

YAC-BUS - AN INTERFACE FOR
MICROPROCESSOR-CONTROLLED PERIPHERALS

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
by
C. P. Stott

University of Canterbury

1980

ABSTRACT

The use of the microprocessor to control a computer's peripheral changes the criteria for the interface between the peripheral and computer. An interface is described in which the data handled by the peripheral is stored in the microprocessor's own memory and the minicomputer is provided with the means to access this data through its IO system.

The interface allows up to 16 microprocessor-controlled peripherals to be interfaced in a manner that places few requirements on the capabilities of a minicomputer's IO system, a feature which permits portability between various types of minicomputer. Details of implementation for a PDP-11 minicomputer and a 6800 microprocessor are given. Finally, the weaknesses of the implementation are examined and a comparison between the interface and conventional direct memory access methods is made.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisors, Dr. M.A. Maclean and Mr. R.M. Harrington for their advice, guidance and criticism during the course of this investigation. In particular, I would like to thank Mr. Harrington for suggesting the initial problem and making available laboratory facilities and test equipment used during the development of YAC-Bus.

TABLE OF CONTENTS

	<u>PAGE</u>	
CHAPTER 1	INTRODUCTION	1
CHAPTER 2	ARCHITECTURE OF IO SYSTEMS	3
2.1	CPU-PERIPHERAL COMMUNICATION	3
2.1.1	A SIMPLE APPROACH TO IO	4
2.1.2	POLLING PERIPHERALS	5
2.1.3	INTERRUPTS	6
2.2	PERIPHERAL-MEMORY COMMUNICATION	8
2.2.1	DIRECT MEMORY ACCESS FACILITIES	8
2.2.2	THE IO PROCESSOR	9
2.3	THE PERIPHERAL AS A PROCESSOR	11
CHAPTER 3	THE MICROPROCESSOR AS A PERIPHERAL CONTROLLER	13
3.1	THE PRESENT FORM OF A COMPUTER-PERIPHERAL INTERFACE	13
3.2	THE MICROPROCESSOR	15
3.2.1	REPLACING EXISTING HARDWARE WITH A MICROPROCESSOR	15
3.2.2	EXPLOITING THE MICROPROCESSOR'S CAPABILITIES	16
3.3	DESIGN AIMS FOR A NEW HARDWARE INTERFACE	17
3.3.1	EFFICIENCY	17
3.3.2	TRANSFORMATION OF THE HARDWARE INTERFACE	18
3.3.3	FLEXIBILITY	18
3.3.4	PORTABILITY	19
3.4	MINICOMPUTER IO FACILITIES	19
3.4.1	REGISTER INTERFACES	20
3.4.2	DMA INTERFACES	20
3.5	AN INTERFACE FOR A MICROPROCESSOR- CONTROLLED PERIPHERAL	21
3.6	THE SUITABILITY OF OTHER BUS SYSTEMS	24
CHAPTER 4	YAC-BUS ARCHITECTURE	26
4.1	YAC-BUS COMPONENTS	26
4.1.1	YAC-BUS	26
4.1.2	THE MINICOMPUTER INTERFACE	28
4.1.3	THE PERIPHERAL INTERFACE	31
4.1.4	THE INTERRUPT SYSTEM	31

	<u>PAGE</u>	
4.2	YAC-BUS PROTOCOL	33
4.2.1	BUS CONTROL	34
4.2.2	BUS SYNCHRONISATION	37
CHAPTER 5	YAC-BUS IMPLEMENTATION	39
5.1	THE PDP-11 INTERFACE	39
5.1.1	UNIBUS BUFFERING AND TIMING	41
5.1.2	YAC-BUS INTERFACE REGISTERS	41
5.1.3	TAC-BUS TIMING	44
5.1.4	UNIBUS INTERRUPT	48
5.2	OTHER MINICOMPUTER INTERFACES	50
5.3	THE 6800 INTERFACE	51
5.3.1	PERIPHERAL CONTROL AND TIMING	53
5.3.2	DUAL-PORTED MEMORY	57
5.4	USING OTHER MICROPROCESSORS	57
CHAPTER 6	USING YAC-BUS	59
6.1	SOFTWARE FOR YAC-BUS	59
6.1.1	ALLOCATION OF TASKS	59
6.1.2	THE USER INTERFACE	61
6.1.3	USER FACILITIES	63
6.2	YAC-BUS EFFICIENCY	66
6.3	A MONITOR FOR YAC-BUS	68
6.3.1	OVERVIEW	69
6.3.2	THE 6800 MONITOR	69
6.3.3	THE PDP-11 MONITOR	73
CHAPTER 7	COMMENTS AND CONCLUSIONS	77
7.1	CONCLUSIONS	77
7.2	FUTURE WORK AND IMPROVEMENTS	81
BIBLIOGRAPHY		84
APPENDIX A	6800 MONITOR LISTING	A-1
APPENDIX B	PDP-11 MONITOR LISTING	B-1

