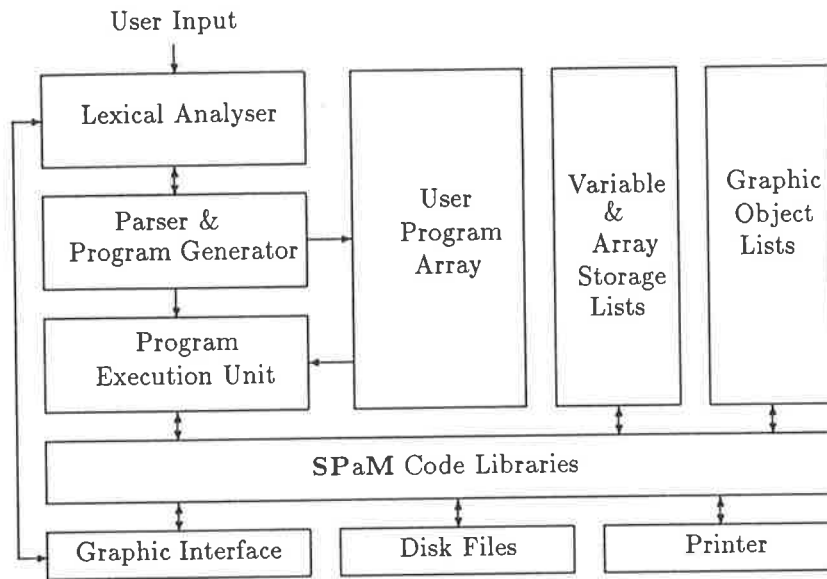


Figure 4.4: SPaM block diagram. The logical link between the lexical analyser and the graphic interface represents the mixing of user-generated events from the keyboard and mouse.

### 4.3.1 The Lexical Analyser

The lexical analyser is a piece of code which examines the source input to SPaM (which may be text commands typed by the user at the keyboard, or fetched from a file on disk), and breaks them up into 'tokens'. The tokens are then passed to the parser (see next item).

A token is an atom of language. It consists of one or more character symbols. These symbols may be alphabetic letters, numeric digits, or other keyboard symbols. The boundaries of a token are defined by either the occurrence of a whitespace character (space, tab, or newline) or a character not consistent with the current token's type.

For instance, if the lexical analyser encounters a numeric character, then that character forms the beginning of a numeric token. Further characters will be read (and a number will be constructed) until a non-numeric character is encountered. When encountered, this boundary defines the end of the current token and the beginning of the next (but whitespace is skipped.)

Consider the following line of text.

```

WHILE (i<100)
  i=i+1;
END

```

The tokens in this short program are shown below, each token surrounded by [].

```

[WHILE] [(] [i] [<] [100] [)] [i] [=] [i] [+] [1] [;] [END]

```

In the example above, the analyser can identify all of the tokens except for the occurrences of the letter *i* which are not part of the token `WHILE`. When confronted by such an unknown token, the

lexical analyser must come to a decision as to how to classify the token, so that the parser may decide whether the user is making sense or not. The following rules are used by the lexical analyser to make its decision.

1. If the token begins with a numeric digit, assume it is a number (a constant). Keep reading characters until the first non-numeric character is encountered, and form a number from those digits. Return that number to the parser.
2. If the unknown token appears on a line all by itself, it may be the name of a command file<sup>1</sup> which the user wishes to run. Look for a command file of that name in the search path. If such a file exists, open it for reading and make it the standard input (so that the file will be read as command input.)  
If no such file exists, fall through to rule 3.
3. Search the list of user defined functions. If the token is the same as the name of a function, return a corresponding symbol to the parser.
4. Search the list of existing variables. If the unknown word is the same as the name of one of those variables, notify the parser of that fact.
5. If the unknown word corresponds to none of the above, assume that it is the name of a variable which does not exist (simply because this is the first time it has been encountered), and mark it as having an undefined type.

From this point on, an occurrence of the same token will be caught by rule 4.

Variables need not have a known type at compile time, as long as their type is clear at run time. A variable's type is fixed when a value is assigned to the variable.

The lexical analyser is called from the parser (described below). As the lexical analyser splits the input stream into tokens, each token is returned to the parser.

### 4.3.2 The Parser

The parser is that part of SPaM which inspects the tokens being produced by the user, and decides how they must be processed.

When the user types a stream of character into the SPaM command line, it will either make sense or it will not. In software jargon, it is said to either parse correctly, or not.

The parser decides whether something makes sense by comparing it to a set of rules which define the SPaM language. In fact, the parser was created from these very rules by the automatic parser generator called *bison*, which is a public domain derivative of the Unix *yacc*[8] program.

For each rule in the SPaM grammar, there is a corresponding program action to be performed when an occurrence of that rule is encountered in the user's program text.

An example of a rule is shown in figure 4.5. This rule describes how a valid expression is constructed. According to this rule, an expression may consist of either a number or variable (both of which yield a value without having to be broken down further), or some arithmetic combination of other (simpler) expressions.

In the case of SPaM, each an occurrence of a rule causes SPaM to add the appropriate code to its pseudoprogram for execution after the entire input has been parsed.

---

<sup>1</sup>A command file is simply a text file containing a SPaM program. Programs may be entered into SPaM interactively, or read from a disk file.

An example of the code generated by SPaM for a numeric expression is shown in table 4.1.

```
expression :      number          ( action: yield a value )
                | variable       ( action: yield a value )
                | expression '+' expression ( action: add )
                | expression '-' expression ( action: subtract )
                | expression '*' expression ( action: multiply )
                | expression '/' expression ( action: divide )
```

Figure 4.5: Sample yacc rule. The SPaM parser is built by yacc from a complete, formal, specification of the language, similar to the above. The vertical bar | means 'or'. The parentheses enclose the actions to be performed when an occurrence of that rule is found.

The parser can understand self-referencing rules, which makes it a powerful tool for breaking down complicated expressions. In the rule of figure 4.5, expressions consist either of literals (ie constants, variables, or symbols like '+'), or combinations of smaller expressions. Thus complex program structures are broken down into manageable pieces.

In the rule of figure 4.5, the `number` and `variable` cases correspond to actual tokens identified by the lexical analyser. If the tokens arriving from the lexical analyser do not correspond to any known rules, the parser will signal the user that a syntax error has occurred.

As rules are successfully matched to the arriving tokens, corresponding program code is written to an array maintained by SPaM. This array holds pointers to functions which will carry out the actions specified by the rule (and its matching tokens supplied by the user.)

This array of pointers becomes the 'program' which is executed after the compilation process has been successfully completed.

### 4.3.3 The Program Execution Unit

After the parsing step, during which the 'compilation' actually occurs, the array of function pointers which now represents the user's program must be 'executed.' It is during this execution phase that the results which the user seeks are generated.

Execution of the 'program' is accomplished by stepping through the array of function pointers which was generated during the parsing phase, and calling each one of those functions in turn.

The execution phase of SPaM emulates a virtual processor. The functions which are called from the 'program' array represent the object-code instructions of this virtual processor. The 'object code' representation of the user's code is generated during the parsing phase.

The nature of the virtual processor is important since it determines how complex arithmetic expressions will be processed. The SPaM virtual processor uses a stack-based architecture[6], which is very convenient for arithmetic computation (see table 4.1.)

The virtual processor stack is used to hold pointers to numeric objects, and arguments and return-values for functions calls. The program for a stack-based processor resembles programming languages such as FORTH. The stack is also called the 'evaluation stack', due to its role in evaluating expressions. A stack based processor does not have general purpose registers; the stack is used instead.

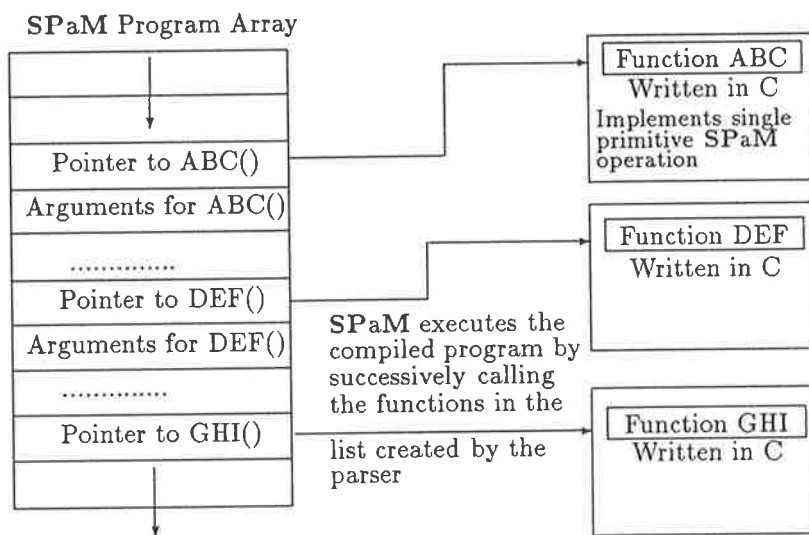


Figure 4.6: The SPaM execution model. During compilation (*parsing*), SPaM places pointers to functions in the Program Array. This pointer table is the pseudoprogram which is then executed by calling each of the corresponding functions in turn. In this diagram, execution begins at the top of the array and proceeds toward the bottom.

For example, consider the following text line as the input to the parser.

$x = ((y + 3) * z)$

After passing through the parser, the program array will have the contents shown in table 4.1.

Program Index	Operation/Operand	Comment
1	PUSH CONSTANT	The item from the following array element is pushed onto the evaluation stack.
2	3	This is an argument to the preceding instruction, and is skipped.
3	PUSH VARIABLE	The following variable is pushed onto the evaluation stack.
4	y	This is an argument to the preceding instruction, and is skipped.
5	ADD	Pop two arguments from the evaluation stack, add them in a manner appropriate to their type, and push the result back onto the evaluation stack.
6	PUSH VARIABLE	The following variable is pushed onto the evaluation stack.
7	z	This is an argument to the preceding instruction, and is skipped.
8	MULTIPLY	Pop two arguments from the evaluation stack, multiply them in a manner appropriate to their types, and push the result back onto the evaluation stack.
9	PUSH VARIABLE	The following variable is pushed onto the evaluation stack.
10	x	This is an argument to the preceding instruction, and is skipped.
11	ASSIGN	Pop a variable from the evaluation stack, and then pop a value from the stack. Assign the value to the variable.
12	PRINT	Pop one value from the evaluation stack, and print it to the screen.
13	STOP	Halt program execution.

Table 4.1: An example of the program which the parser creates. This one would implement the expression  $x = ((y + 3) \times z)$ .

The 'program index' value in column 1 of table 4.1 is the effective 'address' of each program instruction. It states where an instruction may be found in the array, and is therefore analogous to the address in a microprocessor system. The Program Array in figure 4.6 can be considered to have address 0 corresponding to the top of the array.

To understand the pseudoprogram in table 4.1, one must remember that SPaM implements a *stack based* virtual processor to execute the pseudo programs. All fundamental operations of that processor (ie its assembly language, as exemplified by `PUSH VARIABLE`, `ADD`, `MULTIPLY` in table 4.1) are stack operations.

This means that all pseudo-instructions expect to find the correct number of arguments on the stack when they are called, and all results are placed on the stack. The evaluation stack provides a supremely simple mechanism for connecting multiple instructions in sequence.

The example of table 4.1 has been somewhat simplified by omitting the actual way in which objects such as the constant '3' and the variable 'x' are treated. SPaM is an object-oriented system which manages objects in a linked list. All atoms of the language, be they constants (such as the '3'), variables (such as x,y,z), or other, are represented by objects. An object can be considered as a 'box' containing some value. The numeric value, and the type of numeric value, can change during the calculation. Section 4.3.9 further discusses SPaM's internal object types.

Operators, such as the `ADD` and `MULTIPLY` operators seen in table 4.1, process *objects*. When called, these operators pop their requisite number of objects from the evaluation stack, and based on the *type* of the arguments, perform the required action. The addition operator, `ADD`, pops two objects from the stack, adds their values, and places the result back on the stack.

Due to the variety of data types found within SPaM, it is possible to specify arguments of inappropriate type, in which case a runtime error will be signalled, and the user's program will be halted.

#### 4.3.4 The Math Function, and Primitive Math Operation Libraries

The math function library implements transcendental and other high level functions, while the primitive math library contains elementary operations such as `+`, `-`, `*`, `/` and logical-test operations. The main difference between the two is that the primitive library performs its operations in the representation of the arguments, while the math function library performs much of its calculations in floating point representation.

SPaM uses object oriented program techniques to process its multiple object types. Object oriented programming (OOP) is a paradigm for software systems in which similar operators (for instance, the standard `*`, `/`, `+`, `-` operators) are required to operate on many different classes of objects. The code which implements the operation must be capable of recognising arguments of different type, and adjusting its behaviour accordingly.

SPaM implements many numeric attributes, as shown in section 4.2. Numeric variables of any type may be mixed freely. Where possible, SPaM carries out operations in the same representation as the arguments, and produces a result of in the same representation. For instance, multiplying an integer by an integer yields an integer.

If the operands are of different type, SPaM will choose the type most appropriate to represent the result. For instance, multiplying a complex integer by a real floating-point number yields a complex floating point number.

The advantages of operators which operate on different types differently are:

1. The user has control over which representation is used, allowing integer based algorithms (such as those found on integrated DSP devices) to be simulated.
2. Computation can be sped up. On PCs without floating point accelerators, floating point calculation is cumbersome. Integer operations provide a way of speeding up such computation.

The numeric representations available within SPaM are summarised in table 4.2. There are several groups of attributes, with the attributes within each group being mutually exclusive.

All numeric objects have one attribute which determines the numeric representation used within the object (integer, floating, etc), another attribute determines whether the numeric quantity is real or complex, and yet another determines whether the object is a scalar or matrix quantity.

Attribute	Meaning
INTEGER	The number is represented as a 16-bit 2's complement integer.
LONG	The number is represented as a 32-bit 2's complement integer.
FLOATING	The number is represented as a double precision floating-point number.
REAL	The numeric object represents a real quantity.
COMPLEX	The numeric object represents a complex quantity.
SCALAR	The numeric object is scalar.
MATRIX	The numeric object is a matrix.
VLM	The numeric object is a Very Large Matrix (refer to 5.5).
STRING	The numeric object is a text string. This attribute alone defines strings, the above attributes are meaningless in the presence of this one.

Table 4.2: SPaM numeric object attributes. One attribute from each group (groups are separated by a thick line) applies to any numeric object.

### 4.3.5 The Graphics Library

The graphics library contains code to manage the SPaM graphic environment, including all of the graphical display items outlined below.

- **CRT Windows.**

CRT windows are designed for displaying waveforms without any additional information (such as axis numbering or titling.)

The waveform to be displayed is stored in a matrix variable. The dimensions of the data matrix determine how it will be displayed:



1. *Matrix is  $1 \times N$*

In this case, the the X axis represents the index value of the array. The first array element (eg  $\text{foo}(1)$ ) is displayed on the leftmost side of the window, and the last element (eg  $\text{foo}(N)$ ) is displayed on the right-hand edge of the window.

The array is first scanned to determine the maximum and minimum value, to allow the display to be autoscaled (autoscaling can be turned off if desired.) Then the array elements are plotted, with each array element determining the vertical ordinate of the waveform at that index point.

2. *Matrix is  $M \times N, N \neq M$*

If the matrix is rectangular, and the smaller dimension is greater than 1, then SPaM assumes that the matrix represents a set of vectors to be plotted in the same window.

SPaM assumes that the smaller dimension is the number of vectors, and that the larger dimension is the length of each of those vectors.

Thus, a  $3 \times 4$  matrix would be plotted as a set of 3 distinct vectors of 4 elements each. Autoscaling in this case is performed based on the value of the first vector in the matrix.

3. *Matrix is  $N \times N$*

If the matrix is *square*, then each row is assumed to be a data vector.

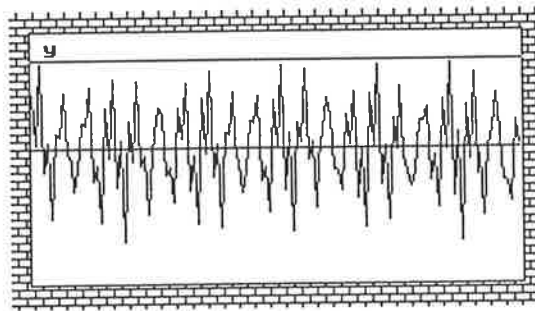


Figure 4.7: A typical CRT Window

- **Graph Windows.**

The *graph* window is similar to the *CRT* window, the difference being that a *graph* window is divided into 2 parts. One part is used for waveform display, and behaves in exactly the same manner as the *CRT* window.

The remaining area of the *graph* window is dedicated to the display of axis labels and numbering. By default, the X-axis is numbered according to the index position of the element being examined. The user may specify X-axis start and end values, which SPaM uses to calculate the value for any intermediate index position in the array.

The Y-axis is numbered simply according to the values in the array elements. The user may perform any desired algebra on these values before displaying them to get the required 'units' of display.

- **Argand Windows.**

The *CRT* and *graph* windows exist specifically to display one dimensional data, eg a signal the amplitude of which varies over time, sampled at some rate.

The *argand* window exists to display two dimensional data. The data is displayed on a complex plane, and so the data must be complex. The real part is displayed as an X-coordinate, while the imaginary part is displayed as the Y-coordinate.

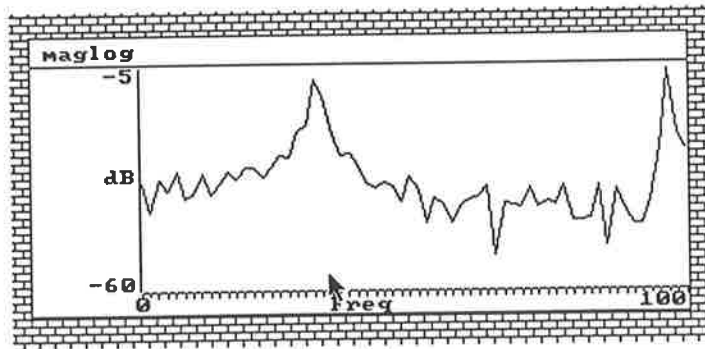


Figure 4.8: A typical *graph* window

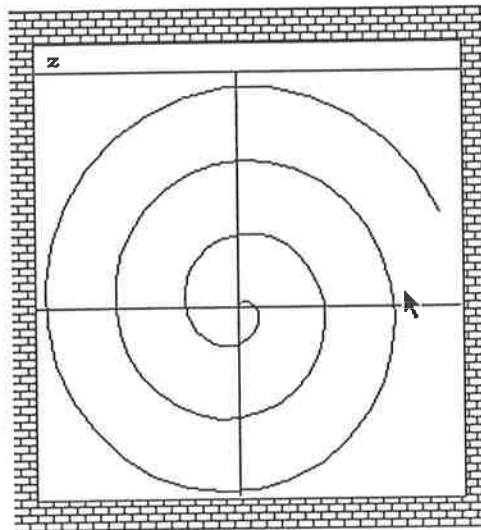


Figure 4.9: A typical *argand* window

The *argand* window allows polar diagrams to be displayed by SPaM (though the polar coordinates must be converted to rectangular coordinates.)

- **Buttons.**

A *button* is a rectangular window on the screen designed to be 'pressed' by the user, in the same way as a physical push-button switch would be pressed. The user 'presses' a *button* by clicking the left-mouse-button (LMB) over it.

When a *button* is activated by clicking on it with the LMB, SPaM looks for a *handler* with the same name as the button. A *handler* is analogous to a user-defined function, or subroutine.

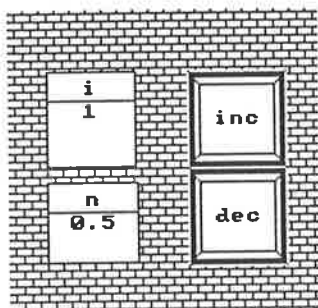


Figure 4.10: A typical *button* and *numeric*.

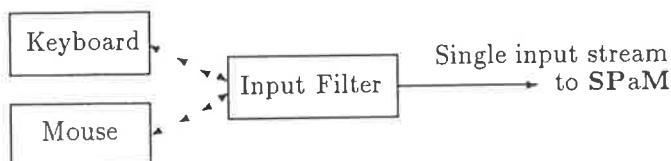
SPaM will execute the code in the *handler* corresponding to the *button*. This allows the user to create screen objects which can directly cause the execution of specific code modules.

- **Numerics.**

A *numeric* is a window dedicated to the task of displaying the value of a scalar variable. Whereas the *CRT*, *graph*, and *argand* windows display vector or matrix data, the *numeric* displays scalar data.

### 4.3.6 The Input Event Filter

SPaM accepts user input from the keyboard, and the mouse. To make the mouse and keyboard interchangeable, all input events (an input event is any operation which the user may do with the keyboard or mouse, such as pressing a key or a button) pass through a software 'filter' where they are converted into a single stream which SPaM can process. This allows the keyboard to take the place of the mouse, and vice versa, without additional code being required.



The input events generated by the user include:

- Normal keyboard character input.

- Keyboard ABORT signal (ESC, CTRL-BRK, CTRL-C)
- Mouse LEFT button, clicking on *button* windows.
- Mouse MIDDLE button, used to make measurements on waveforms in *CRT* windows.
- Mouse RIGHT button, used to pop up menus.

By combining input events into a single stream, the mouse and keyboard can be made equivalent.

### 4.3.7 The GI Hardware Control Library

A library of commands exists for communicating with the Generalised Instrument hardware. The library contains functions to perform the following:

- Download and upload data to and from the GI hardware.
- Download DSP program code from host to GI.
- Control the execution state of the signal processor in the GI.
- Communicate with the GI onboard monitor software.

### 4.3.8 Online Help

SPaM has an online help facility which may be called at any time from within SPaM.

The text database is external to SPaM, allowing it to grow without being limited by system memory.

### 4.3.9 SPaM as an Object-Oriented Environment

The diversity of data types which are encountered in signal processing is considerable, with the main structures being scalars, vectors, and matrices. Additional variation exists in the choices of real or complex data, and the form of the numeric representation (see 4.2).

A simplistic programming methodology would dictate that those types of data most likely to be encountered be adopted and the rest discarded, despite their usefulness.

SPaM implements most operations directly on the various combinations of argument types. Within SPaM, there are libraries of code which operate, at the user level, not on numbers or vectors, but on *objects*. An *object* is an abstract entity which may take the form of a constant, variable, or graphical entity (such as a *CRT* window.)

When the lexical analyser scans the user's program, it creates new objects to represent tokens which it does not understand. In all cases (barring spelling mistakes), these objects will refer to some user defined entity, for instance a variable or a function. Pointers to these objects are embedded in the program generated by the parser.

The parser does not, however, type-check the use of objects. This is sensible, since in most cases the objects will be undefined until assigned a value in some part of the user's program. The benefit resulting from not type checking at the parse stage is that objects can change type on-the-fly during the execution of a program. For example, matrices can change dimension during program execution, as shown below:

```

x=1;           % x is now a scalar of value 1
print x
x=eye(3);     % x is now a 3x3 identity matrix
print x
x=[0,x;x,0]; % build a new matrix which looks like
              %
              % new x =  [ 0 1 0 0 ]
              %          [ 0 0 1 0 ]
              %          [ 0 0 0 1 ]
              %          [ 1 0 0 0 ]
              %          [ 0 1 0 0 ]
              %          [ 0 0 1 0 ]
              %

```

In the above example, the variable `x` changes type from scalar to  $3 \times 3$  matrix to  $6 \times 4$  matrix. Variables may change type at any time without problem if the operations carried out on those variables are compatible with all of the forms which the variable takes.

Though type checking is not done at compile (parse) time, it is certainly performed at run time. All library code within SPaM checks the types of the arguments on which it is about to operate. Based on these types, the code chooses the most appropriate action. This action may include aborting program execution if the arguments are quite incompatible.

Consider the addition '+' operation as an example of how SPaM treats its arguments.

## 4.4 Control Processor Software

The control processor (a 68000 family device) exists within the GI hardware to control the operation of the GI hardware system, and to facilitate communication between it and the host computer.

The software executed by the 68000 is fixed in EPROM. This ensures that on power-up, the GI powers up in a known state, allowing the host to communicate with it immediately. The control processor is the only module of the GI system which has a program fixed in hardware (though it may be updated by changing EPROMs.)

The control facilities provided by the 68000 include

1. Direct control of the TMS320C25 `RESET` and `HOLD` signals.
2. Monitoring of TMS320C25 program execution to determine when DSP algorithms have completed execution.
3. Bidirectional transfer of data words from host to TMS320C25 data memory.
4. Bidirectional transfer of program words from host to TMS320C25 program memory.
5. Direct read and write operations to TMS320C25 I/O ports.

Communication between the host and the control processor is (at time of writing) via an RS232 link. Communication over the RS232 link is accomplished using a custom protocol. The protocol is packet based, and has error-detection and correction provisions. A typical command sequence is shown in figure 6.2.

$x$	$y$	$z = x + y$
scalar	scalar	scalar $z = x + y$
scalar	matrix	matrix $z_{ij} = y_{ij} + x$
matrix	scalar	matrix $z_{ij} = x_{ij} + y$
matrix	matrix	If $x$ and $y$ are of the same dimension, then matrix $z_{ij} = x_{ij} + y_{ij}$ otherwise an error is signalled and program execution is stopped.
real	real	real $z = x + y$
real	complex	complex $(z_R + jz_I) = (y_R + x) + jy_I$
complex	real	complex $(z_R + jz_I) = (x_R + y) + jx_I$
complex	complex	complex $(z_R + jz_I) = (x_R + y_R) + j(x_I + y_I)$
integer	integer	$z$ is integer
integer	float	$z$ is floating point
float	integer	$z$ is floating point
float	float	$z$ is floating point

Table 4.3: Example of type-dependent addition

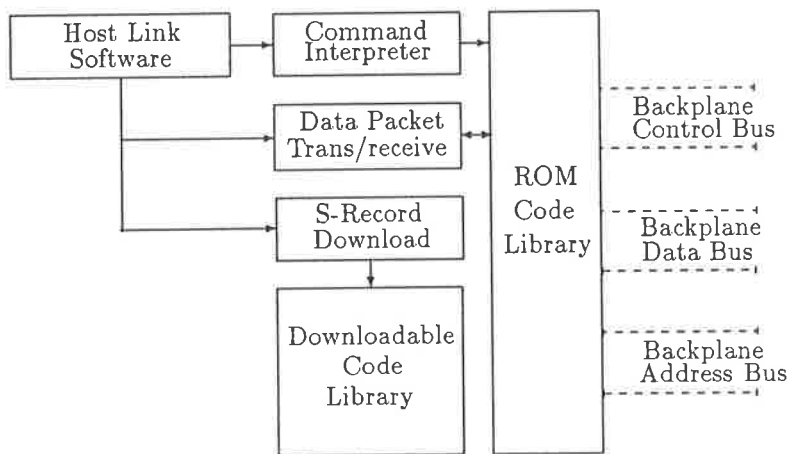


Figure 4.11: The GI Control Processor Software

The link may also be used to download new control processor software using Motorola S-record format. This facility is for development and is not used within the GI system under normal circumstances.

As shown in figure 4.11, the control processor receives command packets from the host via the RS232 link, and controls the backplane bus according to those commands. Typical commands will order the control processor to change the state of TMS320C25 control signals, and to transfer data to and from the TMS320C25 address spaces.

## 4.5 DSP Software

At the core of the Generalised Instrument lies the signal processor, At the time of writing a TMS320C25 device. The great computational rate achieved by this device (and comparable devices from other manufacturers), combined with its self-contained nature and low cost has made the GI realisable with reasonably little circuitry.

The GI implementation of the TMS320C25 provides the signal processor with full address spaces of alterable RAM. Therefore, no executable DSP code exists in the GI when the machine is powered up. All DSP code is downloaded from the host computer.

This scheme provides maximum flexibility since

- It is envisioned that many different DSP code modules will be written in the future, making a host-based library necessary to ensure that the latest version of DSP code is available.
- The user can modify existing DSP code to suit her own needs. Source code to DSP code modules will normally be available, allowing customisation.
- Code which may be too large to fit into the address space of the TMS320C25 can be split into multiple modules which can be downloaded and executed in turn.
- Efficient DSP software development requires the fastest turn-around time on the edit-compile-test sequence, which the downloading of code permits.

Despite these advantages, there are situations (such as embedded DSP) where DSP algorithms must be available without being supplied by the host computer. Though the GI does not support the inclusion of non-volatile storage in the DSP program address space, the DSP code may be stored in the control processor (68000) EPROMs, and the control processor may be programmed to transfer the DSP code to the TMS320C25 at powerup. This is an example of a simple behaviour modification which would allow the GI to be used in a different way, and may be implemented at some future time.

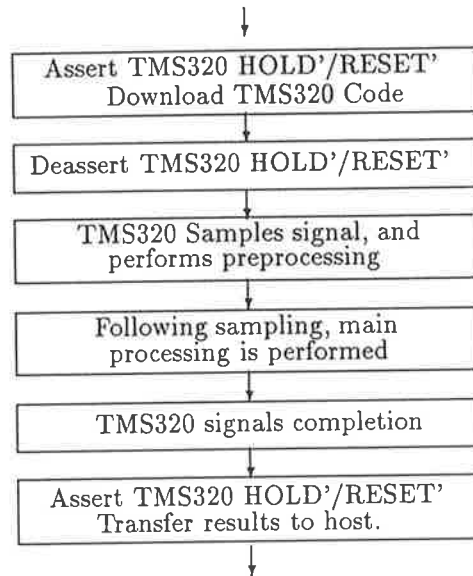


Figure 4.12: DSP code execution sequence

There are several broad categories of DSP code which have been written for the GI:

#### 1. Sample Acquisition.

With the prototype analogue interface module, all signal sampling is performed under software control by the TMS320C25 itself, at a rate determined by the DSP's onboard timer.

Sampling code must read sample values from A/D converters, and store them as arrays of values in TMS320C25 data memory.

#### 2. Preprocessing

This includes windowing functions, and the subtraction of DC levels from signals.

#### 3. Processing

Examples of processing implemented so far include Fast-Fourier Transform, correlation, and transfer function analysis [4].

#### 4. Postprocessing

Postprocessing is performed to reduce the workload of the host.

An example is the FFT algorithm, which produces its results as rectangular complex numbers (though this is a result of the implementation). To display a meaningful frequency spectrum to the user, there are two choices available:

One is to upload the real and imaginary data to the host, where the magnitude and phase are calculated. For an  $N$ -point FFT, this requires the transfer of  $2(N/2) = N$  words to the host (since half of the spectrum is redundant for a real signal.)

The second alternative is to cause the signal processor, (which is a faster mathematical processor than the host, after all) to calculate the magnitude. This would require only  $N/2$  words to be transferred to the host, thus saving time in both computation and transfer.

Clearly the second choice is preferable. This is an example of postprocessing the data within the GI to reduce workload on the host, thus improving the response time of the instrument as a whole.)



Following execution of the DSP code, the system must alert the host computer that the results of the computation are available. In the present implementation this signalling is done through the interprocessor port (see 3.3.2). The host then initiates a command to upload that data into an array variable.

## 4.6 Application Level Software

From the beginning, the aim of the Generalised Instrument project has been to produce a *system* which could emulate other test instruments, or serve as a testbed to create completely new instruments.

The preceding discussion has addressed the hardware and software issues of building such a system, but it has not brought the ideas together. This section examines how the system can be customised and controlled by the user's program script to achieve the desired function.

In order for the GI to function successfully as a test instrument, several things must be accomplished.

1. The hardware of the GI must be made compatible with the signals to be examined. We will assume that the GI is limited to signals which are compatible with its input range.
2. Appropriate DSP code must be written to perform the necessary sampling and processing of the signal data.
3. A SPaM script must be written which does the following:
  - Sets up the graphics screen with all necessary waveform display windows, buttons, and numerics to provide control of the virtual instrument.
  - Sends the necessary DSP code to the GI, and causes its execution.
  - Downloads any necessary parameters to the GI (eg sampling rate, gains, etc).
  - Causes the DSP to execute the code, and waits for the DSP to signal completion of execution.
  - Uploads the processed data from the GI, and displays it to the user.

We will now examine the SPaM script in figure 4.14 in more detail.

- *Lines 1-6*

This is a comment field. The '%' character marks the beginning of a comment. All characters between the '%' and the end of the text line are ignored.

The comments at the beginning of a file are of special significance. If the user were to type:

```
help spectrum
```

then SPaM would look for a disk file called SPECTRUM.M. If such a file exists, then the comment field at its beginning is printed. This provides a means of putting accessible documentation into script files.

- *Line 7*

This line sets the PC screen to display graphics.

- *Lines 9-15*

The variables which are used in the script are initialised here. Initialising them ensures that there will be some default waveforms displayed when the script is run.

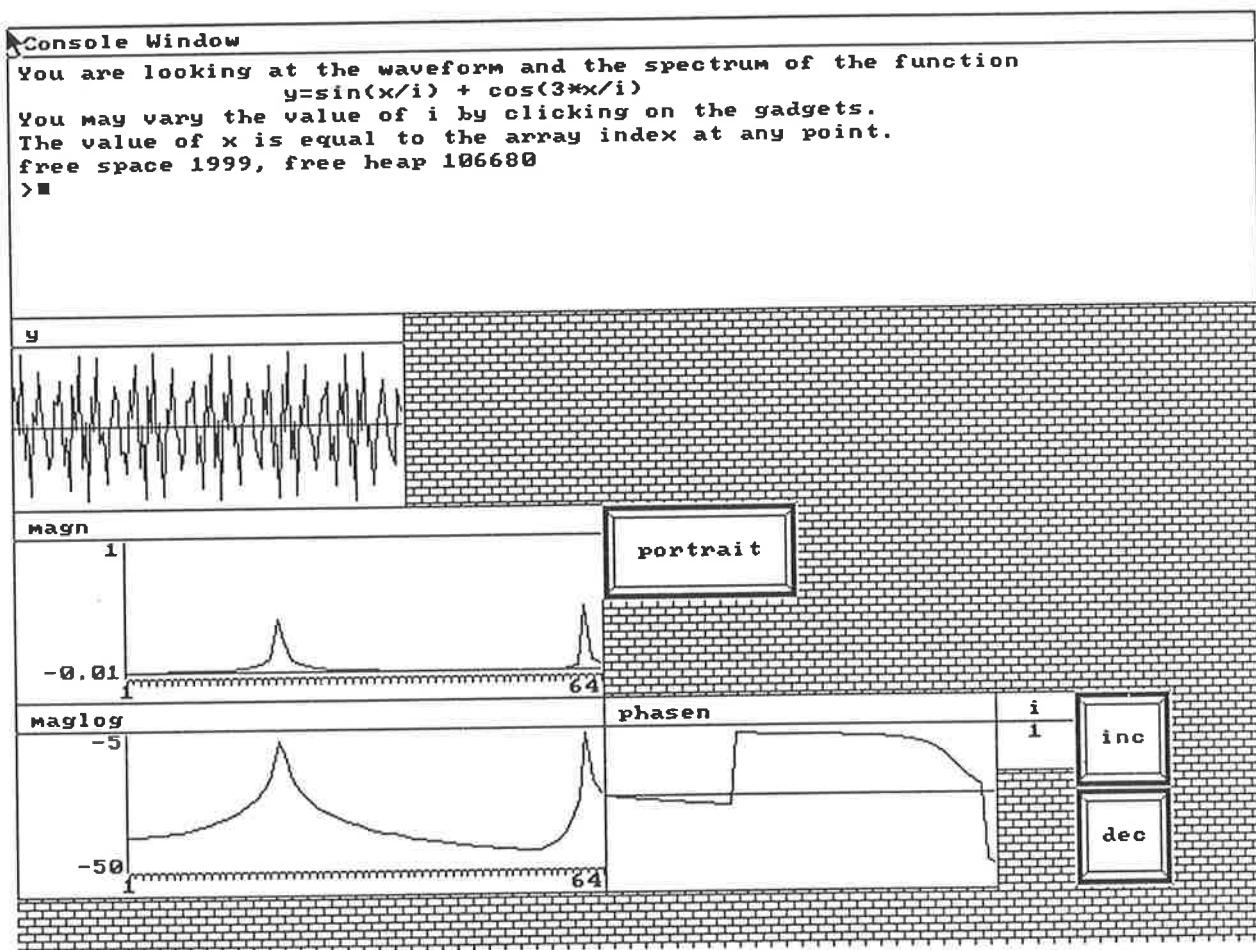


Figure 4.13: Application generated graphics screen

```

1 %      This script demonstrates thbe use of CRT and Graph windows,
2 %      as well as Buttons for control, and Numerics for the display
3 %      of scalar quantities.
4 %
5 %      George Vokalek, 1990.
6 %
7 graphic                                % set graphic display mode
8 %
9 x=0:127;                                % set some initial variable values
10 i=1.0;                                  %
11 y=sin(x/i) + cos(3*x/i);                % generate a signal waveform
12 z=fft(y);                                % calculate the FFT of the signal
13 magn = z[:,1:64];                        % extract the magnitude of the FFT.
14 maglog = 20*log(magn)/log(10);           % calculate the LOG() of magnitude
15 phasen = z[:,65:128];                    % extract the phase of the FFT
16 %
17 auto crt y 200 100                       % make a display window for y
18 auto graph magn 300 100                  % one for magn
19 set label "magn" "Freq" "Amplitude"
20 set xaxis "magn" 0 100                   % set the axis numbering
21 auto graph maglog 300 100                % display maglog
22 set label "maglog" "Freq" "DB"
23 set xaxis "maglog" 0 100                 % set axis numbering
24 auto crt phasen 200 100                  % make window for phasen
25 auto numeric i 40 40                     % display the value of i
26 %
27 auto button "inc" 50 50                  % make a screen BUTTON called INC
28 auto button "dec" 50 50                  % another called DEC
29 auto button "portrait" 100 50            % and another called PORTRAIT
30 %
31 handler portrait                          % do this code when PORTRAIT
32     poster("magn")                        % button is depressed.
33     end
34 %
35 handler inc                               % do this code when INC button
36     i=i+1;                                % is depressed.
37     y=sin(x/i) + cos(3*x/i);              % Force recalculation of all
38     z=fft(y);                              % data in accordance with the
39     magn = z[:,1:64];                      % new value of i.
40     phasen = z[:,65:128];
41     maglog = 20*log(magn)/log(10);
42     update
43     end
44 %
45 handler dec                               % do this code when DEC button
46     i=i-1;                                % is depressed.
47     if(i==0.0)                             % Do not let i be decremented to
48         i=1.0;                             % a value less than 1.
49         print "cant decrement i below 1.0"
50     end
51     y=sin(x/i) + cos(3*x/i);              % Force recalculation of all data
52     z=fft(y);                              % using the new value of i.
53     magn = z[:,1:64];
54     phasen = z[:,65:128];
55     maglog = 20*log(magn)/log(10);
56     update
57     end
58 %
59 print "You are looking at the waveform and the spectrum of the function"
60 print "      y=sin(x/i) + cos(3*x/i)"
61 print "You may vary the value of i by clicking on the gadgets."
62 print "The value of x is equal to the array index at any point."

```

Figure 4.14: Application script example. The line numbers are not part of the script: they are present only for ease of reading in this listing.

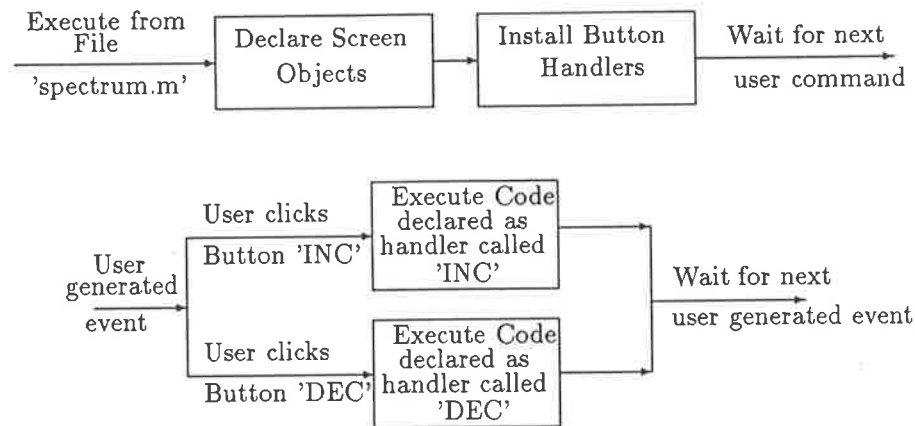


Figure 4.15: Application script flow chart

- *Lines 17-29*

These lines contain the declarations for the graphic display objects which are to be present on the screen. The objects which are used in this script include:

1. **CRT Window**

*CRT* Windows display waveforms without axis labelling.

2. **Graph Window**

*Graph* Windows display waveforms with axis labelling.

3. **Buttons**

*Buttons* are similar to real pushbutton switches. When the user clicks on one with the mouse, a corresponding *handler* (defined in the user's script) is executed.

4. **Numerics**

A *numeric* is a window which displays a scalar numeric value, unlike *graph* and *CRT* windows which display matrix values.

For further information about these objects, refer to section 4.3.5.

- *Lines 31-33*

A *handler* is a subroutine. When the user clicks a *button* on the graphics screen, SPaM checks whether a *handler* of the same name has been defined by the user. If so, that handler is executed immediately as a subroutine.

These three lines define a *handler* called *portrait*. When the *button* called *portrait* is clicked by the user, this handler is executed. Its effect is to expand the *graph* window called *magn* to full screen.

- *Lines 35-43*

The *handler* called *inc* increments the value of variable *i*, a simple integer value, and recalculates the main waveform arrays. The waveform used in this example is generated according to the formula:

$$y = \sin \frac{x}{i} + \cos \frac{3x}{i}$$

so that increasing the value of *i* increases the wavelength of the observed waveform.

- *Lines 45-57*

This *handler*, called `dec`, is similar to `inc`, described above, except that it decrements the value of variable *i*. Lines 43-46 contain a conditional statement to ensure that the value of *i* does not fall below 1.

- *Lines 59-62*

The `print` statement prints the following text in the console window, as can be seen in figure 4.13.

Examples of controlling the GI hardware from a script file can be found in section 6.3.

A simplified flowchart of the script can be seen in figure 4.15. The script sets up the display screen with the programmed objects, installs the handlers for the screen buttons, and then waits for the user's next command. The user may continue to type commands in the same way as before the script was executed. If the user clicks a screen *button*, the handler for that button will be executed, and control will be returned to the user. Since the script contains no loops, `SPaM` will always return to wait for the next user command after any part of the script is executed.

## Chapter 5

# SPaM Reference Information

SPaM is the latest in a set of signal processing packages which have evolved over recent years in the Department of Electrical & Electronic Engineering at the University of Adelaide.

Some of these packages were stand-alone software systems, for experimenting with signal processing, while others were designed also to control programmable signal processing hardware.

The SPaM software system has followed in the footsteps of a package called Sigproc[5], an adaptation<sup>1</sup> of which was used to control the Generalised Instrument. The Generalised Instrument is a programmable signal processing 'box' which together with SPaM, emulates test instruments.

SPaM is a self-contained package for generating, processing, and displaying data. It has been created for use in signal processing work, but its use is in no way restricted to that field alone. SPaM features a built in algebraic language, which should be familiar to users of the popular Matlab package. The programming language, discussed in section 5.2, allows the user to implement algorithms not hard-coded into SPaM.

As well as providing a responsive command line environment, SPaM provides a full-screen graphical user interface, with which the user can interact using keyboard and mouse. This screen may be used to display and manipulate waveforms in a more intuitive manner, and is ideally suited for use with programmable hardware such as the Generalised Instrument. The graphical environment is discussed in section 5.3.

### 5.1 SPaM : Philosophy of Design

In this section some of the important design decisions behind SPaM will be discussed. Great changes have occurred in the style of the DSP software which the author has experienced, almost all of which has been at the top level, the so called 'user interface.'

The algorithms of DSP itself remain similar or identical across implementations, but the way the user interacts with those algorithms is of profound significance in determining whether a package will be pleasant to use, or a burden. The ideal package would be so intuitive that the user can concentrate on signal processing and not the vagaries of a computer program.

---

<sup>1</sup> Sigproc was originated by Prof.R.E.Bogner at the University of Adelaide as a set of Fortran signal processing programs. The same name was given to a single DSP program written by Richard Lane in the C language. The latter was the basis for prototype software in the GI project, before SPaM was written.

### 5.1.1 The User Interface

The 'user interface' is that part of the software with which the user must interact directly. It may simply be a command line, or a graphics screen with icons symbolising functions.

While the next section espouses the virtues of the graphical, intuitive, user interface, it is worth remembering that SPaM actually has both a graphical and command-line user interface. The former is most suitable for novice users and fixed applications, but the latter is essential for wizards who wish to access fully the signal processing algorithms lurking within the machine.

#### What came before, and what was wrong with it.

Earlier signal processing packages with which the author has had experience had all consisted of a command interpreter style interface. Such programs accept a textual command, and respond to it immediately. The command may instruct the program to generate a sinusoid, transform a signal, or any one of many possible activities. Command words are followed on the command line by arguments, for example the frequency of the sine wave to be generated.

After the program has finished its task, it asks the user for the next command, and so on. Such systems are *imperative*, and break signal processing tasks down into a sequence of 'orders'.

In many cases, especially for teaching purposes, such an interface is quite reasonable, since the number of commands required is often small, errors are reported and acted on immediately, and the individual commands can question the user further if they require more information than was given on the command line.

The main disadvantage of command line interfaces is that the user is required to read documentation, and create a logical sequence of actions to get the desired result. Instead of saying 'I want to see the spectrum of the signal on channel 2', the user has to think 'acquire from channel 2' - 'spectrum analyse result' - 'display spectrum' etc.

Another disadvantage of imperative interfaces is that they have no *structure*. A sequence of commands in an imperative interface is executed starting at the top, and ending at the bottom (assuming no errors occur.) There is no way to modify the flow of execution of the 'program'.

Such 'imperative' interfaces have been dominant in the computer industry until recently. Witness the MS-DOS user interface prevalent on most PCs, and the interfaces of their predecessors. Only since the mid 1980's have user interfaces steered toward a more symbolic and intuitive approach.

Instrumental in popularising such interfaces was the Apple MacIntosh. Since then, similar interfaces have appeared on personal computers and workstations. As the display and processing capabilities of computers improve, such interfaces will become dominant.

SPaM adopts the 'WIMP'<sup>2</sup> environment of these modern GUIs (Graphical User Interfaces), though it copies none in detail. One of the pleasant features of the GUIs available today is that though they may be numerous, and each having specific features, they all obey enough common rules of behaviour to enable a novice user to learn the interface quickly (usually by experimenting.)

Old-style signal processing packages like Sigproc were akin to MS-DOS. Unless the user reads a manual, she simply does not know what commands are available. A WIMP interface alleviates this by allowing the author of software to place icons (gadgets, menus, etc) on the screen which represent a core of possible activities, allowing the user to start immediately.

If the programmer is sufficiently careful, all the user's required activities may be directly represented

---

<sup>2</sup> Windows, Icons, Mouse, Pull-down menus - a modern paradigm of user interface design.

by objects on the screen. In this way the computer becomes a machine with a finite repertoire of actions. It may be capable of other actions, but they are irrelevant to the user, and may be hidden from her sight.

Potential user's are often discouraged when presented with a manual for a system (be it software or hardware), where there is far more detail than is required to solve the problem at hand (the author certainly is.) A WIMP style interface largely removes the mental anguish associated with the realisation that there is much studying to be done before even the simplest problems can be solved.

### The SPaM User Interface

As mentioned in the previous section, the SPaM user interface consists of a command-line interface and a GUI.

The command-line interface is loosely modelled on Matlab, a commercial numerical mathematics package. Matlab is widely used, since it is very convenient for implementing numerical algorithms without detailed programming. Many DSP algorithms exist in the Matlab language (or may be quickly implemented,) and many of its features have been incorporated into SPaM to allow the execution of those programs (with minor changes, if any.)

SPaM's Graphical User Interface (GUI) is loosely consistent with commercial GUIs. It is designed to be used in conjunction with a mouse (though all mouse actions can be emulated from the keyboard.)

### The Command Line Interface

The Command Line Interface (CLI) simply presents a prompt to the user, and waits for the user to type in a line of text. The line of text is analysed (called 'parsing'), and if it is a syntactically correct (ie the right types of words appear in the correct places) then the appropriate actions are carried out.

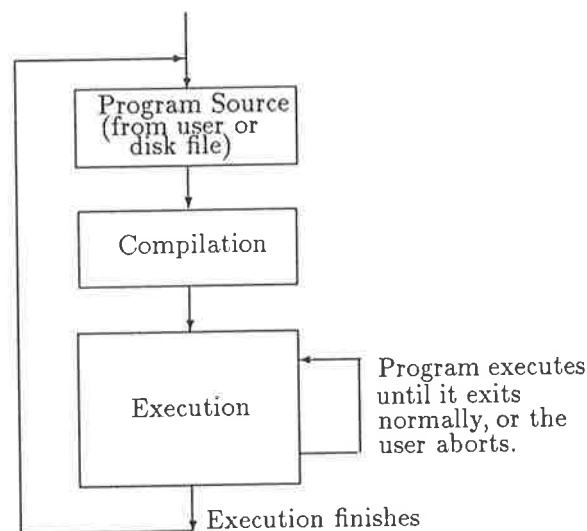


Figure 5.1: SPaM execution flow

It is worth emphasising at this stage that there are two phases in the response of SPaM to commands. The first phase is the compilation phase, where the user's input is analysed for correctness according



to rules of grammar defined for SPaM, while the second is the execution phase, where the actions specified by the user are actually executed.

In this regard, SPaM differs from command interpreters such as Sigproc, which begin execution of commands before all of the user's input had been analysed. This allowed the program code for individual commands to carry out analysis of their own arguments. In some cases this was an advantage, since optional qualifiers could be defined, and missing arguments prompted for. If SPaM finds an error in its input, it signals the user to correct the error and aborts the compilation process. The user must correct the error before resubmitting her program source to SPaM.

The disadvantage of an interpreter such as Sigproc is the difficulty (for the programmer) in efficiently processing complex elements such as numeric expressions and loops. A large amount of text scanning is required to interpret such structures, and this processing must be repeated each time the structure is encountered as the program runs. SPaM performs its interpreting (generally called parsing) only once. During the parsing step, it constructs a more concise and efficient representation of the algorithm to be performed. This new representation is then 'run' to generate the desired results.

The commands can be either accepted from the keyboard, or they can be stored in files. The commands in a file may be executed by simply typing the name of the file. The file must have an extension of '.m', so that to execute the commands in file 'foo.m', you would type 'foo'. The file 'foo.m' is simply a text file of commands, arranged just as if they were to be typed directly into SPaM.

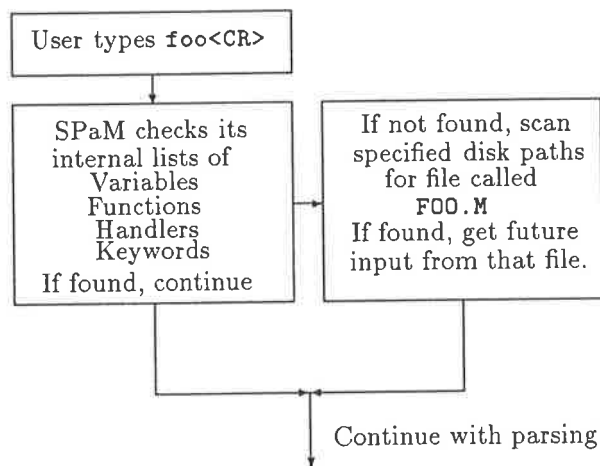


Figure 5.2: Execution of commands from scripts

Files of commands may be nested to a depth of *five*. That is, one file can cause commands from another file to be compiled as part of its own compilation.

Once a program is executing, the CLI will not respond until that program has completed execution. You will know when this happens because a new prompt will be printed to the screen.

To abort the execution of a program, the user may press CTRL-C (the CTRL and C keys simultaneously.), the ESC key, or the CTRL-BRK key.

For more detailed information about the commands which can be used from the CLI, see section 5.6.

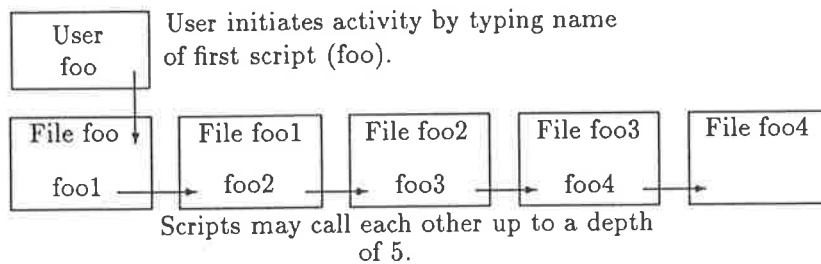


Figure 5.3: Nesting of script files

### The Graphical User Interface

To allow simulated instruments to be created on the PC screen, and to allow the display of information in flexible manner, SPaM has a versatile Graphical User Interface (GUI).

The GUI allows the user to manipulate objects, and cause actions, in a very intuitive way. The GUI is designed to be used with a mouse, so that many actions simply become a matter of 'point-and-shoot'.

The reduction in workload for a casual user will be enormous, since applications may be prepared which have all options easily visible on screen, and require little or no documentation to be read.

In its elemental state, the GUI provides the following functions:

- A mouse pointer or cursor.

The pointer may be moved by moving the mouse across the desktop. It is important to remember that the mouse behaves differently when positioned over different classes of object. These objects are summarised in table 5.1.

- Pop-up menus.

If the mouse is positioned over the background (brickwork pattern), and the right-mouse-button (RMB) is depressed and held, then a pop-up menu will appear.

This particular menu is the top-level menu. It allows you to create new objects, and also to select actions which affect the *whole screen*.

Once you have created some objects, different menus will appear depending over which object you have the pointer positioned, and those menus will affect only the object over which the mouse pointer is positioned.

Various classes of objects may then be added to display screen to perform specific operations, as listed in table 5.1. Once created, screen objects may be manipulated by the user in various ways, as described in section 5.3.5.

## 5.2 SPaM Programming Language

The SPaM programming language is a simple language, designed mainly for the rapid writing of mathematical expressions and procedures. It is based loosely on Matlab, but also closely resembles BASIC.

<i>buttons</i>	<p><i>Buttons</i> are displayed on the screen as a box which has the appearance of an electrical pushbutton switch. When the user positions the mouse pointer over the <i>button</i> and clicks the left-mouse-button, SPaM will search its internal lists for a user-defined handler (see section 5.2.6 and execute the code defined in that handler.</p> <p>In this way, simple click actions of the mouse can cause preprogrammed events to occur within SPaM and the GI.</p>
<i>numerics</i>	<p>A <i>numeric</i> is a window on the screen whose purpose is to display the value of a scalar variable.</p>
<i>CRT</i>	<p>A <i>CRT</i> is a window whose purpose is to display the value of a vector or matrix variable as a two dimensional graph.</p> <p>A <i>CRT</i> is analogous to a <i>graph</i> window, except that a <i>graph</i> window displays axis titles and numbering, whereas a <i>CRT</i> window does not.</p> <p>The xaxis of a <i>CRT</i> window represents the index value into the array, while the vertical axis represents the value of the array element at that particular index value.</p> <p>If the variable being displayed in a <i>CRT</i> or <i>graph</i> window is a matrix, it is displayed as a collection of vectors of data, with each vector being a separate trace in the window. The longer dimension of the matrix is assumed to define the length of the vectors, with the shorter dimension defining the number of vectors.</p>
<i>graph</i>	<p>A <i>graph</i> window, like a <i>CRT</i> window, can be opened to display the value of a vector or matrix as a two dimensional graph. A <i>graph</i> has other properties, such as axis labels, and axis numbering.</p>
<i>argand</i>	<p>Whereas the <i>CRT</i> and <i>graph</i> windows display the values of real vectors and matrices, the <i>argand</i> diagram is designed to display the values of complex vectors.</p> <p>The <i>argand</i> window occupies a region of the complex plane (hence it is called an Argand diagram). Each complex number in the vector being displayed is plotted as a point on the complex plane. Points from consecutive array elements are connected by lines.</p>

Table 5.1: SPaM graphical screen objects

SPaM's essential features include:

- No variable declarations.

Normal<sup>3</sup> variables need not be declared. They are simply used as required. Variables can be of one of several different types, as detailed in the following pages.

The type of a variable is checked each time an operation is to be performed on it, and if a method for handling that type exists, then the operation proceeds normally.

<sup>3</sup>some special variables, such as VLMs, do need to be declared

See section 5.2.3.

- Multiple numeric types.

Three base numeric types are supported: 16-bit integers, 32-bit integers, and double-precision floating point numbers. The integer types are present to allow easy interfacing of SPaM to integer signal processors, without the need for explicit conversions to take place.

See section 5.2.3

- Real and Complex numbers.

Complex numbers are fully supported by SPaM (except in most transcendental functions). By default, the variable 'j' has the value of  $\sqrt{-1}$ , so that a complex number  $3 + j4$  is formed by typing `3 + j*4`.

See section 5.2.3.

- Scalar and Matrix quantities.

SPaM allows scalar and matrix quantities to be easily defined and used. Vector quantities are simply a trivial case of a matrix (having one row and many columns, or vice versa.)

Most mathematical operations operate directly on matrix data as easily as scalar data.

Assignment of individual elements of a matrix are allowed, as are assignments of regions within a matrix. Portions of a matrix, as defined by ranges of rows and columns, may be extracted.

Matrices can be as large as memory will allow, and larger matrices still (called VLMs) may reside on disk.

See sections 5.2.3, 5.5

- Execution flow control.

Statements may simply be executed one after the other, or loops constructed. SPaM supports the following loop constructs: `GOTO`, `WHILE...`, `FOR...`, `IF...THEN...ELSE...`

See section 5.2.4.

- Conditional tests.

Operators such as `<`, `>`, `<=`, `>=`, `==`, `!=` are available for testing of numeric quantities. These tests can be used in conjunction with the `WHILE...` and `IF...` statements to control the execution of the user's program.

Refer to section 5.2.4

- User defined functions.

The user may define her own functions. The functions may have any number of arguments, and return any number of results.

The variables used within a user-defined function are all *local*. That is, they are not visible to other parts of the program, and assignments to these internal variables will not affect the values of similarly named variables in other parts of the program.

The only way to get values into a user-defined function is to pass them as arguments. The only way to get values out of a function is to return them as results.

See section 5.2.5.

- User defined handlers.

A 'handler' is akin to a user-defined function except that it takes *no* arguments (ever), and returns *no* results. Handlers are designed to allow the user to implement often used code only once, and from then on to call that code with a single word (the handler's name.)

Also unlike a user-defined function, a handler has *no* local variables. It has full access to all of the variables of the block in which it was compiled.

If the declaration of the handler was part of the main body of the program, then the variables which the handler access will be the (globally accessible) variables of the main program.

If, however, the handler was declared within a user-defined function, the variables which it is able to access will be those local variables of the user-defined function.

See section 5.2.6.

- Built-in functions.

SPaM includes a set of built-in functions for numeric computation, graphic display control, and communication with the Generalised Instrument hardware

See sections 5.6.4, 5.6.3, 5.6.2.

- Loading and Saving Variables.

Variables may be loaded from, and saved to, disk. They are stored in a readable ASCII format, which allows the user to prepare files of data outside SPaM, and allows other programs to interface to SPaM via disk files.

See section 5.4

- Externally defined text editor.

SPaM allows you to call an external text editor from within it, so that you can edit program scripts without leaving SPaM (thus preserving your environment and variables.)

The name of the editor is fixed by an environment variable, and any small text editor can be used. The author recommends the shareware editor QEDIT.

See section 5.2.2.

- Shelling to DOS.

SPaM allows the user to drop into DOS to carry out any operations which are compatible with the reduced memory available in such a situation. On completing her operations, the user can return to SPaM by typing 'exit'. The SPaM environment and variables will be restored to its previous state.

See section 5.2.7

- Communication to Generalised Instrument hardware.

SPaM includes multiple functions to allow the transfer of data to and from the GI box, and other functions control the state of the signal processor in the GI.

A terminal mode is also implemented to allow direct communication to the GIs onboard monitor, for debugging of signal processing code.

## 5.2.1 General Rules

The following are general rules which must be observed when using SPaM:

- SPaM is case sensitive. All keywords must be in lower case.
- Variable names must begin with an alphabetic characters (a-z, A-Z), but may also contain numeric digits in the remaining body of the variable name.
- Variable names may not be the same as any of SPaM's reserved .words, which are listed in section 5.6.1.
- User defined function and handler names may not be the same as any of SPaM's reserved words. The set of reserved words are listed in section 5.6.1.

- Expressions separated by a space may be concatenated. For instance, the character sequence "1 -1" (where a space exists between the minus sign and the preceding digit) will be interpreted as the expression  $1 - 1$  which evaluates to zero. To ensure correct separation, use a comma. This is especially important in matrix and vector assignments.
- Variable assignments such as ' $x = 3$ ' will cause the value of  $x$  to be echoed to the screen unless the assignment is followed by a semicolon, eg ' $x = 3;$ ', in which case nothing is printed to the screen.

### 5.2.2 SPaM Entering Program Text : Script Files

The user interacts with SPaM by typing in program statements.. SPaM compiles one statement at a time, and executes it. A statement can consist of an expression, in which case SPaM will return a numeric result once the code which represents the expression is executed, or a command which causes SPaM to change some aspect of the system.

It would be tedious in practice if the user had to type in commands from the keyboard, and thus SPaM has the ability to execute commands from disk based files, called *script files*. A script file is simply a text file written by the user using a text editor or word processor, which contains program text in the same form as the user would type to the keyboard. Unlike keyboard input, SPaM compiles the entire text of a script file before it begins executing the code.

The user can create a script from within SPaM by calling an external editor. SPaM allows this by simply typing

```
edit
```

at which point SPaM shells to the user defined external text editor. The default editor is the shareware QEDIT.EXE program, but the default can be changed by modifying the SPAMEDIT environment variable (see the local installation guide.)

After invoking the editor, the user creates the script file, ensuring that it has a '.m' extension. The file can be saved to any of the directories which SPaM searches for scripts. SPaM first searches the current directory, then those directories specified in the SPAMPATH environment variable (see the local installation guide.)

To invoke the script file, the user must simply type its name. For instance, if the file which was created was called 'foo.m', then it can be invoked by typing

```
foo
```

SPaM will then compile the all of program statements from that file, execute them, and then return control to the keyboard (unless the script contained the **chain** command, see section 5.6.6.)

### 5.2.3 SPaM Variables and Numbers

A variable is a box in which a value is stored. It is like the memory in a pocket calculator. SPaM understands two fundamental types of variables : numbers and strings.

String variables hold a sequence of characters, usually just text, which the user may want to print at certain times, use as a label for a graph, and so on.

Numeric variables are not so simple. They may have several attributes associated with them, as detailed in the following pages.

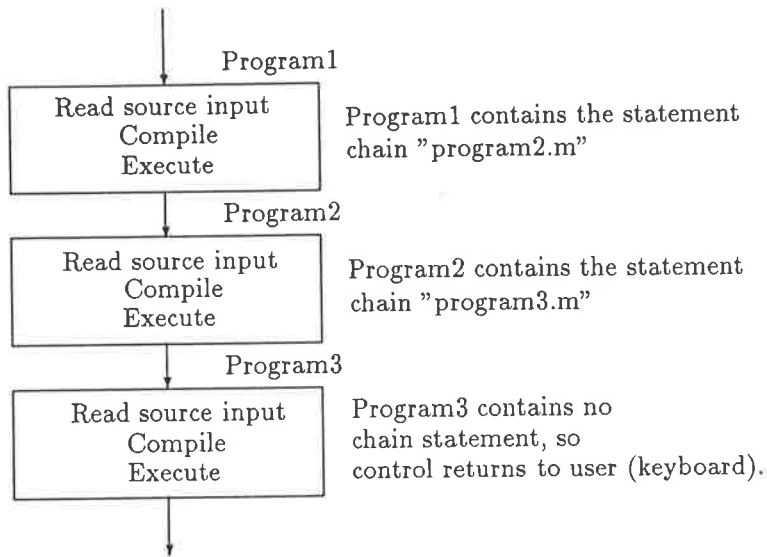


Figure 5.4: Chaining multiple input scripts

Variables are referred to by *name*. Usually, if SPaM encounters a word which is not in its vocabulary, it assumes that the word is the name of a variable, and inserts that name into its list of known variables.

Examples:

```

speed = 200.0
name = "Lamborghini"
  
```

The first assignment would create a new variable called `speed` which would take the floating point value 100.0, and the second assignment would create a string variable called 'name' whose value would be the string 'Lamborghini.' The name of a variable must begin with an alphabetic character (a-z, A-Z), but may include numbers and the underscore '\_'. *A variable may not be given a name which is the same as a SPaM keyword (see section 5.6.1 for a list of keywords). Note this fact, since SPaM will detect this as a syntax error, and the cause may not be obvious.*

If a variable already exists when an assignment takes place, the contents of that variable are first deleted. If any extra storage was associated with the variable (for instance a matrix), it is reclaimed by SPaM for future use.

Note that in the above cases SPaM echoes the results of the assignment. In response to the first case, SPaM would have printed the following:

```

speed =
200.0
  
```

To suppress the printing of the result of an assignment statement, the statement should be followed by a semicolon ';'. For example:

```

speed = 200.0;
  
```

would not echo the result.

To see a list of presently defined variables, type `who` followed by `<CR>`.

## Numeric Representation

A number in SPaM can be represented in one of the following forms.

### 1. 16-bit integer.

Numbers of this type faithfully represent any integer in the range  $(-32768 \dots 32767)$ .

example: `counter = 100;`

A numeric literal (eg the '100' in the above example) is stored as a 16-bit integer *if and only if* it is in the above range *and* it does not contain a decimal point.

### 2. 32-bit integer.

Numbers of this type faithfully represent any integer in the range  $(-2^{31} \dots 2^{31} - 1)$ .

example: `counter = 100000;`

A numeric literal (eg the '100000' above) is stored as a 32-bit integer *if and only if* it lies within the above range, but outside the range for a 16-bit integer *and* does not contain a decimal point.

### 3. Double precision floating point.

Any number which contains a decimal point is converted into a floating point number and stored in this format.

examples:

```
a = 1.23;
```

```
b = 1.0e10;
```

## Real and Complex

Numbers can be either real or complex. For scalar values, a real value is essentially the same as a complex value with a vanishingly small imaginary part. For matrix values, however, a real value only requires half of the storage space of a complex value of the same numeric representation.

Generally, numbers are interpreted as being real by SPaM. Expressions may return a complex result if they involve a complex variable, or the square root of a negative number. For instance, '`x=3+j*4`' is an expression containing only real literals, but a complex variable (`j`) which causes the expression to evaluate to a complex result.

By default, SPaM assigns the variable '`j`' with the value of  $\sqrt{-1}$  on startup. There is nothing, however, to prevent the user from assigning a different value to `j`, or assigning the value of  $\sqrt{-1}$  to a different variable.

Examples of assigning a complex number:

```
x=3+j*4;
```

```
y = sqrt(-1);
```

## Scalar and Matrix

Matrices can be created in a number of ways.



1. Returned by functions which create matrices.

For instance, the function `eye(n)` returns an identity matrix of size  $n \times n$ .

2. By using a range constructor.

A range is a vector of numbers built as an arithmetic sequence from user specified start and end values, and optionally a step value. For example,

```
x=0:10; y=0:0.1:10;
```

The first case creates a  $1 \times 11$  matrix. The values of the elements of that matrix are [0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0].

The second case creates a matrix of size  $1 \times 101$ . The first element contains the value 0.0, the second 0.1, the third 0.2, and so on, until the 101th element which contains the value 10.0.

Thus if only the start and end points of the range are specified, the default increment of 1.0 is used. A user specified increment may be used, in accordance with the syntax:

```
range = start_value : increment : end_value
```

The elements of a matrix generated using a range constructor are always real floating-point types.

3. By explicit matrix specification

An entire matrix may be explicitly specified by using square brackets '[' ]' to enclose the contents. Within the square brackets, elements separated by a space or comma will be assumed to exist on the same row. When a semicolon ';' is encountered, the current row is terminated, and the next one begun.

The size of the matrix is defined by the number of rows encountered, and the longest row encountered (which determines the number of columns).

Example:

```
x = [1; 2 3; 4 5 6; 7 8 9 10];
```

will create the matrix

$$x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{bmatrix}$$

*NOTE: To avoid confusion, the comma ',' rather than the space ' ' should be used as the element separator, for the following reason. Consider the matrix specification -*

```
x = [1 -2 -3 4];
```

in response to which, SPaM will create the following matrix:

```
x = [-4 4]
```

because SPaM reads '1 -2 -3' as  $1 - 2 - 3$  which equals  $-4$ . The syntax which should be used for the declaration is-

```
x = [1,-2,-3,4];
```

4. By referring to a specific matrix element.

The assignment '`x(4,5)=1.2;`' will create a matrix `x` which has 4 rows and 5 columns. The element at (4,5) is assigned the value 1.2.

Now, there are some points to remember about such assignments.

- (a) If the variable  $x$  did not exist, it is created from scratch. The numeric representation of the elements of matrix  $x$  will be set entirely by the right hand side of the equality. In this case, the value is 1.2, which is a real floating point value. Consequently, matrix  $x$  is created as a matrix of real floating point values.
- (b) If the variable  $x$  did exist, but was not a matrix, then the same rules apply as if  $x$  had not existed.
- (c) If the variable  $x$  did exist, and it was a matrix, then the right hand side of the equality is converted to the same type as the elemental type of the existing matrix  $x$ .

**WARNING:** If the matrix  $x$  is a matrix of integers, then the value (1.2) in the above example would be truncated (to 1) before insertion into the matrix.

Now, it is possible that one or both of the indices of assignment (4 and 5 in the above example) are outside of the range allowed by the current size of matrix  $x$ . In this case, matrix  $x$  will be enlarged enough so that the assignment can take place.

Since the enlargement of matrix  $x$  may make available more new elements than the assignment will affect, all of the new elements are set to zero first.

The values inside a matrix can be extracted as single elements, or as submatrices of the parent matrix. For instance,

```
y=x(3,4);      y = x34
z=x(2:3,1:3); z = [ x21 x22 x23
                   x31 x32 x33 ]
```

The first case simply extracts the element in row 3, column 4 of matrix  $x$  and assigns its value to the variable  $y$ . The second case extracts a submatrix from matrix  $x$ , consisting of rows 2 and 3, columns 1, 2, and 3, and assigns that submatrix to the variable  $z$ .

A special type of range, called a *maximum extent range* allows either all rows or columns to be specified. For instance,

```
a=x(1,:);
b=x(:,1);
c=x(:,:);
```

Consider the following matrix as an example.

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

After execution of the preceding commands, the arrays  $a$ ,  $b$ , and  $c$  would have the following values.

$$a = [ 1 \ 2 \ 3 \ 4 ], b = \begin{bmatrix} 1 \\ 5 \\ 9 \\ 13 \end{bmatrix}, c = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

The first case creates a row vector 'a' to which it assigns the value of the first row of matrix  $x$ . The lone ':' means 'all columns' in this case.

The second case creates a column vector 'b' to which it assigns the value of the first column of matrix  $x$ . In this case the ':' means 'all rows'.

In the third case, the two ':' symbols mean 'all rows and columns', so that the resulting matrix  $c$  is identical to matrix  $x$ .

So far, we have discussed only methods of extracting numbers from a matrix. What about methods of putting numbers *into* a matrix? An example above showed how to set individual matrix elements, but **SPaM** allows the user to do more than that. **SPaM** will allow matrices to be copied *into* matrices. Consider the following example.

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$y = \begin{bmatrix} 100 & 101 & 102 \\ 103 & 104 & 105 \\ 106 & 107 & 108 \end{bmatrix}$$

`x(3,3)=y;`

The result will be-

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 0 \\ 5 & 6 & 7 & 8 & 0 \\ 9 & 10 & 100 & 101 & 102 \\ 13 & 14 & 103 & 104 & 105 \\ 0 & 0 & 106 & 107 & 108 \end{bmatrix}$$

The matrix  $y$  has been pasted into the matrix  $x$  starting at row 3, column 3. Because matrix  $y$  was larger than the remaining space available within matrix  $x$  (to the right of element (3,3), and down from element (3,3)), the matrix  $x$  had to be extended by 1 row and 1 column.

A **VLM** is a special case of a matrix. Normal matrices are stored in main memory, and processed without difficulty. A **VLM** is a matrix whose data is not entirely stored in main memory, but is kept on disk (usually hard disk).

Since the memory resources of a PC are finite, the **VLM** method allows matrices to be built which by far exceed the memory available. Some mathematical operators can operate on **VLMs** directly, but most will not<sup>4</sup>. The way around this is to copy data into and out of the **VLM** as necessary. Most **DSP** functions operate on packets of data from a larger set, and such operations can be readily implemented using the matrix addressing methods described above. **VLMs** are discussed further in section 5.5.

### Variable Assignments and Undefined Variables

As shown in the above examples, variables are simply given values by using statements such as -

`var = expression;`

Now the assignment will only proceed if the value of **expression** can be evaluated. The value of **expression** may not always be defined. One reason for this is that it may contain references to

---

<sup>4</sup>Currently, **VLMs** are only allowed to contain 1 type of data, which is floating point real numbers. This may change in future

other variables whose values have not yet been defined. SPaM does not assume any default value (such as zero) for an undefined variable.

Attempting to access the value of an undefined variable will cause SPaM to abort the execution of a command (or program) and signal an error.

If the expression can be evaluated, then the variable `var` is checked to see whether it currently has a value. If so, any extra storage associated with that value (such as the storage associated with a matrix of elements) is returned to the system. The value of `expression` is finally assigned to `var`.

## Multiple Assignments

A mechanism for assigning multiple variables with values in the same statement exists within SPaM. It is provided mainly to cater for functions which return multiple values. One example of a built-in function that returns two values is the `size()` function, which returns the row and column dimensions of a matrix. An example of the multiple assignment of this functions results is shown below.

```
{row,column} = size(mymatrix);
```

The variables to be assigned (`row,column`) are enclosed by braces `{}`. Any number of variables can be assigned in this way. See section 5.2.5 for examples on how user declared functions may return multiple values.

## Predefined Variables

Several variables are defined automatically by SPaM. This is done simply for the convenience of the user. The variables are listed in table 5.2.

These variables are not protected in any way, so the user is free to reassign them. However, after the execution of a `clear` statement, the variables will all return to the values shown in table 5.2.

<i>Variable Name</i>	<i>Default Value</i>
pi	3.14159265359
e	2.718281828
j	$\sqrt{-1}$

Table 5.2: SPaM predefined variables

### 5.2.4 SPaM Control Statements

Rather than simply executing statements in the same order, SPaM has control statements which allow the order of execution of a program to be changed dynamically, based on decisions made by the program.

The following control statements are implemented in SPaM.

- `if expression statements... else statements... end`  
`if expression statements... end`

Following the `if` must be an **expression**. That expression is evaluated, and if the result is non-zero, then the statements between the *expression* and the `else` keyword will be executed.

If the result of the control expression is zero, then the statements between the `else` and the `end` will be executed.

The control expression may be any expression which returns a scalar value. Matrix values (ie a value which is a matrix itself) are not acceptable as the result of the control expression<sup>5</sup>

The following examples indicate some typical expressions.

```
if(a==b) print "a=b" end
if(all([1;2;3]==([2;4;6]/2))
    print "elements of second are twice that of first"
end
if(a-b>0) print "a>b" end
```

The conditional tests which SPaM allows are listed in table 5.13.

- `for v=m statements... end`

The `for` statement allows the execution of the enclosed statements a precise number of times, each time with a different value of the control variable `v`.

Each time the loop starts, `v` is assigned the value of the *next* column of matrix `m`. The first time through the loop, `v` will take the value of the first column of matrix `m`, the second time through it will have the value of the second column of matrix `m`, and so on.

If you simply want to use the loop as you would in BASIC, say, where `v` takes on scalar consecutive values, you would make `m` a 1×N matrix, as shown in the following example:

```
for i=[1:10]
    print i
end
```

The expression `'[1:10]'` generates the matrix `[1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0]`, and each of the (scalar) elements will be assigned to `i` in turn.

If the matrix in the control expression is not a vector, for example

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
for i=x
    print i
end
```

will produce the following results.

```
i =
    [ 1 ]
    [ 4 ]
i =
    [ 2 ]
```

---

<sup>5</sup>To make a test on a matrix expression, you can use the `all()` function to convert a matrix result into a scalar result which the `if` statement can digest.

```

    [ 5 ]
i =
    [ 3 ]
    [ 6 ]

```

As can be seen, the variable `i` is successively assign the value of each column from the matrix which is the the result of the control expression.

- **while expression statements... end**

The **while** loop will execute its enclosed statements only if the result of **expression** is not zero. It will keep executing the statements until the result of **expression** is zero. For example:

```

i=1;
while (i<10)
    print i
    i=i+1;
end

```

The conditional tests which **SPaM** allows are listed in table 5.13.

## 5.2.5 User Defined Functions

As well as providing a library of predefined (built-in) functions, **SPaM** allows the user to define her own. The following examples shows how to define a function.

```

function y=myfunc(x)
    y = x*x + x + 1;
end

```

The first line contains the interface information for the function. Following the **function** keyword (which begins the definition) is the form which the function will take in practice.

The text '`y=myfunc(x)`' states that the function is to be called `myfunc`, it will have one argument which will be referred to as `x` within the function body, and it will return one argument called `y` within the body of the function.

Now note that the names of the arguments (in this case `x`) and the names of the return variables (in this case `y`) are only for use within the body of the function. When the function is actually called from outside, the names of arguments and return values on the outside can be quite different. At the time the function is called, the arguments from outside are copied to the argument variables used inside the function (this prevents the outside arguments from being corrupted by processes within the function). At the end of the function, the return variables used within the function are copied to the outside return variables.

Functions need not be restricted to one argument or return value. In fact, they can return as many values as the user wishes, and take as many arguments as the user wishes. Consider the following example.

```

function half,quarter,eighth=fractions(x,y,z)
    sum = x+y+z;
    half = sum/2;
    quarter = half/2;
    eighth = quarter/2;
end

```

In the above example, the function `fractions` takes three arguments, and returns three values. To call such a function, you would use the multiple assignment syntax (see section 5.2.3) as shown below.

```
{h,q,e} = fractions(1,2,3)
```

Which would result in `h` being given a value of 3, `q` of 1.5, and `e` of 0.75. The multiple targets of the assignment must be surrounded by braces `{}`.

A point worth mentioning about the declaration of `fractions()` is that the variable `sum` which is used within the function is a *local* variable, and exists only within the function. It is not visible from outside of the function, and will not affect the value of an outside variable with the same name.

Finally, the return values `half`, `quarter`, `eighth` are treated simply as normal variables within `fractions()`. They can be assigned to multiple times (it is only the last assignment before the end of the function which determines the returned value) and used within expressions.

## 5.2.6 User Defined Handlers

Unlike functions, which take arguments, return values, and have local variables, **handlers** take *no* arguments, return *no* values, and have *no* local variables.

Handlers operate on global variables only. They should be thought of as subroutines, not as functions. An example is shown below.

```
handler increment
  i=i+1;
  j=j+1;
  k=k+1;
end
```

The user can invoke the handler simply by typing its name. When invoked from within a program script, the handler will be executed, and then the main program will resume immediately after the call to the handler. For instance,

```
i=1;
j=2;
k=3;
increment
print i
print j
print k
```

will result in the display

```
i =
  2
j =
  3
k =
  4
```

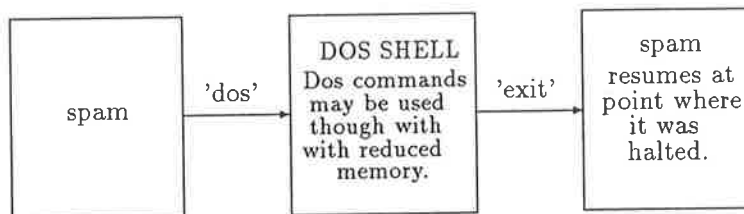
User-defined handlers serve the dual purposes of providing a shorthand way of invoking operations on global variables, and of providing a means to perform operations from the GUI (see section 5.3.5.)

### 5.2.7 Shelling to MS-DOS

The user can execute MS-DOS commands without leaving SPaM by invoking a DOS shell. By typing

`dos`

the user calls the DOS command interpreter `COMMAND.COM`, making it appear as though she has quit SPaM altogether. SPaM is, however, still resident in memory, but is 'asleep'. If the screen was in graphic display mode before the `dos` command was executed, it is first switched to text display mode.



Since SPaM is still resident in memory, there may not be sufficient program memory left to execute large programs, but most small ones should run without problem. The situation also depends on how much memory was occupied by variables (esp. arrays) before the user shelled to DOS.

After completing her DOS operations the use can return to SPaM by typing

`exit`

to the DOS command line. If SPaM was originally in graphic mode, the graphics screen will be reopened and redrawn. The text which was in the text window before the `dos` command was invoked is not refreshed, however.

There is no need to shell to DOS to run a text editor, since this can be done with SPaM's `edit` command (5.2.2.)

## 5.3 The Graphical User Interface

SPaM's Graphical User Interface (GUI) is a graphical environment which acts as a buffer between the user and SPaM's internal processes. In essence it consists of a graphical display with which the user interacts through the keyboard and the mouse.

Certain classes of objects can be created (as described briefly in section 5.1.1) and manipulated using the GUI. These operations are intuitive and do not require the user to memorise any SPaM commands. Creation of screen objects is described in section 5.3.4.

Once created, the user can interact with screen objects in various ways. *Buttons* can be clicked with the left mouse button to cause certain preprogrammed events to occur, *Numerics* can be used to



explicitly set the values of variables, and *CRTs/graphs* can be used with to make measurements on waveforms represented by arrays of data. These interactions are detailed in section 5.3.5.

As well as creating such objects interactively, the objects can be created by a script executed from disk (see section 5.2.2 where the execution of scripts is described), which is detailed in section 5.3.6.

### 5.3.1 Mouse & Keyboard

The use of the mouse (or its emulation via the keyboard) is fundamental to the operation of the GUI. The mouse is assumed to be a 3 button mouse, with the three buttons fulfilling the functions shown in table 5.3.

The mouse is of prime importance in its interactions with screen objects, such as *buttons*, *CRTs* and *graphs*. These interactions differ for each object, and are described fully in section 5.3.5.

Left Button (LMB)	Clicking on buttons to active handlers (5.2.6), or when in poster mode the positioning of annotation
Middle Button (MMB)	If the MMB is depressed and held when the mouse pointer is over the waveform display in a <i>CRT</i> , <i>graph</i> , or <i>argand</i> window, a cursor will appear in that window. If the mouse is then moved left to right (while the MMB is held depressed) the cursor will move back and forth along the waveform. The current index value, and the value of the waveform at that index position, is displayed in the top right corner of the window. Using this technique, measurements can be made on the waveform.
Right Button (RMB)	The RMB is used exclusively to pop up menus by depressing the RMB and holding it down until a menu choice has been made. Once a menu is opened, the user chooses one of the options contained in that menu by moving the mouse so that the desired option is highlighted (inverted). At that point, the RMB is released and SPaM performs the requested action.

Table 5.3: Mouse Button Functions

### 5.3.2 The Console Window

When the SPaM display is set for text mode, the user may interactively type statements and expressions, which SPaM then evaluates. When in graphic display mode, this facility is still available through the *console window*, which emulates the display seen in the text screen mode.

If no SPaM program is running, the console window will provide interaction of the same type as the text-only display.

When a SPaM program is being run, messages printed by that program (caused by either the *print* statement, or by variable assignments which are not followed by ';') will appear in the *console*

INS	Toggle state of Left Mouse Button
HOME	Toggle state of Middle Mouse Button
PGUP	Toggle state of Right Mouse Button
END	Simulate click of left mouse button
CURSOR-KEYS	Move mouse pointer
CTRL-CURSOR-KEYS	Move mouse pointer quickly
CTRL-HOME	Move to next screen <i>button</i>
CTRL-END	Move to previous screen <i>button</i>
CTRL-PGUP	Move to next screen window
CTRL-PGDN	Move to previous screen window

Table 5.4: Keypad key functions

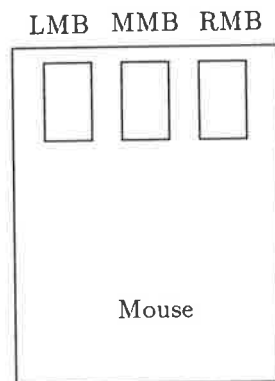
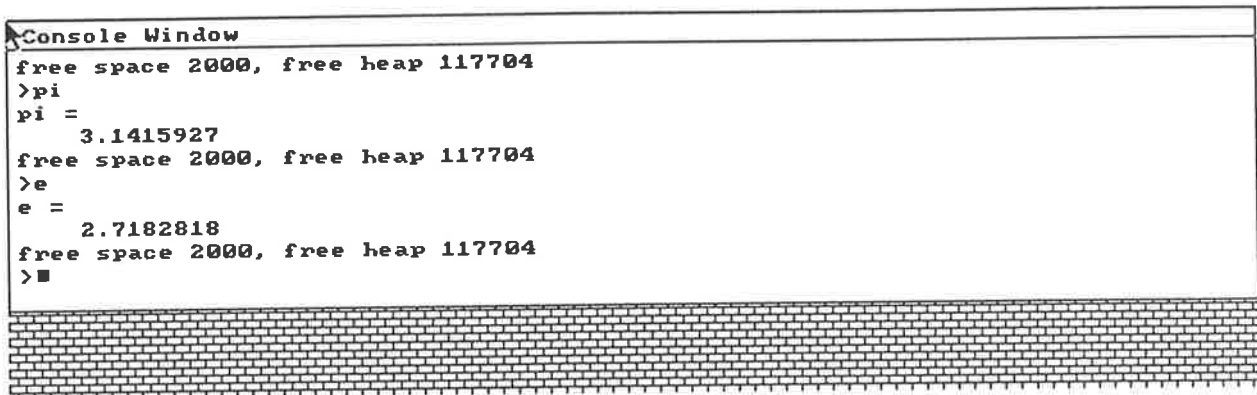


Figure 5.5: Mouse buttons  
LMB = Left Mouse Button  
MMB = Middle Mouse Button  
RMB = Right Mouse Button

window, as will any error messages encountered during program execution.

The *console window* may be resized and moved to other parts of the screen (The default location is the top third of the screen.)



```
Console Window
free space 2000, free heap 117704
>pi
pi =
  3.1415927
free space 2000, free heap 117704
>e
e =
  2.7182818
free space 2000, free heap 117704
>■
```

Figure 5.6: Console window on graphic screen  
The console window allows direct interaction with SPaM's programming language while in graphic display mode.

### 5.3.3 The Backdrop Window

The backdrop is that part of the screen which is not covered by some user created object, and is tiled with a brickwork pattern. The only important operation which the user can perform with a mouse over the backdrop window is to use the RMB to call up the menu shown in table 5.5.

### 5.3.4 Creating Screen Objects With Menus

As described in section 5.3.3, the user may create all of the available screen objects with a mouse, by selecting the appropriate creation option from the menu which opens over the backdrop screen pattern.

All of the creation operations are similar, and the user is prompted through them with text messages, so there should be little confusion.

For instance, to create a *CRT* window, the user selects the **Make CRT** option as shown in table 5.5. After doing this, the user will see a message similar to the following:

```
Move mouse to upper left corner of CRT.
Click LEFT mouse button once.
Move mouse to lower right corner of CRT.
Click LEFT mouse button once.
Click any other button to abort.
```

These instructions are self-explanatory. The user is simply being asked to point to the diagonally opposite corners of the rectangle which will define the bounds of the new *CRT* window. Once the boundary of the window is defined, the user is presented with a input window in which the following question is asked:

<b>Print Screen</b>	Dump the whole screen to the printer.
<b>Refresh Screen</b>	Cause the whole screen to be redrawn
<b>Make Button</b>	Prompt the user through the process of making a <i>button</i> on the screen.
<b>Make Numeric</b>	Prompt the user through the process of making a <i>numeric</i> on the screen.
<b>Make CRT</b>	Prompt the user through the process of making a <i>CRT</i> window on the screen.
<b>Make Graph</b>	Prompt the user through the process of making a <i>graph</i> window on the screen.
<b>Make Argand</b>	Prompt the user through the process of making a <i>argand</i> window on the screen
<b>Dump Screen Setup</b>	<p>Choosing this option causes <b>SPaM</b> to write a text file called <b>SCRLOG.M</b> in the current directory.</p> <p>This file contains the <b>SPaM</b> script commands to recreate the screen exactly as it appeared at the time the file was created.</p> <p>This feature is useful when the user manually designs a screen layout, and wants to incorporate that layout into her own script file. The setup can first be dumped to the file <b>SCRLOG.M</b>, and then incorporated into the user's program script using a text editor.</p>

Table 5.5: Menu for the Screen Backdrop

What is the name of the variable to display in the CRT -

The user simply types the name of the variable which she wishes to be displayed in that window, for instance 'x', and presses the return key. The *CRT* window will then be created on the screen.

If the variable which the user has specified for that window is already defined, a graph should appear in that window. If the variable is undefined, the symbols '???' will be drawn at the center of the window to indicate that the variable is undefined.

The creation process for all other screen objects is similar to the one described above, and certainly no more difficult.

Once screen has been set out according to the user's wishes, she can keep a permanent record of the layout by selecting the **Dump Screen Setup** option described in table 5.5.

### 5.3.5 Interactions with Screen Objects

Interactions with screen objects occurs in two distinct ways. Firstly, there is direct interaction, such as clicking on a *button*. Secondly, there is the less direct interaction of using menus.

Menu operations fall into two main categories: those that are generic to all screen objects, and those that are object specific. The generic operations are detailed in the following paragraphs, while the object specific menu operations are detailed in the following sections.

- **Move/Resize**

This option allows screen objects to be repositioned or resized. Basically, the user is asked to define a new bounding box for the object, in a process similar to the one used to create the object (if it was created interactively, of course.)

After the user has defined the new bounding box, the object will be erased and redrawn in the position.

- **Delete**

The selected object will be erased from the screen, and deleted from SPaM's internal list of objects. If the object is linked to a variable, for example a *CRT* is linked to the variable which it displays, the variable will not be deleted when the object is deleted.

#### Interactions with Buttons

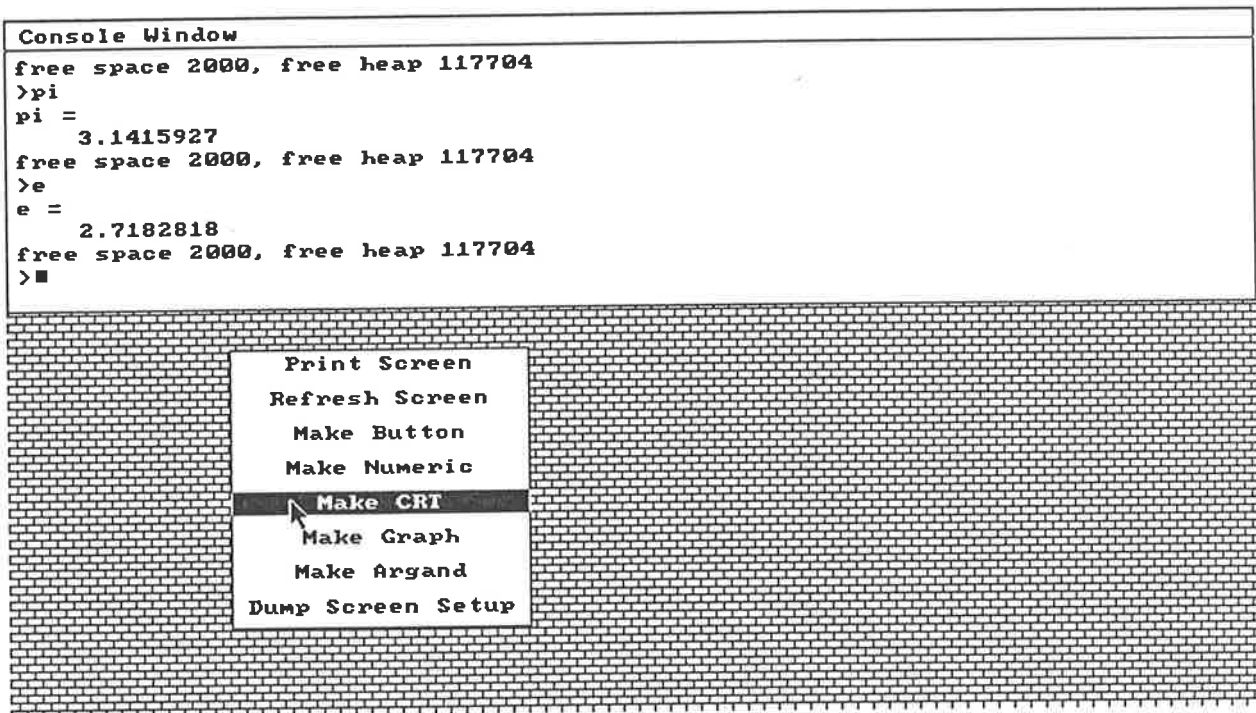
The most important button interaction with a *button* is the clicking of the Left Mouse Button (LMB) over the screen *button*. A *click* is defined as the pressing of the LMB and releasing it immediately.

When the user clicks on a *button*, SPaM searches through its internal lists for a user-defined handler which has the same name as the button. If such a handler is found, its code is executed immediately. If no corresponding handler is found, then an error message is printed. See section 5.2.6 for details on defining handlers.

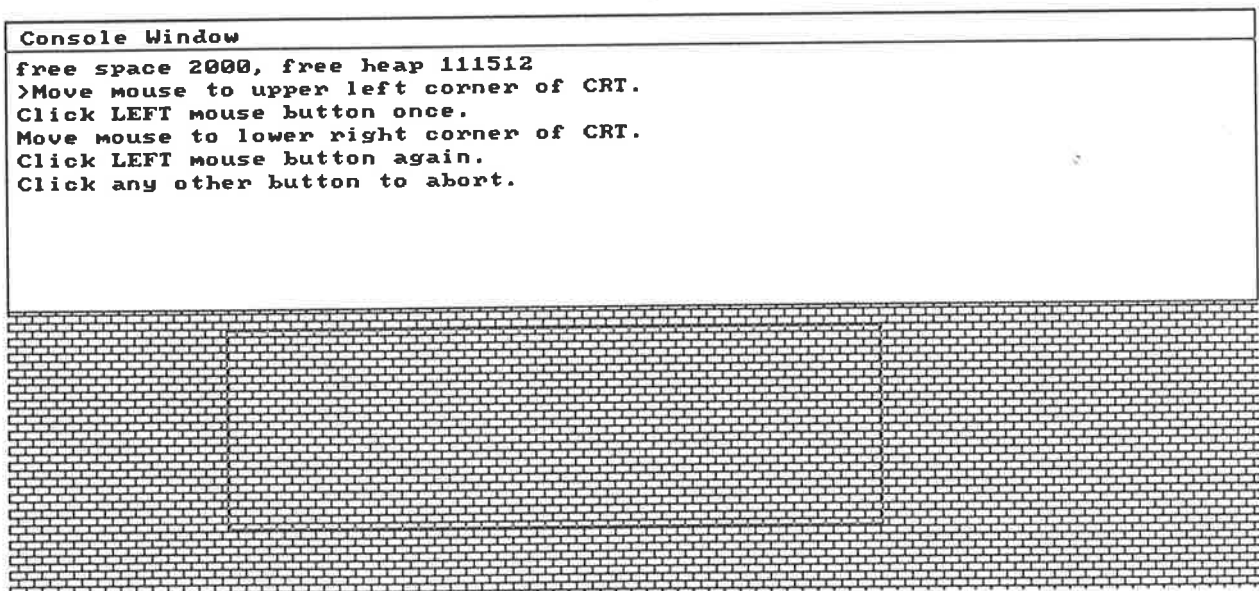
There are no menu options specific to *buttons*.

#### Interactions with Numerics

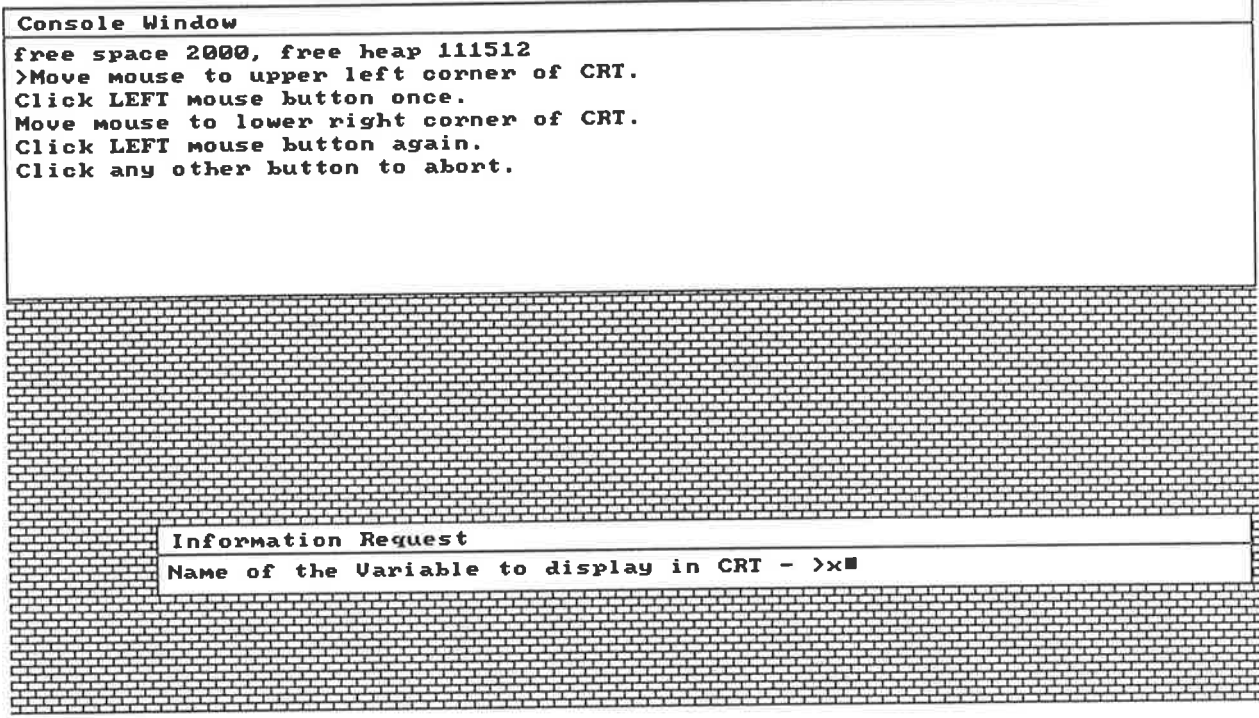
The only interaction possible with a numeric is the explicit setting of the value of the variable which it displays. This action is generated by the one *numeric*-specific menu option, called **Set Value**.



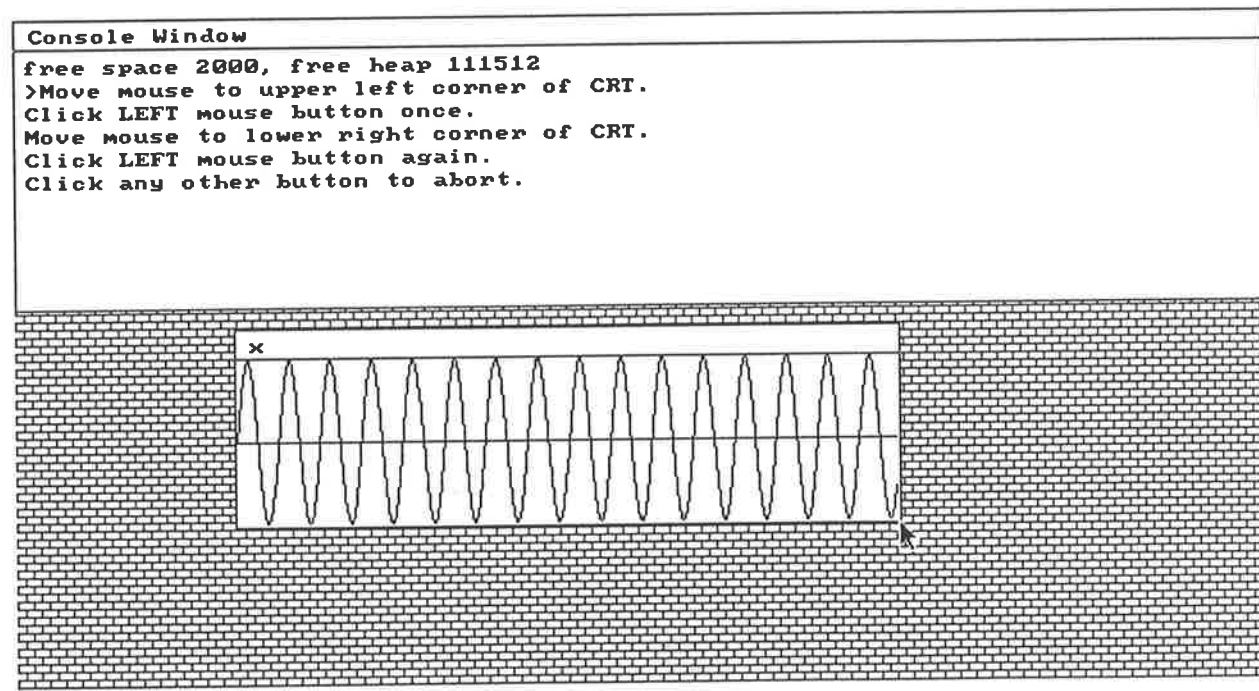
(a)



(b)



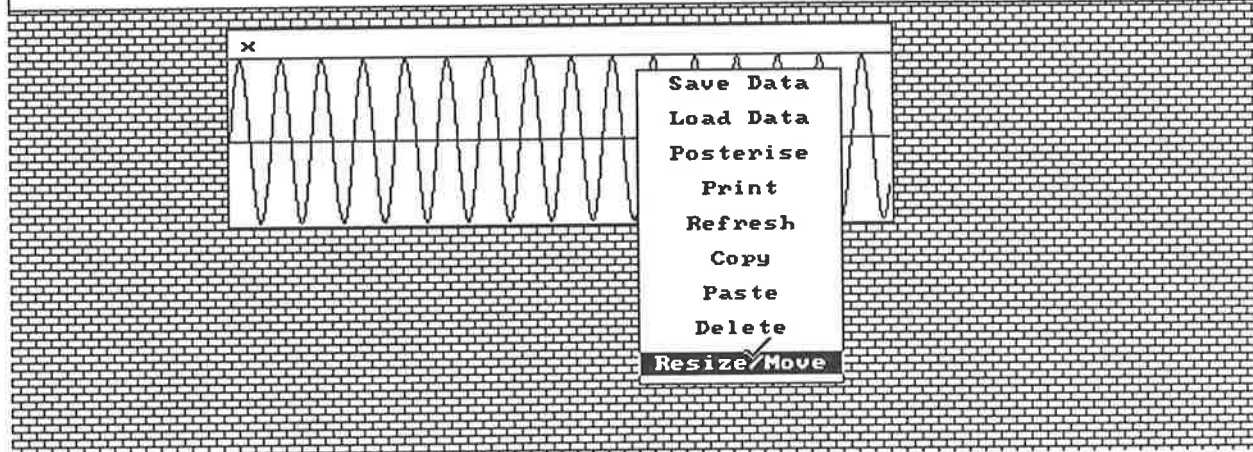
(c)



(d)

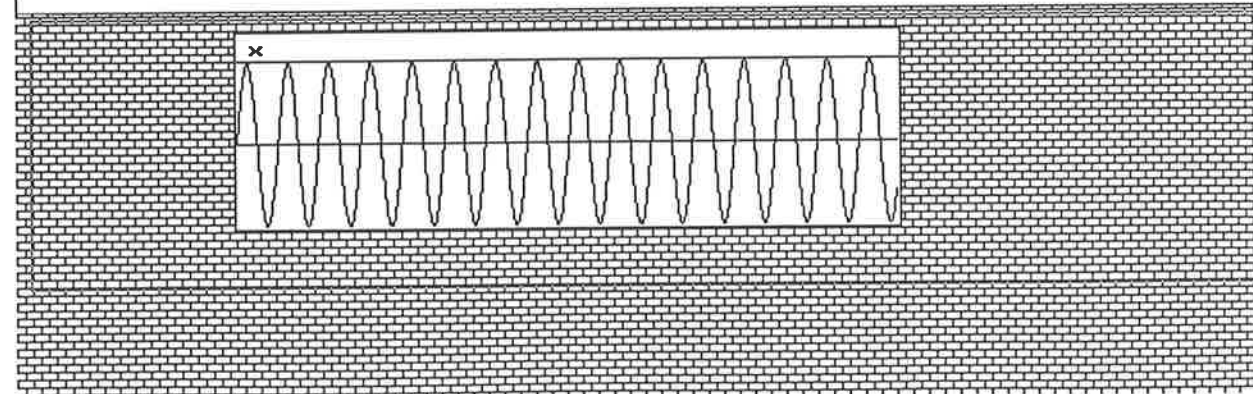
Figure 5.7: Creating a display window with the mouse  
(a) Select Option from menu  
(b) Outline boundary for desired window  
(c) Enter name of variable to display in this window  
(d) Resulting window.

**Console Window**  
free space 2000, free heap 111512  
>Move mouse to upper left corner of CRT.  
Click LEFT mouse button once.  
Move mouse to lower right corner of CRT.  
Click LEFT mouse button again.  
Click any other button to abort.



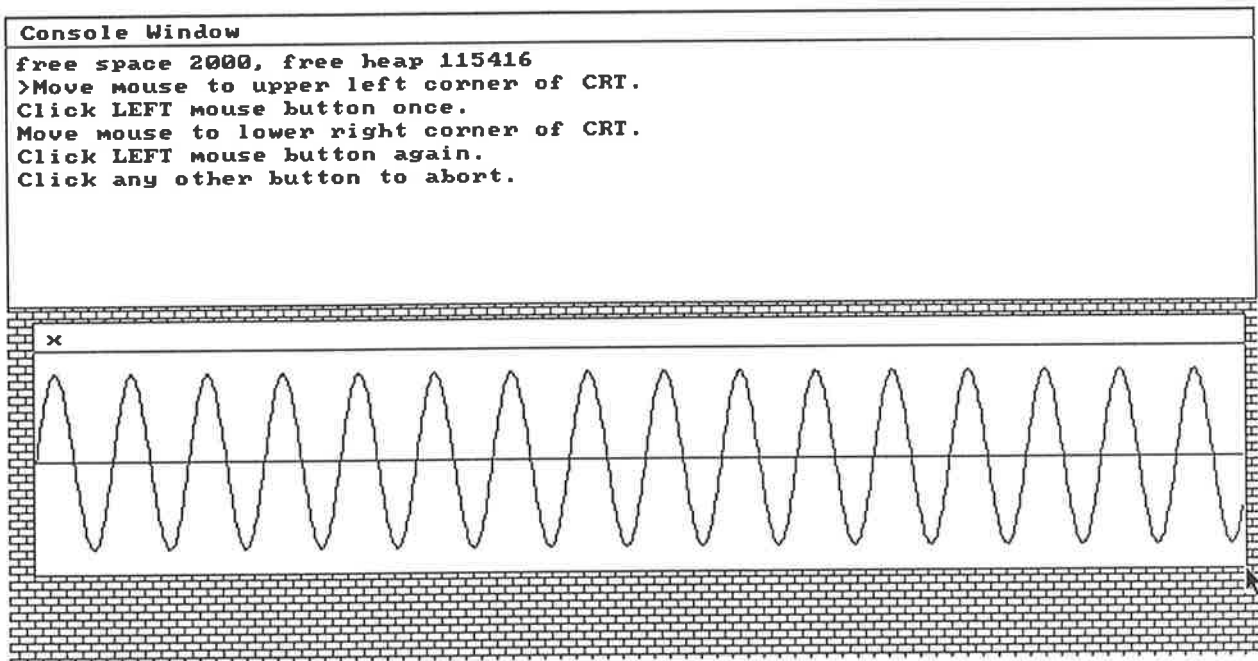
(a)

**Console Window**  
Click LEFT mouse button once.  
Move mouse to lower right corner of CRT.  
Click LEFT mouse button again.  
Click any other button to abort.  
Move mouse to upper left corner of CRT.  
Click LEFT mouse button once.  
Move mouse to lower right corner of CRT.  
Click LEFT mouse button again.  
Click any other button to abort.



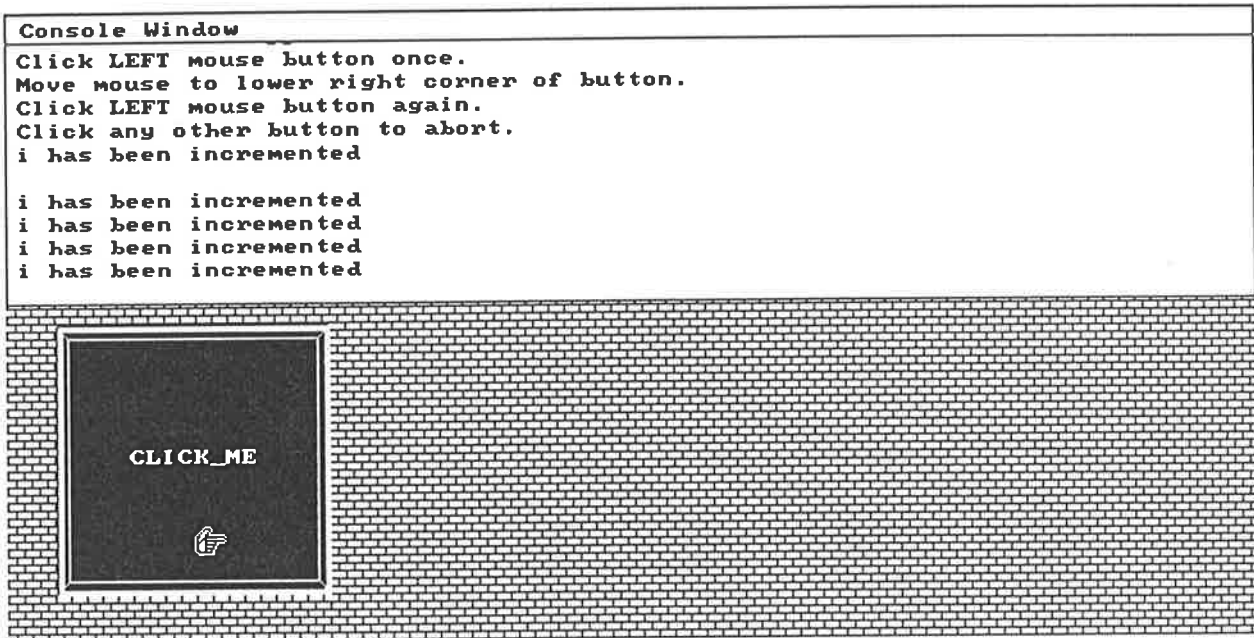
(b)





(c)

Figure 5.8: Moving/resizing graphics windows  
(a) Selecting the menu option with the right-mouse-button  
(b) Drawing the new window boundary  
(c) The result.



(a)