LIST OF FIGURES

FIGURE		PAGE
3.1	YAC-Bus	22
4.1	YAC-Bus Configuration	27
4.2	YAC-Bus Memory System	29
4.3	YAC-Bus Interrupt System	32
4.4	YAC-Bus Timing	36
5.1	Block Diagram - Minicomputer Interface	40
5.2	PDP-11 Interface Registers	42
5.3	Schematic Diagram - Unibus Buffering and Timing	43
5.4	Schematic Diagram - YAC-Bus Interface Registers	45
5.5	Schematic Diagram - YAC-Bus Timing	46
5.6	Timing Diagram - Minicomputer Interface	47
5.7	Schematic Diagram - Unibus Interrupt	49
5.8	Block Diagram - Peripheral Interface	52
5.9	Schematic Diagram - Peripheral Control and Timing	54
5.10	Timing Diagram - Peripheral Interface	55
5.11	Schematic Diagram - Peripheral Dual-Ported Memory	56
6.1	YAC-Bus Access	65
6.2	Accessing YAC-Bus with the PDP-11 - a comparison	67

CHAPTER 1

INTRODUCTION

Peripherals are very important parts of any computer - they are the means by which that computer communicates with its environment. A peripheral may be considered to consist of two parts:

1) the peripheral device - that hardware that interacts directly with the computer's environment; this hardware covers a very diverse range of equipment. It includes such things as

- electromechanical actuators and transducers
- storage media such as magnetic disks and tapes
- display equipment such as lights and cathode ray tubes
- communication equipment

to name but a few.

2) the peripheral controller - that hardware that interfaces the peripheral device with the rest of the computer; this hardware is essentially a collection of digital logic and signal level translators.

An important function of the peripheral controller is to provide conversions between the form of information used by the peripheral device and that used by the computer. A common process performed is the conversion between the serial data required by a device and the parallel data handled by the computer. Where the task performed by the peripheral is quite complicated (such as required in the operation of a disk drive or a line printer), the logic required within the peripheral controller is quite substantial and is expensive and difficult to design.

If one uses a microprocessor as the basis upon which to design a peripheral controller, the amount of logic required in the controller can be substantially reduced. Additional benefits result from the fact that the microprocessor is programmable and much flexibility is gained in the way in which the controller can be designed and performs its tasks.

In this thesis we are not so much interested in the task of the microprocessor as a peripheral controller but more in providing an interface between a computer and a microprocessor-controlled peripheral that makes use of the microprocessor's capabilities. Traditionally, the structure of the peripheral controller has influenced the design of the interface between the peripheral and the computer. Because the controller consisted of hardwired logic, it meant for instance that the information transferred across the interface was invariably rigidly formatted. The use of a microprocessor changes the criteria by which a peripheral controller is designed, especially in respect to the interface.

It is the object of the thesis to examine how the use of a microprocessor may change the design of the interface between the peripheral and the computer. In the next chapter, chapter 2, the traditional role of the peripheral controller is examined with regard to the type of information that is required by the computer to control the peripheral and the actions required of the controller. In chapter 3, we consider the use of a microprocessor as a peripheral-controller. This leads to a set of design aims for the interface and we discuss in broad details a design that meets these aims.

The specifications of YAC-Bus are presented in chapter 4. YAC-Bus is a bus system proposed as a means of interfacing microprocessor-controlled peripherals to a minicomputer.

Chapter 5 details the implementation of interfaces to YAC-Bus for the PDP-11 minicomputer and the 6800 microprocessor. We also examine the changes necessary to the interfaces to allow them to work with different types of minicomputers and microprocessors.

In chapter 6, the use of YAC-Bus is explored from a software point of view and there is an example illustrating some of the points raised.

Finally, in chapter 7, suggestions for the future development of YAC-Bus are made and conclusions reached in the current research are presented.

CHAPTER 2ARCHITECTURE OF IO SYSTEMS

The purpose of this chapter is to examine the architecture of the Input-Output system of computers, how the designs of two minicomputers typify present designs and what considerations influence the design of IO systems in larger computers.

The IO system provides the communication path between a computer's peripherals and the other components of a computer such as the central processing unit (CPU) and, in some cases, the memory. In considering these two paths of communication, this chapter examines some aspects of the design and structure of the IO system.

2.1 CPU - PERIPHERAL COMMUNICATION

The CPU - peripheral path is a means of communication in which the CPU plays an active and important role in IO operations. If we consider this path as being the sole means by which peripherals communicate with the rest of the computer then the CPU is responsible for the transferring of data between the peripherals and memory in addition to the task of controlling the peripheral. Thus the implementation of the interface between the peripherals and the CPU is important on the grounds of efficiency and ease of use by the programmer.

If one or more registers within the peripheral controller are addressable by CPU instructions, data and control information can be exchanged between the peripheral and the CPU. The design of the Data General Nova minicomputer [DG 78] typifies the traditional approach in the design of this register interface. The main component of the IO system on the Nova is a bus system. This is capable of addressing 64 peripherals and up to 6 16-bit registers (3 read-only, 3 write-only) within each peripheral controller. These registers are accessible by the CPU using a special set of instructions devoted to servicing the registers within each peripheral and performing specialised operations for the IO system.

In contrast, the Digital Equipment PDP-11 [DEC 76] range of minicomputers treats the registers within the peripheral controller as if they were part of memory. Each register is assigned an address within the memory address space so that all instructions that reference memory may gain access to the registers. Thus no special IO instructions are needed and a large number of registers may be accommodated with few restrictions on the number available to a peripheral. This convenience is achieved through the sacrifice of 4096 words of the memory address space of the PDP-11.

We now consider the control requirements for IO operations carried out by the CPU and show their implication for CPU and overall computer efficiency.

2.1.1 A SIMPLE APPROACH TO IO

An IO operation involves the transfer of data between peripherals and the rest of the computer. When this involves the CPU, it could be so arranged that the CPU waits for the peripheral to present data to the interface register on an input operation. In the case of an output operation, the CPU waits until the peripheral is able to accept data for output. In either case, the operation of the CPU is held up until the IO operation has been completed by the peripheral.

An IO operation also involves the issuing of command information by the CPU to the peripheral. For a simple peripheral, this command might imply an input operation to be carried out to fill this register. For peripherals capable of many functions, the function to be carried out may have to be "stated" by the loading of a special register with the appropriate command.

Although this type of IO operation has many serious drawbacks, it was used in early computers and can be found today in some microprocessors. It should be pointed out that the abstract model of IO assumed by most programming languages requires that IO operations be completed before continuing execution. This may be convenient for the applications programmer using the computer but not for the systems programmer involved in keeping the computer working efficiently.

The main drawback to this type of IO operation is that the CPU is effectively halted for the duration of the operation. Such a system would be useless in a real-time control application where several peripherals are being controlled at the same time. Since a typical CPU is capable of executing many instructions during the IO operation of a peripheral, the computer could be doing other and perhaps more important tasks. To allow the computer to do this, we must provide the peripheral with some means of indicating that it is busy performing an operation and the CPU with instructions that test this information and allow the computer to take alternative courses of action.

2.1.2 POLLING PERIPHERALS

By providing the CPU with information about the status of its peripherals, more efficient use can be made of the computer. The status information should be sufficient to allow the CPU to determine what the peripheral is currently doing. Since the CPU can determine if a peripheral is busy or finished an operation, by periodically polling the peripherals for this information, the CPU has the ability to keep the computer running efficiently.

The status information is generally presented in a series of 1-bit flag registers. One flag may indicate if the peripheral is busy performing some IO operation, another may be set when this is finished. Other flags may indicate error conditions or if there is valid data in other registers and so on.

As an example, the PDP-11 [DEC 75] has a status register associated with each of its peripherals. Although the format and contents of this register depend on the design and function of the peripheral, one finds that one bit is designated the Done bit - when set to 1 by the peripheral it indicates that the peripheral has completed the current operation. Other bits indicate the presence of error conditions and other status information. A bit test instruction allows these bits to be tested.