```
handler increment
  i=i+1; % semicolon so new value is not printed each time
  print "i has been incremented"
end
```

(b)

Figure 5.9: SPaM button activation

(a) Button being clicked. Results are printed in the console window. Buttons are clicked with the left-mouse-button.

(b) Handler which is executed when button is clicked. The value printed by the handler called 'increment' can be seen in the console window.

After choosing this menu option, a requestor window will open and the user will be requested to type in the numeric value to be assigned to the variable. When the user hits RETURN, the variable will be assigned that value.

Note that the value typed in by the user will be coerced to the same number-representation present in the variable before the operation was selected. As an example, if the *numeric*-window was displaying a variable whose type was 16-bit integer, and the user types in the new value of 1.23, the value will be coerced to a 16-bit integer, and will end as a simply 1.

## Interactions with CRTs

### The effect of using Middle Mouse Button on CRT

This section applies equally to *graph* and *argand* windows. If the user positions the mouse pointer over the waveform displayed in the window, and depresses (and holds) the MMB, a vertical cursor (or a crosshair for *argands*) will appear. Moving the mouse left and right while holding the MMB down will move the cursor along the waveform.

As the cursor is moved, the current index value of the cursor, and the value of the waveform at that index position, is displayed in the top right corner of the window.

Releasing the MMB freezes the cursor at its last position, and the last numeric values remain in the top right corner of the display.

A new cursor may be generated by repeating the operation any number of times, but only the numeric values of the current (or last used) cursor will remain in the top right corner of the window.

### Menu options for CRT windows

The menu options applicable to *CRT* windows are detailed in table 5.6.

## Interaction with Graphs

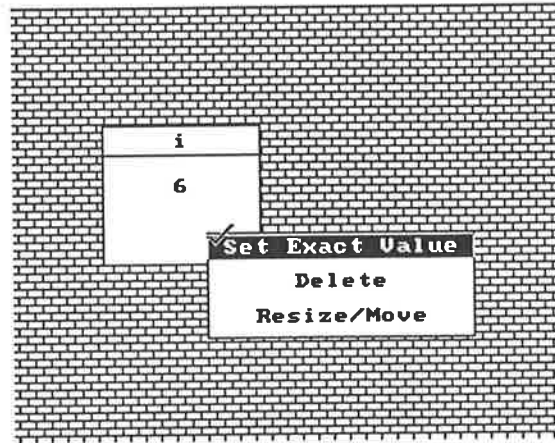
A *graph* window consists of two parts. The area where the waveform is drawn behaves exactly like a *CRT*, and the operations described in section 5.3.5 are applicable.

The area where the axis labels and numbering are displayed is the place where *graph*-specific operations are performed. In this area, the menu operations shown in table 5.7 are applicable.

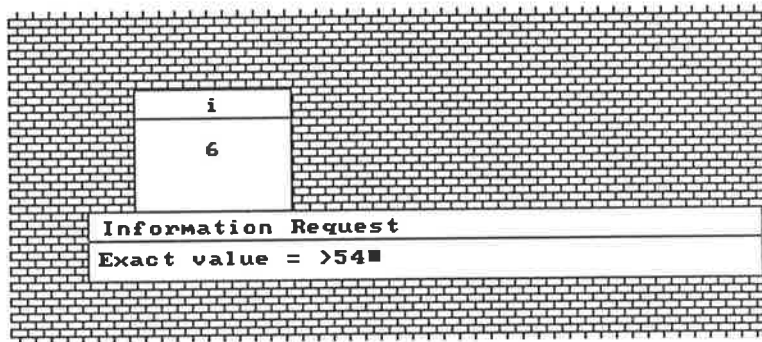
## Interaction with Argands

Interaction with an *argand* window is very much like that with a *CRT* window, except that instead of a vertical line cursor, a cross-hair cursor appears when the MMB is pressed and held over the window. The crosshair travels over the complex plane, following the data, as the mouse is moved left and right. The complex value at the current crosshair position is displayed in the top right corner of the *argand* window. Refer to figure 5.12.

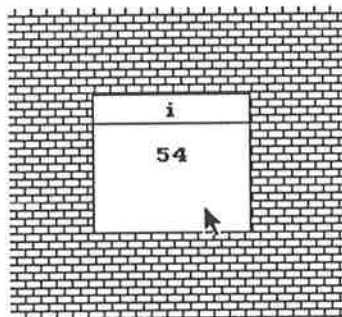
The menu operations of the *CRT* and *graph* windows also apply to the *argand* window.



(a)



(b)



(c)

Figure 5.10: Setting the value of a *numeric*  
(a) choosing the menu option  
(b) entering the new value for the numeric variable  
(c) the resulting numeric *window*

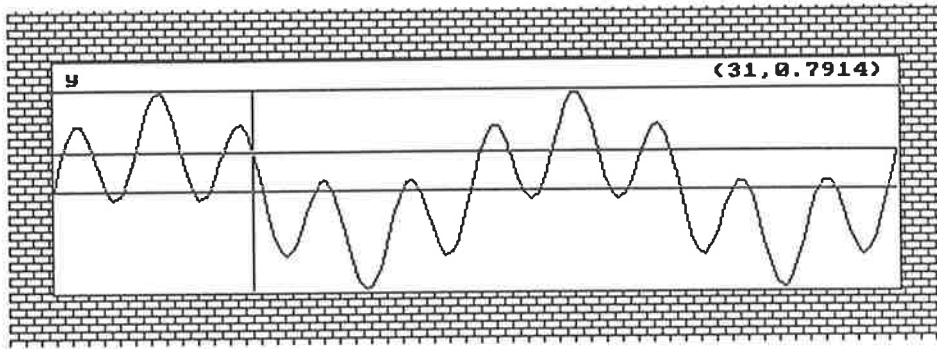


Figure 5.11: Making measurements in *CRT* window  
 The crosshair indicates the measurement point on the waveform, while the top right hand corner of the window displays the  $(x, y)$  values.

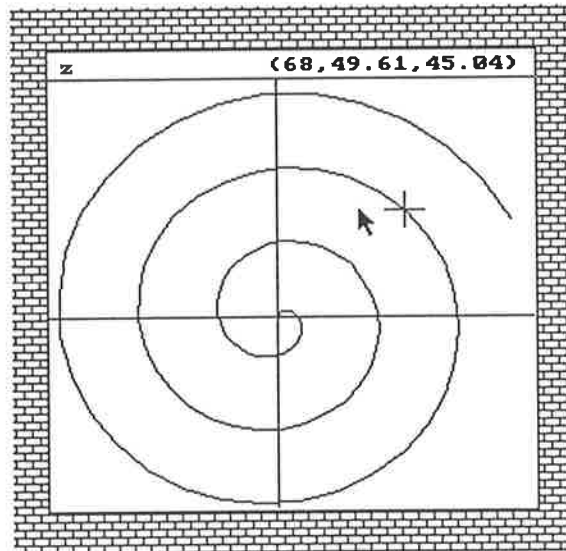


Figure 5.12: Making measurements on *argand* window

<b>Save Data</b>	The user will be prompted to type in a file name. SPaM will then write the contents of the variable displayed in the <i>CRT</i> window into the specified disk file.
<b>Load Data</b>	The user will be prompted for a file name. SPaM will attempt to read the contents of the file into the variable displayed in the current <i>CRT</i> window. The window will then be refreshed to display the data.
<b>Posterise</b>	The current <i>CRT</i> window is expanded to full-screen size, and SPaM switches into poster-mode. A new set of menus applies here, as detailed in section 5.3.7.
<b>Print</b>	The image within boundaries of this <i>CRT</i> window is dumped to the printer.
<b>Refresh</b>	The contents of this <i>CRT</i> are redrawn.
<b>Copy</b>	Before this option is invoked, the user must have generated two 2 different cursors, as detailed in the preceding paragraphs. When this menu option is selected, the user is prompted for a variable name. The variable need not already exist. When the name of the variable has been given, the range of the array (which is displayed in the current <i>CRT</i> ) bounded by the 2 cursors is copied to the variable whose name the user has given. In effect, the piece of waveform between the last 2 cursors in that <i>CRT</i> window is copied to a new variable.
<b>Paste</b>	Prior to selecting this option, the user must have defined 1 cursor on the current <i>CRT</i> window. When this option is selected, the user will be prompted for the name of a variable. That variable must already have been defined. Once the user has supplied the name, SPaM will insert the element values from the specified variable into the variable in the <i>CRT</i> window at the last cursor position. After the insertion, the window will be refreshed.

Table 5.6: *CRT* Window Menu Options

### 5.3.6 Programmed Generation of Screen Objects

It would be tedious indeed if the user had to manually create all of the necessary display objects each time SPaM was run. Fortunately, there is a way to automate the process.

Screen objects can be created by commands typed on SPaM's command line, and therefore from within SPaM program scripts. Objects can be fully specified by the user, or simply the size of the object specified and SPaM allowed to place it in a free part of the screen.

A mid-point between these two approaches allows the user to interactively create objects on the screen with the mouse, and then dump the setup in text form to a file. The file may be edited to

<b>X Axis Label</b>	When this option is selected, the user is prompted for a text string. This text string is then displayed in the <i>graph</i> window as the x-axis label.
<b>X Axis Min Val</b>	<p>By default, the x-axis in a <i>graph</i> window is numbered with the array index numbers. For example, if the array of data it displays has 1024 elements, the x-axis is numbered 1..1024.</p> <p>If the X-Axis minimum value, and the maximum value (see below) are both defined, then they will be used to generate an alternate numbering scheme for the x-axis of the waveform display, according to the following formula.</p> $label(i) = X_{min} + \frac{i}{i_{max}}(X_{max} - X_{min})$ <p>where <math>i_{max}</math> is the maximum index value of the array (ie the array has index values 1..<math>i_{max}</math>).</p> <p>To reverse this numbering, and revert to index numbering, the <b>X Axis Index</b> menu option (below) should be selected.</p>
<b>X Axis Max Val</b>	Together with the X-Axis minimum value (above), setting this value allows the numbering of the x-axis to be customised.
<b>Y Axis Label</b>	<i>See X Axis label.</i>
<b>Y Axis Min Val</b>	<p>By default, the y-axis of a <i>graph</i> window is auto-ranging. That is, the array of data is scanned for maximum and minimum values, and the display is scaled to those proportions.</p> <p>If the user wishes to have a constant scaling on the y-axis, she must set the y-axis minimum and maximum values, using this menu option and the one below. The minimum value will be the one at the bottom of the <i>graph</i> window, and the maximum value will be one at the top of the <i>graph</i> window.</p> <p>If the waveform to be displayed extends outside of the range limits imposed by the user, the display will be clipped.</p>
<b>Y Axis Max Val</b>	<i>See Y Axis Min Val.</i>
<b>X Axis Index</b>	If the user has set custom x-axis numbering using the <b>X Axis Min Val</b> and <b>X Axis Max Val</b> menu options, the user can revert back to index-only numbering of the x-axis by selecting this menu option.
<b>Y Axis Auto</b>	If the user has used the menu options <b>Y Axis Min Val</b> and <b>Y Axis Max Val</b> to set the range of display on the y-axis, the display can be made to return to vertical autoranging by selecting this menu option.

Table 5.7: Graph window menu options

make minor changes, and executed directly as a script to recreate the same display at some future time. The method for dumping the screen state to a file is described in table 5.5.

The graphics screen is a bit mapped display of some vertical and horizontal resolution. Objects are

generated at specific positions on this bit-mapped display, and have specific sizes. SPaM allows the exact sizes and positions of objects to be specified, giving full control over the way the screen is drawn. It also allows the user to specify only the sizes of objects, trusting SPaM to place them automatically. The latter method represents less work to the user, but can have unpredictable results. It is most useful for quickly placing objects on the screen for test purposes.

### Automatic Placement of Objects

Objects are automatically placed using commands of the following syntax.

```
xsize = 100;
ysize = 50;
s_var = 10;
v_var = 0:100;

auto button "name" xsize ysize
auto numeric s_var xsize ysize
auto crt v_var xsize ysize
auto graph v_var xsize ysize
```

In each case, only a variable must be specified (or a string constant in the case of a *button* creation), and the x and y size of the object in screen pixels.

SPaM maintains a tiling list of objects which are declared in this way, and will place subsequent objects in a screen tile which is not occupied. If it cannot find sufficient unallocated screen space for the object, it will return with an error message.

### Controlled Placement of Screen Objects

To retain full control over the placement of objects on the graphics screen, the following functions must be used.

```
lu_x = 10; % Left Upper x
lu_y = 10; % Left Upper y
rb_x = 10; % Right Bottom x
rb_y = 10; % Right Bottom y

button{"name",lu_x,lu_y,rb_x,rb_y)
numeric{var,lu_x,lu_y,rb_x,rb_y)
crt{var,lu_x,lu_y,rb_x,rb_y)
graph{var,lu_x,lu_y,rb_x,rb_y)
```

Objects placed in this way are not tracked by SPaM in the same way as the automatically placed objects. A screen using both automatically and manually placed objects will be prone to corruption, so the user should use one or other method.

By interactively creating objects (using mouse & menus), the user can create a screen which can be preserved by dumping the screen setup (see table 5.5). The dump file has a format much like the above example, and may be incorporated directly into a script to generate the required screen.



## Changing Attributes of Screen Objects

Some screen objects have attributes which can be set separately from the initial creation operations of the preceding section. For example, the *graph* window has optional parameters such as axis labels and axis minima and maxima.

These parameters may be set separately, using the menu operations listed in table 5.7, or using the following commands.

```
auto graph x 400 100           % first create a graph window
set labels "x" "X Axis" "Y Axis" % set the axis labels
set xaxis "x" 0.0 1.0          % x axis is now labeled 0.0 ... 1.0
set yaxis "x" -10.0 10.0       % y range is now -10.0 ... 10.0
```

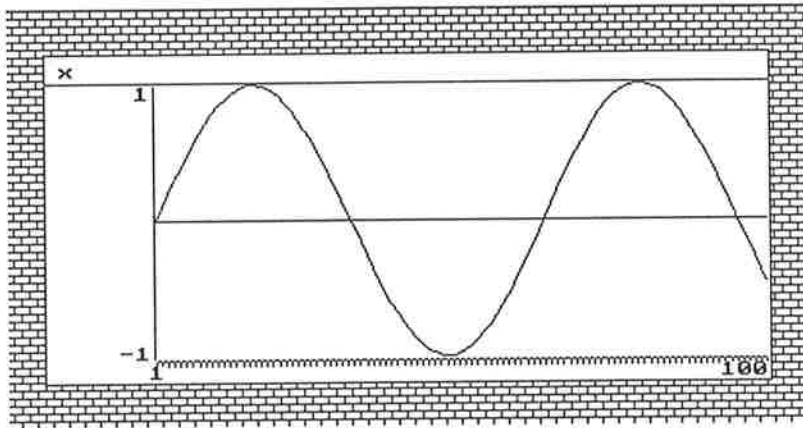


Figure 5.13: Graph before  $x$  and  $y$  scaling

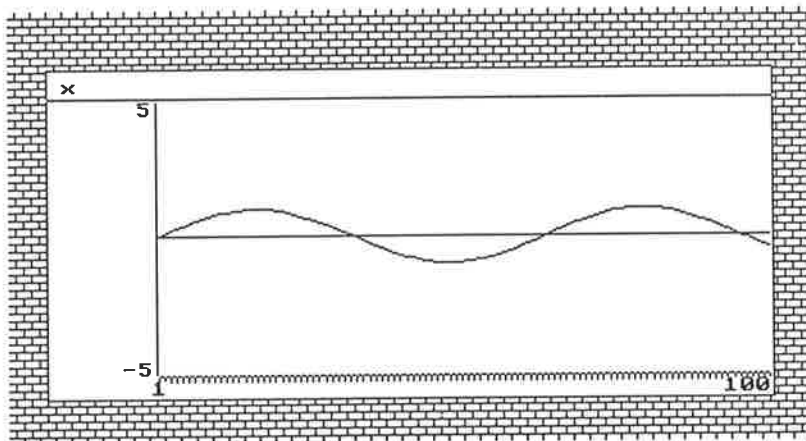


Figure 5.14: Graph following  $x$  and  $y$  scaling

Note that in the above example, the third parameter following the `set...` statement is the name of the variable whose display window is being modified, surrounded with double quotes "".

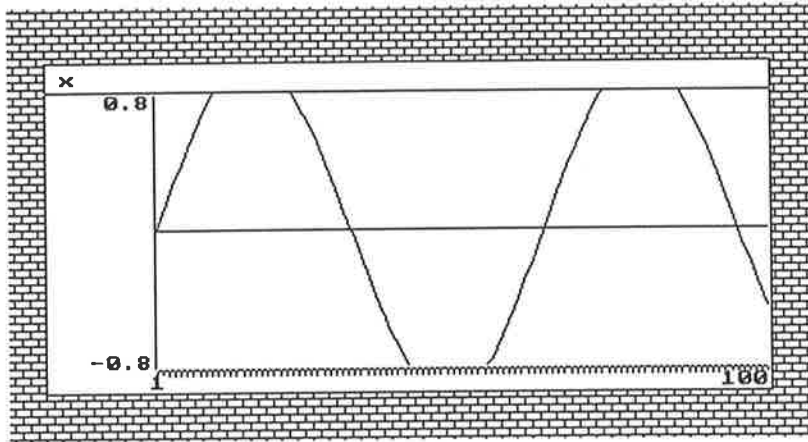


Figure 5.15: Graph showing waveform clipping. Clipping occurs if the vertical limits over too narrow a range are imposed.

The x-axis of a displayed waveform corresponds to the index of the array element at that point. The default numbering of the x-axis is according to the array index at each point. The 'set xaxis ...' statement simply causes SPaM to change the numbering on the x-axis, without any effect on the waveform display itself.

The statement 'set yaxis ...' actually sets the display range for the y-axis in that window. Only those parts of the waveform which lie in that range will be displayed on the screen, the remainder will be clipped at the top and bottom of the display window, as shown in figure 5.15.

To reverse the above operations, and return the attributes of the display window to their default values, the following statements can be used.

```
clear labels "x"
clear xaxis "x"
clear yaxis "y"
```

### Updating Screen Objects

Screen objects which represent SPaM variables, such as *numerics*, *CRTs*, and *graphs*, are not automatically redrawn when the values of those variables are changed.

The user must force a redraw operation. This is achieved using the `update` statement, which causes SPaM to check its internal lists for variables which have changed value since the screen was last redrawn. Screen objects which display these variables are redrawn. Since this can be a time consuming operation, it is left for the user to specify when the screen should be refreshed.

The user can force a selective object refresh by selecting the **Refresh** option from the object's menu (see table 5.6). A full screen refresh can be produced by selecting the **Refresh Screen** option from the backdrop window menu (see table 5.5.)

### 5.3.7 Poster Mode

The poster-mode of display is designed for producing hard-copy prints for documentation purposes. Only one *CRT*, *graph*, or *argand* can be displayed on the screen in poster-mode, and it occupies the whole screen area.

In poster mode, the menu shown in table 5.8 is called up by the RMB.

<b>LOAD DATA</b>	SPaM prompts for a file name, and loads data from the specified file into the variable which is currently displayed in the poster window. The window will be updated to show the new value.
<b>SAVE DATA</b>	SPaM prompts for a file name, and will save the contents of the variable currently being displayed in the poster window to the specified file
<b>EXIT POSTER</b>	Returns to normal screen display mode.
<b>ANNOTATE (border)</b>	<p>Annotation is a means of labeling points of interest on the waveform being displayed.</p> <p>To annotate the waveform, the user first uses the MMB to position a cursor at the point of interest, and then selects this menu option.</p> <p>At this time, the mouse pointer will disappear, and will be replaced by a rectangle which represents the size of the annotating text.</p> <p>The user moves the mouse to position the text in a desired part of the screen (where it will not obstruct or be obstructed), and presses the LMB to lay the text down.</p> <p>This particular option draws a black border around the annotating text to highlight it. The text that is printed is in the form:</p> <p><math>(x, y)</math></p> <p>where <math>x</math> is the <math>x</math>-value of the cursor, and <math>y</math> is the waveform value at that point.</p> <p>Note that annotations and comments will disappear if a <b>REFRESH</b> is caused by the user.</p>
<b>ANNOTATE</b>	<p>Identical to the <b>ANNOTATE (border)</b> menu option except that no highlighting border is drawn around the annotating text.</p> <p>Note that annotations and comments will disappear if a <b>REFRESH</b> is caused by the user.</p>
<b>COMMENT</b>	<p>This option is used to place text strings on the screen display, such as a title for the waveform plot.</p> <p>Note that annotations and comments will disappear if a <b>REFRESH</b> is caused by the user.</p>
<b>PRINT</b>	Dump the current screen to printer.

Table 5.8: Poster mode menu

The user may annotate the waveform displayed in poster mode with comments and numeric values. Numeric annotation is accomplished by moving a cursor (using middle mouse button, refer to section

5.3.5) to the desired part of the waveform, and then choosing the **Annotate** option from the poster menu. The user then positions the displayed rectangle (which represents the size of the text to be placed) in the desired position on the screen.

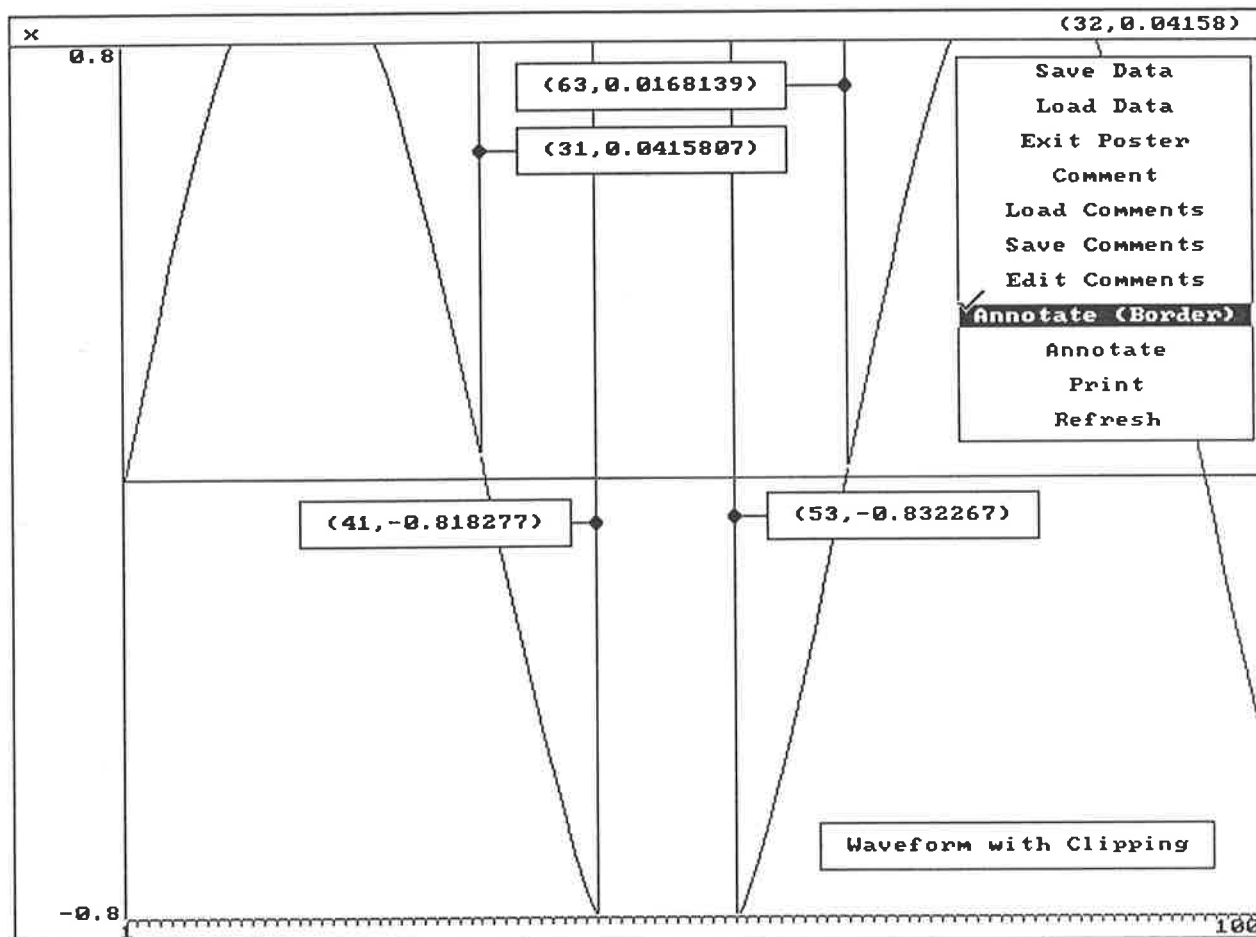


Figure 5.16: Annotated poster display

### 5.3.8 Printing the Screen Contents to a Printer

The GUI screen can be printed either in whole or in part. By selecting the **PRINT SCREEN** menu option over the background tile pattern, or by pressing **SHIFT-PRTPSCRN** on the keyboard, the whole screen will be printed.

Similarly, when in poster-mode, the **PRINT** menu option will cause the whole screen to be printed.

To print only the contents of a *CRT*, *graph* or *argand* window, position the mouse cursor over the window and select the **PRINT** menu option. Only the contents of that window will then be printed.

To be able to print, the program **PRTSCRN.EXE** must be run before **SPaM** is run from DOS. Currently, printing can only be done on EPSON compatible impact printers. Table 5.9 summarizes the means by which hardcopy can be generated.

<i>Action</i>	<i>How to</i>	<i>Result</i>
SHIFT-PRTSCRN	Keyboard	Pressing the two keys SHIFT and PRTSCRN simultaneously will cause the current screen to be dumped to the printer <i>provided that the PRTSCRN.EXE program was run before SPaM.</i>
Print Screen	Menu	The whole screen is dumped to printer, <i>provided that the PRTSCRN.EXE program was run before SPaM.</i>
Print	Menu	The object over which the mouse pointer was positioned before the menu was opened is dumped to the printer, <i>provided that the PRTSCRN.EXE program was run before SPaM.</i>

Table 5.9: Various ways of printing graphics

## 5.4 Loading and Saving Variables

Variables can be loaded from disk, and saved to disk. This enables data values to be preserved on disk, and recalled into SPaM at some later stage.

The values of variables can be saved using the `write()` function, and reloaded using the `read()` function. The following example shows how.

```
x=[1 2 3
4 5 6
7 8 9]

write(x,"x.dat"); % semicolon since write() returns 1 if ok.
                  % file "x.dat" created in current directory
x=1;              % the matrix value of x is now erased, replaced with 1

x=read("x.dat"); % contents of file "x.dat" in current directory are
                  % reloaded into variable x.
```

The files created by `write()` are created in the current directory, and those read by `read()` are assumed to lie in the current directory.

The loading and saving of variables may also be accomplished by menu operations if the screen is set for graphic display, and the variable is bound to a display window (either *CRT*, *graph*, or *argand*). The relevant menu operations are shown in tables 5.6 and 5.8.

### 5.4.1 SPaM disk file format

SPaM saves and reads ASCII data files, in which the data is represented textually. This allows data files to be created from scratch by the user or other programs, at the expense of file size and processing speed.

The file format used is documented in table 5.10.

Line No.	Alternatives	Effect
1	<p><b>matrix</b></p> <p><b>scalar</b></p>	<p>The data in this file represents the elements of a matrix.</p> <p>The datum in this file represents a scalar value.</p>
2	<p><b>real</b></p> <p><b>complex</b></p>	<p>The data/datum in this file is to be read as real numbers.</p> <p>The data/datum in this file is to be read as complex numbers.</p>
3	<p><b>integer</b></p> <p><b>long</b></p> <p><b>float</b></p>	<p>The data/datum in this file is to be stored in memory as 16-bit integer values.</p> <p>The data/datum in this file is to be stored in memory as 32-bit integer values.</p> <p>The data/datum in this file is to be stored in memory as double-precision floating point numbers.</p>
4	<p><i>Rows Columns</i></p> <p><i>real_part</i></p> <p><i>real_part imag_part</i></p>	<p>If the entity is a matrix (as specified in line 1 of the file, this line holds the number of rows, followed by the number of columns.</p> <p>If the entity is a real scalar, this line holds the real part of the the number, and represents the last valid line of the file.</p> <p>If the entity is a complex scalar, this line holds the real part and the imaginary part, separated by a space, and represents the last valid line of the file.</p>
5 : N + 4	<p><i>real_part</i></p> <p><i>real_part imag_part</i></p>	<p>If the entity is a real matrix, the remaining N lines of the file each contain 1 element of that matrix.</p> <p>The matrix <math>m</math> is filled starting with element <math>m_{11}</math>, which is read from line 5, proceeding across the row to element <math>m_{1C}</math> where C is the number of columns, then wrapping around to the start of the next row.</p> <p>The total number of elements in the file must be <math>N = Rows \times Columns</math>. If there are not sufficient elements in the file to meet this criterion, <b>SPaM</b> will abort the read process with an error. If there are more elements than necessary, <b>SPaM</b> will ignore the remainder.</p> <p>If the entity is a complex matrix, the real and imaginary parts of each element must be placed on the same text line in the file. The filling of the matrix proceeds as indicated for the real matrix above.</p>

Table 5.10: SPaM file format

## 5.5 Very Large Matrices

It is inevitable that some users of SPaM will wish to process arrays or matrices of data which are larger than permitted by the available memory of their computer. To provide a partial solution, a mechanism was created whereby the data objects are stored on disk (which usually has a larger capacity than main memory), and referenced elements extracted from the disk files as needed. Such objects are called Very Large Matrices (VLM).

Of course, the penalty is a long access time for elements, but reasonable performance can be achieved by loading a set of values (rather than individual values) into memory where they can be processed quickly.

At the time of writing of this document, the only numeric type which can be stored in a VLM is a real floating point number. This will change at some future time to include all numeric types supported by SPaM.

### 5.5.1 Creating a VLM

Unlike normal, memory resident variables which are simply created whenever used, VLMs must be explicitly declared before use. This allows SPaM to create a disk file of the appropriate size (first checking to make sure that there is sufficient disk space), and to fill that file with zeros. VLM is created with a command such as the following following:

```
create vlm x 100 200
```

A VLM variable called 'x' is created, of size 100 rows by 200 columns. The only restrictions on the size of VLMs is that the product of the number of rows and the number of columns must be less than  $2^{31}$ , and the VLM must fit within the available disk space.

A double precision floating point number occupies 8 bytes of storage, so that a VLM requires 8 bytes of disk space per element of the matrix. Thus for the above example, where the matrix has 20000 elements, the disk space required would be 160000 bytes.

### 5.5.2 Using a VLM

Using a VLM is little different from using a memory resident matrix. The main difference is that with a memory resident matrix, any assignment to a matrix element outside of the current matrix dimensions will cause the matrix to be suitably enlarged.

A VLM cannot be enlarged in this way. Its size specified in the declaration (see section 5.5.1) is represents the boundary for that variable for all time.

In all other ways, a VLM variable behaves like a normal memory resident matrix. The properties of matrices are detailed in section 5.2.3.

### 5.5.3 Caching a VLM

Since VLM variables are stored on disk, references to their internal elements are slowed down by the disk access time. Often, the operations performed on VLMs are ones which scan the elements of the VLM in a linear fashion, so that if a whole row (or column) were read in at one time, the average access time would be decreased.

SPaM provides a caching mechanism which allows SPaM to read in more than just the one element being sought in a reference to the VLM. The user defines how many elements are to be read in at a time. When the user (or her program) script refers to an element of a VLM, the cache is first examined to determine whether the element has already been read from disk. If it has, it is simply returned directly from the cache. SPaM therefore provides not only caching, but look-ahead fetching of matrix elements.

If the required element does not exist in the cache, it is read from the disk file which represents the VLM. At the same time, the cache is filled with the elements in the file which follow the specified element.

The user specified cache is split into 2 halves of identical size, to form two distinct caches. These are used according to a simple LRU (Least Recently Used) rule, so that the cache which is filled with new data is not the one which was last read from. The cache is not associated with any one VLM in particular.

This principle is important when a mathematical operation is being carried out on two distinct VLMs. Since the cache will try to service both, then thrashing would result if only one cache buffer existed. By having two buffers with an LRU algorithm, each VLM will effectively have its own buffer. Similarly, operations which refer to elements in two distinct areas of a VLM will benefit from this dual buffer scheme.

Since most mathematical functions in SPaM operate on one or two operands, the dual buffer scheme provides the major increase in performance for VLM operations. Adding more buffers would increase performance further, at the expense of memory consumption and complexity, but with decreasing improvement. Most operations carried out in SPaM will be of the form:

```
x=func(y(i,j));    % where y is a VLM.  
z=x+y;           % where z,x,y are VLM.
```

and these operations typically have only one or two VLMs on the right hand side of the assignment.

To cache a VLM the user must first create a cache variable. A cache variable must be a floating point matrix. The data area of the matrix is used as the cache buffer. Since the user can create the cache variable to be of a given size, she can determine the cache buffer size. An example of how to create a cache variable follows:

```
cache_v(40)=0.0;
```

Assuming that the variable `cache_v` had previously been undefined, this operation would create a  $1 \times 40$  matrix, which would be used by SPaM as two 20 element caches. Note that there is no simple formula for determining the optimal cache size. It depends on the operations being performed, the access speed of the computer's disk, and the size of the VLM matrices.

To cause SPaM to use the cache variable, the following instruction must be executed.

```
cache(x);
```

SPaM will now use the variable `x` as the cache for all subsequent VLM operations. The user should forget that `x` exists, since no useful data can be read out of it by the user, and setting the value of `x` will corrupt the cache contents.

To remove caching, the following statement is used.

```
cache off
```



## 5.6 SPaM Language Reference

In the SPaM language there are a several types of tokens. A token is simply a collection of contiguous characters. Tokens are separated by spaces, tab characters, or newline characters, and therefore a token may not include any of those characters<sup>6</sup>.

Tokens fall into broad classes, each of which has certain rules associated with it. The rules are shown in the following table.

keyword	A keyword is part of the SPaM language. An example of a keyword is <code>while</code> , which is used as a loop constructor. SPaM's repertoire of keywords is shown in the following tables.
function (built-in)	SPaM has a collection of built-in functions. The names of these functions are reserved words, which means that they may not be used in any context but as function calls. The user may not use variables with names identical to the names of built-in functions, but she may define her own functions with the same name as a built-in function. In this case, her function will replace the built-in one.
variable name	If a token begins with a alphabetic character, and does not correspond to a keyword, or a function or handler, SPaM will assume it is a variable. If the token is found to be an existing variable, that variable will be referenced. Otherwise, a new variable is created, and declared as empty until a value is assigned to it. A variable name must begin with an alphabetic character, but may contain numeric digits and the underscore symbol <code>'_'</code> .
numeric constant	Any token beginning with a digit is assumed to be a numeric constant. Numeric constants are converted into one of three representations : 16- bit integer, 32-bit integer, and double precision floating point. The applicable rules are listed in section 5.2.3.
<code>','</code>	The semicolon is used to prevent the printing of the results of an expression evaluation.

Table 5.11: SPaM tokens

### 5.6.1 SPaM Reserved Words and Symbols

The following table lists SPaM's reserved words. They are explained in detail in the following sections.

- **Mathematical functions**

<sup>6</sup>except when the token is surrounded by double quotation marks `' '`, indicating that it is a string constant



sin, cos, abs, tan, log, ceil, sinh, cosh, asin, acos, atan, tanh, floor, exp, real, imag, mag, phase, rand, root, sqrt, unwrap, det, fft, eye, size, zero, all, range, mean, integrate, deriv, stddev, compare, inv, solve, mod, atantwo, chirp, int, long, float, max min, lu

- **Graphics Screen Manipulation**

yaxis, xaxis, window, screen, hide, mouse, update, move, crt, button, slider, argand, numeric, graphic, graph, mmb, lmb, rmb, reqltext, poster, movemenu, auto

- **Generalised Instrument Control**

cts, cfs, cits, cifs, upload, term, mon, send, run, download, upload, sdownload, pdownload, supload, getports, sendports, getport, getcd, freqgen, sendport, set, dsp, port, restart, baud

- **Miscellaneous functions**

chain, input, banner, prstack, time, cache, cachestat, read, write, beep, , while, if, else, end, abort, exit, for, function, procedure, goto, label, print, info, whof, who, whoc, whoi, clear, dir, handler, dos, edit, wait, new, show, help, cls, trace, notrace, memfree, vlm, bind, create, declare, external,

Neither variables, functions, nor handlers may have names which are identical to reserved words. Any attempt to use reserved words as names will be flagged as a syntax error by SPaM.

Since SPaM is case sensitive, reserved words may be used as names if the case of at least one character in the word is changed.

*	Multiplication operator example <code>c=a*b</code> ;
+	Addition operator example <code>c=a+b</code> ;
-	Subtraction operator example <code>c=a-b</code> ;
/	Division operator example <code>c=a/b</code> ;
"	Double quote, surrounds string literal example <code>s="hello"</code> ;
=	Assignment operator example <code>x=1</code>
;	Echo suppression operator example <code>x=1</code> ; will not echo the value of <code>x</code> after the assignment.
^	Exponentiation operator example <code>x=2^8</code> ;
@	Unstructured multiplication operator For scalars this operator is identical to the '*' operator. However, for matrices, <code>z=x@y</code> ; yields $z_{ij} = x_{ij} \times y_{ij}$
\	Pre-inversion operator example <code>z=x\y</code> is equivalent to $z = y/x$ if <code>x</code> is a scalar, and $z = x^{-1} \times y$ if <code>x</code> is a matrix.
'	Conjugate - transpose operator example <code>x=y'</code> ; will assign to <code>x</code> the transposed and conjugated value of <code>y</code> .
'	Transpose operator example <code>x=y'</code> ; will assign to <code>x</code> the transpose of the value of <code>y</code> . Note that no complex conjugation is performed for this operator.

Table 5.12: SPaM special symbols (operators)

>	Greater than example if (x>1) ...
<	Less than example if (x<1) ...
>=	Greater than or equal to example if (x>=1) ...
<=	Less than or equal to example if (x<=1) ...
==	Equal to example if (x==1) ...
!=	Not equal to example if (x!=1) ...
&&	Logical AND example if (x>1)&&(y>0) ...
	Logical OR example if (x==1)  x==2) ...
!	Logical NOT example if !(x==1) ...

Table 5.13: SPaM special symbols (conditionals & logicals)

## 5.6.2 Mathematical Functions

---

### abs

abs(x) - return the absolute value of x

If x is real, x is returned.

If x is complex, the magnitude of x is returned.

x may be scalar or matrix, but not VLM.

---

### acos

acos(x) - return the arc cosine of x

x and y must be real, and must both be either scalar or matrix. If matrices, x and y must be of identical size.

---

### all

all(x) - returns 1 if all elements of matrix x are not zero 0 if any elements of matrix x are zero.

This function allows matrices to be used in conditional tests by turning a comparison of every matrix element into a single logical result. For instance, if x and y are matrices of the same dimension, then

if(all(x==y)) print matrices are the sameend

---

### asin

asin(x) - return the arc sine of x

x and y must be real, and must both be either scalar or matrix. If matrices, x and y must be of identical size.

---

### atan

atan(x) - returns the arctangent of x

The argument x may be scalar or matrix. If a matrix, the result is a matrix whose elements are the arctangents of the elements of x. At present x must be real.

---

### atantwo

atantwo(x,y) - returns the arctangent of x/y

The arguments x and y may be scalar or matrix. If a matrix, the result is a matrix whose elements are the arctangents of the elements of x/y. At present x/y must be real.

---

### **ceil**

`ceil(x)` - returns the smallest integer larger than  $x$ .

$x$  must be real, but may be either scalar or matrix.

---

### **compare**

`compare(x,y)` - compares two matrices  $x,y$ .

The elements of  $x,y$  must be greater than or equal to zero, and real.  $X$  and  $Y$  must also have the same dimension.

The value returned is the probability that  $x$  and  $y$  are samples of the same body of data. The closer to 1.0 the result is, the higher the probability.

---

### **cos**

`cos(x)` - returns the cosine of the argument  $x$ .

The value returned is floating point. If  $x$  is a matrix, the result will be a matrix whose elements are the cosines of the elements of  $x$ . At present  $x$  must be real.

---

### **det**

`det(x)` - returns the determinant of matrix  $x$

The matrix  $x$  must be real. The determinant is calculated using the cofactor method.

---

### **deriv**

`deriv(x)` - estimate the first derivative of data.

$y = \text{deriv}(x)$ ; the result is an array of data in which  $y(i) = x(i+1) - x(i)$  and  $y(N) = y(N-1)$  since  $x(N+1)$  is undefined.

---

### **exp**

`exp(x)` - returns the exponential of  $x$

The argument  $x$  may be scalar or matrix. If a matrix, the result is a matrix whose elements are the exponentials of the elements of  $x$ . At present  $x$  must be real.

---

### **eye**

`eye(n)` - returns an identity matrix of size  $n \times n$

The identity matrix will be of type integer.

---

---

**fft**

`fft(x)` - returns the FFT of vector `x`

Note that `x` must be a  $1 \times n$  or  $n \times 1$  vector, where  $n = 2^k$  for some integer  $k$ . FFT of matrix is not presently supported. `X` may be either real or complex.

---

**float**

`float(x)` - returns the floating point value of `x`

The value of `x` is returned, represented as floating point. If `x` is a matrix, a matrix is returned.

---

**floor**

`floor(x)` - returns the largest integer smaller than `x`.

`x` must be real, but may be scalar or matrix.

---

**imag**

`imag(x)` - returns the imaginary part of complex number `x`.

The imaginary part will be returned in the same format as `x`, ie floating-point, integer, or long integer.

If `x` is real, then the an error will be signalled. `Imag()` operates on scalars and matrices.

---

**int**

`int(x)` - returns the integer value of `x`

The value returned consists of the truncated integer value of `x`. If `x` is a matrix, a matrix is returned. The result consists of 16bit integers.

---

**integral**

`integral(x)` - estimate the integral of an array of data.

This function returns the arithmetic sum of all elements in `x`. `X` must be real and a matrix, but not a VLM.

---

**inv**

`inv(x)` - return the inverse of a matrix

The inverse is calculated by LU decomposition and back substitution. The determinant of the matrix is checked, and if less than  $1e-14$ , the matrix is assumed to be singular, and an error will be signalled.

---

## **log**

`log(x)` - returns the natural logarithm of `x`

The argument `x` may be scalar or matrix. If a matrix, the result is a matrix whose elements are the logarithms of the elements of `x`. At present `x` must be real.

Although not defined, a logarithm of 0 is signalled as a warning at present, and is assigned a value of 0. In the future this may be flagged as an error.

---

## **long**

`long(x)` - returns the long integer value of `x`

The value returned consists of the truncated integer value of `x`, represented as a 32bit integer. If `x` is a matrix, a matrix is returned.

---

## **lu**

`lu(x)` - return the LU decomposition of matrix `x`.

---

## **mag**

`mag(x)` - returns the magnitude of complex number `x`.

If `x` is real, then the value returned is identical to `x`. If `x` is complex, then each value in the result is the magnitude of the corresponding complex number in `x`, calculated as:

$$\text{mag}(k) = \text{sqrt}( X_r(k)*X_r(k) + X_i(k)*X_i(k) )$$

where  $X_r(k)$  and  $X_i(k)$  are the real and imaginary parts of `x`, respectively.

---

## **max**

`max(x,y)` - returns the larger of two arguments.

The arguments must be real and scalar. The larger of the two is returned.

---

## **mean**

`mean(x)` - return the mean of the values in matrix `x`.

`x` must be real.

---

## **min**

`min(x,y)` - returns the smaller of two arguments.

The arguments must be real and scalar. The smaller of the two is returned.

---



---

**mod**

`mod(x,y)` - return the remainder of  $x/y$

$x$  and  $y$  must be real, and must both be either scalar or matrix. If matrices,  $x$  and  $y$  must be of identical size.

---

**phase**

`phase(x)` - returns the phase values of complex number  $x$ .

If  $x$  is real, an error is signalled. If  $x$  is complex, the result consists of phase values calculated as :  
 $\text{pha}(k) = \text{atan}(\text{Xi}(k)/\text{Xr}(k))$

where  $\text{Xr}(k)$  and  $\text{Xi}(k)$  are the real and imaginary parts of  $x$ , respectively.

---

**rand**

`rand(x)` - returns  $x$  random floating point numbers.

If  $x = 1$ , then the result is a scalar.

If  $x > 1$ , a vector of size  $(1,x)$  is returned, the elements of which are different random numbers.

If  $x < 1$ , an error is signalled.

---

**range**

`range(x)` - prints the range of the argument  $x$

The minimum and maximum real and imaginary values of  $x$  are displayed, but not returned. `Range()` should be thought of as a procedure and not a function, though this may change.

---

**real**

`real(x)` - return the real part of complex number  $x$ .

The real part will be returned in the same format as  $x$ , ie floating point, integer, or long integer. If  $x$  is real, then the value returned is identical to  $x$ . `Real()` operates on scalars and matrices.

---

**root**

`root(x)` - returns the complex roots of polynomial.

The polynomial is represented by vector  $x$  which is a  $(1,M)$  vector, which represents the coefficients of a polynomial of degree  $M-1$ .

The  $x(1,1)$  element represents the zero-power coefficient, while the  $x(1,M)$  element represents the  $M-1$  power coefficient.

---

The vector  $x$  may be real or complex, but the result will always be complex.

---

### **sin**

$\text{sin}(x)$  - returns the sine of the argument  $x$ .

The value returned is floating point. If  $x$  is a matrix, the result will be a matrix whose elements are the sines of the elements of  $x$ . At present  $x$  must be real.

---

### **size**

$\text{size}(x)$  - returns the row and column dimensions of the matrix  $x$

This function is normally used in a double assignment such as  $\text{row,col}=\text{size}(x)$  If only the row value is required, an assignment such as  $\text{row} = \text{size}(x)$  can be used.

---

### **solve**

$\text{solve}(A,B)$  - solve a system of simultaneous equations.

Given a system of simultaneous equations represented as  $AX = B$  where  $A$  is an  $n \times n$  matrix, and  $B$  is an  $n \times 1$  vector, the  $\text{solve}()$  function will return the solution vector  $X$ .

If  $A$  is not square, an error will be flagged.

---

### **sqrt**

$\text{sqrt}(x)$  - returns the square root of  $x$

The argument  $x$  must be scalar, but can be either complex or real. Negative real  $x$  results in a complex result.

---

### **stddev**

$\text{stddev}(x)$  - returns the standard deviation of the elements in matrix  $x$ .

$x$  must be real.

---

### **tan**

$\text{tan}(x)$  - returns the tangent of the argument  $x$ .

The value returned is floating point. If  $x$  is a matrix, the result will be a matrix whose elements are the tangents of the elements of  $x$ . At present  $x$  must be real.

---

---

**zero**

`zero(x,y)` - returns a zero matrix with x rows, y cols.

The matrix will be of type integer.

### 5.6.3 Generalised Instrument Control Functions

---

**download**

`download(variable,start)` - download array to HarPS.

The 'variable' must be an array or scalar of 16-BIT INTEGER type (either real or complex). An error will be signalled if any other type is specified.

The 'start' parameter is the address in TMS320C25 data memory to which the array will be downloaded.

example:

```
x=[10 20 30 40];  
download(x,0);
```

See also  
upload

---

**getport**

`getport(n)` - returns the value on TMS320C25 input port n.

The value returned is a 16-bit integer, representing the value seen on the input port n of the DSP device. The allowed range for n is  $0 \leq n \leq 15$ .

See also  
`getports`, `sendport`, `sendports`

---

**getports**

`getports(mask)` - return the values on TMS320C25 input ports

'mask' is a 16-bit integer number. A '1' bit in one of its bit positions (say N,  $0 \leq N \leq 15$ ) will cause the value to be read from input port N of the TMS320C25. A '0' bit in a position prevents the reading of the corresponding input port.

`getports()` always returns an 1x16 matrix. The first element corresponds to input port 0 of the DSP, while the last element corresponds to input port 15 of the DSP.

Values in the array at positions corresponding to '0' bits in the value of 'mask' have a default value of zero.

Example:

```
regs = getports(15); % read input ports 0,1,2,3 only.
```

See also  
getport, sendport, sendports "

---

### **monitor**

monitor - talk directly to HarPS onboard debug monitor.

This command turns the PC into a terminal, and connects it to the HarPS onboard debug monitor. This is a debugging facility which should not normally be needed.

See also  
terminal

---

### **restart**

restart - restart the DSP in the HarPS system.

A reset PULSE of short (microsecond) duration is sent to the DSP in the HarPS hardware system, causing it to restart execution of its current program. The 'restart' command waits for the DSP to request attention before returning control to the user spam script.

See also  
send, run

---

### **run**

run "filename" - execute DSP code in disk file.

Issuing this command is equivalent to issuing:

```
send "filename"  
restart
```

See also  
send, restart

---

### **send**

send "filename" - send a file of TMS320C25 code to HarPS.

The file called 'filename' is assumed to be an unformatted binary file of code for the DSP, which is loaded into the DSP program memory starting at address 0 (the code must therefore include the reset vector).

The processor is NOT released from its indefinite RESET state (and therefore cannot execute the code ) until the 'restart' command is issued.

Example:

```
send "c:\dsp\sampler.bin"
```

See also  
restart

---

### sendport

sendport(n,data) - causes a value to be written to a TMS320C25 output port.

The output port number n (where  $0 \leq n \leq 15$ ) of the DSP will have the value of 'data' written to it. The value must be an integer, so 'data' is first converted to a 16-bit integer.

See also  
getport, getports, sendports

---

### sendports

sendports(mask, data) - set TMS320C25 output ports.

This function causes the values in the matrix 'data' to be written to those TMS320C25 output ports which correspond to a 1 bit in the value of 'mask'.

'data' must be a matrix of 16 elements (the shape is not important), and it must be real. It is first converted to 16-bit integer type. The 16-bit integer values are then written to the ports.

'mask' is converted to a 16-bit integer, which has bits numbered 15 to 0 (from left to right). If a bit N ( $0 \leq N \leq 15$ ) is set to 1, then element number N is taken from the matrix 'data' and stored to TMS320C25 output port N.

If the value of bit N is zero, no write operation to output port N occurs.

For example,

```
foo(16)=0;           % cause foo to be 1x16 vector
foo(9)=1234;         % the value to write to port 8
sendports(256,foo) % 256 = binary 0000000100000000
                    % so only port 8 is written to.
```

See also  
getport, getports, sendport

---

### terminal

Terminal - enter direct terminal mode to the serial port.

This turns the PC into a terminal to the serial port.

---

## upload

upload(start,end) - return data from DSP data memory.

The 'upload' function returns as its result an array of number uploaded from the HarPS DSP data memory. The range begins at TMS320C25 data memory address contained in the variable 'start', and ends at the address given by 'end'.

The total number of words uploaded is therefore (end-start+1).

example:

```
x = upload(0,1023);
```

See also  
download

## 5.6.4 Graphics Management Functions

---

### argand

argand - create a window to display complex array data.

The complex data contained in the array is displayed on an Argand diagram, with the real axis horizontal, and the imaginary axis vertical.

See also:  
set, crt, graph.

---

### auto

auto - automatically place objects on the screen.

Auto must be used with other keywords, as shown in the example below.

```
auto numeric k 50 50
auto slider volt 50 50
auto button "START" 100 100
auto graph myarray 200 100
auto crt myarray 200 100
auto argand mycomplex 200 200
```

Instead of having to specify exact screen coordinates for objects, the 'auto' keyword lets you specify only the desired size of the object, and it tries to place that object in a spot on the screen that is free.

See also:  
crt, button, graph, argand, slider, numeric.

---

## button

A button is a rectangle on the screen which simulates a pushbutton switch. When the LEFT MOUSE BUTTON is pressed and released within the rectangle, spam tries to execute HANDLER (a user defined procedure) to do perform some action corresponding to the button (see 'handler').

There are two ways to create buttons. The first is to call a function:

```
% make a button called START with top left corner (10,100)
% and bottom right corner (110,200).
button("START",10,100,110,200)
```

The second is to use the 'auto' keyword to automatically place the button on the screen.

```
% we want a button 100 pixels by 100 pixels, put it anywhere
% on the screen where there is room.
auto button "start" 100 100
```

See also:

auto, handler, buttons, mouse, lmb

---

## crt

crt - create an array display window.

See 'graph'

---

## dump

Graphic screen dump - printing the graphics screen.

Printing of the graphic screen can be done in several ways.

1. type the command 'print screen'
2. Press the keys CTRL-PRTSCR on the IBM keyboard. Note that the PRTSCRN.EXE file must have been executed before spam was run.
3. Use the mouse and menus to selectively print objects. If you hold down the RIGHT MOUSE BUTTON to bring up a menu, you will see an option in the menu called 'PRINT'.

MENU 'print' over background pattern prints whole screen.

MENU 'print' over object prints only that object.

In all cases, an EPSON compatible printer is assumed.

---

## graph

graph - create a graphing window to display array data.

There are two ways to display array data. One is in a window called a CRT, which displays no numbering or labeling on its axes, the other called GRAPH, which displays both. The numbering and labeling is done using the 'set' command.

Both CRT and GRAPH can only display arrays of REAL data. To display arrays of complex data, use the ARGAND window.

To create a GRAPH window, use either of the following.

```
% create a graph window to display values of array 'x'.  
% Top left hand corner at (10,100), bottom right at  
% (110,200).  
graph(x,10,100,110,200)
```

```
% create a graph window 100x100 pixels, put it anywhere  
% on the screen where there is room.  
auto graph x 100 100
```

See also:

auto, crt, set

---

## graphic

graphic - move from text display mode to graphics display mode.

either:

```
graphic<return>
```

enters graphics mode and creates a console window about 600x200 pixels in size.

or:

```
graphic N1 N2<return>
```

enters graphics mode and creates a console window N1xN2 pixels in size.

Currently supported graphics devices include EGA, VGA, HERCULES.

See Also,

text

---

## lmb

lmb(x) - set the state of the LEFT MOUSE BUTTON.

This launches an event into the input stream of SPaM which looks like the user doing something with the LEFT MOUSE BUTTON.

```
% simulate the user pressing the left mouse button  
lmb(1);  
% simulate the user releasing the left mouse button  
lmb(0);
```



Since the LEFT MOUSE BUTTON is used to press BUTTONs, its effect will depend on the current location of the mouse cursor.

See also:  
button, rmb, handler

---

## move

move - move an object

MOVE is used in conjunction with other keywords to move screen objects from their current screen location to a new one.

```
% the following command moves the mouse cursor to screen
% coordinates (200,200)
move mouse 200 200
```

```
% the following command moves the mouse cursor to the center
% of the button called 'START'
move mouse button "START"
```

---

## numeric

A NUMERIC is a rectangle on the screen which displays the value of a scalar variable (as opposed to a GRAPH or CRT which displays the value of an array.)

To create a NUMERIC, two methods can be used:

```
% the first is to specify exactly the position of the numeric on the
% screen. In this case, we want the value of variable 'rate' to
% be displayed in a rectangle with top left hand corner at (10,100)
% and bottom right hand at (110,200).
numeric(rate,10,100,110,200)
```

```
% or we can simply specify the size of the rectangle, and let SPaM
% place it in a free part of the screen.
auto numeric rate 100 100
```

See also:  
auto

---

## rmb

rmb(x) - set the state of the RIGHT MOUSE BUTTON.

This launches an event into the input stream of SPaM which looks like the user doing something with the RIGHT MOUSE BUTTON.

```
% simulate the user pressing the right mouse button
```

```
rmb(1);  
% simulate the user releasing the right mouse button  
rmb(0);
```

Since the RIGHT MOUSE BUTTON controls pull-down menus, its effect will depend on the position of the mouse cursor at the time.

See also:  
menu, lmb

---

### slider

slider - create a sliding bar button.

This command produces a control simulating a linear control such as a potentiometer. It is not concretely defined at present.

---

### update

update - update all objects on the graphics screen.

This command causes all CRTs, NUMERICs, GRAPHs, and ARGANDs to be redrawn, so that any changes which may have occurred in the variables that they represent are reflected on the screen.

---

### xaxis

xaxis - refer to x axis of a GRAPH or CRT or ARGAND.

The 'xaxis' keyword is used in conjunction with either the 'set' or 'clear' command to carry out operations on graphic objects.

When used with 'set', it allows minimum and maximum values to be specified for GRAPH windows, and with the 'clear' command to clear those values.

The X axis of a GRAPH window is by default numbered with the index values of the array which it is displaying. If the 'set' command is used to set minimum and maximum values, then the left hand end of the graph (index = 1) will take the minimum value, and the right hand end (index = size of array) will take on the maximum value. Index positions in between the two extremes are appropriately scaled.

See also  
set, clear, yaxis, label

---

---

## yaxis

yaxis - refer to the y axis of a GRAPH or CRT or ARGAND.

In conjunction with the 'set' or 'clear' command, this keyword causes either setting of the minimum/maximum y axis display values , or the clearing of those values, respectively.

Unlike the setting of X axis min/max values, the Y axis min/max values have a profound effect on the way that the data is displayed. The bottom of the display window, corresponding to the minimum Y value, takes on the value specified, as does the top of the display window, corresponding to the maximum Y value.

The waveform of the array is then drawn to scale on the new range of the y axis. If the waveform fits within the range specified, it will be seen in its entirety. If the waveform wanders outside of the specified range, it will be clipped.

If no min/max Y values have been specified, or if they have been cleared using the 'clear yaxis..' command, the waveform will be automatically scaled to fit exactly into the display window.

See also  
set, clear, xaxis

## 5.6.5 Program Flow Control

---

### abort

abort - abort execution of the current program and resume source input.

There may be occasions when you wish an event (such as the user hitting a BUTTON) to stop the execution of a program script. The 'abort' command does this, and source entry resumes. By default, source is read from the keyboard.

If a call to the 'chain()' function was made beforehand, then source input will resume from the file specified by that call.

See also  
chain

---

### for

for - loop construct.

The use of the FOR loop is demonstrated in the following example:

```

for i=0:100
    print i
end

```

The control expression (in the above example it is '0:100' must evaluate to a vector (ie 1xn matrix). On consecutive passes, the loop variable (in the above example it is 'i') will take the value of successive elements of the vector, starting with the first, and ending with the last.

---

## goto

goto - goto a label.

The goto statement allows program execution to jump from one part of a program to another. The destination of the GOTO is defined by a LABEL statement. If no LABEL statement of the specified name has been encountered, an error will be signalled.

example

```

label start <-----
....          |
goto end     --- |
....          | |
goto start   -|---
....          |
label end   <----

```

See also  
label

---

## if

if - start a conditional statement

The syntax for the IF construct is shown by the following example:

```

if(1>2)
    print "There is something wrong in our universe!"
end

```

If the conditional test is true, all statements after the conditional will be executed until a 'end' keyword is encountered. The IF-ELSE construct is shown below.

```

if(1>2)
    print "There is something wrong in our universe!"
else
    print "Normality has been resumed."
end

```

---

## while

while - loop construct.

The use of the WHILE loop is demonstrated in the following example:

```
i=1;
while (i<10)
    print i
    i=i+1;
end
```

If the condition is true, all statements following the conditional expression and the 'end' keyword are executed. Execution is continued until the condition fails.

Note that if the condition is never met (even the first time), then the inner statements may never be executed.

## 5.6.6 Disk Access Functions

---

### chain

chain("filename") - get input from file after current program finishes.

When the current program finishes, instead of waiting for the user to type in a command, SPaM will get its input from the specified file. The current input file is closed, so that there is no limit on how many times one file can chain to a new input file.

The commands in the new file are compiled/executed only after the current program has finished, they are NOT included into the current program.

---

### dir

dir - display names of available .m files

The names of SPaM script files (.m files) in the current directory, and those directories defined by the environment variable SPAMPATH, will be displayed.

The name of the path where the script is located will also be displayed.

To find out what a script does, simply type

help name

where the name is the name of the script WITHOUT the '.m' extension. If the script contains any help information in its first few lines, that information will be printed.

See also  
scripts

---

### read

`read(fname)` - returns the value of the data in the data file whose name is in the variable `fname`.

`fname` can either be a string variable, or a string literal (surrounded by double quotes).

The disk file itself is a text file of a specific format, see the documentation for further details. eg

```
x = read("x.dat");
```

---

#### **write**

`write(fname,x)` - writes a variable to a disk file.

The variable `x` is written to the disk file whose name is in `fname`. eg

```
write(x,"x.dat")
```

### 5.6.7 Miscellaneous Functions

---

#### **clear**

`clear` - clear a value in an object which has been SET, or clear the value of one variable, or clear the value of all variables.

The 'clear' keyword must be used in conjunction with other keywords, as shown below.

```
% remove axis labels on graph called 'y'  
clear label "y"  
% clear minimum and maximum x axis values in graph called 'y'  
clear xaxis "y"  
% clear minimum and maximum y axis values in graph called 'y'  
clear yaxis "y"
```

Typing `clear` on its own will cause all of SPaM's internal lists to be purged. All variables, user defined functions and handlers, and all graphics objects will be forgotten.

Typing `clear` followed by the name of a variable will cause that variable to be forgotten, and any memory that was associated with it to be returned to the free memory pool.

See also  
label, xaxis, yaxis, set

---

#### **cls**

`cls` - clear console window.

If in text display mode, the text screen is cleared.  
If in graphics mode, the console window is cleared.

---

## dos

DOS - shell out to MSDOS.

The user is placed into a DOS COMMAND.COM shell, where he may use DOS commands as normal, though with much reduced memory available.

To return to SPaM, the user must type 'exit'.

If the user was in GRAPHIC mode, the screen will be restored to its state before the 'dos' command was issued.

See also  
edit

---

## edit

edit - invoke an external text editor.

A DOS shell is created and a text editor is run, allowing the user to edit SPaM command scripts without having to exit SPaM.

Once the editing is finished, the user should save the text file and quit the editor. This will cause SPaM to resume in the state which existed before the 'edit' command was given.

The text editor to be used is specified by the DOS environment variable 'SPAMEDIT', set in DOS as follows:

```
set SPAMEDIT=c:\dos\q.exe
```

This will cause an editor called 'q' to be run each time the 'edit' command is given.

See also  
dos

---

## input

input(prompt) - prompts for and returns a floating point value.

The argument prompt is either a literal string (surrounded in double quotes), or a variable to which a literal string has been assigned. The prompt will be displayed, the program will wait for the user to type a number, and the value of that number will be returned. eg

```
x = input("Input the value for x - ");
```

---

## notrace

notrace - prevent the compiler from echoing input

This command reverses the effect of the 'trace' command, so that source code input is not echoed to the screen.

See also  
trace

---

## print

print - prints the value of a symbol

PRINT prints the value of a variable, constant, or expression. For instance:

```
> print "This is a string"
This is a string
> print x
x =
    42.0
> print 1+2+3+4
    10
```

Note that in all cases the same result could be obtained by just typing the expression to be printed. Unless the expression (or assignment) is followed by a semicolon ';', its value will be printed by default.

Information about graphics screen printing can be found under the title 'dump.'

See also  
dump

---

## time

time(x) - returns either:

if x=0 , time in seconds since SPaM started,  
if x=1 , time in seconds since time() last called.

---

## trace

Trace - echo source input to the compiler

This command causes the compiler to echo all input until a 'notrace' command is given.

Echoing of input allows the user to easily see where syntax errors are occurring in the source code.

See also  
notrace

---

## vlm

vlm - stands for Very Large Matrix.

Normal matrices are memory resident - that means they are stored in the PC's memory between uses. The memory of the PC imposes limits on the size of a matrix. To overcome this limit, VLMS were created.

A VLM never exists in memory, but is instead stored in a file on the hard disk. When elements are to be accessed, they are read/written from/to that file. This means access is slower for a memory



resident matrix.

The best way to use VLMs is to read parts of them into memory resident matrices, which the user can then process. Many math functions are not able to directly access VLMs directly, so the user will have to process them in pieces in any case.

VLMs must be explicitly created using the 'create vlm' command.

A VLM may be any size allowed by the computers disk capacity, the only restraint being that (rows \* columns)  $\leq 2^{31} - 1$ . The elements in a VLM are REAL FLOATING numbers, though more types may be implemented in the future. Each element occupies 8 bytes, so an  $N$  element VLM will require at least  $8N$  bytes of disk space. In practice, it will require *more* due to operating system overhead.

**SPaM** checks the default drive to make sure that there is enough space for the VLM to be created, and if there is not an error will result.

See also  
create

---

## wait

wait(x) - wait for x seconds before continuing.

The number x may be floating point to pause the program for a fraction of a second, but the resolution of the interval timing is only 1/(18.2) second.

## 5.7 Online Help

SPaM has an built-in help system which the user may invoke to obtain information about SPaM language and operation. An index of help topics is displayed by simply typing:

```
help
```

This will display a screen similar to the following:

The following topics exist in my help index:

```
abort      abs        acos       all        argand
asin       atan       atantwo    auto       button
ceil       chain      clear      cls        compare
cos        create     crt        det        deriv
dir        dos        download   dump       edit
environment exp        eye        fft        float
floor      for        functions  getport    getports
goto       graph      graphic    harps      if
imag       input     int        integral   inv
keys       lmb       log        long       lu
mag        matrix    max        mean       menu
min        mod        monitor    move       notrace
numeric operators paths       phase      print
rand       range     read       real       restart
rmb       root      run        send       sendport
sendports  set       sin        size       slider
solve     spam      sqrt       stddev     tan
terminal  time      trace      update     upload
vlm       wait      while      write      xaxis
yaxis     zero
```

Type HELP "subject" where 'subject' is from the above list.

The user can obtain more specific information about any of the topics named in the index. For instance, if help about mathematical operators is required, the user types:

```
help "operators"
```

# Chapter 6

## Integration Issues

This chapter deals with system integration issues which were encountered during implementation of the GI. The GI consists of many different parts, some hardware and some software, which were integrated to form the whole.

The interaction between the GI hardware and SPaM is carried out on several levels, as shown in figure 6.1. The Application layer (user program) is the highest level of software within SPaM. The user program calls various functions which interface to the protocol layer, which in turn transmits and receives characters to and from the GI hardware.

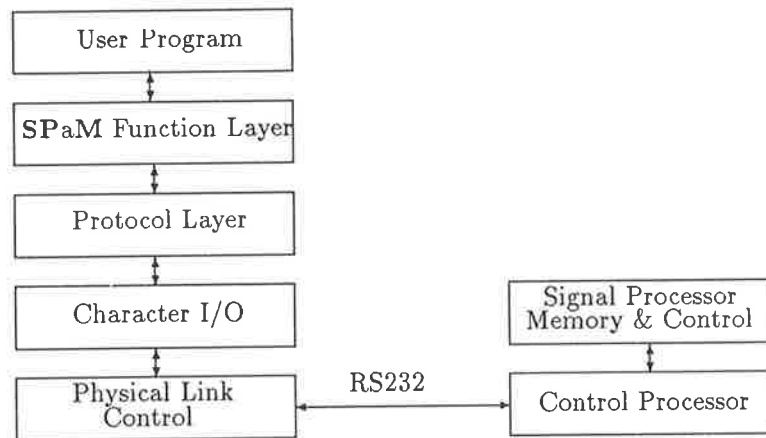


Figure 6.1: SPaM software layers

### 6.1 Aspects of the RS232 link between host and GI

The RS232 link connecting the GI hardware and the host computer poses several problems on its own. As a long distance link, it is susceptible to outside interference. The high data rates involved also mean that internal PC activity can sometimes interfere with communication. For this reason, effective management of transmission errors is essential, as discussed in the following section.

The synchronisation of host and GI is another crucial point. Communication occurs according to a state-transition diagram such as that shown in figures 6.2 and 6.3. Any deviation from the predefined paths would cause a loss of control. These issues are discussed in section 6.1.2

### 6.1.1 Link Error Management

The link between the host and GI is an RS232 interface for reasons stated in section 3.2, and this brought its own problems.

In order to increase the responsiveness of the system (by *decreasing* its response time), the highest possible serial bit rate is used. The bit-per-second rate of the GI hardware is 57600, which still allows a maximum throughput of 5.76kbytes per second (8 bits per byte, plus start bit and stop bit, yield 10 bits per byte).

Though communication on most PCs with standard serial ports is reliable at this speed, transmission errors can and do occur. The main reason is interference by higher-priority interrupt devices in the PC. Even the smallest probability of a link transmission error occurring means that the software at both ends must be programmed to detect and correct errors. The error detection and correction mechanism exists within the protocol layer of figure 6.1.

The way this was done was to implement a packet based protocol in which packets of bytes of known size are transmitted between host and GI. These packets have an embedded cyclic redundancy code which allows the receiver to determine, with a high degree of certainty, whether the packet has been corrupted during transmission.

Two types of packets are currently implemented: Command packets and data packets. Command packets originate only from the host computer, and contain commands for the control processor, as well as any numeric arguments required by those commands. Data packets can originate in either the host or control processor module.

On receipt of a packet, the receiving device must issue a response which tells the sender whether the packet was correctly received or not. If the packet arrived intact, the sender may proceed to send the next logical packet. If the packet arrived corrupted, the sender will retransmit it. A limit is placed on the number of times packets may be resent during a transmission. If this limit is exceeded, the link is considered faulty and the user notified.

### 6.1.2 Host to GI Synchronisation

In order for communication to begin, the host and control processor must be synchronised — that is, they must be in the same state (idle state.)

The host can monitor the state of GI by sending NULL characters to the GI, and waiting for a '?' character in response to each one. If the response character is received, then the GI is in the idle state and may be sent a command packet to perform a given action.

If no response character is received, the host may force the GI to pay attention by sending a break character (actually a non-character, consisting of an extended time of transmit data signal inversion.)

Figure 6.4 shows in highly simplified form the activity of the control processor (CP) in and around its idle state. In this state, the CP waits for the host to send a 'magic number' which signifies the start of a command block. Just as the '?' character is used to inform the host that the CP is alert, the magic number is used to synchronise to the start of a command block.

After receiving the magic number, the CP assumes that the following group of characters represents

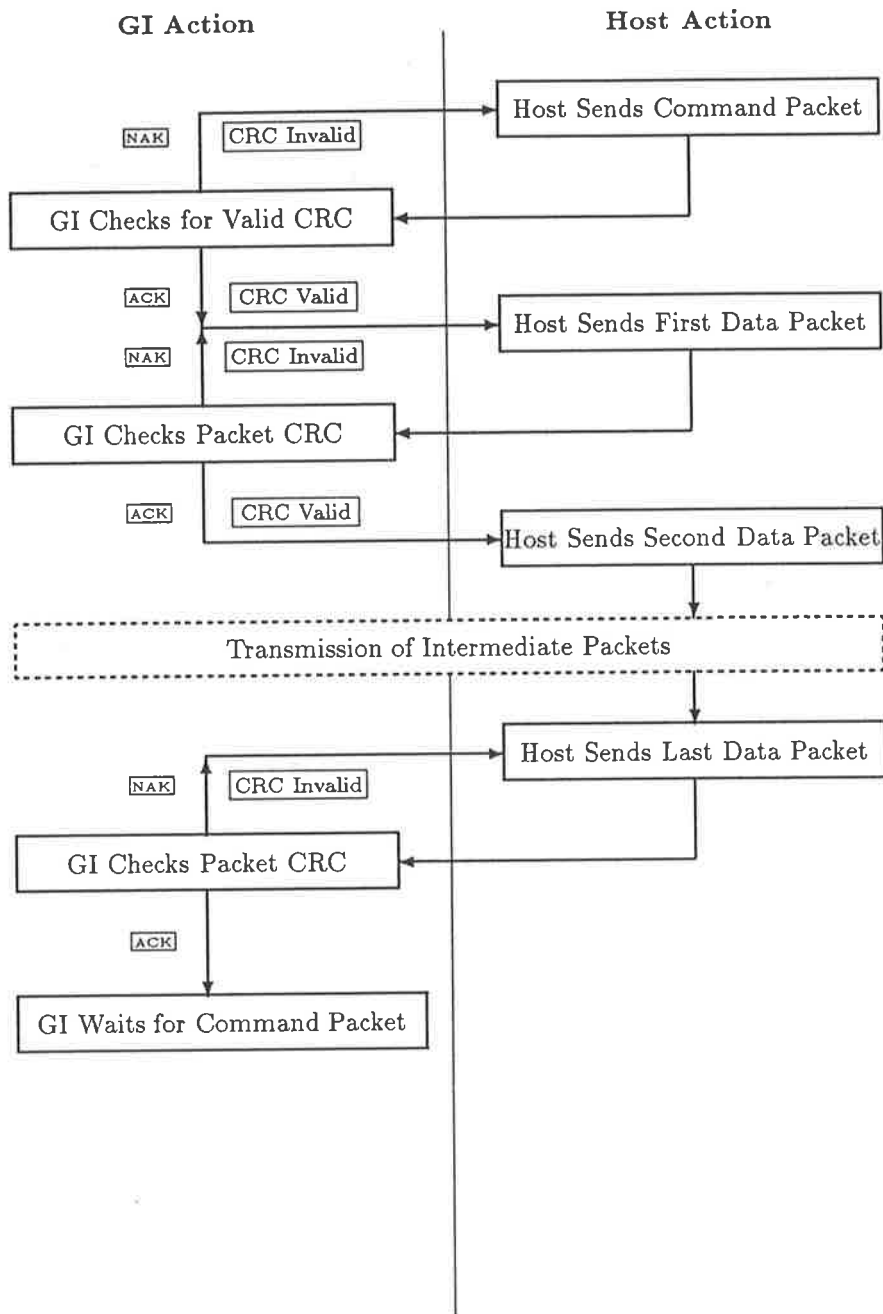


Figure 6.2: The state-diagram for Host to GI data transfer.

Note:

CRC = Cyclic Redundancy Check

ACK = Acknowledge

NAK = Negative Acknowledge

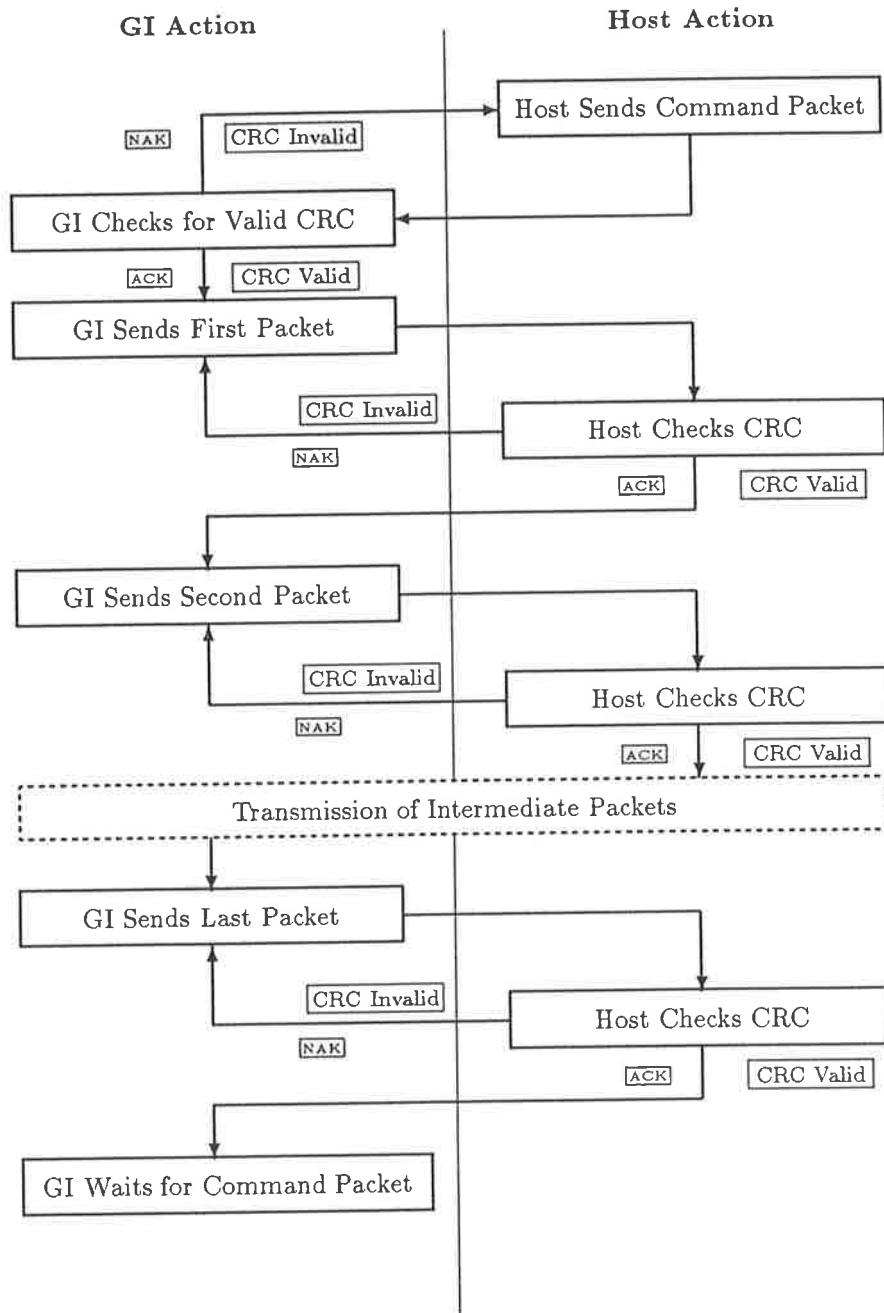


Figure 6.3: The state-diagram for GI to Host data transfer.

Note:  
 CRC = Cyclic Redundancy Check  
 ACK = Acknowledge  
 NAK = Negative Acknowledge

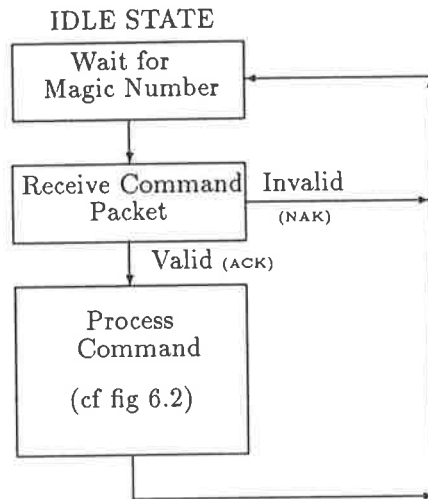


Figure 6.4: Control Processor IDLE State

a command block. After receiving the block, the CP checks the embedded cyclic redundancy code (CRC) to confirm the validity of the packet. If the block is valid, the CP acknowledges it, and proceeds to carry out the actions specified within the command packet.

Should the command packet prove to be invalid, the CP sends a NAK character to the host, and returns to the idle state. Under most circumstances, the host will attempt to resend the command packet (the exception being when the maximum number of retries is exceeded.)

## 6.2 Control Processor to Signal Processor Synchronisation

In practice, the user (or her program) sends a file of DSP code to the GI, and causes the code to be executed by the TMS320C25. In order for the following operations to be correctly sequenced, the host must be informed when the signal processor has completed execution of the code.

This synchronisation is achieved using the interprocessor port, which connects the TMS320C25 and the 68000 control processor. The DSP code, as a final action, asserts an attention-request signal which is detected by the control processor. The host is then informed that calculation is complete, so that it can issue commands to retrieve the results.

The commands provided within SPaM for control of the GI handle this sequence of operations in a manner which is invisible to the user. The only requirement is that users who write their TMS320 code must follow convention and end their program with instructions which assert the attention request signal. An example of such code can be found in the sample TMS320 program in section C.

## 6.3 Integrating Software and Hardware

This section discusses the steps which must be performed by a GI application script to emulate a test instrument. A script is a disk based text file containing SPaM commands (see section 4.6.)

The following actions must be performed by the user's script:

1. Set up graphics screen.

The graphics screen must be set to display the required waveforms, provide the necessary control *buttons*, and display any necessary numeric values.

An example of such a screen is given in figure 4.13, and the accompanying script in figure 4.14. See lines 2-5 in figure 6.5.

2. Send the necessary DSP code to the GI.

The code module for the TMS320 is loaded from disk, and sent to the TMS320 program memory. This operation is carried out using SPaM's `send` command.

Unless multiple separate code modules must be run within the single application, the code module needs to be sent only once, at the start of the script.

See line 6 in figure 6.5.

3. Download any operational parameters to the DSP.

An interactive instrument allows the user to set various parameters, such as sampling rate, offset voltages etc. These can be downloaded into TMS320 program or data space as required (the TMS320 code must be written to read these parameters and use them.)

See lines 18-22 in figure 6.5.

4. Start DSP program execution.

At this point, the TMS320 can be taken out of the RESET state, so that execution of the DSP code begins. This is accomplished using SPaM's `restart` command.

See line 24 in figure 6.5.

5. Wait for signal processor to finish.

This is done using the mechanism described in section 6.2. In practice, the user does not need to take any specific action here, since spam's `restart` command does not return control to the user's script until the DSP has finished execution.

See line 24 in figure 6.5.

6. Uploading of the computation results to the host.

SPaM's `upload` function fetches words from a specified area in TMS320C25 data memory, and stores them in a SPaM array variable, which may then be displayed.

See line 26-27 in figure 6.5.

7. Repeat.

For as long as required, repeat the operations from step 3.

As demonstrated in figure 6.5, commands which control the GI hardware are implemented either as direct keywords (eg `send`), or as functions which return a value (in the case of `upload()`, the value returned is an array of numbers.)