The Nova minicomputer assumes each peripheral has two 1-bit registers known as the Busy and Done registers. These registers can be tested with a special set of conditional skip instructions. Other status information such as the presence of error conditions depends on the design of the peripheral and is usually provided within one of the 16-bit registers within the peripheral controller.

A particular problem with polling arises with input operations. Keyboard input, for example, can be infrequent and at random intervals but the CPU is obliged to check if there has been any data input at a rate that has to ensure that all key depressions are registered - the majority of such checks will have nothing to report.

For a computer with a lot of peripherals, the polling of peripherals can be a considerable task. Polling is complicated by the presence of fast peripherals which, because of their higher data transfer rates, have to be checked more frequently than others.

Polling is an attempt at trying to match the orderly execution of programs in a computer with random events occurring with the operation and use of its peripherals. Its inefficiencies result from the active role of the CPU required in the polling operation. If the peripherals could signal the CPU when they require servicing, the task of the CPU would be reduced considerably.

2.1.3 INTERRUPTS

An interrupt is an event which breaks into the normal operation of the CPU and evokes some response. This event may be the completion of an operation by a peripheral, or when some error condition arises in a peripheral, or it may be some event internal to the CPU such as an arithmetic error (overflow, divide by zero etc) or power failure. The response by the CPU has two parts:

- 1) the currently running program is halted and sufficient information about the "state" of the CPU is saved to allow the program to be continued later. Such information

always includes the contents of the program counter and may include the contents of other important registers.

2) automatic transfer of control to a routine to service the interrupt.

An interrupt system frees the CPU from the task of constantly polling peripherals to check their status. However, there are extra requirements in the design of the computer that are needed in order to cope with interrupts. Because there may be many sources of interrupts, the CPU requires information to identify the source to allow for fast servicing. Interrupts can happen at any time, even simultaneously, so there must be a method of determining which interrupts have priority over others for servicing by the CPU. The interrupt service routine may itself be interrupted as it may be important for very fast peripherals to be able to preempt the servicing of slower ones.

By examining the interrupt system of the two mini-computers being considered, it is possible to see how such issues are resolved. The PDP-11 uses a vectored interrupt system in which the peripheral requesting the interrupt supplies the CPU with an "interrupt vector", an address from which new values of the program counter and program status word (PSW) are loaded, the previous values being pushed onto the stack. The PSW contains important information about the class of peripherals that may interrupt the program currently running. The effect of the vectored interrupt system is to quickly identify the interrupting peripheral as well as to branch to the particular interrupt service routine for the peripheral and change the priority conditions for future interrupts. The interrupt service routine is responsible for the saving and restoring of the contents of any CPU registers used.

The Nova minicomputer requires more on the part of the operating systems programmer to handle interrupts. Upon an interrupt, the Nova CPU places the current value of the program counter in location 0 of memory and picks up the address of the interrupt service routine from location 1.

Identification of the peripheral requesting the interrupt must be done by software, an "interrupt acknowledge" instruction being provided to obtain this information from the peripheral itself. Those classes of peripherals that may interrupt during the servicing of an interrupt are set through the use of the "mask out" instruction.

An enhanced version of the Nova, the Data General Eclipse minicomputer [DG 75], has a special instruction that does a lot of the work required to service an interrupt. Known as the "vector on interrupting device code" instruction, it transfers control to the service routine appropriate to the interrupting peripheral with a series of options. These provide for changing the stack (important when a user program is interrupted), providing a new interrupt mask and saving important CPU registers such as the accumulators.

2.2 PERIPHERAL - MEMORY COMMUNICATION

Peripherals such as disks and magnetic tapes are capable of quite high data transfer rates once the recording medium has been positioned correctly. To improve the efficiency of such peripherals, it is usual for the data to be stored in blocks of several hundreds or even thousands of words or characters. IO operations involving these blocks of information would be particularly wasteful of CPU capacity if carried out in the manner suggested in the previous section. In practice, it is usual to find that such peripherals have specialised hardware to allow blocks of data to be transferred directly to and from memory once the transfer has been initiated by instructions from the CPU. The ability to transfer data directly to memory requires further capabilities on the part of the IO system.

2.2.1 DIRECT MEMORY ACCESS FACILITIES

An approach used in the design of most minicomputers is to provide as part of the IO system a path to memory for use by those peripherals requiring block transfer facilities. This path provides what is termed direct memory access (DMA). In addition to data and peripheral addressing capabilities

of the IO system, memory address information is required as well for DMA transfer of data. In minicomputers, this address information is held by the peripheral together with information as to the amount of data to be transferred.