```

00 time_interval = 10000;           % sampling interval
01 old_time = 0;
02 auto graph x 400 100             % create auto graph window for array 'x'
03 auto numeric time_interval 50 50 % create a numeric display window
04 auto button "start" 100 50       % make a pushbutton called 'start'
05 auto button "stop" 100 50       % make a pushbutton called 'stop'
06 send "dspcro.bin"               % download TMS320 code to GI
07
08 handler stop                     % executed when 'stop' button clicked
09     stop = 1;                    % set this variable to stop loop.
10     end                          % end of handler.
11
12 handler start                    % this code is executed when the 'start'
13                                     % button is clicked by the user
14     stop = 0;                    % while stop==0, repeat the action.
15
16     while(stop==0)               % while the user has not clicked 'stop' ...
17
18         if(time_interval!=old_time) % then the user has entered a new
19                                     % time interval value
20             pdownload(time_interval,20) % download sample interval value to
21             old_time=time_interval;    % DSP program memory.
22             end                      % end of if statement
23
24             restart                % make the DSP execute its code,
25                                     % and wait for it to finish.
26             x=upload(4096,4096+1023) % upload 1024 words into array x.
27             update                  % update the graphic window
28                                     % which displays x.
29         end                        % end of while loop.
30     end                            % end of handler.

```

Figure 6.5: SPaM GI control script  
The line numbers are not part of the SPaM script.

## Chapter 7

# System Evaluation

This section compares the predicted performance of the system and its various parts to that measured on manufactured units.

### 7.1 DSP Throughput

The TMS320C25 signal processor has achieved its rated performance in the DSP module. The only possible barrier to such performance is the access time of the external memory, which may slow accesses and thereby reduce throughput.

The DSP module has been run with both 0 and 1 wait-state in the DSP hardware. To achieve full speed, fast memories of <50ns access time are required. Until recently, such memories (in suitable configurations) were expensive, and so the DSP module was designed to allow standard (inexpensive) commercial 100ns parts to be used with 1 wait-state accesses.

Fortunately, there has been a drastic decline in the price of fast SRAM, pushed largely by its widespread use in consumer computer technology (eg cache RAMs), so that a 10-MIPS DSP module can now be constructed for little more than one utilising slower SRAMs.

With 0 wait-state external accesses, the TMS320C25 has a maximum throughput of 10MIPS (millions of instructions per second). The actual throughput for particular algorithms is much more complex and will not be discussed here, since it depends on the actual program and data flow. From the hardware perspective, the DSP module is achieving its nominal design throughput.

Cost of the DSP hardware can be cut without sacrificing a great deal of performance, especially when the signal processing algorithms to be used consist of small software loops. The TMS320C25 has internal memory which may be configured as program memory. Small programs may be loaded from external memory into this internal memory, and executed at full speed, even though the external memory may require 1 wait state accesses.

The throughput of signal processing software cannot be simply defined or measured in a general fashion. One software task which can be benchmarked is that of signal sampling. The DSP module with the prototype AIM can achieve sampling rates of 1.2MSPS (performed by software). A minor redesign of the AIM would double this to 2.4MSPS.

### 7.1.1 A/D System Performance

While only a prototype, the Analogue Interface Module (AIM) clearly showed the path which must be followed for future designs. As an emulator of test instruments, the GI must be able to set all of the essential characteristics of its circuitry under software control. These characteristics include:

1. Input stage gain.
2. Voltage offset at the A/D signal input.
3. Anti-aliasing filtering
4. Output range.
5. Output offset.

The prototype AIM does have programmable gain amplifiers, allowing one from a set of gains between 1 and 256. However, the amplifiers also amplify a selection of DC offsets within the AIM module, causing a loss in effective resolution of the A/D. For this reason, as well as the obvious one of DC signal cancellation, programmable offset removal must be provided.

### 7.1.2 Link Throughput

The link transfer *burst* rate of 57.6kbps was approached closely by the *effective* rate. In tests<sup>1</sup>, the link protocol did not measurably degrade the serial link throughput.

During data transmission, the protocol overhead consists of 3 bytes per transmitted packet: two bytes of cyclic redundancy code, and one handshake byte. The control packet preceding each transfer of data represents an additional 15 byte overhead. Assuming that the link is used to send 1024 16-bit words of data in packets of 256 words each (these are typical values used in the system), then the link efficiency will approach 98.7%. The protocol therefore has negligible effect on link performance when the link is error free.

The serial link does represent the major bottleneck in the system, however, when real-time performance is required. The sample-transfer-display cycle for real-time instrument simulation can benefit from improvements in each of those separate tasks.

The serial link may disappear in the future in favour of a high-speed parallel link, which would give up to 20 times the throughput.

## 7.2 Experience with the GI in a Teaching Laboratory

Six Generalised Instruments were built and used by students in the undergraduate laboratories at the University of Adelaide. The units worked well, but deficiencies were observed. These fell into the following categories:

- Slow response.

The PC-host computers used were XT-clones with processor speeds of 10MHz, and without a numeric coprocessor. The refreshing of the display screen with new waveform data was slow.

---

<sup>1</sup>host is a 80386/33MHz based machine

- aliasing in the A/D process.

The prototype analogue interface modules were built without antialiasing filters. It was therefore necessary for the user to choose sampling rates which would not produce aliasing. Such a situation is not acceptable in the general sense, but it did prove to be educational for the students.

- Input voltage range.

The limited input voltage range of the prototype analogue interface modules ( $\pm 10V$ ) required users to avoid clipping. The input circuitry of the GI features programmable gain but not (at this time) programmable attenuation. Programmable attenuation will be added to future models.

A disturbing observation was that students were less likely to question the results of a measurement presented to them by a computer, compared with other test instruments. One possible reason for this is that normal test instruments have front panel controls which may be changed (twiddled) to provide the desired display. During this 'twiddling' the user is able to judge the consistency of what she is seeing, and to judge whether or not the display is valid.

The time lags in the GI measurement process cause the user to be less inclined to change settings, since the process may take some seconds and will not necessarily produce a desirable result.

This psychological factor in the use of the GI will be tackled by improving the response time of the whole system, and increasing the amount of control available to the user.

### 7.3 Areas for Future Progress

The development and use of the GI has revealed areas where improvements may be made:

- **the host to GI communication link.**

The RS232 link which links the GI with the host computer has too low a throughput to permit rapid updating of displayed waveforms.

A faster interface will be implemented using the parallel port on the control processor module. This will allow transfer rates at least an order of magnitude faster than those currently attained. The disadvantage will be that a parallel interface card will need to be designed for the host computer.

- **the host to GI communication protocol.**

The communication protocol between the host and GI allows the host to fully control the behaviour of the signal processor, and other circuitry within the GI. This was desirable during the development of the GI, but is now a limiting factor.

All events in the GI must currently be initiated by the host computer. For instance, to retrieve the results of a computation by the DSP, the host commands the GI to notify the host when the computation is complete, and another command to upload the data. Until the data arrives, the host is idle.

A more efficient method would be for the signal processor to control the dynamics of the communication, by simply sending the data to the host when complete. The signal processor could then commence another calculation without being explicitly ordered to do so by the host. The host would not need to be idle during this time; it would be processing commands from the user or updating the display.

Such a protocol would be very different to the current implementation, though it may not appear to be so. In the current system, the host acts as the master processor, while the signal processor is the slave. In the proposed protocol, the host and signal processor would be peers on a network. This network could link more than one signal processor. Communication within the network would take place using message routed between processors.

The GI control processor would act as a central exchange where messages arrive and are routed on toward their destinations. Possible destinations for a message would be signal processors on a common backplane, or the host computer (via the RS232 link, or a faster link).

The `upload()` function, which is currently used to retrieve data from the signal processor, would be made obsolete. Packets of data would arrive invisibly and be assigned to specified variables (whose names would be contained in the packets) in the host's memory.

- **the host's graphical interface.**

The present graphical interface lacks the ability to manage overlapping windows. Consequently, only the area of the display screen is available for graphic rendering. By writing new display software which is capable of managing overlapping windows, the total display area can be made greater than the area of the screen, since windows could be stacked and overlapped, placing those of immediate interest at the top of the pile.

The majority of functions of the present interface are not immediately visible on the display screen eg menus which require a mouse button to be depressed before appearing.

This creates an uncluttered screen, but has the disadvantage making plain all of the *possible* functions. An instrument such as an oscilloscope has all of its functions clearly visible on the front panel. This often produces a cluttered control panel, but the presence of all controls provides cues to the operator as she adjusts the instrument to achieve the desired display.

By increasing the overall display area (with overlapping graphic windows), more controls can be placed onscreen.

- **the GI's analogue interface.**

The prototype Analogue Interface Modules suffered from:

1. the lack of antialiasing filters.

Strategies for implementation of antialiasing filters were discussed in section 3.2.5.

2. the lack of programmable signal attenuation.

The current AIM has programmable gains of unity and greater. In many cases the user will wish to sample signals approaching or exceeding the specified  $\pm 10V$  input range.

3. the lack of autocalibration for input signals.

The prototype AIM suffered from drift of DC levels within its analogue circuitry. Provision was made to trim both the gain and offsets at the converter. Unfortunately, the offset voltage at the A/D is not constant, since it undergoes amplification in the programmable gain amplifiers along with the signal.

A future AIM would incorporate auto-calibration circuitry with the ability to perform the following:

- switch the input to an AC reference voltage of known amplitude.
- either trim the input stage gain, or store the digitised amplitude. The latter method will allow subsequent sample values to be scaled back to a voltage, thus correcting for linear effects within the circuit.
- switch the input to the analogue ground, and adjust the offset voltage until a reading of zero is obtained from the A/D converter. In this way all voltage offsets up to the ADC input can be cancelled.

- **the creation of a TMS320 code generator for the SPaM compiler.**

At present, the TMS320 code and SPaM script are written separately. The SPaM script is compiled and executed by the host. The script controls the execution of the (separate) signal processor code.

An alternative to this scheme is to compile the SPaM script directly into TMS320 code. Instead of executing SPaM code itself, the host would simply act as a terminal and resource server for the TMS320.

Such a compiler would eliminate the need to separately code TMS320 programs in assembly language, and would allow direct programming of the TMS320 in a high level language (SPaM).

As it is currently implemented, SPaM includes a number of code libraries which would need to be rewritten for the TMS320 (eg the Math Function and Primitive Math Operation libraries.) The size of the resulting code would very likely exceed the 64k-word address space of the TMS320C25. A SPaM compiler would be more suitable for the TMS320C30, which has the advantage of a greater address space and hardware support for floating point calculations.

## Chapter 8

# Conclusion

The GI has been used successfully in the laboratory as both a research and teaching tool. The integration of multiple software and hardware modules into one system has been achieved, and the seams are invisible in the resulting system.

This work has addressed the issues relevant to programmable instrumentation systems, which are now appearing in the marketplace. The Generalised Instrument software and digital hardware has been verified and used successfully.

The issue of the analogue front-end has been discussed, illustrating the many obstacles that must be overcome in that part of the system. At the time of writing, the A/D-D/A module in existence was an 8-bit system lacking many of the features discussed in section 3.2.5. Considerable future work will be devoted to this area of the system. Techniques such as multirate digital signal processing may need to be used if the system is to achieve a wide dynamic range of frequency. By transferring the (anti-aliasing) operations into the digital domain, the cost of exotic analogue systems is spared (for this reason multirate techniques are widely used in commercial digital audio equipment.)

A great advance in was made in the construction of the SPaM program for control of the GI. Previous software, based on command driven software packages such as Sigproc, were limited in their capabilities. The SPaM software, with its embedded algebraic language compiler, promises rapid development of new applications by both the author and new users.

Unlike common commercial software, which is designed to implement specific signal processing operations, with or without DSP hardware, SPaM provides a general control and display environment within which customised signal processing applications can be created. Since the signal processing carried out by the hardware is external to SPaM, the user can alter the performance of the system by changing the signal processor code which is executed.

The SPaM program provides flexibility on several levels, including the DSP code itself, the SPaM script which determines the behaviour of the host, and SPaM's interactive support which allows the user to perform new operations on the data which may not have been allowed for when the program script was written.

One area which was touched on during this project was that of automated software generation tools. Such tools allow software to be generated automatically from a specification of the functioning of that software (as *yacc* generates a parser from a language specification file). Such tools are currently available for filter generation (FIR/IIR), but new tools will be created which allow all phases of the DSP process to be specified and the corresponding code generated. Such tools will be valuable, since a great deal of time is spent writing and testing low-level signal processing software, usually in assembly language.

Though based on the TMS320C25, there is no reason why the GI cannot use newer signal processor devices as they emerge. The partitioning of the system means that typically only the DSP module and the DSP code libraries will need modification. The recent emergence of low cost floating point signal processors (eg TMS320C30) will significantly broaden the usefulness of the GI not only as a signal processing system, but as a numeric processing system in general.

The modularity of the GI hardware lends itself to the implementation of multiple processor signal processing systems. Such systems would fall into one of three broad categories:

- Systems in which multiple signal processors are responsible for processing distinct sets of inputs and outputs, with either low bandwidth or no communication between processors. An example of this is real-time filtering of multiple signals, and spectral analysis of multiple, separate signals.

The characteristic of such a system is the absence of communication between signal processors on separate modules within the GI.

SPaM directly supports such a system, which is functionally identical to a single DSP system repeated  $N$  times.

- Fixed topology multi-processors. In these systems the multiple signal processors are interconnected in a fixed, predetermined topology which has been chosen to suit the application. For higher performance, the interconnections should be made with hardware. Software connection is possible and is discussed in the next item.

SPaM is compatible with such systems, as the system is functionally similar to that discussed in item 1, except that sample data flows will in many cases lead to other signal processors rather than analogue signals (via A/D, D/A).

- Systems in which a set of inputs feeds into a digital signal processing system consisting of  $N$  processors. The processors will be arranged in some sort of topology (either physically or logically) to provide greater throughput than could be achieved using a single processor. This topology may be a pipeline, or processors working on a partitioned data set.

The GI lends itself directly to logically connected signal processors, as data can be moved in preprogrammed ways across the backplane between DSP modules. Physical topologies may be created, but are expensive and give little or no flexibility.

Managing a multiprocessor DSP system with programmable topology requires sophisticated control, something which neither SPaM nor the GI control processor provide at present. An example of the way in which such a system may be built is to use the control processor as a routing switch for data flowing from one DSP module to another.

Some recent developments in parallel software engineering, such as the Linda language promise parallel processing software which is easy to understand and combine with current programming languages, for example the C language. Replacing rigid processor scheduling at compile-time by dynamic scheduling at run-time (as performed by systems such as Linda) is desirable, since the result is more efficient use of a multiprocessor system under varying loads.

To practice the science of digital signal processing it is necessary to have a hardware and software system which is suitable for use in research (and development), and which can then be used in implementation with as few changes as possible.

The Generalised Instrument is such a system. By being separate from its host, it can be used with any host (providing that the software is available). The inclusion of a built-in language in SPaM allows the rapid prototyping and development of virtual instruments without extensive programming.



# Bibliography

- [1] Texas Instruments, *TMS320C25 Users Guide*, December 1987.
- [2] Texas Instruments, May 8 1987. *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide*
- [3] Texas Instruments, May 14 1987 *TMS320C25 C Compiler Reference Guide*
- [4] G.Y. Yuan. *Transfer Function Analysis Using Correlation Technique*, Dept. Electronic & Electrical Engineering, University of Adelaide, 1989
- [5] R. Lane, G. Vokalek, *Sigproc Manual*, Dept. Electrical & Electronic Engineering, University of Adelaide, 1989.
- [6] Brian W. Kernighan, Rob Pike, *The Unix Programming Environment*, Prentice Hall 1984.
- [7] Ronald E. Crochiere, Lawrence R. Rabiner *Multirate Digital Signal Processing*, Prentice Hall 1983
- [8] Stephen C. Johnson, *Yacc: Yet Another Compiler-Compiler*, document in electronic form, origin unknown.
- [9] Arthur B. Williams, Fred J. Taylor *Electronic Filter Design Handbook*, 2nd Edition, McGraw Hill.
- [10] Motorola Inc., *MC68681 Dual Asynchronous Receiver/Transmitter*, Sept. 1985

## Appendix A

# The Generalised Instrument Hardware Design

This document describes, in detail, the hardware design of the Generalised Instrument, a programmable signal processing system developed for research and teaching in the Department of Electronic & Electrical Engineering, University of Adelaide.

At the time of writing, the GI control processor, and signal processor had been finalised and verified. A prototype acquisition board was working, which is described herein.

The organisation of the first three sections is as a set of drawings. All of the drawings in a given section together form the design for one circuit board module.

Section A.1 describes the 68000 control processor board which controls the GI system as a whole, by forming an intelligent link between the host computer and the GI backplane.

Section A.2 describes the TMS320C25-based Digital Signal Processor module.

section A.3 describes the design of a prototype analogue-interface module. This module is not a final production design, and has certain deficiencies discussed in that section.

Section A.4 discusses design considerations for those readers planning on building custom modules for the GI.

### A.1 The 68000 Control Processor Module

This section contains the drawings which comprise the design of the control processor module of the GI. The control processor is essentially a self-contained 68000 board, with the following features:

1. Dual RS232 ports.
2. Dual unidirectional 8-bit parallel ports (1-input, 1- output).
3. SCSI port.
4. 64/128kB EPROM
5. 64kB RAM.

- 6. Bus timeout & inactivity monitors.
- 7. Backplane interface.

The drawings which are included in this section are summarised in table A.1.

<i>Drawing</i>	<i>Contents</i>
CON-0000	68000 CPU CPU Clock Power on reset CPU single step circuit
CON-0001	EPROM RAM EPROM/RAM decoding
CON-0002	DUART RS232 Transceivers
CON-0003	Parallel port transceivers Handshaking PAL
CON-0004	Wait state generator PAL
CON-0005	Bus timeout detector Inactivity monitor
CON-0006	Interrupt generator
CON-0007	Empty
CON-0008	Backplane bus interface
CON-0009	Control processor onboard decoding Memory Map
CON-0010	Backplane bus attention request circuit
CON-0011	Empty
CON-0012	Board control register
CON-0013	Board status register
CON-0014	SCSI Interface

Table A.1: Drawing index for Control Processor Module

### A.1.1 Control Processor Module Specifications

Processor	68000
Processor clock rate	any of 8, 10, 12, 16MHz. Set by replacing crystal oscillator.
Onboard EPROM	64k-bytes or 128k-bytes.
Onboard RAM	64k-bytes
Onboard Interfaces	2 RS232 interfaces 2 unidirectional 8-bit parallel 1 SCSI 1 backplane interface
Other	Bus activity timer to detect $\overline{DTACK}$ timeout Bus inactivity timer to detect 68000 loss of control
Card Format	220mm×100mm Eurocard format
Backplane Connector	DIN41612 (proprietary bus)

Table A.2: Control processor module specifications

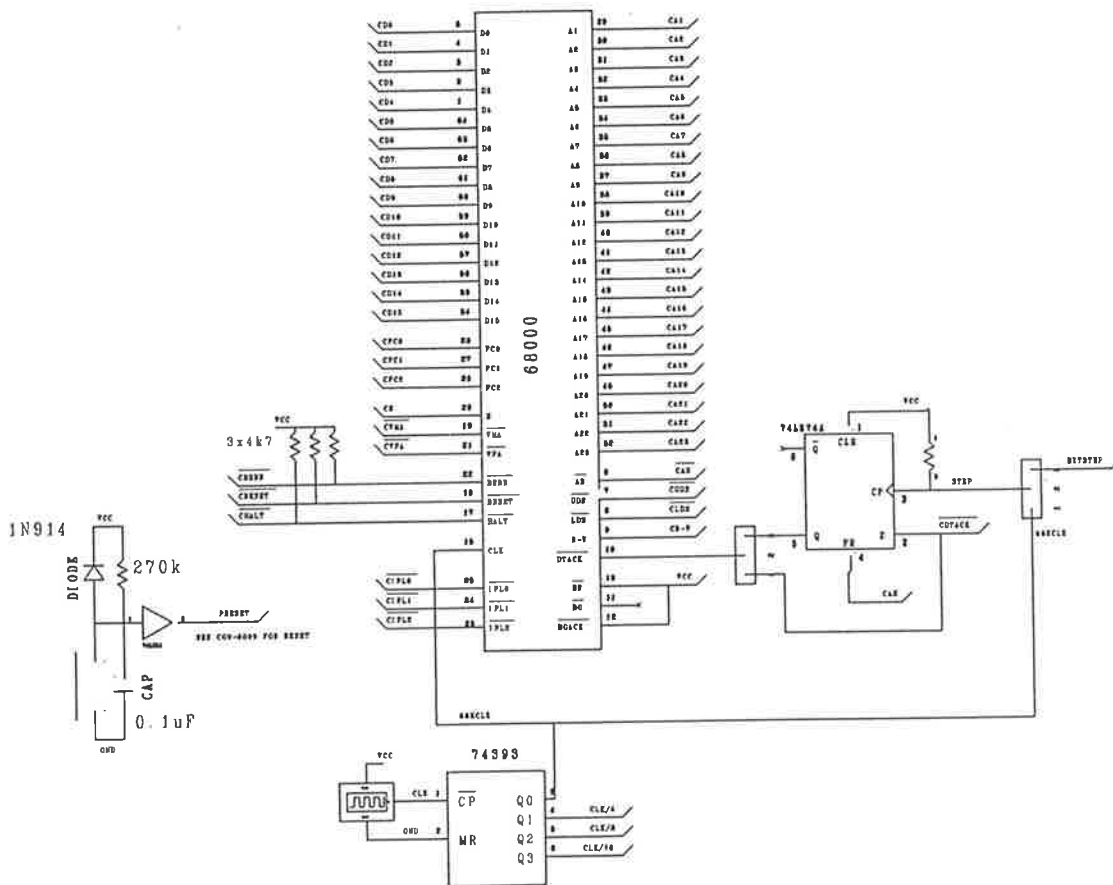
## A.1.2 CON-0000

### CPU AND CLOCK

*Notes:*

1. CPU clock rates up to 16MHz have been successfully used, though the speed of EPROMs and SRAMs used on the board will mean that wait-states may need to be inserted (see CON-0004). Backplane modules must generate their own wait states if needed.
2. The clock is provided by a TTL oscillator module which supplies 2×clock, and may be easily changed by using a different oscillator.
3. A single step circuit can be selected by jumper (68KJMP1) to hold off  $\overline{DTACK}$  to the 68000 until the user depresses a pushbutton connected to the pins called EXTSTEP.

This has the effect of extending the bus access indefinitely (until the switch is pressed), so that control, address, and data lines can be examined with a logic probe/analyser.



### A.1.3 CON-0001

#### EPROM and SRAM

*Notes:*

1. 2 EPROMs are catered for. They may be 256kbit or 512kbit devices, with a jumper (CJUMP1) determining type. The EPROMs must be identical, and both must be present simultaneously (the 68000 is a 16-bit processor).

CJUMP1 omitted	256kbit EPROMs
CJUMP1 inserted	512kbit EPROMs

Table A.3: Setting jumpers for EPROM capacity

2. 2 SRAMs are catered for. They may be 64kbit or 256kbit devices, with the jumper CJUMP2 determining which.

CJUMP2 omitted	64kbit SRAMs
CJUMP2 inserted	256kbit SRAMs

Table A.4: Setting jumpers for RAM capacity

3. Wait states may be generated separately for EPROMs and SRAMs, as indicated in drawing CON-0004 on page 156.

EPROM	256kbit devices	\$000000-\$00FFFF
	512kbit devices	\$000000-\$01FFFF
SRAM	64kbit devices	\$040000-\$043FFF
	256kbit devices	\$040000-\$04FFFF

Table A.5: EPROM & SRAM Address Map

```
Name      ramdec1;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p16L8;
Format    j;
```

```
/* SHEET CON-0001 */
/* eprom and sram decoder pal */
```

```
Pin 1 = CJUMP1; /* if CJUMP1=1 then 27256, else 27512 */
Pin 2 = CJUMP2; /* if CJUMP2=1 then 8kx8 SRAM, else 32kx8 sram */
Pin 3 = CA13;
Pin 4 = CA15;
Pin 5 = CA16;
Pin 6 = CA17;
Pin 7 = CA18;
Pin 8 = !OFFBOARD;
Pin 9 = !CAS;
Pin 11 = CA14;
```

```
Pin 19 = EPP1; /* pin 1 of EPROM */
Pin 18 = !EPCEO;
Pin 17 = !EPCE1;
Pin 16 = !SRCEO;
Pin 15 = !SRCE1;
Pin 14 = !EPROM;
Pin 13 = !SRAM;
Pin 12 = SRAMP26; /* pin 26 of SRAM socket */
```

```
$DEFINE ONBOARD !OFFBOARD
```

```
EPP1 = CA16 # CJUMP1;
```

```
SRAMP26 = CA14 # CJUMP2;
```

```
EPCEO = ONBOARD & CAS & !CA18 & !CA17 & !CA16 & CJUMP1 #
        ONBOARD & CAS & !CA18 & !CA17 & !CJUMP1;
```

```
EPCE1 = ONBOARD & CAS & !CA18 & !CA17 & CA16 & CJUMP1 #
        ONBOARD & CAS & !CA18 & CA17 & !CJUMP1;
```

```
SRCEO = ONBOARD & CAS & CA18 & !CA17 & !CA16 & !CA15 & !CA14 & CJUMP2 #
        ONBOARD & CAS & CA18 & !CA17 & !CA16 & !CJUMP2;
```

```
SRCE1 = ONBOARD & CAS & CA18 & !CA17 & !CA16 & !CA15 & CA14 & CJUMP2 #
        ONBOARD & CAS & CA18 & !CA17 & CA16 & !CJUMP2;
```

```
EPROM = ONBOARD & CAS & !CA18 & !CA17 & !CA16 & CJUMP1 #
        ONBOARD & CAS & !CA18 & !CA17 & !CJUMP1 #
        ONBOARD & CAS & !CA18 & !CA17 & CA16 & CJUMP1 #
        ONBOARD & CAS & !CA18 & CA17 & !CJUMP1;
```

```
SRAM = ONBOARD & CAS & CA18 & !CA17 & !CA16 & !CA15 & !CA14 & CJUMP2 #
        ONBOARD & CAS & CA18 & !CA17 & !CA16 & !CJUMP2 #
        ONBOARD & CAS & CA18 & !CA17 & !CA16 & !CA15 & CA14 & CJUMP2 #
        ONBOARD & CAS & CA18 & !CA17 & CA16 & !CJUMP2;
```

```

Name      ramdec2;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p16L8;
Format    j;

```

```

/* SHEET CON-0001 */
/* GENERATES MISC WR' AND RD' STROBES. */

```

```

Pin 1 = CRW;
Pin 2 = !CUDS;
Pin 3 = !CLDS;

```

```

Pin 19 = !CHIOE;
Pin 18 = !CHIWE;
Pin 17 = !CLOOE;
Pin 12 = !CLOWE;

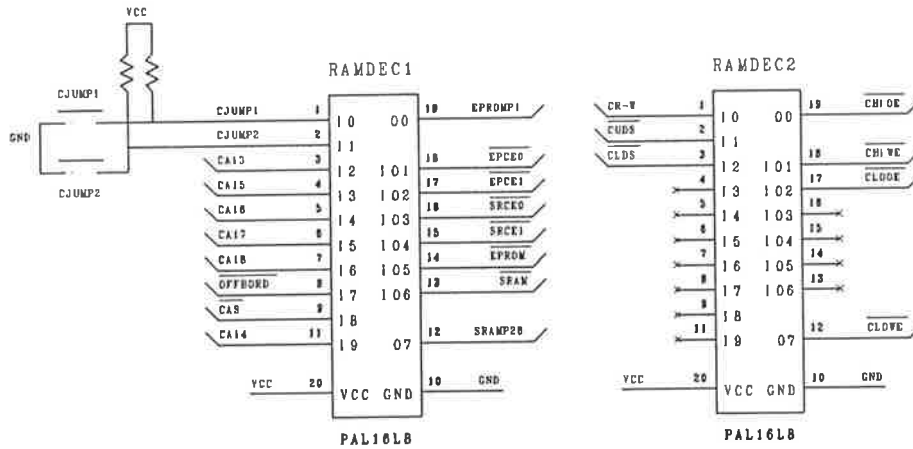
```

CHIOE = CUDS & CRW;

CLOOE = CLDS & CRW;

CHIWE = CUDS & !CRW;

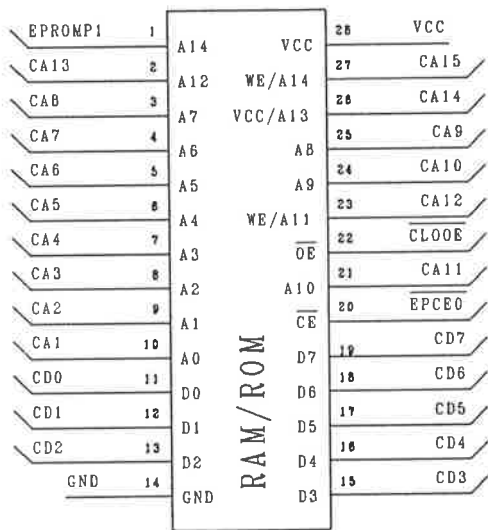
CLOWE = CLDS & !CRW;



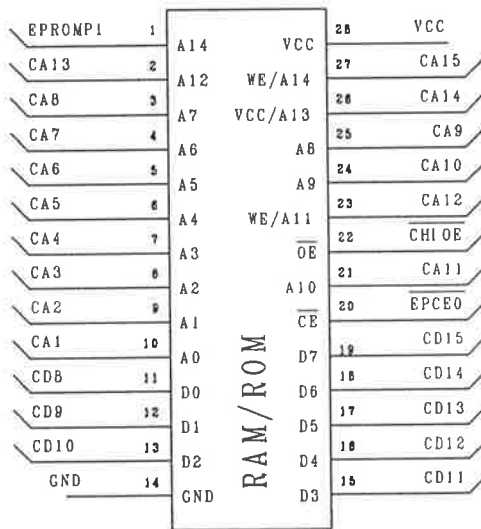


epr

### EPROML

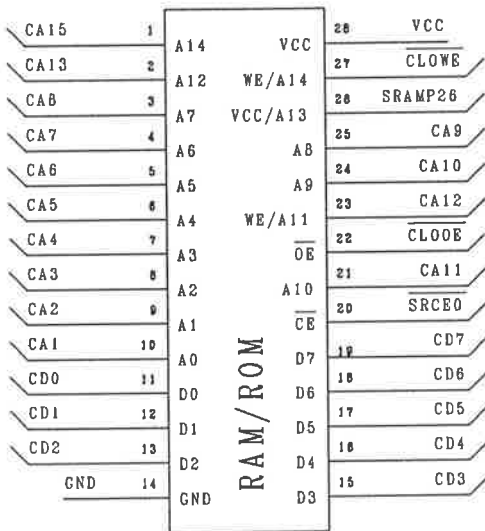


### EPROMH

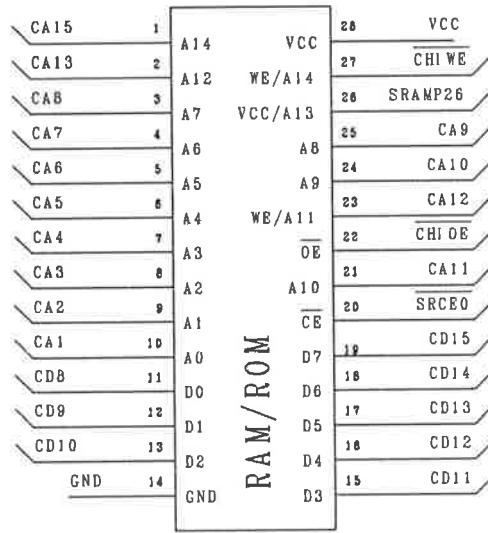


sra

### SRAML



### SRAMH



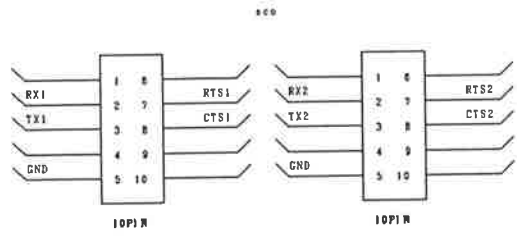
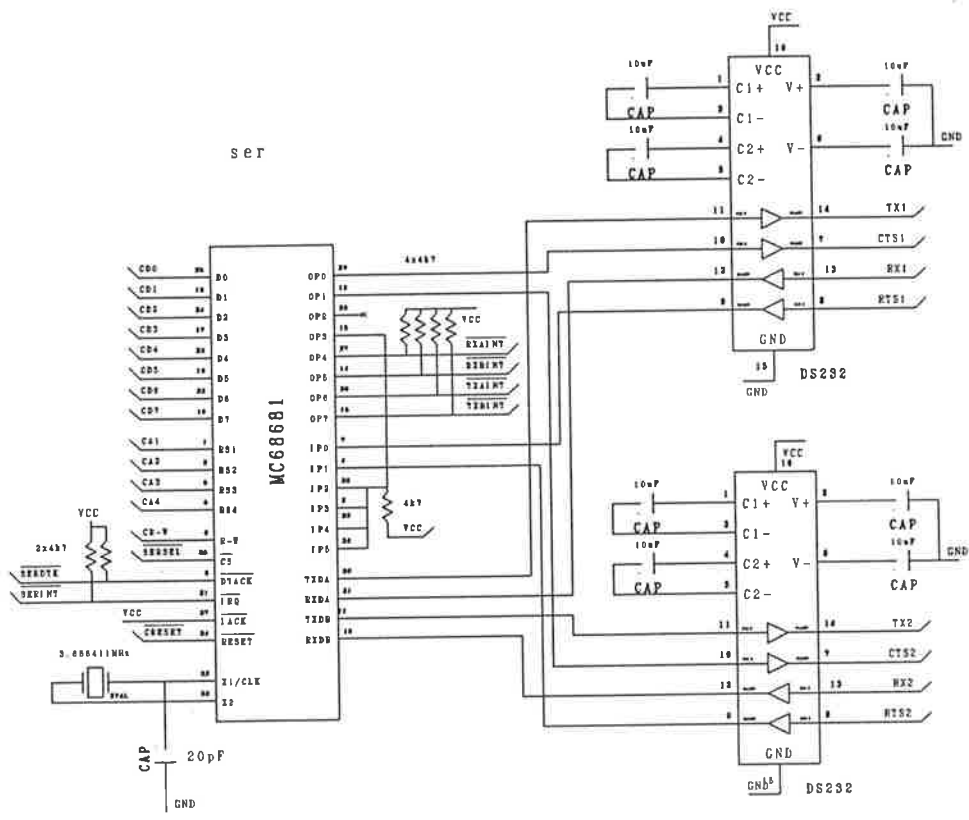
## A.1.4 CON-0002

### Serial Ports

#### *Notes:*

1. There are 2 serial ports provided by the dual UART.
2. The port transceivers are MAX232 type devices, which internally generate the  $\pm 10V$  RS232 rails.
3. Connection to the serial ports is via pin strips to which ribbon cable and header can be connected.
4. The serial port may generate up an interrupt on several conditions (see [10]), including
  - Transmit register empty (2 channels).
  - Receive register empty (2 channels).
  - Break signal detect (2 channels).

To aid in finding the cause of interrupts, the TXBINT, TXAINT, RXBINT, and RXAINT signals may be read directly through the board status register (CON-0013).



## A.1.5 CON-0003

### Parallel Port

The parallel port system consists of two separate parallel port. One is an 8-bit output port with handshaking, the other is an 8-bit input port with handshaking.

The parallel port(s) were added to facilitate a fast parallel interface to the host computer, to take the place of the RS232 link at some time in the future.

#### *Notes:*

1. The state of the transmit and receive port may be separately determined by reading bits in the status register (see CON-0013).
2. The board control register (CON-0014) may be programmed to allow either the receive port or the transmit port (or both) to generate a hardware interrupt (PARINT).  
For the transmit port, the interrupt is asserted when the host acknowledges the last transmission, thus freeing the port for the next one.  
The receive port can generate an interrupt when a byte is received from the host.
3. Terminator sites are provided around the pinstrip connector, allowing terminator resistor packs to be used for impedance matching of long cables.

```
Name      PARPORT;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p16L8;
Format    j;
```

```
/* SHEET CON-0003 */
```

```
/* This PAL implements the handshaking for the parallel port */
```

```
Pin 1 = !LPARRD;
Pin 2 = !LPARWR;
Pin 3 = !RLACK;
Pin 4 = !RLDATA;
Pin 5 = PARIREN;
Pin 6 = PARIWEN;
```

```
Pin 8 = TIMEOUT; /* from CON-0005, this signal indicates DTACK' timeouts */
Pin 9 = !STATUSR; /* from decoder, CON-0009 */
Pin 11 = STATUS; /* from status register, CON-0013 */
```

```
Pin 19 = !LRDATA;
Pin 18 = LWFULL;
Pin 17 = !RLACKB;
Pin 16 = !LRACK;
Pin 15 = !PARINT;
Pin 14 = LRFULL;
Pin 13 = !CBERR;
Pin 12 = CD15;
```

```
LWFULL = LPARWR # /* local write port is full from the time we */
(LWFULL & !RLACKB); /* write a value to it, to the time its acknowledged */
```

```
LRFULL = RLDATA; /* local read port is full when the incoming
RLDATA strobe is asserted */
```

```
LRDATA = LWFULL; /* signal to tell remote end that data is waiting
for it, just a buffered version of LWFULL */
```

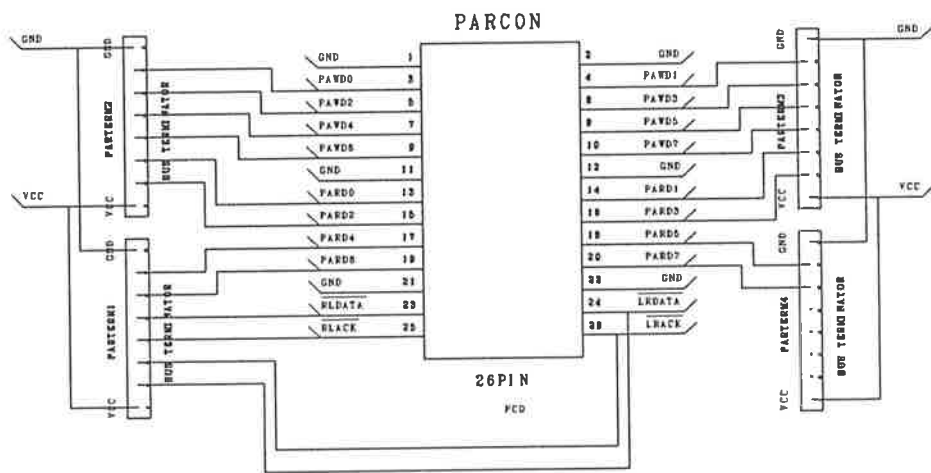
```
LRACK = LPARRD; /* acknowledge to other end consists of this end
reading the incoming data port. */
```

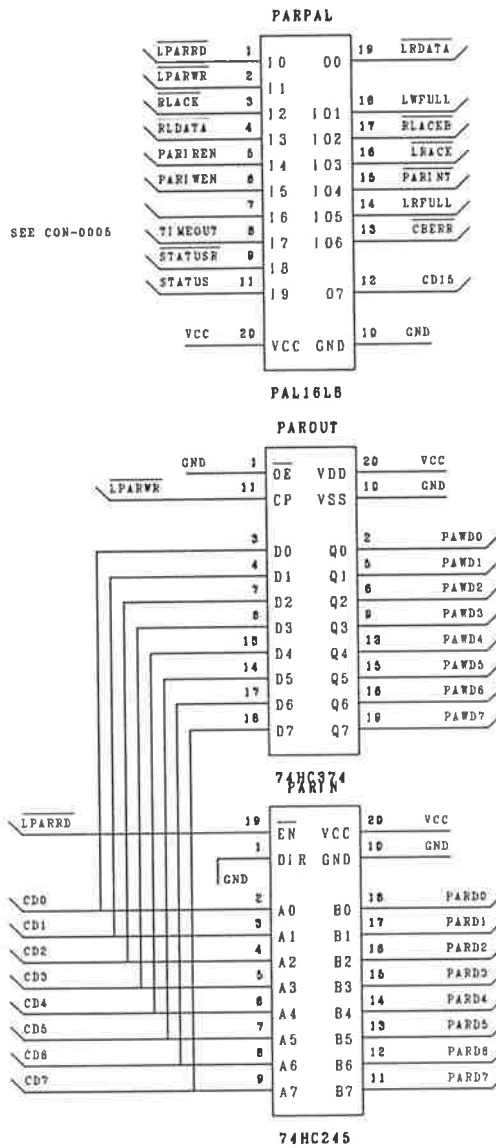
```
RLACKB = RLACK; /* a buffered version of RLACK */
```

```
PARINT = LRFULL & PARIREN # /* we get an interrupt when our read port is full */
RLACKB & PARIWEN # /* or other end has just ACKed our last xmission */
PARINT & !(LPARWR # LPARRD); /* make this interrupt hang around until
something happens */
```

```
CD15 = STATUS;
CD15.OE = STATUSR; /* this provides the tristate buffer for the
board status register */
```

```
CBERR = 'b'1;
CBERR.OE = TIMEOUT;
```







## A.1.6 CON-0004

### Wait State Generation

*Notes:*

1.  $\overline{DTACK}$  must be generated for EPROM and SRAM in a manner which allows jumpers to determine the number of wait states added.
2. Onboard devices other than EPROM and RAM are to be run with zero wait states.
3. Offboard devices may generate their own wait states. No  $\overline{DTACK}$  is generated by the control module for devices outside of its immediate address space (\$000000- \$7FFFFFFF).

Should the 68000 attempt to access an address for which no  $\overline{DTACK}$  is generated at all, the bus activity timer will detect this condition and generate a bus-error exception (see CON-0005.)

Jumper SPEED0	Jumper SPEED1	EPROM WAIT STATES	SRAM WAIT STATES
inserted	inserted	0	0
omitted	inserted	1	0
inserted	omitted	2	0
omitted	omitted	3	1

Table A.6: Jumpers for wait state control

```
Name      dtack;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    g16v8;
Format    j;
```

```
/* SHEET CON-0004 */
/* DTACK generator for 68000 */
```

```
/* SPEED0 and SPEED1 have the following meanings:
```

```
   SPEED1,SPEED0 = 0,0  clock rate = 8MHz
   SPEED1,SPEED0 = 0,1  clock rate = 10MHz
   SPEED1,SPEED0 = 1,0  clock rate = 12MHz
   SPEED1,SPEED0 = 1,1  clock rate = 16MHz
```

```
*/
```

```
Pin 1 = 68KCLK; /* ref CON-0000 */
Pin 2 = !EPROM; /* ref CON-0001 */
Pin 3 = !SRAM; /* ref CON-0001 */
Pin 4 = SPEED0;
Pin 5 = SPEED1;
Pin 6 = !DEVDTK;
Pin 7 = !OFFBOARD;
Pin 8 = !CAS;
Pin 9 = !BDTACK;
/* Pin 11 = GND, !OE of registered outputs */
```

```
Pin 19 = !DTACKE; /* from eproms */
Pin 18 = !DTACKS; /* from SRAM */
Pin 13 = !DTACKB; /* from backplane */
Pin 12 = !DTACKD; /* from devices */
```

```
Pin 17 = !CAS1; /* !CAS delayed by 1 clock */
Pin 16 = !CAS2; /* !CAS delayed by 2 clock */
Pin 15 = !CAS3; /* !CAS delayed by 3 clock */
Pin 14 = !CAS4; /* !CAS delayed by 4 clock */
```

```
CAS1.D = CAS;
CAS2.D = CAS1 & CAS; /* make sure !CAS2 rises soon after !CAS */
CAS3.D = CAS2 & CAS;
CAS4.D = CAS3 & CAS;
```

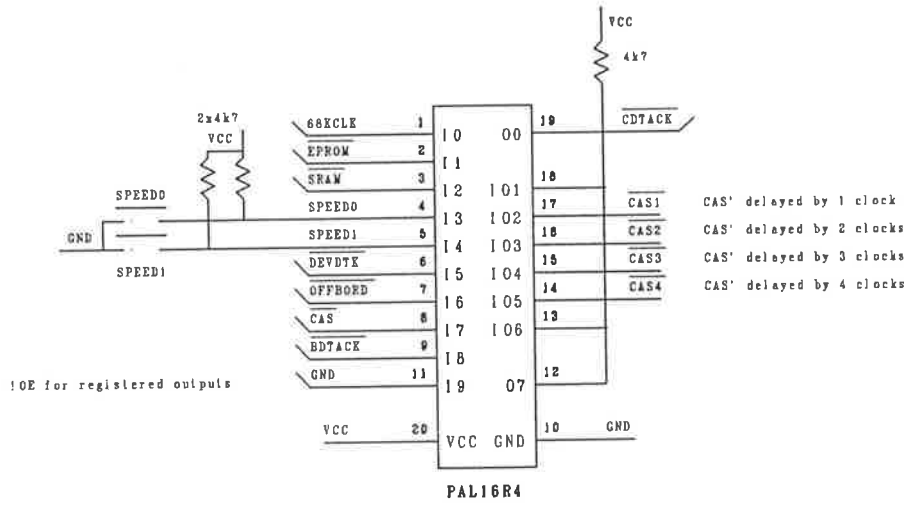
```
DTACKE.OE = EPROM;
DTACKE = EPROM & !SPEED0 & !SPEED1 #
        EPROM & SPEED0 & !SPEED1 & CAS2 #
        EPROM & !SPEED0 & SPEED1 & CAS3 #
        EPROM & SPEED0 & SPEED1 & CAS4 ;
```

```
DTACKS.OE = SRAM;
DTACKS = SRAM & SPEED0 & SPEED1 & CAS2 #
        SRAM & !SPEED0 #
        SRAM & !SPEED1 ;
```

```
DTACKB.OE = OFFBOARD;
DTACKB = OFFBOARD & !SPEED0 & !SPEED1 & BDTACK #
        OFFBOARD & SPEED0 & !SPEED1 & BDTACK & CAS2 #
        OFFBOARD & !SPEED0 & SPEED1 & BDTACK & CAS3 #
```

OFFBOARD & SPEED0 & SPEED1 & BDTACK & CAS4;

DTACKD.OE = DEVDTK;  
 DTACKD = 'b'1;



## A.1.7 CON-0005

### Bus Activity Monitor, Inactivity Monitor, Time Interrupt

This circuit consists of three parts.

1. Bus activity monitor.

This circuit measures the length of 68000 bus cycles. Should a bus cycle last longer than 64 master clock cycles, this circuit will assert the  $\overline{\text{BERR}}$  signal to initiate exception processing. Such a timeout will occur if the 68000 attempts to read non-decoded memory locations.

2. Bus inactivity monitor.

This circuit measures the time between successive bus cycles as initiated by the 68000. If this time exceeds 2048 master clock cycles, this circuit will generate a  $\overline{\text{RESET}}$  signal to the 68000, causing it to reboot.

It is assumed that such an extended period of non-processing can be caused only by system faults which cause the 68000 to halt program execution.

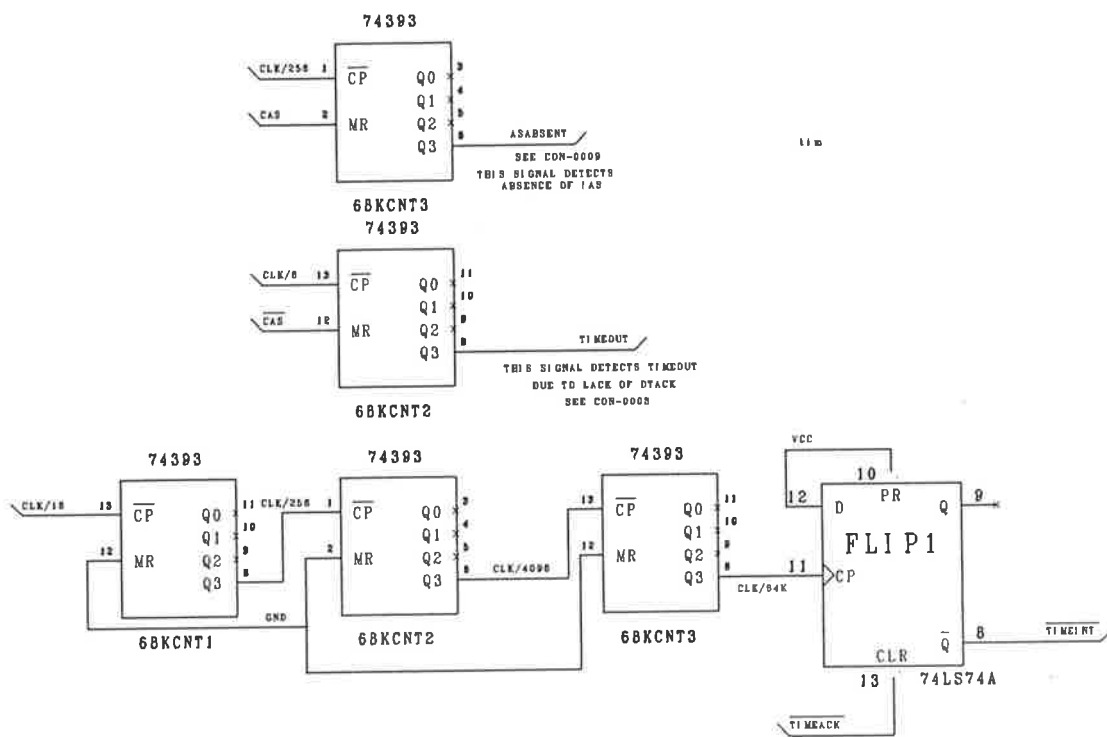
3. Time Interrupt.

This circuit generates real-time interrupts to the 68000 at a rate of one every 65536 master clock cycles. At a master clock frequency of 32MHz (CPU clock = 16MHz), this corresponds to an interrupt rate of 488Hz.

The 68000 must acknowledge each time interrupt before the next one can occur. However, timing accuracy is not compromised by the time taken to respond to the time interrupt.

Master Oscillator	12MHz	16MHz	20MHz	24MHz	32MHz
CPU clock	6MHz	8MHz	10MHz	12MHz	16MHz
Activity timeout	5.3 $\mu$ s	4 $\mu$ s	3.2 $\mu$ s	2.7 $\mu$ s	2 $\mu$ s
Inactivity timeout	171 $\mu$ s	128 $\mu$ s	102.4 $\mu$ s	85.3 $\mu$ s	64 $\mu$ s
Time Interrupt period	5461.3 $\mu$ s	4096 $\mu$ s	3277 $\mu$ s	2731 $\mu$ s	2048 $\mu$ s

Table A.7: System time intervals for various master clock oscillator frequencies.



## A.1.8 CON-0006

### Interrupt Generator

The interrupt generator generates the prioritised interrupt signals for the 68000 from both onboard interrupt request signals, and the backplane attention request signals.

The sources of interrupts are:

1. Serial Port receive register full, and transmit register empty.
2. SCSI interface data transfer.
3. Time interrupt.
4. Parallel port receive register full, and transmit register empty.
5. Backplane attention request.

Two PALs implement the interrupt system. The PAL called 68KINT is responsible for generating the prioritised interrupt signals  $\overline{IPL0}$  ...  $\overline{IPL2}$ . The PAL called VECTOR generates 8-bit interrupt vector numbers.

The interrupts from the control processor onboard devices are autovectored according to priority. The large number of offboard interrupts (attention requests) possible, demands a more efficient treatment to prevent the waste of CPU time in polling boards. The VECTOR PAL allows each of the attention request signals to have its own interrupt vector, by generating the appropriate vector during the interrupt acknowledge phase of the 68000 exception processing cycle.

```
Name      68kint;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p20L10;
Format    j;
```

```
/* SHEET CON-0006 */
/* Interrupt encoder for 68000 control processor on Siglab */
```

```
Pin 1 = !ATN0;
Pin 2 = !ATN1;
Pin 3 = !ATN2;
Pin 4 = !ATN3;
Pin 5 = !ATN4;
Pin 6 = !ATN5;
Pin 7 = !ATN6;
Pin 8 = !ATN7;
Pin 9 = !CAS;
Pin 10 = CFC0;
Pin 11 = CFC1;
Pin 13 = CFC2;
Pin 15 = SCSIRQ;
Pin 16 = !SERINT;
Pin 17 = !PARINT;
Pin 18 = !TIMEINT;
Pin 19 = ATTENTION;
```

```
Pin 23 = !CIPL0;
Pin 22 = !CIPL1;
Pin 14 = !CIPL2;
Pin 21 = !VECIACK;
Pin 20 = !CVPA;
```

```
FIELD INTLEVEL = [CIPL2..CIPL0];
```

```
CIPL0.OE = 'b'1;
CIPL1.OE = 'b'1;
CIPL2.OE = 'b'1;
```

```
$DEFINE ACHTUNG ATN0 # ATN1 # ATN2 # ATN3 # ATN4 # ATN5 # ATN6 # ATN7
```

```
ATTENTION = ACHTUNG;
```

```
INTLEVEL = ['b'0,'b'0,'b'1] & TIMEINT #
            ['b'0,'b'1,'b'0] & PARINT #
            ['b'0,'b'1,'b'1] & SERINT #
            ['b'1,'b'0,'b'0] & SCSIRQ #
            ['b'1,'b'0,'b'1] & ATTENTION;
```

```
VECIACK = CFC0 & CFC1 & CFC2 & CAS;
```

```
CVPA = CFC0 & CFC1 & CFC2 & CAS;
```

```

Name      VECTOR;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p22v10;
Format    j;

```

```

/* SHEET CON-0006 */
/* Interrupt vector generator */

```

```

Pin 1 = !ATN0;
Pin 2 = !ATN1;
Pin 3 = !ATN2;
Pin 4 = !ATN3;
Pin 5 = !ATN4;
Pin 6 = !ATN5;
Pin 7 = !ATN6;
Pin 8 = !ATN7;
Pin 9 = CA1;
Pin 10 = CA2;
Pin 11 = CA3;
Pin 13 = !VECIACK;

```

```

Pin 23 = CD0;
Pin 22 = CD1;
Pin 21 = CD2;
Pin 20 = CD3;
Pin 19 = CD4;
Pin 18 = CD5;
Pin 17 = CD6;
Pin 16 = CD7;
Pin 15 = !DTACKI; /* the interrupt dtack */
/* Pin 14 = !ENDEC ; */ /* enable decode of EPROMS */

```

```

FIELD INTLEVEL=[CA3..CA1];
$DEFINE ATNINT 'H'A /* ['b'1,'b'0,'b'1] = 5 * 2 , need this since we are using ca3-1 not ca2-0 */
$DEFINE SCSINT 'H'8 /* = 4 * 2 */
$DEFINE SERINT 'H'6 /* = 3 * 2 */
$DEFINE PARINT 'H'4 /* = 2 * 2 */
$DEFINE TIMEINT 'H'2 /* = 1 * 2 */

```

```

FIELD DATA = [CD6..CD3];

```

```

FIELD ATNLEVEL = [CD2..CD0];
$DEFINE ATNLEVO ['H'0]
$DEFINE ATNLEV1 ['H'1]
$DEFINE ATNLEV2 ['H'2]
$DEFINE ATNLEV3 ['H'3]
$DEFINE ATNLEV4 ['H'4]
$DEFINE ATNLEV5 ['H'5]
$DEFINE ATNLEV6 ['H'6]
$DEFINE ATNLEV7 ['H'7]

```

```

ATNLEVEL.OE = VECIACK;
DATA.OE = VECIACK;

```

```

CD7 = VECIACK;