The exact way in which a DMA is provided depends on the computer architecture. The PDP-11 is based on a centralised bus system called Unibus which provides access to memory for its peripherals as well as the CPU. While the CPU is solely responsible for determining who has control of the bus, it can surrender control to any peripheral that requests the use of Unibus. This then allows the peripheral to access memory to transfer data to and from memory etc.

In contrast, the Nova CPU is responsible for the control of DMA, including the generation of timing signals for the peripherals to make DMA requests and for strobing address and data information onto the IO bus.

Although DMA greatly facilitates the fast transfer of blocks of data for IO operations, there are some limitations and problems. A particularly serious problem is that of memory contention on the larger minicomputer systems. When the CPU and peripherals compete with each other over obtaining access to memory, it is usual for the CPU to have lowest priority. This is because the peripherals which use DMA such as disk and magnetic tape drives require the data to be transferred within a certain time which depends on the speed of the peripheral and any buffering of data within the peripheral controller. Where several operations involving DMA are running at the same time, the CPU is essentially "frozen" resulting in an overall degradation of performance. Improving performance under such circumstances requires the use of multiple-path access to memory, the interleaving of memory cycles, or simply the use of faster memory.

2.2.2 THE IO PROCESSOR

On computers larger than minicomputers, it is normal to find that the IO system is based on a specialised processor - the IO processor. The main function of the IO processor is to relieve the CPU of many of the control functions and

routine operations associated with the IO system. As an example of the CPU's involvement, the Nova minicomputer CPU is responsible for all IO system timing and control as well as providing the interface to memory for the DMA facilities.

The IO processor may also centralise the DMA facilities. In a minicomputer, a substantial part of the operation of DMA depends on hardware provided within the peripherals themselves. If a computer has many peripherals, each with its own DMA capability, there is considerable duplication of hardware. This duplication is inefficient and a waste of resources since the capacity of the memory limits the number of peripherals that can access memory simultaneously, regardless of the use of an IO processor. By centralising the operation of DMA facilities, duplication is avoided and also allows for the provision of DMA for all peripherals. This reduces the task of the CPU in the management of IO operations to simply initiating transfers of data and the servicing of interrupts.

Other features of an IO processor can be seen in examining a typical example, the IO processor of the Burroughs B6700 [Burroughs 72]. A B6700 configuration can support up to 3 IO processors controlling up to a total of 256 peripherals. A peripheral can be connected to several IO processors so that, if one is busy or non-operational, the peripheral can still be used.

The facilities provided by the IO processor include DMA for all peripherals. Within the IO processor are buffers for the packing and unpacking of data from the 48-bit word size of memory to the 8 to 16 bits of data transferred by the peripheral. The IO processor supports from 4 to 10 simultaneous IO operations in a manner that suggests its alternative name of a multiplexor.

The IO processor is controlled by the "scan-in" and "scan-out" instructions of the CPU(s). When used to control the operation of the peripherals, these instructions provide information for a single operation of the peripheral (e.g. a read or write operation). This is in contrast with IO processors on other computers. In particular, the

IO processor (or "channel") on the IBM 360/370 range of computers [IBM 74] has the ability to execute a "channel program", essentially a sequence of IO operations. Information contained within a channel program selects the type of IO operations to be performed and the locations of buffers for a particular peripheral. Unfortunately, channel programs are very limited in their capabilities and appear to be no more than a means of stringing together commands to a peripheral to reduce the frequency of IO interrupts.

In conclusion, the use of IO processors provides a means of taking care of the simple and repetitive operations involved in the transfer of data for peripherals by a specialised processor that is available for the use of all peripherals. Although the IO processor reduces the involvement of the CPU in the transfer of data to and from the peripheral, the IO processor itself does little, if any, processing of the information.

2.3 THE PERIPHERAL AS A PROCESSOR

Until now it had been assumed that the peripheral and, in particular the controller, had very limited processing capabilities; the reason being that any substantial processing was the responsibility of the CPU. For a peripheral that requires constant attention and considerable processing by the CPU to support, it may pay if a dedicated processor is used to take over the "low level" control of the peripheral, reducing the involvement by the CPU to that of a supervisory role.