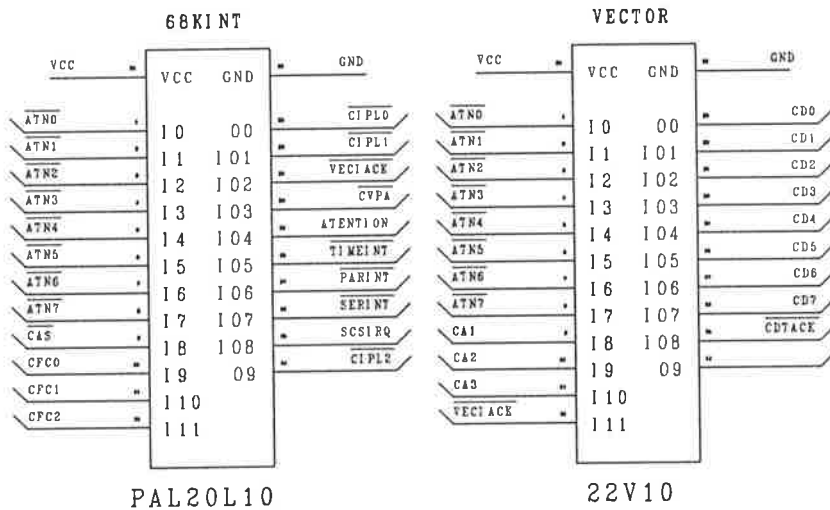
```



```

CD6 = INTLEVEL:ATNINT;
ATNLEVEL = ATNLEV7 & INTLEVEL:ATNINT & ATN7 #
            ATNLEV6 & INTLEVEL:ATNINT & !ATN7 & ATN6 #
            ATNLEV5 & INTLEVEL:ATNINT & !ATN7 & !ATN6 & ATN5 #
            ATNLEV4 & INTLEVEL:ATNINT & !ATN7 & !ATN6 & !ATN5 & ATN4 #
            ATNLEV3 & INTLEVEL:ATNINT & !ATN7 & !ATN6 & !ATN5 & !ATN4 & ATN3 #
            ATNLEV2 & INTLEVEL:ATNINT & !ATN7 & !ATN6 & !ATN5 & !ATN4 & !ATN3 & ATN2 #
            ATNLEV1 & INTLEVEL:ATNINT & !ATN7 & !ATN6 & !ATN5 & !ATN4 & !ATN3 & !ATN2 & ATN1 #
            ATNLEVO & INTLEVEL:ATNINT & !ATN7 & !ATN6 & !ATN5 & !ATN4 & !ATN3 & !ATN2 & !ATN1 #
            ATNLEVO & !(INTLEVEL:ATNINT) ;

```

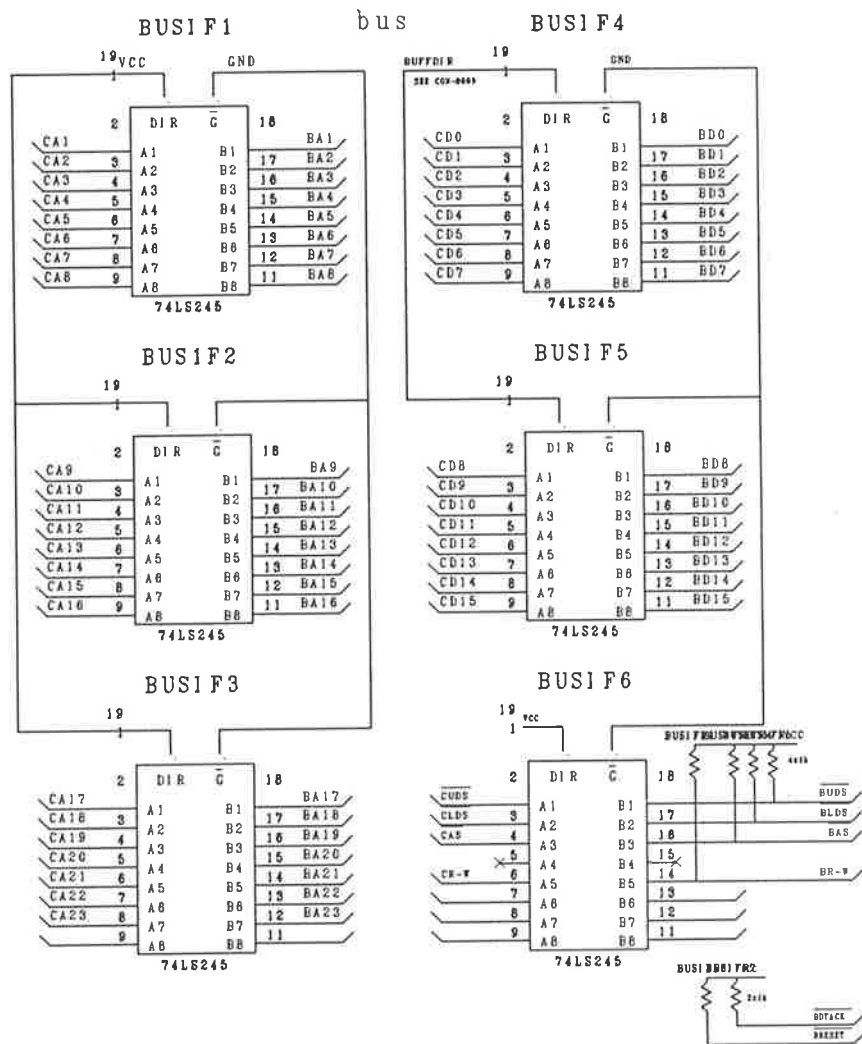


## A.1.9 CON-0008

### Backplane bus interface

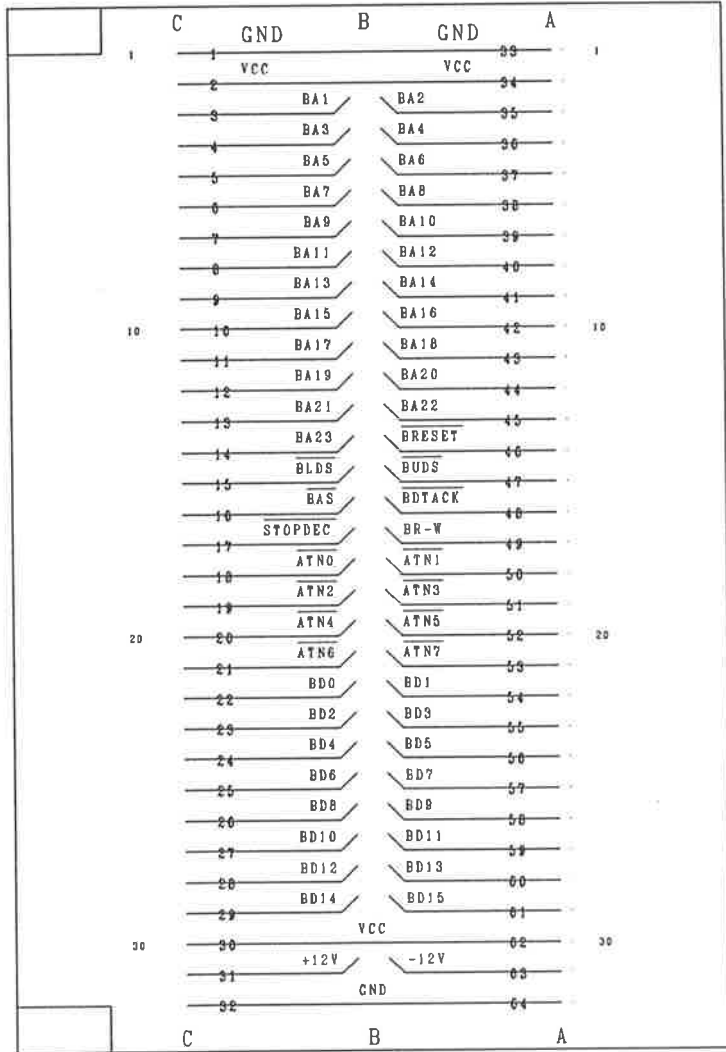
#### Notes:

1. All memory locations at addresses higher than \$07FFFF are assumed to lie offboard, and the bus transceivers are enabled.
2. The backplane signals may be grouped into 4 main categories according to function:
  - (a) Address lines A1-A23.  
Giving a total address space of 16Mbytes (minus the 512kbytes which reside on the control processor board itself).
  - (b) Data lines D0-D15.  
The data bus is 16-bit wide. Operations on 8-bit bytes are supported, by virtue of the presence of the 68000  $\overline{UDS}$  and  $\overline{LDS}$  signals.
  - (c) Control signals.
    - Strokes:  $\overline{AS}$ ,  $\overline{UDS}$ ,  $\overline{LDS}$ , R/  $\overline{W}$ ,  $\overline{BAS}$
    - System control:  $\overline{DTACK}$ ,  $\overline{STOPDEC}$ ,  $\overline{BRESET}$
    - Attention Request:  $\overline{ATN0}$  ...  $\overline{ATN7}$
  - (d) Power supplies.  
Logic supplies are available as standard (0,5V).  
Two pins have been reserved for positive and negative analogue rails. The voltage to be distributed on these rails has yet to be defined, and could be as high as  $\pm 20$  volts (to allow regulators on plug-in modules to generate  $\pm 15V$ ).



bcc

# BUSCON



MALE EUROCONNECTOR

### A.1.10 CON-0009

#### Board decoding & memory map.

Of the 16MB accessible to the 68000 control processor, the bottom 512kB (address \$000000-\$07FFFF) are decoded onboard the control processor module itself. The remainder of the address space is available to slave modules. To function correctly, modules must obey the rules outlined in section A.4.

Address	Device
\$000000..\$03FFFF	EPROM
\$040000..\$07FFFF	RAM
\$060000	DUART select
\$064000	Parallel port read Parallel port write
\$068000	Status register read Control register write
\$06C000	SCSI (NCR5380) device select
\$070000	SCSI pseudo DMA select

Table A.8: Control Processor Memory Map

```
Name      ONBOARD;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p16L8;
Format    j;
```

```
/* SHEET CON-0009 */
/* This PAL detects onboard/offboard accesses. */
```

```
Pin 1 = CA23;
Pin 2 = CA22;
Pin 3 = CA21;
Pin 4 = CA20;
Pin 5 = CA19;
Pin 6 = !STOPDEC;
Pin 7 = CRW;
Pin 8 = !CAS;
Pin 9 = PRESET;      /* active high power on reset signal from CON-0000 */
Pin 11 = ASABSENT;  /* this signal indicates lack of AS' from CON-0005 */
```

```
Pin 19 = INVCAS;    /* inverted !CAS signal */
Pin 18 = BUFFDIR;   /* bus buffer direction signal from CON-0008 */
Pin 17 = !VECIACK;  /* indicates interrupt vector request by 68k */
Pin 16 = !OFFBOARD; /* active when address is offboard */
Pin 15 = !CRESET;   /* processor reset */
Pin 14 = !CHALT;    /* processor halt */
Pin 13 = !BRESET;   /* backplane reset */
Pin 12 = !INVCRW;   /* inverted RW strobe */
```

```
FIELD HIADR = [CA23..CA19];
```

```
OFFBOARD = !(HIADR:000000) # STOPDEC # VECIACK; /* prevent all board decoding when
                                                    STOPDEC or VECIACK are asserted */
```

```
!BUFFDIR = !VECIACK & OFFBOARD & CRW # /* backplane buffers point inward when we are */
        !VECIACK & STOPDEC & CRW;      /* reading from backplane. */
```

```
INVCRW = CRW;
```

```
INVCAS = CAS;
```

```
CRESET = PRESET # ASABSENT;
```

```
CHALT = PRESET # ASABSENT;
```

```
BRESET = 'b'1;
```

```
BRESET.OE = CRESET; /* make backplane reset follow 68000 reset */
```

```
Name      decode;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p16L8;
Format    j;
```

```
/* SHEET CON-0009 */
/* 68000 control board decoder PAL */
```

```
Pin 1 = CA18;
Pin 2 = CA17;
Pin 3 = CA16;
Pin 4 = CA15;
Pin 5 = CA14;
Pin 6 = !CAS;
Pin 7 = CRW;
Pin 8 = !OFFBOARD;
Pin 9 = !SERDTK;
```

```
Pin 19 = !SERSEL;
Pin 18 = !PARR;
Pin 17 = !PARW;
Pin 16 = !STATUSR;
Pin 15 = !CONTROLW;
Pin 14 = !DEVDTK; /* to main DTACK PAL , CON-0004 */
Pin 13 = !SCSDACK; /* used for pseudo DMA transfers */
Pin 12 = !SCSISEL;
```

```
$DEFINE ONBOARD !OFFBOARD
```

```
FIELD DEVADR = [CA18..CA14];
```

```
SERSEL = DEVADR:60000 & CAS & ONBOARD;
```

```
PARR = DEVADR:64000 & CRW & CAS & ONBOARD ;
```

```
PARW = DEVADR:64000 & !CRW & CAS & ONBOARD ;
```

```
STATUSR = DEVADR:68000 & CRW & CAS & ONBOARD ;
```

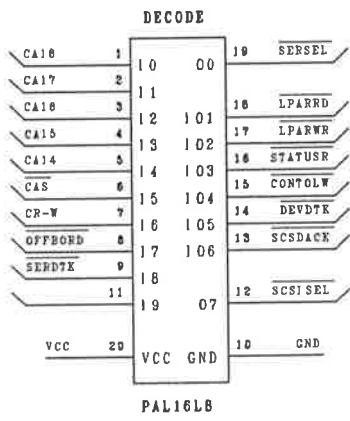
```
CONTROLW = DEVADR:68000 & !CRW & CAS & ONBOARD ;
```

```
SCSISEL = DEVADR:6C000 & CAS & ONBOARD ;
```

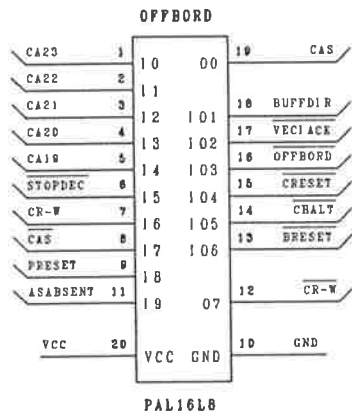
```
SCSDACK = DEVADR:70000 & CAS & ONBOARD ;
```

```
DEVDTK = (      DEVADR:64000 #
                DEVADR:68000 #
                DEVADR:6C000 #
                DEVADR:70000 #
                ( DEVADR:60000 & SERDTK ) ) & CAS & ONBOARD ;
```

DEC



from CON-0000  
from CON-0004





### A.1.11 CON-0010

#### **Bus Attention Signals.**

The bus attention signals are 8 active-low signals which pass directly to the PALs of the interrupt encoder of CON-0006. The signals must be pulled high by resistors, but need no other conditioning.

The signals are asserted by a slave module (such as the DSP module) to request servicing by the control processor. The method by which the attention signals are asserted and cleared is documented in section A.2.9.

### A.1.12 CON-0012

#### Board Control Register.

The board control register is an 8-bit register whose bits directly control signals in the circuit of the control processor.

A control bit is written to by writing a 0 or 1 into data bit 15 of the word at the address shown in table A.9.

Address	Function
\$68000	Not Used
\$68002	Not Used
\$68004	Not Used
\$68006	Not Used
\$68008	Not Used
\$6800A	Time Interrupt Acknowledge ( $\overline{\text{TIMEACK}}$ )
\$6800C	Parallel Read Interrupt enable (PARIREN)
\$6800E	Parallel Write Interrupt enable (PARIWEN)

Table A.9: Control Processor control register bit assignments

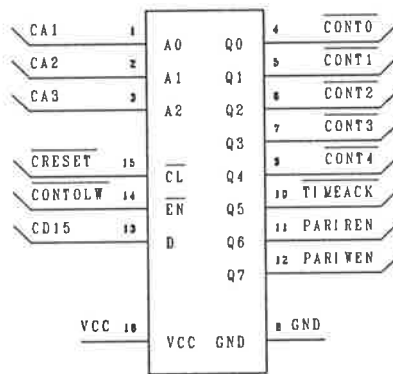
During execution of its coldstart (RESET) routine, the 68000 must set the  $\overline{\text{TIMEACK}}$  bit to 1. On receipt of a timer interrupt, it must pulse the bit to 0 (asserted), returning it to the deasserted (1) state immediately. This pulse acknowledges the timer interrupt, and allows the next one to be generated at the appropriate time. Without an acknowledge pulse, no future timer interrupts will occur.

The acknowledge pulse may be generated at any time after the interrupt. If no timer interrupts are to be missed, the delay must be less than one timer period (65536 master oscillator cycles, or 32768 processor clock cycles).

The PARIREN and PARIWEN signals, when set to 1, enable the respective interrupts from the parallel port.

CON

CONT



74LS259

### A.1.13 CON-0013

#### Board Status Register.

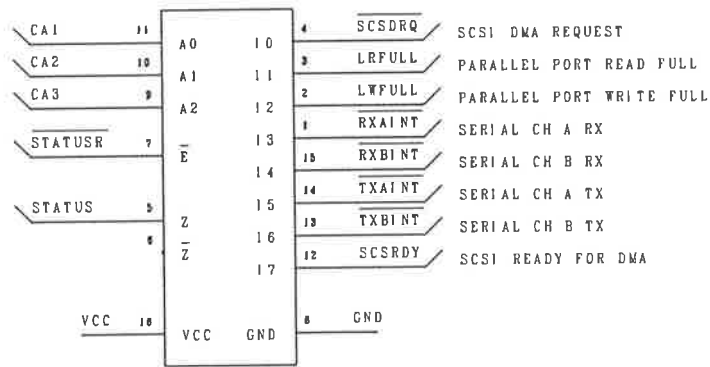
The status register allows the 68000 to sample system signals directly. To read the status of any signal allocated to a status register address (see table A.9), the 68000 must read a 16-bit word from the corresponding address. The state of the signal is then reflected as bit 15 of that word.

Address	Function
\$68000	$\overline{\text{SCSIDRQ}}$ . If this signal is asserted (ie low) then the SCSI controller is requesting data transfer, which is accomplished by pseudo DMA on the control processor board.
\$68002	LRFULL. If this signal is asserted, the parallel port read register is full and waiting to be read.
\$68004	LWFULL. If this signal is asserted, the parallel port write register is full, and another byte may not be written to it until the signal is deasserted.
\$68006	$\overline{\text{RXAINT}}$ . If this signal is asserted, the DUART is asserting its channel A receiver interrupt.
\$68008	$\overline{\text{RXBINT}}$ . If this signal is asserted, the DUART is asserting its channel B receiver interrupt.
\$6800A	$\overline{\text{TXAINT}}$ . If this signal is asserted, the DUART is asserting its channel A transmit interrupt.
\$6800C	$\overline{\text{TXBINT}}$ . If this signal is asserted, the DUART is asserting its channel B transmit interrupt.
\$6800E	$\overline{\text{SCSRDY}}$ . If this signal is asserted, the SCSI controller is indicating it is ready for DMA transfers.

Table A.10: Control Processor board status register

STA

STATUS



MUST TRISTATE THIS  
ONTO CD15 USING  
STATUSR' AS A STROBE

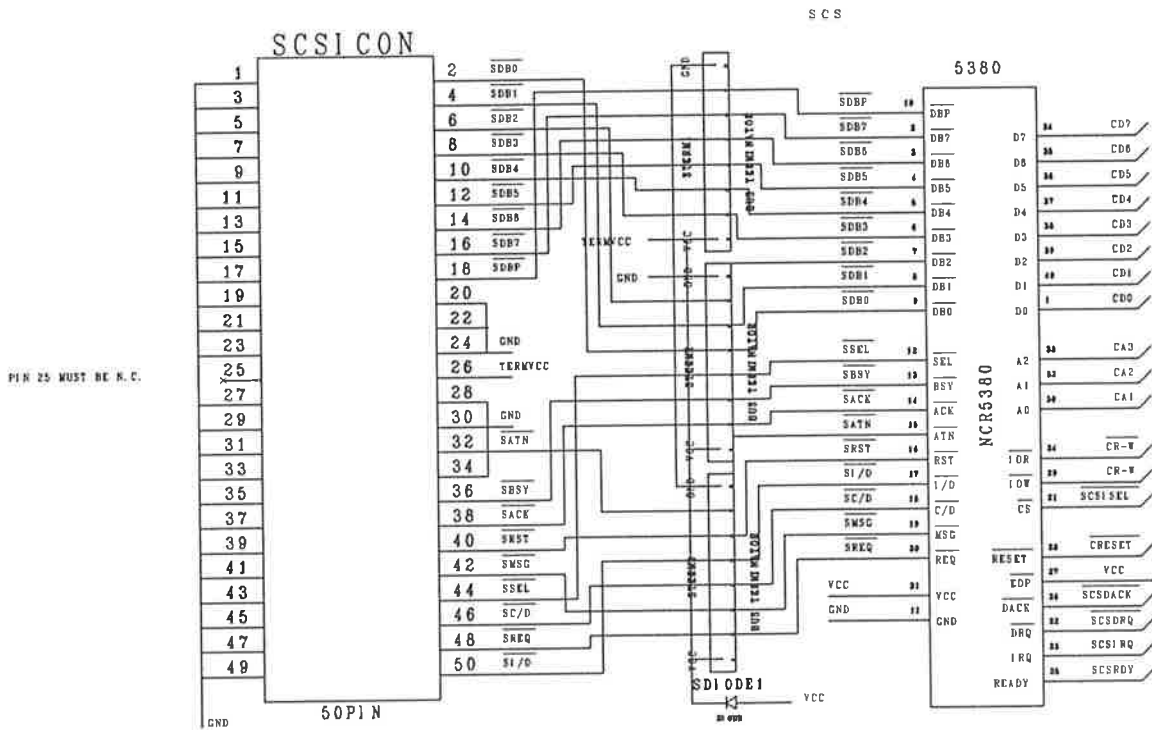
74LS151

### A.1.14 CON-0014

#### SCSI Interface.

The SCSI interface is built around a NCR5380 (or equivalent) SCSI interface controller. No DMA is used in the interface, so all data transfers are performed using pseudo-DMA. The NCR5380 registers are directly memory mapped. The user should consult the NCR5380 data sheet for programming information.

The SCSI connection is provided via a 50-pin strip. Terminator resistors are provided on the control processor board.



## A.2 The TMS320C25 Digital Signal Processor Module

The digital signal processor module is based around the TMS320C25 digital signal processor, which is a fast (10MIP) processor designed specifically for numerically intensive calculations.

The important features of this module are:

- TMS320C25 running at 40MHz.
- Up to 64k-words program RAM.
- Up to 64k-words data RAM.
- 1024x16 FIFO from DSP to backplane.
- 16-bit FIFO from backplane to DSP.
- Backplane interface allows direct access to DSP program, data, and IO spaces.
- Separate IO bus connector.

Drawing	Contents
DSP-0000	TMS320C25 signal processor Clock
DSP-0001	Wait state generation for the DSP
DSP-0002	DSP module onboard decoding
DSP-0004	AIM interface
DSP-0005	Program and Data RAM
DSP-0006	Interprocessor Mailbox
DSP-0007	DSP-accessible control register
DSP-0008	DSP-accessible status register
DSP-0009	DSP Interrupts
DSP-0010	Backplane Attention Request Signals
DSP-0011	Module Control Register
DSP-0012	Module Status Register

Table A.11: Drawings of the DSP Module

The DSP module appears as a collection of memory mapped peripherals to the 68000 control processor, as discussed in section A.2.5.

### A.2.1 DSP Module Specifications

Processor	TMS320C25 (PGA or PLCC)
Processor clock rate	40MHz Card can be configured to run with 0/1 wait state. With 0 wait states, this gives 10MIPS peak performance. With 1 wait state, performance lies between 5-10MIPS, though 10MIPS is still achievable using TMS320C25 internal program memory. Wait states are chosen according to memory speed.
Program RAM	64k-words
Data RAM	64k-words
Analogue Interface	40-pin connector to offboard analogue interface module
Data transfer	The DSP module is fully accessible from the backplane bus. All circuitry (outside of the TMS320C25) can be accessed by the backplane master.
FIFO	1024-word FIFO from DSP to backplane 1-word register from backplane to DSP.
Control Aspects	Backplane master has full control over $\overline{\text{RESET}}$ and $\overline{\text{HOLD}}$ signals to the DSP.
Backplane Connector	DIN41612 (proprietary bus)

Table A.12: TMS320C25 DSP Module Specifications

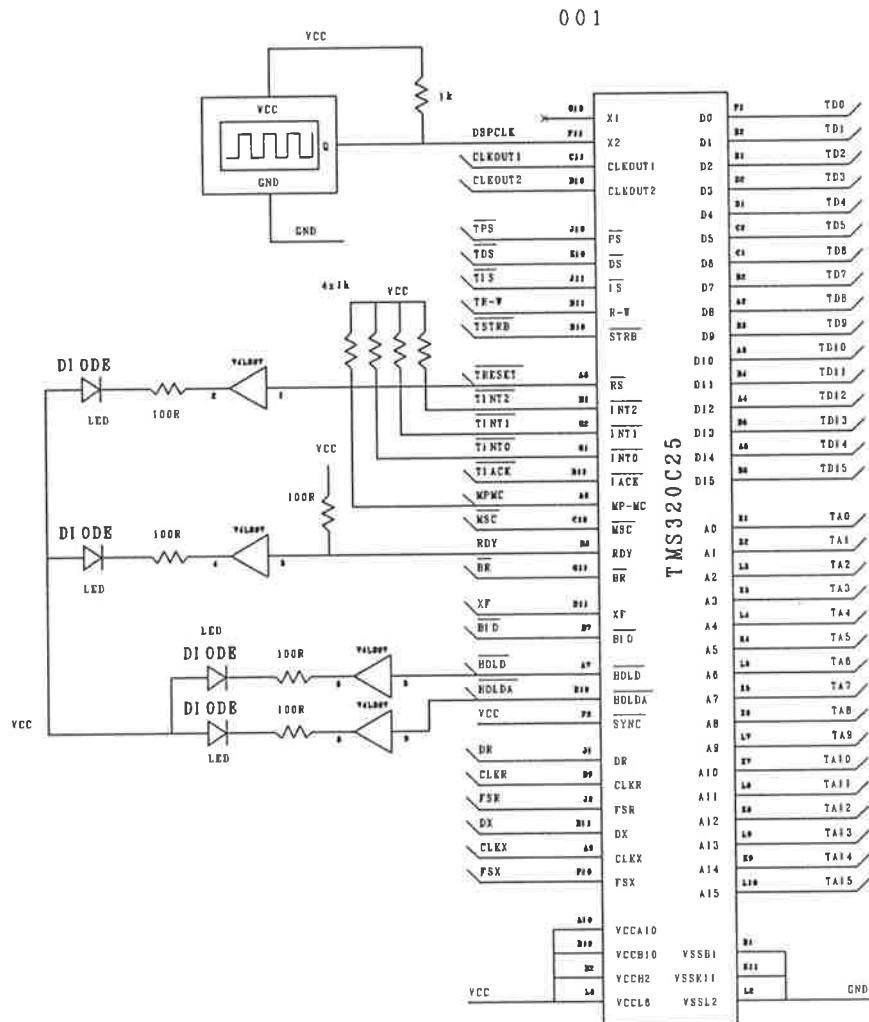


## A.2.2 DSP-0000

### TMS320C25 and Clock.

*Notes:*

1. The clock to the DSP is supplied by a 40MHz crystal oscillator module.
2. The TMS320C25 may be either PGA or PLCC package. They are physically pin compatible (when the PLCC is in a socket).
3. The  $\overline{\text{STRB}}$  signal must have a pull-up resistor to prevent spurious activity when the TMS320C25's busses are tri-stated.



### A.2.3 DSP-0001

#### TMS320C25 Wait State Generation.

*Notes:*

1. The TMS320C25, when clocked at 40MHz, executes a maximum of 1 bus cycle per 100ns CLKOUT period.

Of this 100ns, about 60ns is available (strokes active) for memory access. The cheapest memory devices are of the order of 80ns-100ns, and may or may not function correctly under such circumstances.

Therefore, it is necessary to either use fast, but expensive, memories, or slower memories with 1 wait state.

Both methods are supported in this module. The latter results in a decreased throughput (5MIPs instead of 10MIPs), when executing programs in external program memory, though this can be alleviated by transferring the programs into internal DSP memory and executing them there at the full 10MIP rate (provided the programs are small enough to fit into the internal memory.)

2. Devices such as input/output port latches, ADC, DAC, etc typically require no wait states.
3. No wait states need to be generated when the DSP is tri-stated (ie when the backplane busses have been piped through to the DSP busses.)
4. This module can operate at full speed with zero wait states, or half speed with one wait state. The mode of operation is set using the jumper PSTATES.

Jumper PSTATES	Result
omitted	0 wait state
inserted	1 wait state

Table A.13: DSP wait state jumper

The wait states are generated with the aid of the  $\overline{MSC}$  signal of the TMS320C25, as documented in [1].

5. The PAL called DSPWAIT in this drawing generates the chip- select strobes for the program and data memory.

```
Name      dspwait;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p20L10;
Format    j;
```

```
/* DRAWING DSP-0001 */
```

```
/* This pal provides strobe decoding from the backplane bus onto the
TMS320C25 busses, and generates the RDY signal which determines the
length of bus cycles, and provides decode strobes for RAM. */
```

```
Pin 1 = !DECODE; /* combined with !AS already in the decode process */
```

```
Pin 2 = BA18;
```

```
Pin 3 = BA17;
```

```
Pin 4 = !HOLDA;
```

```
Pin 5 = TA15;
```

```
Pin 6 = TA13;
```

```
Pin 7 = !MSC;
```

```
Pin 8 = PRAMTYPE; /* 1 means 32k device, 0 means 8k device */
```

```
Pin 9 = DRAMTYPE; /* 1 means 32k device, 0 means 8k device */
```

```
Pin 10 = PSTATES; /* 1 means full speed, 0 means 1 wait state (using !MSC) */
```

```
Pin 11 = DSTATES; /* 1 means full speed, 0 means 1 wait state (using !MSC) */
```

```
Pin 13 = ISTATES; /* 1 means full speed, 0 means 1 wait state (using !MSC) */
```

```
Pin 23 = !PRAMOCE; /* lower program memory device select */
```

```
Pin 22 = !PRAM1CE; /* upper program memory device select */
```

```
Pin 21 = !DRAMOCE; /* lower data memory device select */
```

```
Pin 20 = !DRAM1CE; /* upper data memory device select */
```

```
Pin 19 = !TSTRB;
```

```
Pin 18 = !TPS;
```

```
Pin 17 = !TDS;
```

```
Pin 16 = !TIS;
```

```
Pin 15 = ONBDRDY;
```

```
Pin 14 = !DSPSEL; /* asserted when a request for DSP space is made */
```

```
TPS.OE = HOLDA;
```

```
TPS = DECODE & BA18 & BA17;
```

```
TDS.OE = HOLDA;
```

```
TDS = DECODE & BA18 & !BA17;
```

```
TIS.OE = HOLDA;
```

```
TIS = DECODE & !BA18 & BA17;
```

```
TSTRB.OE = HOLDA;
```

```
TSTRB = DECODE & (BA17 # BA18); /* don't assert when accessing control area */
```

```
DSPSEL = DECODE & (BA17 # BA18);
```

```
PRAMOCE = TPS & TSTRB & !TA15 & PRAMTYPE #
```

```
TPS & TSTRB & !TA13 & !PRAMTYPE;
```

```
PRAM1CE = TPS & TSTRB & TA15 & PRAMTYPE #
```

```
TPS & TSTRB & TA13 & !PRAMTYPE;
```

```
DRAMOCE = TDS & TSTRB & !TA15 & DRAMTYPE #
```

```

TDS & TSTRB & !TA13 & !DRAMTYPE;

DRAM1CE = TDS & TSTRB & TA15 & DRAMTYPE #
          TDS & TSTRB & TA13 & !DRAMTYPE;

/*
ONBDRDY =  TPS & PSTATES #
          TPS & !PSTATES & !MSC #

          TDS & DSTATES #
          TDS & !DSTATES & !MSC #

          TIS & ISTATES #
          TIS & !ISTATES & !MSC;
*/

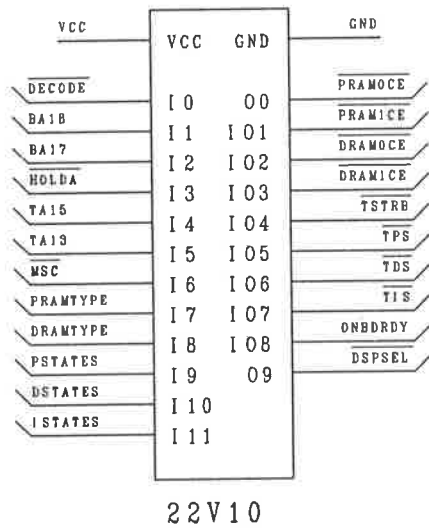
```

/\* The following is a modified one which allows implementation in a 20L10 \*/

```

ONBDRDY = 'b'0;
ONBDRDY.OE = !HOLDA & !PSTATES & MSC;

```



## A.2.4 DSP-0002

### TMS320C25 Onboard Decoding.

This drawing contains only references for the onboard device decoder, as that decoder has been implemented in PALs which are documented elsewhere. This provides read and write strobes for IO devices within the module.

#### *Notes:*

1. The strobes for program and data memory are generated by the DSPWAIT PAL in drawing DSP-0001.
2. IO Devices which must be selected by the DSP are:
  - DSP-accessible status register (see DSP-0008).
  - DSP-accessible control register (see DSP-0007).
  - DSP-to-Backplane FIFO (see DSP-0006).
  - AIM Interface (see DSP-0004).
3. The decoding is performed by the PAL in drawing DSP-0001 and DSP-0002.
4. The IO space of the TMS320C25 has a capacity of 16 read- devices and 16 write-devices. Of these, the first 2 read and first 2 write devices are decoded on the DSP board itself. The remaining 14 of each are decoded to the AIM interface, which is assumed to be connected to the AIM interface connector (see DSP-0004).
5. All IO devices must conform to zero wait-state access by the TMS320C25.

## A.2.5 DSP-0003

### TMS320C25 to Backplane Bus Interface.

To the backplane, the DSP Module appears as a 512kbyte area of memory. This 512k is divided into 4 distinct regions of equal size.

Offset from Base	Function
\$000000	<p>Board Control Area</p> <p>This area contains read-write registers dedicated to board control. The following registers are accessible in this space. The nature of the registers is stated in table A.15.</p>
\$020000	<p>DSP IO Space</p> <p>Only the first 16 16-bit words of this space have any meaning. They correspond to the 16 IO words of the TMS320C25. Reading or writing to this memory space will have the effect of reading or writing straight through to the corresponding TMS320 IO port. This correspondence is stated explicitly in table A.16. As discussed in the text of this section, access to this space causes a <math>\overline{\text{HOLD}} / \overline{\text{HOLDA}}</math> sequence to allow the backplane master possession of the DSP's busses, after which the read/write operation is performed. Note that access to this memory space must be 16-bit only. Byte operations are not supported.</p>
\$040000	<p>DSP Data Space</p> <p>The entire span of 128k bytes of this region is mapped into the 64k words of DSP memory. The text of this section provides information about how the <math>\overline{\text{HOLD}} / \overline{\text{HOLDA}}</math> protocol is used to access this section. Note that access to this memory space must be 16-bit only. Byte operations are not supported.</p>
\$060000	<p>DSP Program Space</p> <p>The entire span of 128k bytes of this region is mapped into the 64k words of DSP program memory. The text of this section provides information about how the <math>\overline{\text{HOLD}} / \overline{\text{HOLDA}}</math> protocol is used to access this section. Note that access to this memory space must be 16-bit only. Byte operations are not supported.</p>

Table A.14: DSP Module Backplane Memory Map

The offset address is relative to the base address of the DSP module, which is set using jumpers to a 512kbyte boundary in the 16Mbyte address space of the control processor.

The TMS320C25 provides a mechanism which allows another processor to gain full control of the

Address Offset	R/W	Function
\$000000	R	Identity PAL
\$004000	R	Module Status Register (see drawing DSP-0012)
\$008000	W	Module Control Register (see drawing DSP-0011)
\$00C000	R	DSP-to-Backplane FIFO (see drawing DSP-0006)
\$010000	W	Backplane-to-DSP FIFO (see drawing DSP-0006)
\$014000	W	Attention Acknowledge (see drawing DSP-0010)

Table A.15: Board Control Area Memory Map  
The Address offset is relative to the DSP module base address.

Offset relative to IO base	Corresponding TMS320 IO Address
\$000000	\$0
\$000002	\$1
\$000004	\$2
\$000006	\$3
\$000008	\$4
\$00000A	\$5
\$00000C	\$6
\$00000E	\$7
\$000010	\$8
\$000012	\$9
\$000014	\$A
\$000016	\$B
\$000018	\$C
\$00001A	\$D
\$00001C	\$E
\$00001E	\$F

Table A.16: IO space address correspondence

DSP's external busses. This mechanism is used to allow the backplane bus master (at present the 68000 control processor) to access all DSP memory and IO devices directly.

This is useful for many reasons, the most important being for testing and the loading of DSP executable code into program memory.

*Notes:*

1. To gain possession of the DSP busses, the backplane bus master must assert the  $\overline{\text{HOLD}}$  signal to the DSP and wait for that request to be acknowledged by the DSP asserting the  $\overline{\text{HOLDA}}$  signal.

Once  $\overline{\text{HOLDA}}$  is asserted, the DSP has relinquished control of its busses, and bus transceivers may be enabled to allow the backplane bus signals into the DSP modules core.

2. In the present DSP module, there are two methods for asserting the  $\overline{\text{HOLD}}$  signal. If the backplane master attempts to access the DSP program, data, or IO spaces directly, the  $\overline{\text{HOLD}} / \overline{\text{HOLDA}}$  protocol will be used for that one bus cycle. Once the bus cycle has terminated, the  $\overline{\text{HOLD}}$  signal to the DSP is deasserted to allow the DSP to resume program execution.

The second method requires the backplane master to set a bit (set it to 0) in the *Board Control Register* (see DSP-0011) which asserts the  $\overline{\text{HOLD}}$  signal to the DSP for as long as that bit is set (to 0). When this mechanism is used, there is no contention for the DSP's busses, since the backplane master always in control of them.

3. Full 16-bit address, 16-bit data, and control signals are buffered from the backplane into the DSP circuit. Thus the backplane master can access all circuitry external to TMS320C25, allowing full testing of the DSP board's subsystems.



```
Name      IDENT;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    g16v8;
Format    j;
```

```
/* SHEET DSP-0003 */
/* Identity PAL */
```

```
Pin 1 = BA3;
Pin 2 = BA2;
Pin 3 = BA1;
Pin 4 = !IDENT;
Pin 5 = !DECODE;
Pin 6 = !DSPSEL;
Pin 7 = !PERMRES;
Pin 8 = !BRESET;
Pin 9 = !HOLDA;
Pin 11 = BRW;
```

```
Pin 19 = BUFD0;
Pin 18 = BUFD1;
Pin 17 = BUFD2;
Pin 16 = BUFD3;
```

```
Pin 15 = !BUF1EN;
Pin 14 = !BUF2EN;
Pin 13 = !TRESET;
Pin 12 = TRW;
```

```
FIELD DATA = [BUFD3..BUFD0];
```

```
DATA = 'b'0101 & !BA3 & !BA2 & !BA1 #
        'b'1010 & !BA3 & !BA2 & BA1;
```

```
DATA.OE = IDENT;
```

```
BUF1EN = DECODE;
BUF2EN = DSPSEL & HOLDA;
```

```
TRESET = BRESET #
        PERMRES;
```

```
TRW = BRW;
TRW.OE = HOLDA;
```

```
Name      DSPBUSiF;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    g16v8;
Format    j;
```

```
/* SHEET DSP-0003 */
/* Interface from bus to dsp */
```

```
Pin 1 = BA16;
Pin 2 = BA15;
Pin 3 = BA14;
Pin 4 = RDY;
Pin 5 = !DSPSEL;
Pin 6 = !DECODE;
Pin 7 = !PERMHOLD;
Pin 8 = !HOLDA;
Pin 9 = BRW;
```

```
Pin 19 = !BFIFOR;
Pin 18 = !BFIFOW;
Pin 17 = !BSR;
Pin 16 = !BCR;
Pin 15 = !HOLD;
Pin 14 = !BDTACK;
Pin 13 = !BATNACK;
Pin 12 = !IDENT;
```

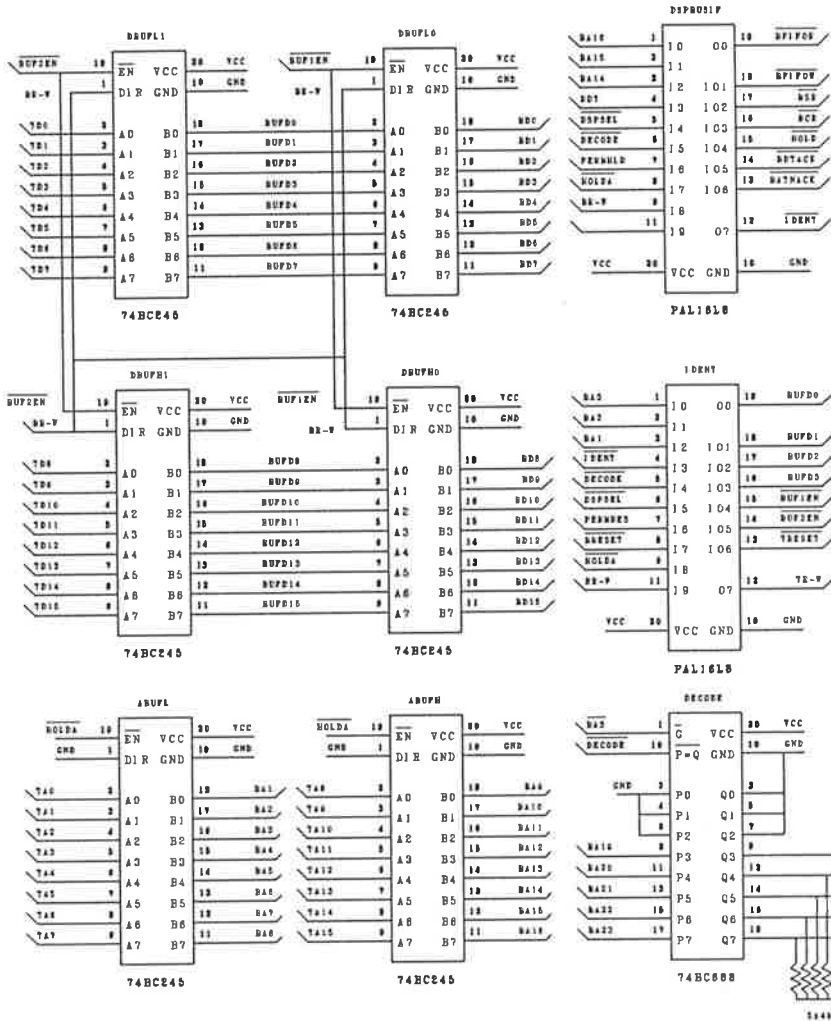
```
IDENT = DECODE & !DSPSEL & !BA16 & !BA15 & !BA14 & BRW;
BSR    = DECODE & !DSPSEL & !BA16 & !BA15 & BA14 & BRW;
BCR    = DECODE & !DSPSEL & !BA16 & BA15 & !BA14 & !BRW;
BFIFOR = DECODE & !DSPSEL & !BA16 & BA15 & BA14 & BRW;
BFIFOW = DECODE & !DSPSEL & BA16 & !BA15 & !BA14 & !BRW;
BATNACK = DECODE & !DSPSEL & BA16 & !BA15 & BA14 & !BRW;
```

```
HOLD = DSPSEL #
      PERMHOLD;
```

```
/*
BDTACK = DSPSEL & HOLDA & RDY #
        DECODE & !DSPSEL;
*/
```

```
BDTACK = DSPSEL & HOLDA # /* this is a hack to make BDTACK work on HOLDA asserted */
        DECODE & !DSPSEL;
```

```
BDTACK.OE = DECODE;
```



## A.2.6 DSP-0004

### AIM Interface.

The TMS320C25 features a separate connector to which Analogue Interface Modules (AIM) may be connected.

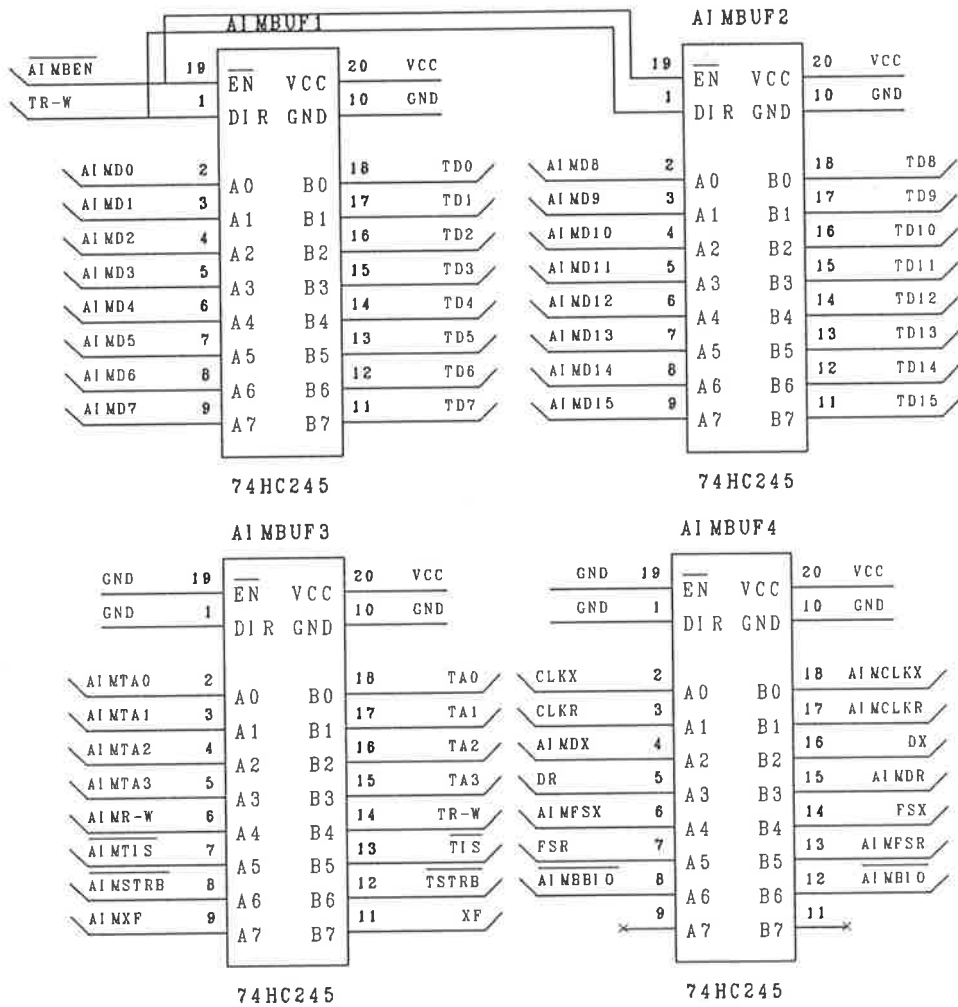
The DSP Module has no analogue interface circuitry on it at all, it must all be provided on a separate module. The AIM must be connected via a separate connector to the DSP module due to the high data bandwidth which will usually be required of the system.

The backplane bus is provided only as a control bus, and while it is feasible for the control processor to provide the data path between separate DSP and AIM boards, the system would fail to perform at high rates, and would involve excessively complex software to coordinate the transfer. For this reason, the AIM has a separate, direct, hardware connection of the DSP module.

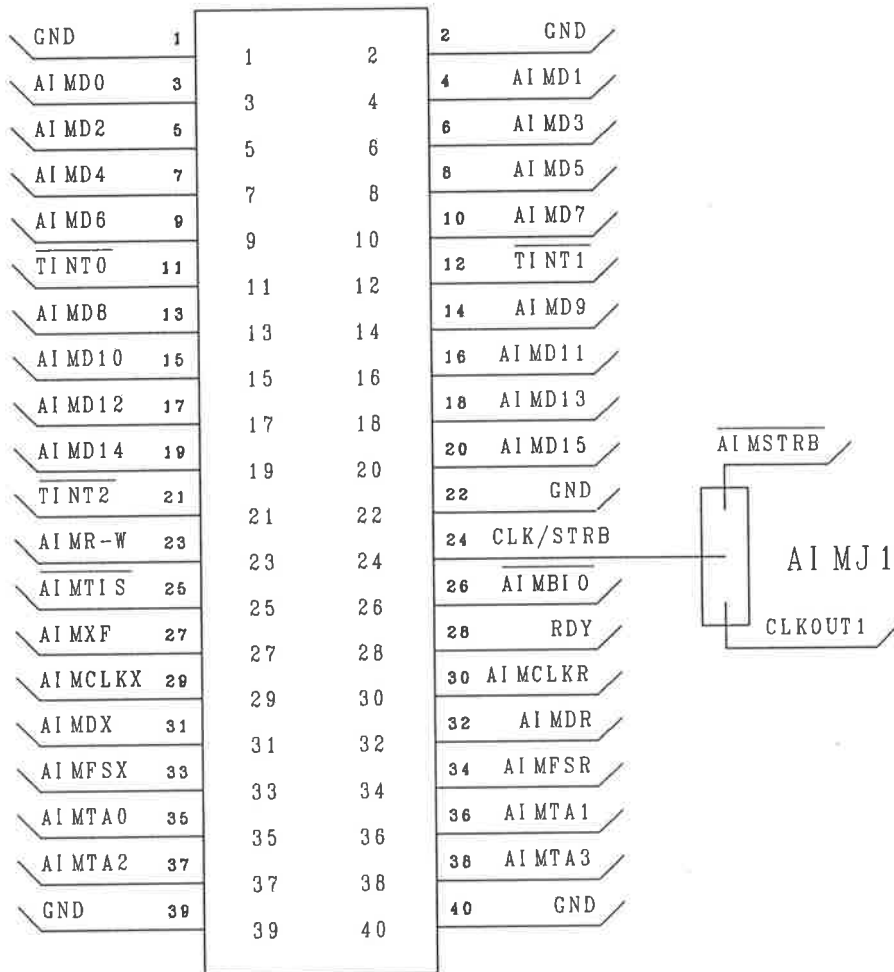
#### Notes:

1. The TMS320C25 can address 16 read-write IO devices. The DSP module reserves the first 2 for onboard use, but the remaining 14 may be decoded on the AIM.
2. All data, address, and control lines passing between the DSP module and the AIM are buffered.
3. The TMS320C25 serial-codec support signals are also passed through the AIM connector.

Jumper AIMJ1 allows the AIM to receive either the buffered  $\overline{\text{STRB}}$  signal from the TMS320C25, or the CLKOUT1 signal (10MHz clock).



# AIMCON



## A.2.7 DSP-0005

### Program and Data RAM.

*Notes:*

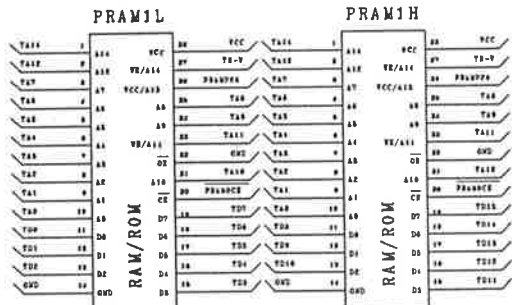
1. Both 8k×8 and 32k×8 SRAM devices are supported. The former result in a full memory of 16k- words in each memory area, and the latter in 64k-words in each area.

The only restriction is that all SRAM devices on the DSP module must be of the same type. The nature of the SRAM used is determined by jumper DRAMTYPE.

Jumper DRAMTYPE	SRAM supported
omitted	32k×8
inserted	8k×8

Table A.17: DSP RAM type jumper settings

2. Devices with access times of below 50ns should be used if zero wait-state operation is required. Devices of access time 150ns or less are suitable for 1 wait-state operation.





## A.2.8 DSP-0006

### Interprocessor Mailbox.

The interprocessor mailbox provides a means by which the TMS320C25 and the system control processor can communicate without the control processor actively taking control of the signal processor's busses.

The mailbox actually consists of two distinct ports, each unidirectional. A single-word-deep port communicates words from the control processor to the signal processor, while a 1024-word-deep FIFO communicates words from the DSP to the control processor.

The asymmetry in the depth of the ports exists for reasons of economics and available PCB real-estate. The DSP-to-backplane direction was given the deeper FIFO since the response time of the control processor to an attention request may be long (if it is servicing an attention request from another DSP, for instance), and this must be the least delay in DSP operations.

The backplane-to-DSP direction is somewhat less critical, since the response time of the DSP to servicing mailbox interrupts will be fast, so that a deep hardware FIFO is less necessary.