A typical and frequent application of such a dedicated processor is for the control of communication lines. There may be considerable processing required to handle line protocols and the conversion of data between the format used internally to the computer and that external to it. Although such a processor may be a separate computer with its own memory etc, as far as the CPU is concerned, it is still a peripheral.

There are several methods by which the idea of a dedicated processor to control a peripheral or group of peripherals may be expanded. One could provide a pool of processors to which one allocates to the peripherals as the need arises. This approach was taken in the case of Control Data's 6600 range of computers [CDC 66] where up to 10 "peripheral processing units" (PPUs) could be provided in a single computer. Although some PPUs may have dedicated uses (e.g. monitor), the rest are available to be allocated to any peripheral.

An alternative approach is to provide each peripheral with its own dedicated processor. With the low cost of hardware, this approach is very feasible and can be seen in computers like the IBM System/38 [IBM 78] . An advantage with this approach is that the processor can be tailored to fit the peripheral and costs can be further reduced by programming into the processor some of the functions of the hardware.

The low cost of microprocessors make them an attractive proposition for use as a dedicated processor for a peripheral in minicomputer applications where cost is important. The rest of the thesis examines this idea further and in particular investigates the interface between a minicomputer and a microprocessor - controlled peripheral with the view of providing an efficient means of communication.

CHAPTER 3THE MICROPROCESSOR AS A PERIPHERAL CONTROLLER

This chapter considers the microprocessor in the role of a peripheral controller. The interface between peripheral and computer is examined in relation to the information passed across it. Using a microprocessor efficiently as a peripheral controller presents difficulties when used with current interfacing techniques and a means of solving these problems is presented in providing the background material for the design of YAC-Bus.

3.1 THE PRESENT FORM OF A COMPUTER-PERIPHERAL INTERFACE

The computer-peripheral interface can be studied at many levels. An applications programmer writing a program to use a peripheral sees that peripheral through a software interface in terms of the abstract model of the peripheral assumed by the programming language. On the other hand, the actual hardware interface between peripheral and computer is generally oriented towards the physical characteristics of the peripheral. Bridging the gap between these two interfaces is software involving the run-time service routines of the programming language and the "device driver" routines of the operating system.

The physical orientation of the hardware interface may mean that the device driver software has a considerable task to do in resolving the differences between the hardware and software interfaces. As an example of the task required of the device driver, a disk drive requires data to be addressed in terms of head, sector and track numbers rather than the more abstract notions of file name and record number.

In some respects, this orientation of the hardware interface has some advantages. It means that many aspects of a peripheral are under the direct control of software. Considering the example of the disk drive, if the file structure is completely under control of the software, the file structure of the disk can be defined by the systems

programmer. The file structure is of particular importance as it is one of the contributing factors to the overall performance of the disk drive. Thus by "tuning" the file structure through the software, it is possible to improve the performance of the system using the disk.

For a complex peripheral, the task of designing its controller and interface is difficult and expensive. One can simplify this task by making the controller as simple as possible and letting the computer take over some of the controller functions. As a consequence, what seems to be a simple IO operation by an applications program may require several more primitive operations on the part of the peripheral and considerable processing by the CPU.

The extent of the differences between the hardware and software interfaces can be gauged in reconsidering the example of the disk. Suppose a program issues a request to read a certain record of a file stored on disk. Assuming that the location of the file is already known, it is then necessary to determine where that record is in terms of head, track and sector numbers and then issue a read command to the disk for the appropriate sector (finding the record may involve more disk reads to get further information about the location of the record). Once the data has been read from the disk, additional processing by the CPU may be necessary to extract the record and provide it in a form suitable for the applications program (e.g. provide trailing blanks, translation (ASCII to EBCDIC perhaps) and formatting of data and so on). Status information returned to the program may indicate if the read was successful; alternatively if the read failed, details of the error may be returned to the program to allow it to attempt to remedy the situation. It should be noted that error conditions returned by a peripheral are usually serious enough to force the operating system to abort the applications program whereas the error information supplied to the program mainly comes from the device driver software and is usually the result of some logical error by the programmer.

Although the information that passes across the hardware and software interfaces may be classified into common categories of commands, status information and data, the above example shows that there are considerable differences between the requirements of the information at the two interfaces. At the hardware interface, one is concerned with the control of the peripheral whereas at the software interface, one is attempting to present a congenial abstract model of the peripheral to the applications programmer. To reduce the task of the device driver software, the differences between the two interfaces must be reduced. Thus one has to produce a peripheral that is closer to the requirements of the software interface. To ease the task of designing the hardware of the peripheral controller, we now consider the use of a microprocessor to control the peripheral.

3.2 THE MICROPROCESSOR

The microprocessor has been successful because of its low cost and the fact that its introduction has meant the lowering of hardware complexity in many applications. The processing capabilities of the microprocessor allow one to design quite powerful functions into a system which would otherwise require complex and expensive hardware to implement.

3.2.1 REPLACING EXISTING HARDWARE WITH A MICROPROCESSOR

As a peripheral controller, the microprocessor is well suited to carry out both control functions and data manipulation. For types of peripherals such as visual display units (VDUs) and matrix serial line printers, the speed of operation is such that a microprocessor is capable of implementing the more sophisticated control functions as well as the more rudimentary ones. As examples, one finds VDUs with complex and comprehensive editing functions and printers with buffer management, bidirectional printing, alternative character sets and graphic mode printing.

Such uses of the microprocessor only replace complex hardwired logic and leave the hardware interface unchanged.

If one uses a microprocessor as a peripheral controller then the rationale of the hardware interface is not necessarily valid. How does the use of a microprocessor change the role of the hardware interface?

3.2.2 EXPLOITING THE MICROPROCESSOR'S CAPABILITIES

The use of a microprocessor as the controller of a peripheral frees the format of the information passed between computer and peripheral, especially the command and status information, from being too dependent on the physical characteristics of the peripheral. The command and status information required depends on the functions carried out by the peripheral. As an example, a card reader has only one command - that to read a card. However, a disk drive has several functions - seek a new track, block reads and writes and so on.

By using the processing capability of the microprocessor, it should be possible to take over the low level functions of the operating system software that involve the peripheral - the device-dependent aspects of the device driver being a good place to start. This has the immediate advantages of releasing memory and processing resources of the computer for other work. It also changes the type of information issued in commands to the peripheral and received in status interrogation.

The degree to which these changes are made depends on the relative processing capabilities of the microprocessor and the computer. A microprocessor with limited processing capabilities would require its information "pre-digested" while a more powerful microprocessor may be capable of taking information in the form the applications programmer uses in his program. The processing capability of the microprocessor could allow the hardware interface to be designed to different criteria and, in particular, closer to the abstract model of IO seen by the applications programmer.

The power of the microprocessor results from the fact that it is programmable. The programs need not be a permanent part of the peripheral - there are many advantages

in having them loaded only when the peripheral is to be used. For instance it allows different programs to be used for different purposes; the program to be loaded depending on the programming language of the applications program running on the computer or even the particular person writing the applications program. Other advantages result from the ability to change the operation of the peripheral as requirements change during its lifetime. This increases the flexibility and usefulness of the peripheral considerably.

Given these possible advantages of the use of microprocessor-based peripheral controllers, let us now consider the design of an interface between the computer and the peripheral that exploits the processing capability of the microprocessor.

3.3 DESIGN AIMS FOR A NEW HARDWARE INTERFACE

The use of a microprocessor-controlled peripheral places different requirements on the design of the hardware interface. These requirements may be considered to lie in four main areas:

- i) efficiency
- ii) transformation of the hardware interface
- iii) flexibility
- iv) portability.

3.3.1 EFFICIENCY

The interface must provide efficient communication. Although the transfer of command and status information involves the handling of small and convenient amounts of information (characters or words), the amount of data encountered in an IO operation may not necessarily allow for such easy handling.

To transfer information efficiently between peripheral and computer requires minimal processing by the CPU and the microprocessor. In particular, there should be no unnecessary handling of the information. For instance, if data is packed into a buffer by an

applications program, the CPU should not have to unpack the data just to output it to the peripheral - the peripheral should be able to handle the data in its original form.

Efficiency is not only required in the mechanism of information transfer but also in the way the information is formatted. There is little point in providing an efficient method of information transfer only to find that the CPU has to do considerable processing to prepare the data in a suitable form for the applications program. Part of the data processing could be done by the microprocessor within the peripheral itself. This would increase the overall efficiency of the computer as it releases memory and CPU resources for other work.