```

Name      mailbox;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p1618;
Format    j;

```

```

/* SHEET DSP-0006 */
/* Mailbox between the bus and the DSP */

```

```

Pin 1 = TRW;
Pin 2 = TA3;
Pin 3 = TA2;
Pin 4 = TA1;
Pin 5 = TAO;
Pin 6 = !TIS;
Pin 7 = PRINTEN;
Pin 8 = PWINTEN;
Pin 9 = !EF;
Pin 11 = !BFIFOW;

```

```

Pin 19 = !TFIFOW;      /* DSP write strobe for DSP->BUS fifo */
Pin 18 = !TFIFOR;     /* DSP read strobe for BUS->DSP fifo */
Pin 17 = FULL;        /* intermediate variable */
Pin 16 = FULLFLAG;    /* high when word in BUS->DSP fifo */
Pin 15 = !INT0;       /* TMS320C25 interrupt */
Pin 14 = !INT1;       /* TMS320C25 interrupt */
Pin 13 = !AIMBEN;     /* AIM data buffer enable */
Pin 12 = !TCR;        /* DSP write strobe for control register */

```

```

TFIFOW = TIS & !TA3 & !TA2 & !TA1 & !TA0 & !TRW;
TFIFOR = TIS & !TA3 & !TA2 & !TA1 & !TA0 & TRW;
TCR =    TIS & !TA3 & !TA2 & !TA1 & TAO & !TRW;

```

```

AIMBEN = TIS & (TA3 # TA2 # TA1); /* enable buffers for all except the
                                     bottom 2 io locations */

```

```

FULL = BFIFOW #
      FULL & !TFIFOR;

```

```

FULLFLAG = FULL & !BFIFOW;

```

```

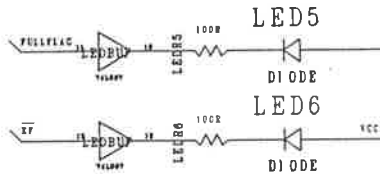
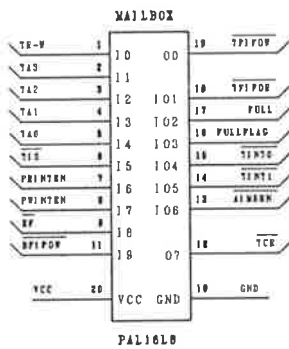
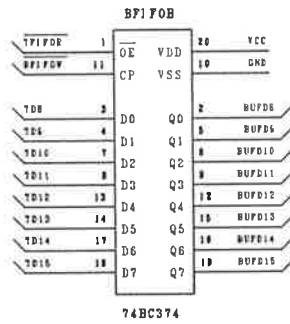
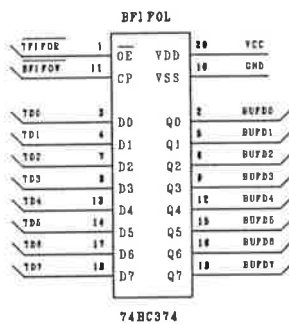
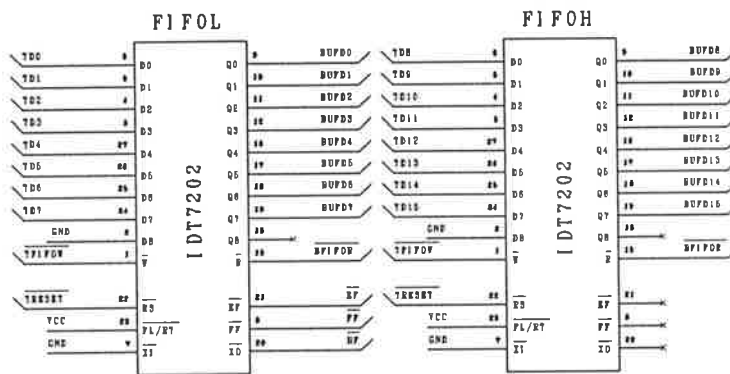
INT1.OE = FULLFLAG & PRINTEN;
INT1     = FULLFLAG & PRINTEN;

```

```

INT0.OE = EF & PWINTEN;
INT0     = EF & PWINTEN;

```



## A.2.9 DSP-0007

### DSP-accessible Control Register.

The DSP-accessible control register allows the TMS320C25 to control the states of various signals within its surrounding circuitry. The 8 control signals are documented in table A.19. The functions of the signals fall into the following main categories:

- Attention Level Control and Assertion

The backplane bus supports an 8-level attention request mechanism which modules may use to request service by the control processor.

A client module asserts an Attention Request signal by performing the following actions:

1. Set the Attention Request Select Bits 0-2 (ARSB0-2) to form the binary representation of the number of the  $\overline{ATNn}$  backplane signal to be asserted.  
For instance, to assert the backplane signal  $\overline{ATN6}$ , the user must set ARSB0=0, ARSB1=1, ARSB2=2.
2. Set the ATNREQ bit of the control register to 1. This bit must only be set momentarily, so it is cleared in the next operation.
3. Set the ATNREQ bit of the control register to 0.

At this time, the  $\overline{ATNn}$  signal on the backplane will be asserted.

There is no way for the TMS320 to directly monitor the state of the attention request (to see whether it has been cleared by the control processor.) However, the user may provide a signal to the TMS320 by writing a value to the backplane-to-DSP FIFO, or the TMS320 can monitor the Empty Flag ( $\overline{EF}$ ) of the DSP-to-backplane FIFO to determine when the control processor has read out all of the waiting data.

- Enabling of Interprocessor Mailbox interrupts.

The Mailbox described in drawing DSP-0006 can generate interrupts for two distinct classes of event:

1. The DSP-to-backplane FIFO is empty.  
This implies that the next set of data can be written to the FIFO.
2. The backplane-to-DSP FIFO is full.  
This means that there is a word waiting to be read from that port. Since this FIFO is only 1 word deep, the DSP should read the value out of the port promptly.

Each of these interrupts is enabled separately by bits in the DSP-accessible control register.

- Selection of  $\overline{BIO}$  signal source.

The  $\overline{BIO}$  signal input of the TMS320C25 provides a convenient, software testable input. It is only one bit wide though, so a multiplexor has been used to allow it to sample one of 4 signals, as described in table A.18.

This multiplexor, together with the  $\overline{BIO}$  input of the TMS320C25, form the DSP-accessible status register discussed in drawing DSP-0008.

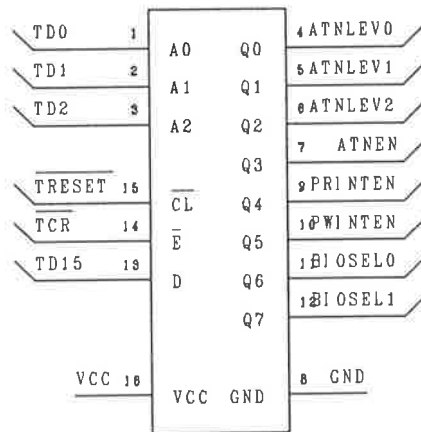
BIOSEL1	BIOSEL0	$\overline{\text{BIO}}$ Source
0	0	$\overline{\text{BIO}}$ signal from AIM connector
0	1	<b>EF</b> (Empty Flag) from DSP-to-backplane FIFO.
1	0	<b>FF</b> (Full Flag) from DSP-to-backplane FIFO.
1	1	FULLFLAG from backplane-to-DSP FIFO

Table A.18: TMS320C25  $\overline{\text{BIO}}$  signal sources

Value written to register	Effect
\$0000	Clear Attention Level bit 0 (ARSB0)
\$8000	Set Attention Level bit 0 (ARSB0)
\$0001	Clear Attention Level bit 1 (ARSB1)
\$8001	Set Attention Level bit 1 (ARSB1)
\$0002	Clear Attention Level bit 2 (ARSB2)
\$8002	Set Attention Level bit 2 (ARSB2)
\$0003	Attention Request remains unchanged (ATNREQ)
\$8003	Force Attention Request assertion (ATNREQ)
\$0004	Disable backplane-to-DSP FIFO interrupt
\$8004	Enable backplane-to-DSP FIFO interrupt
\$0005	Disable DSP-to-backplane FIFO interrupt
\$8005	Enable DSP-to-backplane FIFO interrupt
\$0006	Clear $\overline{\text{BIO}}$ source select bit 0 (BIOSEL0)
\$8006	Set $\overline{\text{BIO}}$ source select bit 0 (BIOSEL0)
\$0007	Clear $\overline{\text{BIO}}$ source select bit 1 (BIOSEL1)
\$8007	Set $\overline{\text{BIO}}$ source select bit 1 (BIOSEL1)

Table A.19: DSP-accessible control register bit assignments

### CONTROL



74LS259

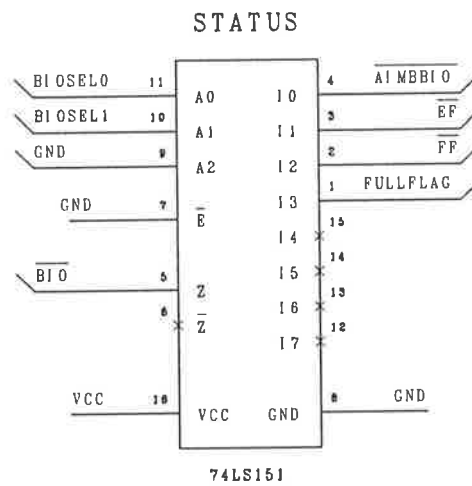
## A.2.10 DSP-0008

### DSP-accessible Status Register.

The DSP-accessible status register allows the TMS320 to monitor the state of three signals within the module, and one signal from the attached AIM. Only one of these 4 signals may be sampled at one time, as they are fed through a multiplexor to the TMS320's  $\overline{\text{BIO}}$  signal input.

The multiplexor is controlled from two bits in the DSP-accessible control register, as described in drawing DSP-0008. To use the status register, the TMS320 must first set these two bits to select the desired signal to be multiplexed onto the  $\overline{\text{BIO}}$  line.

The DSP software may then use the  $\text{BIOZ}$  instruction to test the state of that signal.



## A.2.11 DSP-0009

### DSP Module Interrupts.

The DSP module uses interrupts to alert the TMS320C25 of certain circuit conditions, these being:

1. The arrival of a word at the backplane-to-DSP FIFO.
2. The DSP-to-backplane FIFO being emptied by the control processor.

Each of these interrupts are enabled by a bit in the DSP-accessible control register described in drawing DSP-0007.

The interrupt signals for these two conditions will be active for as long as the circuit condition exists (ie while the word is waiting to be read from the backplane-to-DSP FIFO, or while the DSP-to-backplane FIFO is empty), so that the TMS320C25 should reenable interrupts only after it has cleared the circuit condition which caused the interrupt.

Interrupts may also be generated by the Analogue Interface Module.

TMS320 Interrupt	Causing Signal	Enabling Signal	Comment
$\overline{TINT0}$	$\overline{EF}$	PWINTEN	DSP-to-backplane FIFO empty
$\overline{TINT1}$	FULLFLAG	PRINTEN	backplane-to-DSP FIFO full

Table A.20: TMS320C25 onboard interrupt sources

The Interrupt Enable signals mentioned in table A.20 originate from the DSP-accessible control register (see DSP-0007). The signals which cause the interrupts are associated with the interprocessor mailbox (see DSP-0006).

The interrupt signals  $\overline{TINT0}$ ,  $\overline{TINT1}$ ,  $\overline{TINT2}$  may all be generated by the AIM. In this case the user should program the DSP-accessible control register to disable the onboard interrupts. The signals which cause the onboard interrupts can still be monitored by the TMS320 via the DSP-accessible status register (see DSP-0008).



## A.2.12 DSP-0010

### DSP Module Attention Requests.

The DSP module may request attention from the control processor board in the system by asserting any one of 8 attention request signals on the backplane bus (  $\overline{ATN0}$  ...  $\overline{ATN7}$  ).

The method used to assert the desired attention request signal is described in CON-0007.

The control processor clears the attention request (effectively acknowledging it) by writing to the memory address reserved for this in the DSP module control area (see table A.15.)

```
Name      atnasert;
Partno    20001;
Date      25/9/89;
Revision  01;
Designer  GV;
Device    p1618;
Format    j;
```

```
/* SHEET DSP-0010 */
```

```
/* Misc. functions including tristating of status MUX and attention assert
   control flip-flop. */
```

```
Pin 1 = ATNEN;
Pin 2 = !BATNACK;
Pin 3 = STATUS;
Pin 4 = !BSR;
Pin 5 = TA13;
Pin 6 = PRAMTYPE; /* 1 -> 32K RAM, 0 -> 8K RAM */
Pin 7 = RAMTYPE; /* AS ABOVE */
Pin 8 = ONBDRDY;
```

```
Pin 19 = BUFD15;
Pin 18 = !ATNASRT;
Pin 17 = PRAMP26;
Pin 16 = DRAMP26;
Pin 15 = RDY;
Pin 14 = TRW;
Pin 13 = !TSTRB;
Pin 12 = NEWTRW;
```

```
ATNASRT = ATNEN #
          ATNASRT & !BATNACK;
```

```
BUFD15 = STATUS;
BUFD15.OE = BSR;
```

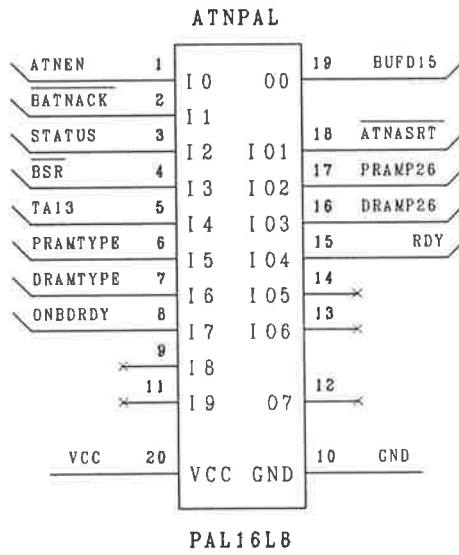
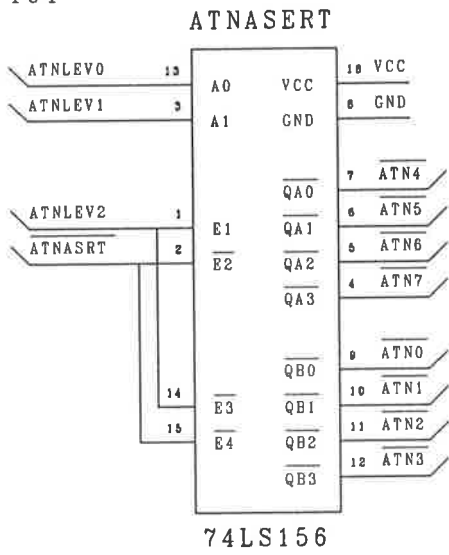
```
PRAMP26 = !RAMTYPE #
          RAMTYPE & TA13;
```

```
DRAMP26 = !RAMTYPE #
          RAMTYPE & TA13;
```

```
RDY.OE = !ONBDRDY;
RDY = 'b'0;
```

```
NEWTRW = !TSTRB # TRW ; /* only get write pulse when TSTRB active */
```

101



### A.2.13 DSP-0011

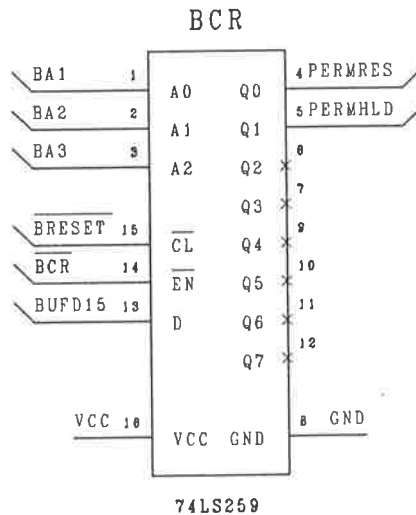
#### DSP Module Control Register.

The Module Control Register (MCR) is a single bit wide register to which only the backplane bus master has access. There are 2 individually addressable bits in the register, which directly control the state of two signals within the DSP module.

The states of the two bit wide words are set by writing a 16-bit word to the addresses shown in table A.19. Bit 15 of that word contains the value that will be written to the register.

Address Offset	Signal	Function
\$008000	PERMRES	Directly controls the state of the TMS320C25's $\overline{\text{RESET}}$ signal.
\$008002	PERMHLD	Directly affects the state of the TMS320C25's $\overline{\text{HOLD}}$ signal. If $\overline{\text{PERMHLD}}$ is asserted (ie low), the $\overline{\text{HOLD}}$ signal to the TMS320 is asserted.

Table A.21: DSP Module Control Register  
The Address offset is relative to the DSP Module base address.



### A.2.14 DSP-0012

#### DSP Module Status Register.

By reading from this register, the backplane master can determine the state of the DSP board at any time, without interfering with DSP program execution at all.

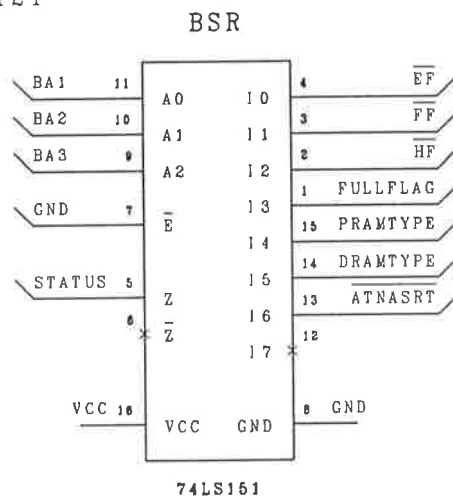
The module status register consists of 8 single bit registers which are read from the addresses shown in table A.22.

Address	Signal	Comment
\$004000	$\overline{EF}$	Empty Flag from the DSP-to- backplane FIFO.
\$004002	$FF$	Full Flag from the DSP-to- backplane FIFO.
\$004004	$HF$	Half Full flag from the DSP-to- backplane FIFO.
\$004006	FULLFLAG	Indicates full state of backplane-to-DSP FIFO.
\$004008	Not Used.	
\$00400A	DRAMTYPE	Indicates type of SRAM chips installed in DSP board (see DSP-0005).
\$00400C	$\overline{ATNASRT}$	If asserted, this signal indicates that this DSP module is requesting attention (see DSP-0010).
\$00400E	Not Used.	

Table A.22: DSP Module Status Register  
The Address is relative to the DSP module base address.

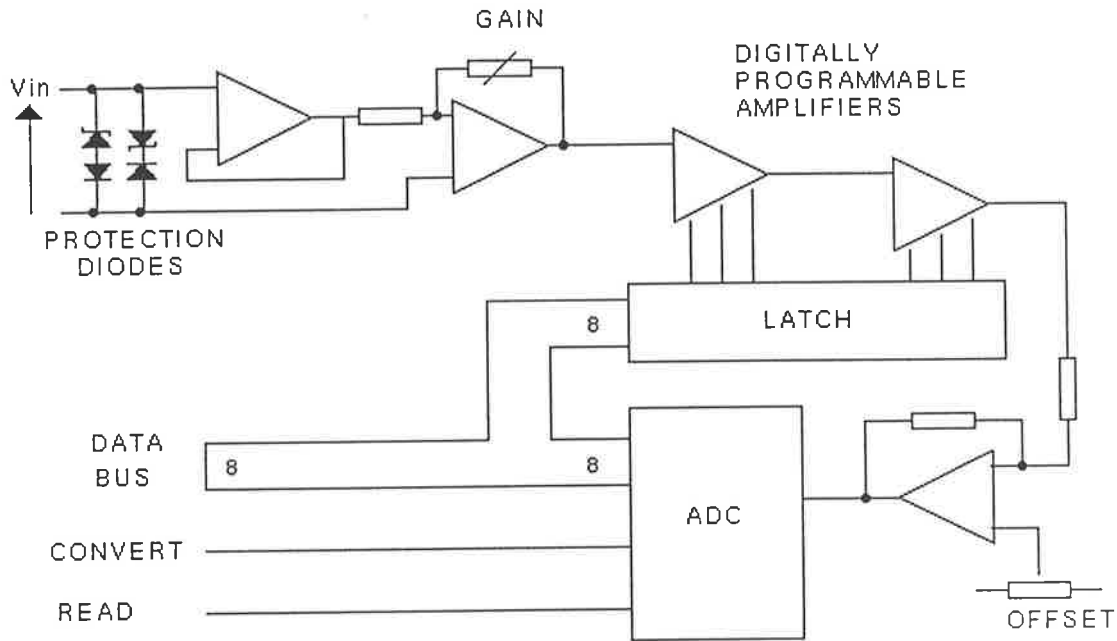
The control processor reads a 16-bit word from the address shown in table A.22, and the value of bit 15 of that word is the value of the corresponding signal.

121



### A.3 A Prototype Analogue Interface Module

The drawings in figure A.1 are not detailed, since this module is undergoing redesign at the time of writing. Detailed drawings are available from the technical staff in the department.



(a)

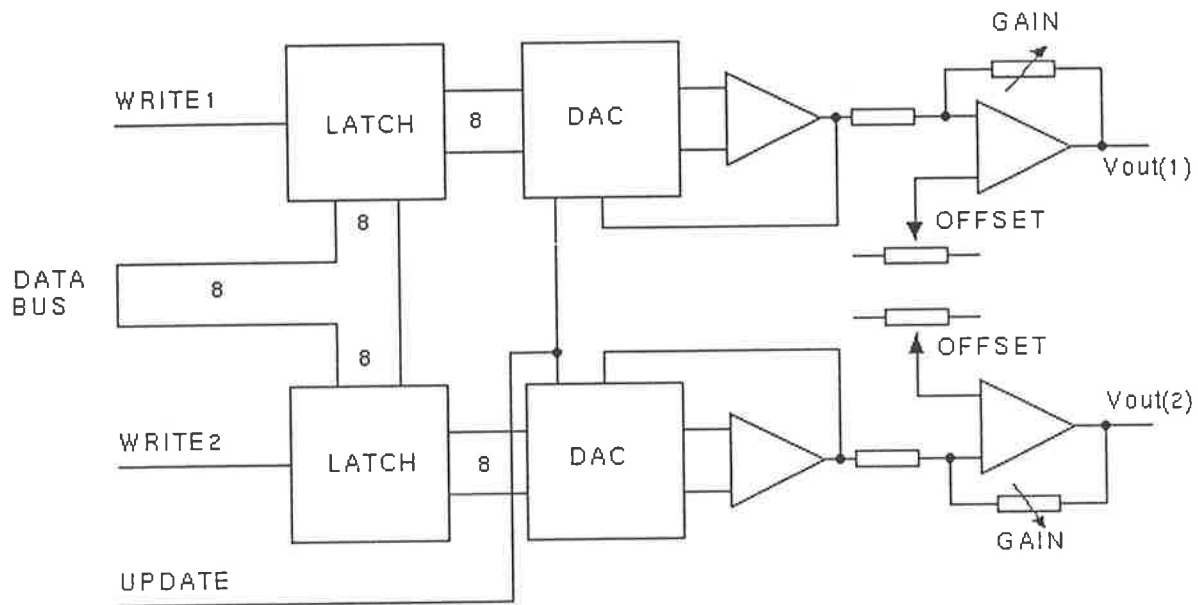
The design is for a prototype analogue interface module. The module provides only very basic facilities, and during testing and use clearly demonstrated the need for complete digital control of more circuit parameters, including anti-aliasing filtering and programmable offset removal. No such features are present on this prototype.

While quite suitable for applications such as the student laboratories, the prototype AIM has certain flaws which must be removed in a future device. These include:

- The lack of anti-aliasing filtering. While it is difficult (if not impossible) to provide analogue anti-aliasing filtering which can be used over the entire range of possible sampling rates, it must be provided in some manner. Digitally programmable switched-capacitor filters offer some promise in this area.
- The lack of programmable voltage offset control. If the signal being digitised has a small AC part superimposed on a large DC component, the system has no way to remove the DC component.

To keep within the ADC input range, a low value of gain must be used, resulting in excessive quantising noise for the AC signal.

Offsets should be removed by subtraction of a DC reference signal. The subtraction should be done as close to the ADC signal input pin as possible, to allow any accumulating voltage



(b)

Figure A.1: Prototype AIM block diagram  
 (a) Input circuitry (one channel shown)  
 (b) Output circuitry (both channels)

Input Channels	2×8-bit.
Input conversion time	50ns
Input sampling	Simultaneous
Input Gain	Digitally programmable $gain = [1, 2, 4, 8, 16] \times [1, 2, 4, 8, 16]$ Only one value is chosen out of the set in [ ]
Input Bandwidth	approx 350kHz
Anti-aliasing filter	None
Output Channels	2×8-bit.
Output updating	Simultaneous

Table A.23: Prototype AIM Specifications



<i>Port Number</i>	<i>Function</i>
Read 10	Start Conversion Signal to both ADCs
Read 11	Read contents of ADC 2 (input channel 2)
Read 12	Read contents of ADC 1 (input channel 1)
Write 10	Gain for ADC 1
Write 11	Load both DACs
Write 12	Next value for DAC 1 (output channel 1)
Write 13	Next value for DAC 2 (output channel 2)
Write 14	Gain for ADC 2

Table A.24: Production IOM Registers

offsets in the analogue circuitry to be removed at the last stage. This is especially important for the programmable gain amplifiers (PGA), which amplify not only the signal by DC offsets within the circuit.

- Too much calibration required. The prototype AIM requires calibration of output voltage ranges and offsets, and of input offsets. Such calibrations are time consuming and prone to error. A production AIM should include fixed, high-precision references in fixed circuit configurations which will not require calibration by a technician.

The DSP IO port addresses for the various components of the AIM are as follows. Note that these addresses are for the prototype GI boards (the single board version), and may be different for the production boards, depending on how the production boards are interfaced to the AIM.

The Analogue to Digital Converters (ADCs) are 8-bit flash converters, and are linked to the data bus as follows:

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	...	Bit 0
ADC	ADC	ADC	ADC	ADC	ADC	ADC	ADC	Not Used	...	Not Used
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Used	...	Used

The ADCs share a common strobe signal which starts the conversion process, so that simultaneous samples of both channels may be made.

The programmable gain amplifiers (PGA) are mapped into their registers as follows.

Bit 15	...	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used	...	Not Used	Not Used	ILE	PGA 1 Bit 2	PGA 1 Bit 1	PGA 1 Bit 0	PGA 0 Bit 2	PGA 0 Bit 1	PGA 0 Bit 0

There are two PGAs in direct cascade for each of the ADCs. PGA 0 is the first in the pair, PGA 1 is the second, feeding into the ADC. Each of the PGAs has the following gains for each 3 digit code in the above register.

The ILE signal (above) is connected to a pin of the DAC0830 converters with a similar name. This pin is not used on the DAC0830, but is used on 12-bit converters (which are pin compatible with the DAC0830) to select between low-byte and high-byte programming. In the current IOM the ILE bit must be set to 1 for correct operation.

<i>Bit 2</i>	<i>Bit 1</i>	<i>Bit 0</i>	<i>Resulting Gain</i>
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	βX	βX	16

Table A.25: AD526 Gain Settings X = Don't Care

The digital to analogue converters are 8-bit devices, and are connected to bits 0-7 of the data buss. This means that only numbers between 0-255 should be written to those ports.

Since there are two PGAs in cascade, any product of the above gains is possible, resulting in a coarse range of 1-256. A trimmable gain op-amp precedes the PGAs, allowing a symmetrical gain range of 0.0625 (1/16) to 16 to be implemented.

Typically, only the low gains of the PGAs should be used, since the amplifiers amplify not only signals, but DC offset voltages within the analogue circuitry, leading to a reduction of effective dynamic range of the converters.

To read the ADCs, code such as the following may be used:

```

*
* EXAMPLE CODE FOR READING THE ADC ON PRODUCTION IOM.
*
ADC_start  =    10 ; read this location to start conversion
ADC2_data  =    11 ; read value of channel 2
ADC1_data  =    12 ; read value of channel 1
*
temp       =          ; dummy data location
val0       =          ; data location to store ADC 0 value
val1       =          ; data location to store ADC 1 value
*
READ_ADC:  IN    temp,ADC_start ; trigger ADCs
           IN    val0,ADC0_data ; get ADC 0 value
           IN    val1,ADC1_data ; get ADC 1 value
*
*

```

*Notes:*

- The lower 8 bits of the 16-bit locations occupied by the ADCs are floating and should be masked in software (by anding with \$FF00, say) before the values are used.
- The ADCs are flash converters, and have valid data available about 50ns after triggering. Thus there is no need to wait for a conversion complete signal.
- There is at present no hardware to generate a sampling clock. This must be generated by software timing at present. The internal timer of the TMS320C25 may be used to generate sampling intervals.

## A.4 Guidelines for Designing New Modules

The designer of a new module for the Generalised Instrument must provide a number of basic services on that module. These services are required if the module is to communicate in any way with the GI's backplane bus.

### 1. Address decoding. (Mandatory only for modules which interact with the bus.)

Notionally, the backplane address space of 16MB is divided in 32 512kB blocks. Boards are assumed to occupy a minimum of 512kB of address space. Since it is unlikely that the full complement of 32 boards will ever be present, there is sufficient memory space available for such a seemingly wasteful scheme.

The reason behind this scheme is that boards should begin on a 512kB boundary, so that the control processor can scan the 32 memory 'slots' and attempt to identify the boards present. This autoconfiguration feature is not presently implemented (though the necessary hardware is present on the TMS320C25 module), but may be in the future. Each board would have an IDENTITY PAL which can be read by the control processor, allowing identification of boards present.

The address decoding for the board therefore relies on A23-A19 of the backplane. Addresses in the range \$000000-\$07FFFF may not be decoded, since they are decoded within the control processor module.

On current modules, the decoded address is set with jumpers. The DSP module is an example, as shown in drawing DSP-0003.

### 2. Assertion of $\overline{DTACK}$ . (Mandatory)

The backplane has a **DaTa ACK**nowledge signal called  $\overline{DTACK}$  which *must* be asserted by the module when a valid module address has been decoded, and the bus cycle may be terminated.

Failure to assert the  $\overline{DTACK}$  signal will lead to a bus timeout after tens of microseconds, causing the control processor to begin exception processing. The circuitry which detects this condition is documented in drawing CON-0005.

### 3. Assertion of Attention Request. (Optional)

The backplane bus provides an 8-level attention request mechanism. This is akin to an 8-level interrupt system. Priority coding and vectoring is performed within the control processor module.

### 4. Access time of devices on the bus.

In order to run without wait-states, devices must conform to certain timing requirements. A general rule of thumb is that the total backplane strobe active duration is 2 control processor clock cycles.

For instance, if the control processor board (assuming 68000 based) is running at 10MHz CPU clock rate, the maximum duration of active  $\overline{AS}$  on the backplane will be 200ns.

The designer must further subtract the delay incurred by her board decoder circuit, and buffers, to determine whether the board will be able to operate without wait-states.

## Appendix B

# Control Processor Onboard Software

The Control Processor operates in two distinct modes: the *command mode* and the *interactive mode*. The interactive mode is designed to allow direct interaction with the user via a suitable terminal (provided by SPaM). The interactive mode is similar in operation to the debug monitors found on many microprocessor systems. The interactive mode of the CP expects ASCII input and output over the RS232 connection to the host.

The command mode uses a binary-only communication protocol, and is designed for efficient and reliable communication between the CP and a host (the PC) computer. Unlike the interactive mode, the command mode uses full-binary representation of numbers.

On powering up the system, the default mode of operation of the control processor (CP) is the command mode. To enter the interactive mode, the user should run SPaM on the host, and use the `mon` command (see 5.6.) The commands available in the interactive mode are explained in detail in the sections which follow. There are some points of interest about the power on sequence which will be noted here.

### B.1 Interactive Monitor Commands

The monitor commands consist of a command word with optional arguments. Arguments are either necessary or optional. If a necessary argument is omitted from the command line, its value is set to *zero* when the command is executed. The following symbols are used to define arguments.

< > surround a necessary argument.

example: `baud <baud-rate>`  
states that the command `baud` be given an argument.

[ ] surround an optional argument.

example: `port [value]`  
states that the command `port` has an optional argument. The command will perform different actions depending on whether an argument is given or not.

| means 'or'. It is used to separate several alternative values.

example: `hold <0|1>`

states that the hold command has one necessary argument which must be either the number 0 or 1.

### B.1.1 Memory Displaying Commands

```
db <start-address> [end-address]
dw <start-address> [end-address]
tpdw <start-address> [end-address]
tddw <start-address> [end-address]
tidw <io-address>
xb <address>
xw <address>
```

**db** displays bytes in 68000 address space, beginning from the highest 16 byte address boundary less than or equal to `<start-address>`, and continues to display memory bytes until it reaches the lowest 16byte boundary greater than or equal to `[end-address]`. If `[end-address]` is omitted, 16 bytes will be displayed.

example:

```
command > db 100 120
00000100: - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000110: - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000120: - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
command >
```

**dw** displays memory words in 68000 address space. Rules are the same as for **db**. Note that since this command operates on 68000 address space, the addresses supplied as arguments are *byte* addresses<sup>1</sup>

example:

```
command > dw 100 120
00000100: - 0000 0000 0000 0000 0000 0000 0000 0000
00000110: - 0000 0000 0000 0000 0000 0000 0000 0000
00000120: - 0000 0000 0000 0000 0000 0000 0000 0000
command >
```

**tpdw** displays words in DSP program address space. Valid addresses are in the range \$0000 to \$FFFF. Note that since we are referring to DSP space (which is comprised only of *words*), these addresses are *word* addresses.

example:

```
command > tpdw 0 20
TMS320C25 Program memory words - (ESC) to quit, any key to pause
00000000: - CE06 C800 CA00 6060 E060 FE80 0018 CA01
00000008: - 6060 E060 FE80 0018 CA02 6060 E060 FE80
00000010: - 0018 CA03 6060 E060 FE80 0018 FF80 0002
00000018: - D101 FFFF CD01 F580 001A CE26 5465 7374
```

<sup>1</sup>compare this with `tpdw` and related commands where they are *word* addresses

```
00000020: - 696E 6720 3638 6B20 5241 4D20 6E6F 772E
command >
```

**tddw** displays words in DSP data address space. Valid addresses are \$0000 to \$FFFF.

**tidw** displays the contents of a location in DSP IO space. Valid addresses here are \$0000-\$000F. Only one value is displayed, that of the specified address <sup>2</sup>.

example:

```
command > tidw 0 10
CD10
command >
```

**xb** displays the contents of one byte in the 68000 address space. The byte value is displayed to the host, the cursor is then backspaced over the value on screen, and an updated value is written over it. This refreshing process continues until a key is hit, at which time control is returned to the monitor program.

**xw** displays the contents of a *word* in 68000 address space in a similar manner to **xb**, which displays bytes. Remember that since this command operates on the 68000 address space, the address argument is a *byte* address.

### B.1.2 Memory Modifying Commands

```
mb <start-address>
mw <start-address>
tpmw <start-address>
tdmw <start-address>
timw <io-address> <value>
```

**mb** modifies bytes in 68000 address space, starting at <start-address>. The byte contents of each consecutive address are displayed, and the user can do one of three things:-

1. Type in a new hex byte value for that location.
2. Hit Return to leave current address value unmodified, and proceed to next byte address.
3. Type **q**↔ to return to monitor without affecting current address contents.

example:

**mw** modifies words in 68000 address space, according to the same rules as **db** modifies bytes.

example:

```
command > mw 3000
Enter <CR> to skip to next word, or new value, or q to quit
00003000: - 202D abdd
00003002: - 4F2D 1234
00003004: - 2D20 a12b
00003006: - 207C q
command >
```

---

<sup>2</sup> this prevents corruption of neighbouring IO registers

**tpmw** modifies words in DSP program address space. Valid values for <start-address> are \$0000 to \$FFFF. The procedure is then the same as for **mb**. Note that it is wise to HOLD the TMS320C25 while such editing is in progress to avoid losing control of the DSP module.

example:

```
command > tpmw 100
TMS320C25 program memory space edit - enter value, or q, or <CR>.
00000100: - 5555 aaaa
00000101: - 5555 dddd
00000102: - 5555 q
command >
```

**tdmw** modifies words in DSP data address space. Valid values for <start-address> are as for **tpmw**.

See **tpmw** for an example.

**timw** writes the value given to a single location in DSP IO space. The IO space may access different devices at the same IO address, depending on whether a read or a write is performed, so that it may not be possible to read the written value back into the DSP.

See **tpmw** for an example.

#### Notes

- Although specific instructions are provided for manipulating words in DSP address space, the standard **dw,mw** commands can be used if the relationship between 68000 address space and DSP address space is kept in mind.

If, for example, one wishes to examine location \$0020 in DSP data address space, then the following arithmetic must be performed to arrive at the correct address in 68000 address space, which is used as the argument to **dw** or **mw**.

\$0020 is a *word* address offset, so multiply by 2 to get the byte address \$0040.

Obtain the byte address in 68000 address space where the DSP data memory is mapped. From Table A.14 we see that this is \$040000 plus the base address of the DSP module. Assume the base address is \$800000, then the address in 68000 space of the DSP data memory is \$840000.

Add the offset to the base address to get the desired address in 68000 space, ie \$840040.

If we now give a command such as

```
dw 840040↔
```

then it is equivalent to giving one such as

```
tddw 40↔
```

since both refer to the same physical memory location.

- When examining words in DSP address space, using any of the above commands, decide whether it is necessary to halt the DSP before doing so. The memory displaying commands, and the memory modifying commands arbitrate dynamically for possession of the DSP busses while going about their business.

For example, the **tpdw** displays 8 words per line on the host screen. This means that during the time it took to display that line, it made eight accesses to DSP memory space. The DSP was tri-stated (effectively halted) during those accesses.

Since the amount of time taken to display the line of information is much longer than that needed to perform 8 memory accesses, the DSP was able to continue execution for most of the

time during which the `tpdw` command was being executed. It is conceivable that the algorithm begin executed by the DSP causes modification of the memory words which are being examined by the control processor, causing those words to lose their relationship between successive read operations.

It may therefore be wise to use the following instruction sequence to read information from DSP address space: `reset 1`↔  
`tddw 0040`↔  
`reset 0`↔

### B.1.3 DSP Module Control Commands

`reset` [0—1]  
`hold` <0—1>  
`port` [value]

`reset` controls the state of the RESET' signal to the DSP Module. The action performed on the RESET' signal is determined by the argument to this command.

`reset 0` causes the RESET' signal to the DSP to be deasserted (ie RESET'=1).

`reset 1` causes the RESET' signal to be asserted (ie RESET'=0).

`reset` with no arguments causes the RESET' signal to be asserted for several microseconds, then deasserted. This is useful for restarting the DSP, and is equivalent to issuing `reset 1` and `reset 0` commands in quick succession.

`hold` is one of two signals which determines the state of the HOLD' signal to the TMS320C25<sup>3</sup>.

The HOLD' signal, when asserted, causes the TMS320C25 to give up control of its external busses when execution of the current DSP instruction is complete. This HOLD' signal is also used for bus arbitration when the 68000 control processor invades DSP address space (as in response to `tpdw`, `tddw` ... commands).

The HOLD' signal provides a useful means of *halting* the TMS320C25, so that locations in its address spaces may be examined without risk of them changing (due to TMS320C25 program execution) during the examination.

`port` is the monitor command used for examining/modifying the contents of the interprocessor port (also called the interprocessor mailbox, see drawing DSP-0006). Values are written/read from the 68000 side of the bidirectional port only at present.

`port`↔ is the syntax used for examining the state of the port and its contents.

`port hhhh`↔ is the syntax for setting the port contents to hhhh, a hex number. Note that only the four least significant hex digits of hhhh will be used, since the port is 16 bits wide.

The state of the port will be returned first, and even if the previous value has not been read out by the DSP, the new one will be written in.

---

<sup>3</sup>The other is the decoded 68000 address strobe which indicates the 68000 is accessing a location in DSP address space, for which it must arbitrate using HOLD'/HOLDA'



## B.1.4 Commands for Data Transfer

baud <baud-rate>  
s  
sg  
lowtpd  
hightpd  
lowtdd  
hightdd

**baud** sets the baud rate of the serial link between host and CP. Valid values for <baud-rate> are 75,110,134,150,300,600, 1200,2000,2400,4800,1800,9600,19200.

**s** allows the downloading of data into 68000 scratch memory. Note that the s-record file is assumed to contain *bytes* which are stored to the 68000 scratch memory as consecutive *bytes*. This command will typically be used to download experimental code for the 68000 to execute, since most cross-assemblers for this processor output Motorola S-record files.

**sg** similar to **s**, but when downloading has finished, execution begins at the address specified in the S9 record of the downloaded file.

**lowtpd** downloads *bytes* into bits 0-7 of DSP program memory words using Intel Hex format. The addresses contained in the Intel Hex format file are assumed to be *word* addresses in DSP program memory. Bits 8-15 of the words are *not* affected.

Since the DSP memory is organised as 16-bit words (the bytes of which can not be address separately as in 68000), and Intel Hex format supports only bytes, then the original file of words (on the host computer) must be split into two files of bytes, one containing bits 0-7 (ie the *low-order bytes* of each word, and one containing bits 8-15 (the *high-order bytes*). The two files must then be downloaded separately using commands such as **lowtpd**, **hightpd**.

**hightpd** downloads *bytes* into bits 8-15 of DSP program memory words, using Intel Hex format. The addresses contained in the Intel Hex format are assumed to be *word* addresses in DSP program memory. Bits 0-7 of the referenced words are *not* affected.

This command is complementary to **lowtpd**.

**lowtdd** downloads *bytes* into bits 0-7 of words in DSP data memory using Intel Hex format. Bits 8-15 are not affected. This command is complementary to **hightdd**.

**hightdd** downloads *bytes* into bits 8-15 of words in DSP data memory, using Intel Hex format. Bits 0-7 are not affected. This command is complementary to **lowtdd**.

## B.1.5 System Control Commands

verbose <0—1>  
warm  
cold  
command

**verbose** determines whether text messages are displayed to the host. Such messages slow down communication, and are useful mainly only debugging of monitor code. An argument of 1 to this command enables the printing of response messages, while an argument of 0 disables them.

**warm** forces a warm restart of the CP. A warm restart is distinguished from a coldstart by the lack of memory testing (to prevent previously downloaded code from being corrupted).

**cold** forces a CP system coldstart. Note that RAM areas will be tested, corrupting *all* RAM contents.

**command** forces the CP control processor to leave the interactive monitor, and begin executing the binary communications software for automated communication with the host. This is described in section B.2.

## B.2 Command Mode

The command mode exists to allow more rapid and reliable communication between the host computer and the Control Processor (CP). All communication is done in packets, which come in several sizes, depending on what is being transmitted.

Command mode may be entered in one of two ways. When power is applied to the system, the CP will default to command mode. Should the user enter the interactive mode of the CP when using the host SPaM software, the CP will reenter the command mode when the user terminates the interactive session.

The first step in using the command mode to talk to the CP is to wait for the synchronizing character from the CP. The character is a '-' (ASCII value 45<sub>10</sub>). The host should continue to send a carriage-return (ASCII 13<sub>10</sub>) until it receives the synch character.

```
while received_character <> '-' do
  send_character(chr(13))
  wait(3 character times)
```

Note that a 'character time' is the length of time needed to transmit a 10bit character at the baud rate used. In the case of (default) 9600 baud communication, a character time is approximately 1 millisecond. Once it has received the synch character, the host can transmit a command packet to the CP. A command packet has the structure shown in Table B.1.

<i>Byte Offset</i>	<i>Value</i>	<i>Comment</i>
\$00-\$01	\$57BD	Command Packet Identifier
\$02	Command Byte	determines operation to perform
\$03	Modifier Byte	determines method of operation
\$04-\$05	Field 1	nature dependent on operation
\$06-\$07	Field 2	nature dependent on operation
\$08-\$09	Field 3	nature dependent on operation
\$0A-\$0B	Field 4	nature dependent on operation
\$0C-\$0D	CRC	used for error checking

Table B.1: Command Packet Structure  
CRC = Cyclic Redundancy Check

Within the command packet, the command byte determines which type of operation is to occur eg. reset the DSP. Following the command byte is the modifier byte, which determines how the operation is to be performed eg. assert RESET to DSP indefinitely.

The four word fields (8 bytes) which follow contain information which is interpreted differently by the various commands. The final value is a CRC (Cyclic Redundancy Check) word which is used to confirm the validity of the command packet. If the CRC sent is the same as the CRC calculated by the CP, then the command packet is assumed to be valid, an ACK character (ASCII 6<sub>10</sub>) is sent to the host, and the command is executed. If the CRC indicates that an error has occurred during the transmission of the packet, then a NAK (ASCII 21<sub>10</sub>) is sent to the host, and the CP will wait for the packet to be resent. *Note that the \$57BD Command Packet Identifier is not included in the CRC calculation.*

### B.2.1 Available Commands

The currently implemented commands are given in Table B.2 along with their corresponding byte values.

<i>Byte Value</i>	<i>Corresponding Command</i>
\$00	NULL Command
\$01	Reset Command
\$02	Hold Command
\$03	Download Command
\$04	Upload Command
\$05	Get IO Ports Command
\$06	Put IO Ports Command
\$07	Exit Command Mode
\$08	Change packet Size Command
\$09	Change baud rate Command
\$0A	Wait for Interprocessor Port Command
\$0B	Clear Interprocessor Port Command
\$0C	Change Timer-based Baud rate Command
\$0D	Write to Interprocessor Port Command

Table B.2: Valid Command Byte values

Some of the commands require one or more arguments to be passed. These arguments are passed in the modifier byte, and the four words which follow the modifier byte in the command packet. The arguments required for each command are listed in the following section.

#### NULL Command

The NULL command does absolutely nothing. When a command packet containing the NULL command is received by the Control Processor (CP), it causes the CP to wait for another command packet. This feature is implemented to catch command packets which have not been initialised properly (assuming a \$00 byte is more likely to occur in uninitialised memory than a real command value).

#### Reset Command

The Reset command performs the same set of functions as the Monitor `reset` command described in section B.1.3. That is, it determines the state of the `RESET` signal to the DSP module.

The modifier bytes for this command dictate whether the RESET' signal is asserted, deasserted or toggled (to simulate pressing and releasing an imaginary reset switch on the DSP module), and are listed in Table B.3.

<i>Modifier Byte Value</i>	<i>Result of Reset operation</i>
\$00	No Change
\$01	RESET' to DSP is asserted
\$02	RESET' to DSP is deasserted
\$03	RESET' to DSP is asserted then deasserted

Table B.3: Reset Command Modifier values

Fields 1...4 of the command packet are ignored by this command.

### Hold Command

The Hold command determines the state of the HOLD' signal (a bus arbitration signal) to the DSP module, and as such it performs a similar role to that of the interactive hold command detailed in section B.1.3.

The modifier values for this command determine whether the HOLD' signal will be asserted, or deasserted according to the assignments shown in Table B.4.

<i>Modifier Byte Value</i>	<i>Result of Hold Command</i>
\$00	No Change
\$01	HOLD' to DSP asserted
\$02	HOLD' to DSP deasserted

Table B.4: Hold Command Modifier Values

Fields 1...4 of the command packet are ignored by this command.

### Download Command

This command is the one to use when transferring hex (whether it be program or data information) from the host *to* the CP. This command can access any address space within the CP, and transfer any number of words in one operation.

The transmission is performed using packets, by default 128 words (256 bytes) in size, though this may be changed (see section B.2.1). Each packet is followed by a 16-bit CRC which the CP examines<sup>4</sup> to determine if each downloaded packet is valid.

As the packet is received, each word is stored to a consecutive address in memory, beginning at an address specified in the command packet which initiated the download operation. After each valid packet is received, the base storage address is incremented by the current size of transmission packets. The order of data within each packet is such that the data to be stored to the lower addresses is sent first, and that to be stored to the higher addresses is sent last. All storage operations are *word*

<sup>4</sup>The CP does this by calculating its own CRC based on the data in the packet, and comparing this to the one sent by the host

store operations performed by the 68000, and it is assumed that the high byte of each word arrives first over the serial link, and the low byte of each word then follows.

After receipt of a valid packet, the CP sends the host an ACK character to initiate the sending of the next packet. See figure 6.2 for a flow diagram of the download process.

Should the received CRC not agree with the calculated one, then the packet has somehow been corrupted during transmission and must be resent. The CP sends the host a NAK character to achieve this. The CP will continue to retry indefinitely: it is up to the host to count the number of retries and abandon transmission when this number exceeds some limit.

As stated before, the command modifier byte determines which address space is to receive the downloaded data, as given in Table B.5.

<i>Modifier Byte Value</i>	<i>Address Space used in Download</i>
\$00	None, operation aborted
\$01	DSP Program Address Space
\$02	DSP Data Address Space
\$03	DSP IO Address Space <sup>5</sup>
\$04	Control Processor Address Space
\$05	LMA Address Space

Table B.5: Download Command Modifier Values

Fields 1..4 in the command packet are *all* used by the download command, for the purposes shown in Table B.6.

Field 1	Bits 31-16 of word count
Field 2	Bits 15-0 of word count
Field 3	Bits 31-16 of address offset
Field 4	Bits 15-0 of address offset

Table B.6: Download Command Field Assignments

Fields 1 and 2 together form the 32bit number which determines how many 16-bit words will be downloaded from the host to the CP. Fields 3 and 4 together form a 32bit address *offset* which is added to a base address determined by the command modifier. The base addresses in 68000 address space are listed in Table A.14.

## Upload Command

The upload command works similarly to the download command, only the directions of data transfers are from CP to host. The CP sends packets of data to the host, each followed by a 16-bit CRC value. The host must decide whether the packets are valid. If the packet is valid, the host should send an ACK character, otherwise a NAK should be sent.

If the CP receives any character other than NAK or ACK, it will abort the upload and wait for a new command block. This makes recovering from an out- of-control upload easier.

Note that there is very little delay between characters in the upload (you should assume it to be no more than 1 character time).

The modifier byte values are those given in Table B.5, and the field usage within the command block is that given in Table B.6.

The sequence of events which occurs when an Upload command is issued is shown in figure 6.3.

### Get IO Ports Command

This command allows the host to read values from selected IO ports in the DSP module (ie locations in DSP IO space). Since these locations usually directly represent real-world devices, such as data-converters or latches, then reading or writing them is a non-trivial matter since it could result in system failure.

To prevent 'dangerous' locations from being accessed, the GetIO command allows the host to mask out those registers which it wants to read. The mask is a 16-bit word sent in the command packet. Each bit of the mask word corresponds directly to 1 of the 16 available Input ports in DSP IO space. Bit 15 corresponds to Input Port 15, Bit 0 corresponds to Input Port 0, and the intermediate ones have the same one-to-one relationship.

The mask word is contained in Field 1 of the command packet. The remaining fields of the command packet are not used. The modifier byte of command packet is ignored by this command.

If a particular bit of the mask word is a '1', then its corresponding Input port is read and the value is sent to the host. If the bit is a '0', then the Input port is *not* read and a zero (\$0000) word is sent to the host in its place.

After receipt of the command packet, the Control Processor (CP) examines the mask word, and proceeds to send to the host a packet of 16 words, followed by a CRC value. The 16 words correspond to the Input port values (wherever the mask bit was 1), or to zero words (where the mask bit was 0). The order of transmission is high byte of Input port 15's word value first, followed by the low byte of Input port 15's word value, and so on until the last byte which is Input port 0's low byte. The CRC value then follows the 16 words (32 bytes) of this packet. If the host responds with an ACK character, the CP assumes the GetIO operation was completed successfully, and waits for another command packet. If a NAK is received by the CP, the entire IO block is resent <sup>6</sup>.

### Put IO Ports Command

This command does the reverse of GetIO, in that it waits for data to arrive from the host which it then sends to the Output ports (specified again by the mask word) in DSP IO space. After receiving the command packet holding this command, the Control Processor (CP) examines the mask word within the command packet. It then waits for a packet of 16 words (the IO packet) to be sent by the host, followed by a CRC value.

The mask word is contained in Field 1 of the command packet. The remaining fields of the command packet are not used. The modifier byte of the command packet is ignored by this command.

As the words of the IO packet arrive, the CP checks the corresponding bit of the mask word to see if the newly arrived word should be stored to an Output port. If the corresponding bit of the mask is a 1, then the word is stored to the corresponding Output port<sup>7</sup>. Note the words arrive at the CP in the order of Output port 15's value first (high byte then low byte), and Output port 0's value last.

<sup>6</sup>Note that at present, the IO ports are not buffered which could result in different values being present on those ports when a re-transmit operation begins following a NAK

<sup>7</sup>Note that since this operation is currently not buffered, words are written to Output ports before the validity of the IO packet is checked using the CRC

## Exit Command Mode

For debugging purposes, this command has been included to allow the host to enter an interactive session with the Monitor on the Control Processor (CP). To re-enter command mode, the following should be entered into the monitor:

```
verbose 0↔  
command↔
```

## Change Packet-Size Command

The Upload and Download commands use data packets with a default size of 128 words (256 bytes). The user can change this value by using the Change Packet-Size command. The size of the packet can be anywhere between 1...65536 words. The size of the packet to use is determined by several criteria:

- Larger packets result in higher throughput on links where transmission is largely error free, but lower throughput on links which have a high error rate.
- Small packets should be used when small amounts of data are being moved. In all cases, a whole number of packets is sent, so using 1000 word packets to send 10 words is a waste of time (and increases the opportunity for errors to occur).
- Larger packets should be used to reduce handshaking overhead on links which introduce a transmission delay (modems for instance).

The modifier byte is ignored by this command. Field 1 in the command packet is the word value representing the packet size to use in future transfers <sup>8</sup>. A word value of \$0000 represents 65536 word packets, \$0001 represents 1 word packets, ..., and \$FFFF represents 65535 word packets.

## Change Baud-Rate Command

In certain circumstances it will be necessary to make the Command Packet (CP) work at a different baud rate to 9600 baud used by default. If the host PC is not capable of reliable 9600 baud communication (or is capable of *higher* speed communication), then it should send a command packet to the CP to change the baud rate.

The modifier byte is ignored for this command. Field 1 and 2 combine to determine the baud rate in the way shown in Figure B.7.

Fields 3 and 4 are not used. The baud rate used by the CP will be changed immediately after the ACK for the command packet is sent. The host should change baud rate after it receives the ACK character.

## Change Timer-based Baud Rate Command

The previous command described how to change the serial-link baud rate by specifying the desired baud rate directly. This only allows baud rates up to 19200 baud to be selected.

---

<sup>8</sup>Note that the GetIO and PutIO commands always send 16word IO packets, only the Upload and Download commands have variable-length packets

<i>Baud Rate</i>	<i>Field 1 Value</i>	<i>Field 2 Value</i>
75	\$0000	\$0075
110	\$0000	\$0110
134	\$0000	\$0134
150	\$0000	\$0150
300	\$0000	\$0300
600	\$0000	\$0600
1200	\$0000	\$1200
2000	\$0000	\$2000
2400	\$0000	\$2400
4800	\$0000	\$4800
1800	\$0000	\$1800
9600	\$0000	\$9600
19200	\$0001	\$9200

Table B.7: Set-Baud-Rate Command Field Allocations

To achieve higher baud rates, it is necessary to bypass the internal baud rate generator of the MC68681 and use the internal timer to generate the data clock. To facilitate this, a command was created which allows the serial link to be switched to timer-based data clocking, and the word value in Field 1 of the command packet is used as the 16-bit timer constant [10].

The lower the value placed in the timer register, the higher the resulting data clock frequency. The minimum value is  $2_{16}$ , which gives a 57600bps serial link speed. Higher speeds are achievable, but require some hardware. Specifically, an output pin of the 68681 must be programmed to be driven by the timer output, and this must be fed to an input pin (input and output pins belong to the parallel port on the device) of the 68681. The serial port can then be programmed to accept its clock from the input port.

Such a roundabout method is necessary to bypass the divide-by-16 counter which normally exists between the data-clock source and the serial port. Using this method, we can go to the maximum data rate of 115200bps, which is the maximum supported by the standard IBM PC serial port.

Fields 2-4 are not used in this command. The **Change Baud Rate** and **Change Timer-based Baud Rate** commands may be used interchangeably.

### Clear Interprocessor Port Command

This command forces the 68000 to read from the 16-bit interprocessor port, thus clearing the flag associated with that port. The value read is discarded. This command has no arguments.

### Wait for Interprocessor Port Command

After receiving this command, the 68000 will wait for 16-bit data words to arrive at the interprocessor (68000←DSP) port. It will read the word from the port, and send it to the host via the serial link.

The number of words to be sent back to the host is passed as a 32-bit value in Fields 1-2 (high word in Field 1, low in Field 2). Once the correct number of words has arrived and been sent on, the 68000 sends the CRC value for all of the returned words. The 68000 does *not* wait for an acknowledge from the host before returning to the command loop, it does so immediately.



Thus, if the returned words contained a transmission error (as indicated by disagreeing CRCs), the error can be either ignored, or the whole operation must be resumed. There is *no* retransmission on NAK for this command.

Note also that there is no fixed time limit on when words arrive at the interprocessor port, this is up to the DSP code author.

Requesting a read of 1 word from the interprocessor port is a useful way of detecting completion of DSP algorithm. The DSP code can be written so that the last instruction causes the DSP to write a word to the interprocessor port. The 68000 detects this, and sends the word on to the host. The arrival of the word indicates to the host that processing has terminated, so that processed data can be uploaded.

Before issuing this command, a command should be sent to the CP to clear the interprocessor port.

### **Write to Interprocessor Port Command**

The 16-bit word in Field 1 of the command packet is written to the interprocessor (68000→DSP) port. At present, no check is made to determine whether the port is empty, thus existing data may be overwritten.

## Appendix C

# Creating a Virtual Instrument Using SPaM

When writing applications which involve the DSP module, it is necessary to follow some conventions if they are to function correctly with SPaM. These conventions are listed below:

1. The TMS320C25 code must be in pure binary format. This is accomplished by using the `INTL2BIN.EXE` program supplied with SPaM to generate the binary file from the Intel format `MYPROG.LO` and `MYPROG.HI` files.

The chain of files created by certain utilities is shown below.

MYPROG.ASM	User's source file
↓	Assembler (DSPA.EXE)
MYPROG.OBJ	Object file
↓	Loader (DSPROM.EXE -i MYPROG)
MYPROG.LO	Intel Hex Format files
MYPROG.HI	Intel to Binary converter (INTL2BIN.EXE)
↓	
MYPROG.BIN	Suitable for use with SPaM

2. When SPaM's `send` statement is used, the `MYPROG.BIN` file is loaded from disk, and downloaded to the program memory of the DSP module.

When the `restart` statement is used, SPaM turns off the RESET signal in the DSP module, and the signal processor begins execution of its code.

The `restart` statement does not immediately return control to the user (or the script). It waits for the signal processor to indicate that processing has finished. This allows the host and the DSP module to synchronise, so that there is no doubt about valid data being available for uploading to the host when the `restart` statement has returned.

To indicate the completion of code execution, the signal processor must enable its attention request signal. This is described in detail in section A.2.9, and in the following code example.

3. If the user is writing real time code which will commence processing as soon as it is downloaded to the DSP module, and will continue processing indefinitely, she should ensure that the code asserts its attention signal immediately, before entering its main processing loop.

In this way, the `restart` statement in SPaM can still be used to begin execution. Note that there is no restriction on the activity of the signal processor after it requests attention. The control processor (at present) merely uses that request to synchronise the host to the completion of DSP code.

4. At present, the data transfer functions in SPaM such as `download()` and `upload()` first place the signal processor in a HOLD state which causes it to cease execution. The data is then transferred. After the transfer, the signal processor is not allowed to resume execution.

A more transparent method of access will be incorporated into the onboard GI software in the near future. At present, the only method by which data can be transferred from DSP memory during real-time DSP processing is to use the interprocessor port, which is not directly accessible through SPaM commands, yet.

The following SPaM script implements a combined signal digitiser and frequency analyser, whose screen display is shown in figure C.1.

```
% This script implements a Waveform Digitiser and Spectrum Analyser
%
% First, do some initialising of variables.
%
FFT=[1,1];
Signal=[1,1];
z=int(zero(1,2048));
samp_rate=100000;
samp_divisor=100;
old_samp_rate=100;
half_sampling_rate=50000;
go = 0;
gos=1;
gof=1;
Gain=1;
%
% Now enter graphic display mode with a small console window
%
graphic 600 50
%
% The following handler is not attached to a button, but is instead
% used as a 'subroutine' by other handlers.
%
handler looper
    if(samp_rate!=old_samp_rate)
        samp_divisor=int(10000000.0/samp_rate);
        samp_rate=10000000.0/samp_divisor;
        pdownload(int(samp_divisor-1),26);
        old_samp_rate=samp_rate;
        half_sampling_rate=samp_rate/2.0;
        update
    end
    if(gos==1)
        pdownload(int(-32768),30);
        restart
        Signal=upload(4096,4096+1023);
        end
    if(gof==1)
        pdownload(int(32768+16384+8192+4096+2048),30);
        restart
        FFT=upload(4096,4096+511);
        end
    update
end
%
% The handler 'AUTO' is executed when the AUTO button is clicked,
```

```

% and continues to execute until the STOP button is clicked.
%
handler AUTO
    go=1;
    print "Press STOP button to halt."
    while(go==1) looper end
    end

%
% The STOP button works by simply setting the variable 'go' to a
% value which will cause the main loop in the handler 'AUTO' to
% fail, thus ending the loop.
%
handler STOP
    go=0;
    print "Stopped." ,
    end

%
% This handler determines whether FFT will be uploaded and displayed,
% or not.
%
handler TOGGLE_FFT
    gof=1-gof;
    end

%
% This handler determines whether the signal waveform will be uploaded
% and displayed, or not.
%
handler TOGGLE_CRO
    gos=1-gos;
    end

%
% This handler causes a screen dump.
%
handler PRINT
    print screen
    end

%
% Now set up the screen objects.
%
graph(FFT,5,55,500,190)
set xaxis "FFT" 0 half_sampling_rate
set label "FFT" "Freq" "Ampl."
graph(Signal,5,195,500,330)
set label "Signal" "Sample number" "Ampl."
button("AUTO",570,55,630,100)
button("STOP",570,105,630,150)
button("PRINT",505,55,560,150)
button("TOGGLE_CRO",505,155,630,200)
button("TOGGLE_FFT",505,205,630,250)
numeric(Gain,505,255,630,290)
numeric(samp_rate,505,295,630,330)

%
% Now set up the serial port to the GI, and download the DSP code
%
set dsp baud 57600
send "dsp\combo.bin"
%
% Now do nothing until the user clicks a button
%

```

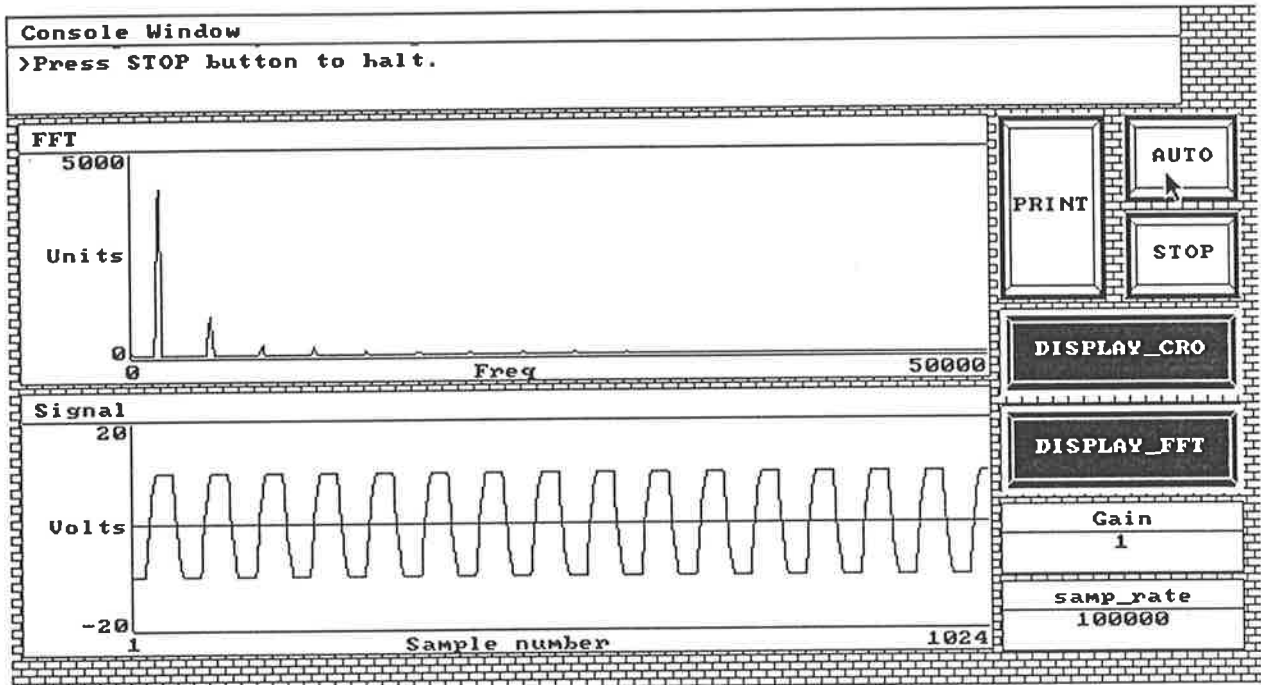


Figure C.1: A Digitiser / FFT Analyser Implemented with SPaM

## C.1 Example TMS320C25 Code

The following TMS320C25 program was designed to be used with the SPaM script shown above, to emulate an instrument which samples a signal, and displays the signal and its FFT.

- Sampling routines
- Processing routines, such as window scaling, FFT, magnitude calculation .

SPaM first downloads this program to the signal processor program memory, and then performs the following actions.

1. Based on the user's wishes (perhaps according to which onscreen buttons she clicks), SPaM downloads a single 16-bit word to the address in program memory corresponding to the address of the word `OPCODE` in the listing below.

Each bit of the word `OPCODE` causes a corresponding operating to be performed when the DSP code is executed. For instance, if bit 15 of the `OPCODE` word is 1, then the sampling routine will be executed first. If bit 15 is 0, then the sampling routine will not be executed. Similar tests are carried out on bit 14 through to bit 0.

2. After setting the `OPCODE` word to the value which will cause the desired operations to be performed, SPaM should then execute the `restart` command to cause the DSP program to execute.

Once the program has terminated, the next SPaM statement will be executed. This will usually be a call to the `upload()` function which will upload the processed data from the GI to the host.

```
* This is a combination sampling and processing program for the GI.
* Operations include those shown in the diagram below.
*
* Written by G.Vokalek, with code contributions from G.Y.Yuan.
*
reset_vector:
    b      start          ; this is the reset vector
    .long  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; unused vectors
timer_vector:
    b      timerhandler   ; this is the vector for timer interrupt
*
* The following are TMS320C25 internal registers.
*
TIM      .set    2 ; timer register
PRD      .set    3 ; period register for TMS320C25 internal timer
MASK     .set    4 ; interrupt mask
*
temp     .set    60H
val0     .set    61H
val1     .set    62H
half     .set    63H
mask     .set    64H
trigger  .set    65H
*
* The following is the only patch site for this module.
*
div      .word   0100 ; PATCH default rate is 10-7/100 = 100kHz.
```

```

trig   .word  0      ; trigger value
gain1  .word  0      ; channel 1 gain
gain2  .word  0      ; channel 2 gain
opcode .word  0      ; this value determines which operations are performed
*
* OPCODE is a 16-bit word
* 0000 0000 0000 0000
* |||| |||| |||| ||||
* |||| |||| |||| ||||
* |||| |||| |||| ||||
* |||| |||| |||| ||||
* |||| ||\ \ \ \ \ \ \ \ - Not used.
* |||| |\----- Perform Complex to Real Conversion
* |||| \----- Perform Magnitude calculation on complex array
* |||\----- Perform FFT calculation
* ||\----- Perform Window Operation
* |\----- Perform Real to Complex Conversion
* \----- Take 1024 signal samples.
* OPCODE is scanned from MSB to LSB, and the bits containing 1 cause execution
* of the corresponding code in that order.
*
* Here comes the code proper.
*
start:
    rsxm
    ldpk 0          ; page 0 contains on chip registers
    larp 0
    lalk opcode     ; get list of operations to do now
    tblr temp
    lac  temp
    andk 8000H      ; is bit 15 set? if so, sample.
    bnz  sample_start

samp_ret:
    lalk opcode     ; get list of operations to do now
    tblr temp
    lac  temp
    andk 4000H      ; is bit 15 set? if so, perform window function.
    bnz  rtoc_start

rtoc_ret:
    lalk opcode     ; get list of operations to do now
    tblr temp
    lac  temp
    andk 2000H      ; is bit 15 set? if so, perform real->complex shuffle.
    bnz  window_start

window_ret:
    lalk opcode     ; get list of operations to do now
    tblr temp
    lac  temp
    andk 1000H      ; is bit 15 set? if so, perform complex fft.
    bnz  fft_start

fft_ret:
    lalk opcode     ; get list of operations to do now
    tblr temp
    lac  temp
    andk 800H       ; is bit 15 set? if so, perform magnitude calculation.
    bnz  mag_start

mag_ret:
    lalk opcode     ; get list of operations to do now
    tblr temp

```

```

        lac    temp
        andk  400H      ; is bit 15 set? if so, perform complex to real shuffle.
        bnz   ctor_start
ctor_ret:
        lalk  8003H
        sacl  temp
        out   temp,1
        lack  3
        sacl  temp
        out   temp,1    ; cause attention request.
*
stop:
        b     stop      ; we have finished, so hang around.
*
*****
* SAMPLER - take N samples of 2 channels.
*
* INPUT 1 - 1024 samples stored at 1000H
* INPUT 2 - 1024 samples stored at 1400H
*
N       equ   1024      ; number of samples
CH1START equ  1000H
CH2START equ  1400H
*
ADC_start .set 7
ADC1_data .set 5
ADC2_data .set 6
CH1DATA   equ   05      ; write
CH2DATA   equ   04      ; write
DACUPDATE equ   06      ; write
*
* Now the gain control registers.
*
CH1GAIN   equ   07      ; write
CH2GAIN   equ   03      ; write
*
sample_start:
        larp  0
        ldpc 0
*
        lalk  timerhandler ; get address of timerhandler
        sacl  temp
        lalk  timer_vector+1 ; get address of vector
        tblw  temp          ; write address of handler to vector
*
        lalk  div          ; load accumulator with address of 'rate' variable
        tblr  PRD          ; load the actual value of rate into PERIOD register
        lark  ar0,8
        sar   ar0,MASK     ; mask timer interrupts
*
        lrlk  ar0,7fffH
        sar   ar0,half     ; subtract from sample to get 2's complement value
        lrlk  ar0,0FF00H
        sar   ar0,mask
        lalk  trig         ; get trigger value
        tblr  trigger      ; read its value into trigger data memory
*
        lalk  gain1        ; address of gain1 word
        tblr  temp

```



```

lac    temp
ork    40H          ; make sure ILE is set for DAC0830
sac1   temp
out    temp,CH1GAIN
lalk   gain2
tblr   temp
lac    temp
ork    40H          ; make sure ILE is set for DAC0830
sac1   temp
out    temp,CH2GAIN
*
lrlk   ar0,N-1
lrlk   ar1,CH1START ; channel 1 table start
lrlk   ar2,CH2START ; channel 2 table start
*
triglp
in     temp,ADC_start ; trigger the converters
in     temp,ADC_start ; trigger the converters
in     temp,ADC1_data
zals   trigger
subs   temp
bgz    triglp
in     temp,ADC_start ; trigger the converters
in     temp,ADC1_data
zals   trigger
subs   temp
bgz    triglp
*
trigd
lalk   div          ; load accumulator with address of 'rate' variable
tblr   TIM          ; load the actual value of rate into TIMER register
in     temp,ADC_start ; cause the first conversion
eint   ; enable interrupts
loop:  b    loop
*
*****
* This is the timer interrupt handler which performs the sampling.
*****
*
timerhandler:
in     temp,ADC_start ; trigger the converters
in     **+,ADC1_data,ar0
*
banz   no,*-,ar1     ; decrement ARO.
yes:
dint
*
lrlk   ar0,N-1
lrlk   ar1,CH1START ; channel 1 table start
ssxm
ldpk   0
larp   1
manglelp:
lac    *
and    mask
subs   half
sac1   **+,ARO
banz   manglelp,*-,ar1
*

```

```

        b      samp_ret
*
no:      eint      ; enable for next interrupt
        ret      ; return from interrupt server
*
*****
* WINDOW - performs hamming window operation over data. (ch1 only).
* Based on code by G.Y.Yuan.
*
ONE      EQU      1H
XI       EQU      2H
YI       EQU      3H
XL       EQU      4H
YL       EQU      5H
XT       EQU      6H
YT       EQU      7H
I        EQU      8H
L        EQU      9H
ITE      EQU      0AH
LTE      EQU      0BH
SIN      EQU      0CH
CON      EQU      0DH
IA       EQU      0EH
IE       EQU      0FH
HOLDN    EQU      10H
QUARTN   EQU      11H
N1       EQU      12H
N2       EQU      13H
J        EQU      14H
TABLE    EQU      15H
HALF     EQU      16H
ZERO     EQU      17H
TEMPE0   EQU      18H
TEMPE1   EQU      19H
TEMPE2   EQU      1AH
TEMP     EQU      1BH
*
*****
*
*      This part of the program is using KAISER-BESSEL window
* function to smooth the spectrum. Uses table lookup of window
* function with table size N.
*
*****
*
PRODUCT:      ; Window applied subroutine
        LAC      TEMPE1
        ADDK     1
        SACL     TEMPE1
        TBLR     TEMPE2
        LT       TEMPE2
        LARP     1
        MPY      *      ; mul by real part
        PAC
        SACH     **      ; store weighted imag part
        LT       TEMPE2
        MPY      *      ; mul by imag part
        PAC

```

```

SACH  **+,0,ar0      ; store weighted imag part
BANZ  PRODUCT,*-
RET

```

```

*
*****
* KAISER window function table
*****

```

```

*
*
TWIDD DATA  0, 199, 209, 218, 228, 238, 249, 259
DATA 270, 281, 292, 304, 316, 327, 340, 352
DATA 365, 378, 391, 404, 418, 431, 446, 460
DATA 475, 489, 505, 520, 536, 552, 568, 584
DATA 601, 618, 635, 653, 671, 689, 707, 726
DATA 745, 764, 784, 804, 824, 845, 865, 887
DATA 908, 930, 952, 974, 997, 1020, 1043, 1067
DATA 1091, 1115, 1140, 1165, 1190, 1216, 1242, 1268
DATA 1295, 1322, 1349, 1377, 1405, 1433, 1462, 1491
DATA 1521, 1550, 1581, 1611, 1642, 1673, 1705, 1737
DATA 1769, 1802, 1835, 1869, 1903, 1937, 1972, 2007
DATA 2042, 2078, 2114, 2151, 2188, 2225, 2263, 2301
DATA 2340, 2379, 2418, 2458, 2498, 2539, 2580, 2621
DATA 2663, 2705, 2748, 2791, 2834, 2878, 2922, 2967
DATA 3012, 3057, 3103, 3150, 3197, 3244, 3291, 3339
DATA 3388, 3437, 3486, 3536, 3586, 3637, 3688, 3739
DATA 3791, 3843, 3896, 3949, 4003, 4057, 4112, 4166
DATA 4222, 4278, 4334, 4390, 4448, 4505, 4563, 4621
DATA 4680, 4740, 4799, 4859, 4920, 4981, 5042, 5104
DATA 5167, 5229, 5293, 5356, 5420, 5485, 5550, 5615
DATA 5681, 5747, 5814, 5881, 5948, 6016, 6085, 6154
DATA 6223, 6292, 6363, 6433, 6504, 6575, 6647, 6719
DATA 6792, 6865, 6938, 7012, 7087, 7161, 7236, 7312
DATA 7388, 7464, 7541, 7618, 7696, 7774, 7852, 7931
DATA 8010, 8090, 8170, 8250, 8331, 8412, 8494, 8576
DATA 8658, 8741, 8824, 8907, 8991, 9075, 9160, 9245
DATA 9330, 9416, 9502, 9588, 9675, 9762, 9850, 9937
DATA 10026, 10114, 10203, 10292, 10382, 10472, 10562, 10652
DATA 10743, 10835, 10926, 11018, 11110, 11203, 11295, 11388
DATA 11482, 11576, 11670, 11764, 11858, 11953, 12048, 12144
DATA 12240, 12335, 12432, 12528, 12625, 12722, 12819, 12917
DATA 13015, 13113, 13211, 13310, 13408, 13507, 13606, 13706
DATA 13806, 13905, 14005, 14106, 14206, 14307, 14408, 14509
DATA 14610, 14711, 14813, 14915, 15017, 15119, 15221, 15324
DATA 15426, 15529, 15632, 15735, 15838, 15941, 16044, 16148
DATA 16252, 16355, 16459, 16563, 16667, 16771, 16875, 16980
DATA 17084, 17188, 17293, 17398, 17502, 17607, 17712, 17816
DATA 17921, 18026, 18131, 18236, 18341, 18446, 18550, 18655
DATA 18760, 18865, 18970, 19075, 19180, 19285, 19389, 19494
DATA 19599, 19704, 19808, 19913, 20017, 20122, 20226, 20330
DATA 20434, 20538, 20642, 20746, 20850, 20954, 21057, 21161
DATA 21264, 21367, 21470, 21573, 21676, 21778, 21880, 21983
DATA 22085, 22187, 22288, 22390, 22491, 22592, 22693, 22794
DATA 22894, 22994, 23094, 23194, 23294, 23393, 23492, 23591
DATA 23689, 23787, 23885, 23983, 24080, 24178, 24274, 24371
DATA 24467, 24563, 24659, 24754, 24849, 24943, 25038, 25132
DATA 25225, 25318, 25411, 25504, 25596, 25688, 25779, 25870
DATA 25961, 26051, 26141, 26230, 26319, 26408, 26496, 26584
DATA 26671, 26758, 26844, 26930, 27016, 27101, 27185, 27269
DATA 27353, 27436, 27519, 27601, 27683, 27764, 27845, 27925

```

DATA 28005, 28084, 28162, 28240, 28318, 28395, 28472, 28548  
DATA 28623, 28698, 28772, 28846, 28919, 28992, 29064, 29135  
DATA 29206, 29276, 29346, 29415, 29484, 29552, 29619, 29686  
DATA 29752, 29817, 29882, 29946, 30009, 30072, 30135, 30196  
DATA 30257, 30317, 30377, 30436, 30494, 30552, 30609, 30665  
DATA 30721, 30776, 30830, 30884, 30937, 30989, 31040, 31091  
DATA 31141, 31190, 31239, 31287, 31334, 31381, 31426, 31471  
DATA 31516, 31559, 31602, 31644, 31686, 31726, 31766, 31805  
DATA 31844, 31881, 31918, 31954, 31990, 32024, 32058, 32091  
DATA 32123, 32155, 32186, 32216, 32245, 32273, 32301, 32328  
DATA 32354, 32379, 32404, 32427, 32450, 32472, 32494, 32514  
DATA 32534, 32553, 32571, 32589, 32605, 32621, 32636, 32650  
DATA 32663, 32676, 32688, 32698, 32709, 32718, 32726, 32734  
DATA 32741, 32747, 32752, 32757, 32761, 32763, 32765, 32767  
DATA 32767, 32765, 32763, 32761, 32757, 32752, 32747, 32741  
DATA 32734, 32726, 32718, 32709, 32698, 32688, 32676, 32663  
DATA 32650, 32636, 32621, 32605, 32589, 32571, 32553, 32534  
DATA 32514, 32494, 32472, 32450, 32427, 32404, 32379, 32354  
DATA 32328, 32301, 32273, 32245, 32216, 32186, 32155, 32123  
DATA 32091, 32058, 32024, 31990, 31954, 31918, 31881, 31844  
DATA 31805, 31766, 31726, 31686, 31644, 31602, 31559, 31516  
DATA 31471, 31426, 31381, 31334, 31287, 31239, 31190, 31141  
DATA 31091, 31040, 30989, 30937, 30884, 30830, 30776, 30721  
DATA 30665, 30609, 30552, 30494, 30436, 30377, 30317, 30257  
DATA 30196, 30135, 30072, 30009, 29946, 29882, 29817, 29752  
DATA 29686, 29619, 29552, 29484, 29415, 29346, 29276, 29206  
DATA 29135, 29064, 28992, 28919, 28846, 28772, 28698, 28623  
DATA 28548, 28472, 28395, 28318, 28240, 28162, 28084, 28005  
DATA 27925, 27845, 27764, 27683, 27601, 27519, 27436, 27353  
DATA 27269, 27185, 27101, 27016, 26930, 26844, 26758, 26671  
DATA 26584, 26498, 26408, 26319, 26230, 26141, 26051, 25961  
DATA 25870, 25779, 25688, 25596, 25504, 25411, 25318, 25225  
DATA 25132, 25038, 24943, 24849, 24754, 24659, 24563, 24467  
DATA 24371, 24274, 24178, 24080, 23983, 23885, 23787, 23689  
DATA 23591, 23492, 23393, 23294, 23194, 23094, 22994, 22894  
DATA 22794, 22693, 22592, 22491, 22390, 22288, 22187, 22085  
DATA 21983, 21880, 21778, 21676, 21573, 21470, 21367, 21264  
DATA 21161, 21057, 20954, 20850, 20746, 20642, 20538, 20434  
DATA 20330, 20226, 20122, 20017, 19913, 19808, 19704, 19599  
DATA 19494, 19389, 19285, 19180, 19075, 18970, 18865, 18760  
DATA 18655, 18550, 18446, 18341, 18236, 18131, 18026, 17921  
DATA 17816, 17712, 17607, 17502, 17398, 17293, 17188, 17084  
DATA 16980, 16875, 16771, 16667, 16563, 16459, 16355, 16252  
DATA 16148, 16044, 15941, 15838, 15735, 15632, 15529, 15426  
DATA 15324, 15221, 15119, 15017, 14915, 14813, 14711, 14610  
DATA 14509, 14408, 14307, 14206, 14106, 14005, 13905, 13806  
DATA 13706, 13606, 13507, 13408, 13310, 13211, 13113, 13015  
DATA 12917, 12819, 12722, 12625, 12528, 12432, 12335, 12240  
DATA 12144, 12048, 11953, 11858, 11764, 11670, 11576, 11482  
DATA 11388, 11295, 11203, 11110, 11018, 10926, 10835, 10743  
DATA 10652, 10562, 10472, 10382, 10292, 10203, 10114, 10026  
DATA 9937, 9850, 9762, 9675, 9588, 9502, 9416, 9330  
DATA 9245, 9160, 9075, 8991, 8907, 8824, 8741, 8658  
DATA 8576, 8494, 8412, 8331, 8250, 8170, 8090, 8010  
DATA 7931, 7852, 7774, 7696, 7618, 7541, 7464, 7388  
DATA 7312, 7236, 7161, 7087, 7012, 6938, 6865, 6792  
DATA 6719, 6647, 6575, 6504, 6433, 6363, 6292, 6223  
DATA 6154, 6085, 6016, 5948, 5881, 5814, 5747, 5681  
DATA 5615, 5550, 5485, 5420, 5356, 5293, 5229, 5167

```

DATA 5104, 5042, 4981, 4920, 4859, 4799, 4740, 4680
DATA 4621, 4563, 4505, 4448, 4390, 4334, 4278, 4222
DATA 4166, 4112, 4057, 4003, 3949, 3896, 3843, 3791
DATA 3739, 3688, 3637, 3586, 3536, 3486, 3437, 3388
DATA 3339, 3291, 3244, 3197, 3150, 3103, 3057, 3012
DATA 2967, 2922, 2878, 2834, 2791, 2748, 2705, 2663
DATA 2621, 2580, 2539, 2498, 2458, 2418, 2379, 2340
DATA 2301, 2263, 2225, 2188, 2151, 2114, 2078, 2042
DATA 2007, 1972, 1937, 1903, 1869, 1835, 1802, 1769
DATA 1737, 1705, 1673, 1642, 1611, 1581, 1550, 1521
DATA 1491, 1462, 1433, 1405, 1377, 1349, 1322, 1295
DATA 1268, 1242, 1216, 1190, 1165, 1140, 1115, 1091
DATA 1067, 1043, 1020, 997, 974, 952, 930, 908
DATA 887, 865, 845, 824, 804, 784, 764, 745
DATA 726, 707, 689, 671, 653, 635, 618, 601
DATA 584, 568, 552, 536, 520, 505, 489, 475
DATA 460, 446, 431, 418, 404, 391, 378, 365
DATA 352, 340, 327, 316, 304, 292, 281, 270
DATA 259, 249, 238, 228, 218, 209, 199, 0

```

```

*
*-----*

```

```

*
DATADR .WORD 1000H ; address of data to window
*

```

```

window_start:

```

```

    LDPK 4
    ssxm

```

```

*
    LARP 0
    LRLK ARO,N ; debug was N
    SBRK 1
    LALK TWIDD
    SUBK 1
    SACL TEMPE1

```

```

*
    LALK DATADR
    TBLR TEMPE0
    LAR AR1,TEMPE0

```

```

*
    CALL PRODUCT
    B window_ret

```

```

*
*****

```

```

* rtoc - interleave consecutive real numbers with zero to
* make them complex.

```

```

*
rtoc_start:

```

```

    rsxm
    ldpk 0
    lac 0
    sacl temp
    lalk N,1 ; N*2
    subk 1 ; Acc is now offset of last imag part
    addk CH1START ; Acc is now address of last imag part of dest
    sacl temp
    lar ar0,temp

```

```

    lalk    N
    subk    1
    addk    CH1START      ; Acc now address of last real part of source
    sacl    temp
    lar     ar1,temp
    lrlk    ar2,N

```

```

rtoclp:
    larp    0
    zac
    sacl    *-,ar1      ; store imag part = 0
    lac     *-,ar0      ; load real part from source
    sacl    *-,ar2      ; store real part in dest
    sar     ar0,temp
    banz    rtoclp,*-

```

```

*
    b       rtoc_ret
*

```

```

*****
* FFT - perform 1024 point complex fft. Based on program by G.Y.Yuan.
*

```

```

M     EQU    10          ; N = 2 ** M

```

```

*****
*
*   This program is for implementing a single butterfly RADIX-2 *
*   Cooley-Tukey N-point FFT. All data is in external data memory *
*   and uses Q15 data format. All DIT butterflies are implemented *
*   with dynamic scaling to avoid arithmetic overflows. Uses table *
*   lookup of coefficients with table size N * (3/4) for sines and *
*   cosines.
*

```

```

*****
*
fft_start:

```

```

    sxxm
    LDPK    4
    LALK    DATADR      ; get address of DATADR variable
    TBLR    TEMPEO
    CALL    SUBFFT

```

```

*
    B       fft_ret

```

```

*
SUBFFT      ; FFT subroutine for two channels

```

```

*
    LACK    1
    SACL    ONE
    SACL    IE          ; Initialize IE = 1
    LALK    SINE
    SACL    TABLE      ; Table has address of cosine table
    LALK    N
    SACL    HOLDN      ; Holdn = N
    SACL    N2
    LAC     HOLDN,14
    SACH    QUARTN     ; Quartn = N/4

```

```

*
    LARK    ARO,M-1    ; ARO contains K counter

```

```

KLOOP     LARP    1
          LAC     N2,15

```

```

SACH N1,1 ; N1 = N2
SACH N2 ; N2 = N2/2
ZAC
SACL IA
SACL J
LAR AR1,N2 ; AR1 contains J value
MAR *- ; Start at N2-1
JLOOP LAC TABLE ; Table is full size
ADD IA
TBLR SIN ; Get twiddle factors
ADD QUARTN
TBLR CON
LAC IA
ADD IE
SACL IA ; IA = IA + IE
LAC J,1
SACL I ; I = J
*
ILOOP LAC I
ADD N2,1 ; L = I + N2
SACL L
*
LAC I
ADD TEMPEO
SACL ITE ; Data stored from >0400
LAC L
ADD TEMPEO
SACL LTE
*
LARP 6
LAR 6,ITE
LAC **,,15 ; scaling by 1/2
SACH XI
LAC *,15 ; scaling by 1/2
SACH YI
LAR 6,LTE
LAC **,,15 ; scaling by 1/2
SACH XL
LAC *,15 ; scaling by 1/2
SACH YL
*
*****
* Compute butterfly *
*****
*
LAC XI
SUB XL
SACL XT ; XT = XI - XL
ADD XL,1
SACL XI ; XI = XI + XL
LAC YI
SUB YL
SACL YT ; YT = YI - YL
ADD YL,1
SACL YI ; YI = YI + YL
LT CON
MPY YT
PAC
LT SIN

```

```

MPY    XT
SPAC
SACH  YL,1      ; YL = COS*YT - SIN*XT
MPY    YT
PAC
LT     CON
MPY    XT
APAC
SACH  XL,1      ; XL = COS*XT + SIN*YT

```

```

*
*****
*   Output results of butterfly                               *
*****
*

```

```

LARP   6
LAR    ar6,ITE
LAC    XI
SACL   **
LAC    YI
SACL   *
LAR    6,LTE
LAC    XL
SACL   **
LAC    YL
SACL   *

```

```

*
*****
*   Add increment for next loop                               *
*****
*

```

```

LAC    I
ADD    N1,1      ; I = I + N1
SACL   I
SUB    HOLDN,1   ; While I < N
BLZ    ILOOP

```

```

*
LAC    J
ADD    ONE       ; J = J + 1
SACL   J
LARP   1
BANZ   JLOOP     ; AR1 <> 0 then pass to JLOOP

```

```

*
LAC    IE,1
SACL   IE       ; IE = 2*IE
LARP   0
BANZ   KLOOP     ; AR0 <> 0 then pass to KLOOP

```

```

*
*****
*   Digit reverse counter for radix-2 FFT computation       *
*****
*

```

```

DRC2   ZAC
SACL   L
SACL   I
LARP   0
LAR    ARO,HOLDN ; For I = 0 to N-2
MAR    *-
MAR    *-
DRLOOP SUB    L      ; If I < L, then swap

```



BGEZ NOSWAP

\*  
\*\*\*\*\*  
\* Swap ITE and LTE values \*

\*  
LAC I  
ADD TEMPEO  
SACL ITE ; Get ITE data address  
LAC L  
ADD TEMPEO  
SACL LTE ; Get LTE data address

\*  
LARP 6  
LAR 6,ITE  
LAC \*\* ; Get I value address  
SACL XI ; Get real and imaginary parts  
LAC \*  
SACL YI  
LAR 6,LTE  
LAC \*\* ; Get L value address  
SACL XL ; Get real and imaginary parts  
LAC \*  
SACL YL

\*  
LAR 6,LTE  
LAC XI  
SACL \*\*  
LAC YI  
SACL \*  
LAR 6,ITE  
LAC XL  
SACL \*\*  
LAC YL  
SACL \*  
LARP 0

\*  
NOSWAP LAC HOLDN  
SACL J ; J = N  
INLOOP LAC L  
SUB J ; If L >= J then  
BLZ OUTL  
SACL L ; L = L - J  
LAC J,15  
SACH J ; J = J/2  
B INLOOP  
OUTL ADD J,1  
SACL L ; L = L + J  
LAC I  
ADD ONE,1  
SACL I ; Increment I  
BANZ DRLOOP ; ARO <> 0 then pass to DRLOOP  
RET

\*  
\*\*\*\*\*  
\* Coefficient table(size of table is 3N/4) \*  
\*\*\*\*\*  
\*  
\*

SINE	DATA	0,	201,	402,	603,	804,	1005
	DATA	1206,	1407,	1608,	1809,	2009,	2210
	DATA	2410,	2611,	2811,	3012,	3212,	3412
	DATA	3612,	3811,	4011,	4210,	4410,	4609
	DATA	4808,	5007,	5205,	5404,	5602,	5800
	DATA	5998,	6195,	6393,	6590,	6786,	6983
	DATA	7179,	7375,	7571,	7767,	7962,	8157
	DATA	8351,	8545,	8739,	8933,	9126,	9319
	DATA	9512,	9704,	9896,	10087,	10278,	10469
	DATA	10659,	10849,	11039,	11228,	11417,	11605
	DATA	11793,	11980,	12167,	12353,	12539,	12725
	DATA	12910,	13094,	13279,	13462,	13645,	13828
	DATA	14010,	14191,	14372,	14553,	14732,	14912
	DATA	15090,	15269,	15446,	15623,	15800,	15976
	DATA	16151,	16325,	16499,	16673,	16846,	17018
	DATA	17189,	17360,	17530,	17700,	17869,	18037
	DATA	18204,	18371,	18537,	18703,	18868,	19032
	DATA	19195,	19357,	19519,	19680,	19841,	20000
	DATA	20159,	20317,	20475,	20631,	20787,	20942
	DATA	21096,	21250,	21403,	21554,	21705,	21856
	DATA	22005,	22154,	22301,	22448,	22594,	22739
	DATA	22884,	23027,	23170,	23311,	23452,	23592
	DATA	23731,	23870,	24007,	24143,	24279,	24413
	DATA	24547,	24680,	24811,	24942,	25072,	25201
	DATA	25329,	25456,	25582,	25708,	25832,	25955
	DATA	26077,	26198,	26319,	26438,	26556,	26674
	DATA	26790,	26905,	27019,	27133,	27245,	27356
	DATA	27466,	27575,	27683,	27790,	27896,	28001
	DATA	28105,	28208,	28310,	28411,	28510,	28609
	DATA	28706,	28803,	28898,	28992,	29085,	29177
	DATA	29268,	29358,	29447,	29534,	29621,	29706
	DATA	29791,	29874,	29956,	30037,	30117,	30195
	DATA	30273,	30349,	30424,	30498,	30571,	30643
	DATA	30714,	30783,	30852,	30919,	30985,	31050
	DATA	31113,	31176,	31237,	31297,	31356,	31414
	DATA	31470,	31526,	31580,	31633,	31685,	31736
	DATA	31785,	31833,	31880,	31926,	31971,	32014
	DATA	32057,	32098,	32137,	32176,	32213,	32250
	DATA	32285,	32318,	32351,	32382,	32412,	32441
	DATA	32469,	32495,	32521,	32545,	32567,	32589
	DATA	32609,	32628,	32646,	32663,	32678,	32692
	DATA	32705,	32717,	32728,	32737,	32745,	32752
	DATA	32757,	32761,	32765,	32766		
COSINE	DATA	32767,	32766,	32765,	32761,	32757,	32752
	DATA	32745,	32737,	32728,	32717,	32705,	32692
	DATA	32678,	32663,	32646,	32628,	32609,	32589
	DATA	32567,	32545,	32521,	32495,	32469,	32441
	DATA	32412,	32382,	32351,	32318,	32285,	32250
	DATA	32213,	32176,	32137,	32098,	32057,	32014
	DATA	31971,	31926,	31880,	31833,	31785,	31736
	DATA	31685,	31633,	31580,	31526,	31470,	31414
	DATA	31356,	31297,	31237,	31176,	31113,	31050
	DATA	30985,	30919,	30852,	30783,	30714,	30643
	DATA	30571,	30498,	30424,	30349,	30273,	30195
	DATA	30117,	30037,	29956,	29874,	29791,	29706
	DATA	29621,	29534,	29447,	29358,	29268,	29177
	DATA	29085,	28992,	28898,	28803,	28706,	28609
	DATA	28510,	28411,	28310,	28208,	28105,	28001
	DATA	27896,	27790,	27683,	27575,	27466,	27356

DATA	27245,	27133,	27019,	26905,	26790,	26674
DATA	26556,	26438,	26319,	26198,	26077,	25955
DATA	25832,	25708,	25582,	25456,	25329,	25201
DATA	25072,	24942,	24811,	24680,	24547,	24413
DATA	24279,	24143,	24007,	23870,	23731,	23592
DATA	23452,	23311,	23170,	23027,	22884,	22739
DATA	22594,	22448,	22301,	22154,	22005,	21856
DATA	21705,	21554,	21403,	21250,	21096,	20942
DATA	20787,	20631,	20475,	20317,	20159,	20000
DATA	19841,	19680,	19519,	19357,	19195,	19032
DATA	18868,	18703,	18537,	18371,	18204,	18037
DATA	17869,	17700,	17530,	17360,	17189,	17018
DATA	16846,	16673,	16499,	16325,	16151,	15976
DATA	15800,	15623,	15446,	15269,	15090,	14912
DATA	14732,	14553,	14372,	14191,	14010,	13828
DATA	13645,	13462,	13279,	13094,	12910,	12725
DATA	12539,	12353,	12167,	11980,	11793,	11605
DATA	11417,	11228,	11039,	10849,	10659,	10469
DATA	10278,	10087,	9896,	9704,	9512,	9319
DATA	9126,	8933,	8739,	8545,	8351,	8157
DATA	7962,	7767,	7571,	7375,	7179,	6983
DATA	6786,	6590,	6393,	6195,	5998,	5800
DATA	5602,	5404,	5205,	5007,	4808,	4609
DATA	4410,	4210,	4011,	3811,	3612,	3412
DATA	3212,	3012,	2811,	2611,	2410,	2210
DATA	2009,	1809,	1608,	1407,	1206,	1005
DATA	804,	603,	402,	201,	-0,	-201
DATA	-402,	-603,	-804,	-1005,	-1206,	-1407
DATA	-1608,	-1809,	-2009,	-2210,	-2410,	-2611
DATA	-2811,	-3012,	-3212,	-3412,	-3612,	-3811
DATA	-4011,	-4210,	-4410,	-4609,	-4808,	-5007
DATA	-5205,	-5404,	-5602,	-5800,	-5998,	-6195
DATA	-6393,	-6590,	-6786,	-6983,	-7179,	-7375
DATA	-7571,	-7767,	-7962,	-8157,	-8351,	-8545
DATA	-8739,	-8933,	-9126,	-9319,	-9512,	-9704
DATA	-9896,	-10087,	-10278,	-10469	-10659,	-10849
DATA	-11039,	-11228,	-11417,	-11605,	-11793,	-11980
DATA	-12167,	-12353,	-12539,	-12725,	-12910,	-13094
DATA	-13279,	-13462,	-13645,	-13828,	-14010,	-14191
DATA	-14372,	-14553,	-14732,	-14912,	-15090,	-15269
DATA	-15446,	-15623,	-15800,	-15976,	-16151,	-16325
DATA	-16499,	-16673,	-16846,	-17018,	-17189,	-17360
DATA	-17530,	-17700,	-17869,	-18037,	-18204,	-18371
DATA	-18537,	-18703,	-18868,	-19032,	-19195,	-19357
DATA	-19519,	-19680,	-19841,	-20000,	-20159,	-20317
DATA	-20475,	-20631,	-20787,	-20942,	-21096,	-21250
DATA	-21403,	-21554,	-21705,	-21856,	-22005,	-22154
DATA	-22301,	-22448,	-22594,	-22739,	-22884,	-23027
DATA	-23170,	-23311,	-23452,	-23592,	-23731,	-23870
DATA	-24007,	-24143,	-24279,	-24413,	-24547,	-24680
DATA	-24811,	-24942,	-25072,	-25201,	-25329,	-25456
DATA	-25582,	-25708,	-25832,	-25955,	-26077,	-26198
DATA	-26319,	-26438,	-26556,	-26674,	-26790,	-26905
DATA	-27019,	-27133,	-27245,	-27356,	-27466,	-27575
DATA	-27683,	-27790,	-27896,	-28001,	-28105,	-28208
DATA	-28310,	-28411,	-28510,	-28609,	-28706,	-28803
DATA	-28898,	-28992,	-29085,	-29177,	-29268,	-29358
DATA	-29447,	-29534,	-29621,	-29706,	-29791,	-29874
DATA	-29956,	-30037,	-30117,	-30195,	-30273,	-30349

```

DATA -30424, -30498, -30571, -30643, -30714, -30783
DATA -30852, -30919, -30985, -31050, -31113, -31176
DATA -31237, -31297, -31356, -31414, -31470, -31526
DATA -31580, -31633, -31685, -31736, -31785, -31833
DATA -31880, -31926, -31971, -32014, -32057, -32098
DATA -32137, -32176, -32213, -32250, -32285, -32318
DATA -32351, -32382, -32412, -32441, -32469, -32495
DATA -32521, -32545, -32567, -32589, -32609, -32628
DATA -32646, -32663, -32678, -32692, -32705, -32717
DATA -32728, -32737, -32745, -32752, -32757, -32761
DATA -32765, -32766

```

```

*
*****

```

```

* mag - calculate magnitudes of complex numbers, using a modified
* form of Newton's method to calculate the square root.

```

```

*
m_temp .set 0H
m_sqrtemp .set 1H
m_shifts .set 2H
m_one .set 3H
m_vlo .set 4h
m_vhi .set 5h

```

```

*
* magnitude root code
*

```

```

mag_start:
    larp 0
    lrlk ar0,CH1START
    ldpk 4
    lack 1
    sacl m_one

```

```

*
    lrlk ar3,N ; count the operations we do.

```

```

*
* assume ARO points to real/imag
*

```

```

    ssxm
*
lp3:
    larp 0
    zac
    lt *
    mpy **
    lt *
    mpya *
    apac
    sach *- ; save in low-high format
    sach m_vhi
    sacl * ; low part comes first
    sacl m_vlo

```

```

*
    bz zero
*
    larp 1
    lark ar1,0

```

```

lp4:
    norm
    bbz lp4

```

```

*
  sfr
  sfr
  addh  m_one           ; make sure estimate will be at least 1
  sach  m_sqrtemp ; first estimate is V/4
*
* ar0 still points to same place, but memory now holds number_hi, number_lo
*
  lrlk  ar2,20 ; iteration counter
*
lp2:
  larp  1           ; use ar1 as a counter
  lark  ar1,0       ; zero it
  zalh  m_sqrtemp ; get x
lp1:
  mar   **
  sfl
  bnc   lp1
*
  sar   ar1,m_temp   ; save counter
  lack  16
  sub   m_temp       ; now we have number of shifts to do
  sacl  m_shifts
*
  larp  0
  zals  **           ; get low part
  addh  *-           ; get high part
*
  lt    m_sqrtemp
  mpy   m_sqrtemp
  spac  ; form v-x*x
  neg   ; form x*x-v
*
  rpt   m_shifts
  sfr   ; do one more shift than indicated by 'shifts'
*
  sub   m_sqrtemp ; form (x*x-v)/2p - x
  neg   ; form x - (x*x-v)/2p
*
  sacl  m_sqrtemp ;
  larp  2
  banz  lp2,*-
*
  larp  0
  sacl  **           ; store it in place
  lack  0
  sacl  **           ; zero high part
*
zcont:
  larp  3
  banz  lp3,*-
*
* now move all the real parts together (remove interspaced complex parts )
*
  larp  0
  lrlk  ar0,1000H
  lrlk  ar1,1000H
  lrlk  ar2,N
movelp:

```

```

    larp    0
    lac     **+,ar1
    sacl    **+,ar0
    mar     **+,ar2
    banz    movelp,*-
*
    b       mag_ret
*
zero:
    larp    0
    lack    0
    sacl    **+
    sacl    **+
    b       zcont
*
*****
* ctor - throw away imag parts to leave reals
*
cr_temp    equ    60H
*
ctor_start:
    ldpk    0
    lrlk    ar0,CH1START
    lrlk    ar1,CH1START
    lrlk    ar2,N
crlp:
    larp    0
    lac     **+
    mar     **+,ar1      ; skip over imag part
    sacl    **+,ar2
    banz    crlp,*-
*
    b       ctor_ret
*
*****

```