3.3.2 TRANSFORMATION OF THE HARDWARE INTERFACE

The use of a microprocessor-based controller allows the interface to be brought closer to the requirements of applications programs using the peripheral. This is an important transformation of the hardware interface as it is usually designed with regard to the physical characteristics of the peripheral. The transformation is the result of the use of the processing capabilities of the microprocessor which allow hardware restrictions (e.g. the format of information) to be removed. The transformation particularly effects command and status information as it is this information that is mostly hardware dependent. As an example, a disk could be accessed in terms of an address somewhat more abstract than the usual head, track and sector numbers. Overall, the effect is that the hardware interface must now be capable of handling information whose format and other attributes are defined only by software.

3.3.3 FLEXIBILITY

The use of a microprocessor-based controller makes the peripheral inherently flexible as changes in

the requirements of its operation can be met by re-programming. In fact, different types of peripherals could be presented to a common interface as the presence of a microprocessor allows their various idiosyncracies to be accommodated. Such standardisation is important to the applications programmer, both in learning about the computer system and in adapting programs for use with different peripherals. At the hardware interface, flexibility is important from the point of view of handling information whose attributes are defined by software and so are liable to change at any time.

3.3.4 PORTABILITY

The hardware interface should be, as much as possible, independent of the particular computer or microprocessor used. This allows the basic design work involved with the hardware interface to be usable with different computers, microprocessors, or peripheral devices.

The above design aims give some guidelines to the design of the hardware interface. Unfortunately, they are not consistent in that considerations of efficiency conflict with those of portability and flexibility and so on. Ultimately, it is the facilities within a computer available to implement the hardware interface that determine the extent to which one may achieve one's objectives. It is now appropriate to examine these facilities.

3.4 MINICOMPUTER IO FACILITIES

Minicomputer manufacturers provide detailed information about the structure and operation of the IO system for those users wanting to design and build their own customised IO interfaces. Interfaces to other parts of a computer are not generally sanctioned and there may be the additional difficulty of finding adequate information for use in designing such interfaces.

The IO systems of most minicomputers generally provide two paths for the transfer of information between a computer and its peripherals:

- registers within the peripheral and accessible by the CPU
- direct access by the peripheral to memory (DMA).

3.4.1 REGISTER INTERFACES

The register interface between a computer and peripheral consists of register storage within the peripheral controller accessible by the CPU. As an example, a disk drive has several registers holding command and status information and other information necessary for the operation of the disk such as memory and disk addresses and so on.

The register interface is a feature common to all minicomputers although the design of a particular minicomputer limits the size and number of these registers. The use of the register interface is the generally accepted method of controlling the peripheral by the CPU - in particular a DMA transfer by a peripheral may have the parameters set up in registers within the peripheral.

3.4.2 DMA INTERFACES

DMA provides the peripheral with a direct access path to the computer's memory which may be shared by several peripherals. In operation, the peripheral requests and is granted a memory access cycle, giving the peripheral access to data anywhere in memory.

The power of DMA results from its ability to allow a peripheral to transfer data to and from memory independently of the CPU. However, this power is at the expense of increased complexity of the hardware within the peripheral. At this point we must consider a fundamental design limitation. Because of space limitations within the minicomputer used for experimentation (a PDP-11/40), if some type of DMA facility were provided within the interface then there would be little room for other hardware. This limitation was accentuated by details such as the need to reconcile the different word sizes used by the minicomputer and the microprocessor.

A decision was made to design an interface that attempted to do without the use of DMA for the transfer of data. Such an interface would have to use the register interface as the sole means of communication between the peripheral and the computer.

3.5 AN INTERFACE FOR A MICROPROCESSOR-CONTROLLED PERIPHERAL

A microprocessor-controlled peripheral may be considered to consist of three main components (see figure 3.1):

- i) the device (actuators, transducers and the like) used to communicate with the computer's environment
- ii) the microprocessor
- iii) memory for storing programs and providing working storage for the microprocessor.

The storage of data handled by the peripheral in the course of its operation is of particular importance as where and how it is stored influences how efficiently such data can be transferred between the peripheral and a computer. Suppose this data is stored in the memory used by the microprocessor. To gain access to this data from outside of the peripheral, one must supply an address and a path to transfer the data. By making both the address and the data available through the register interface of the IO system of a minicomputer, the CPU can gain access to the data handled by the peripheral.

The data can be accessed quite efficiently by the CPU. The address is loaded into a register within the register interface, the data being made available through another register. Note that the CPU cannot address the data directly as would be possible if the data were stored in the computer's memory. Instead one has to use index register techniques where a register is used to hold the address of data being accessed by the CPU.

Inefficiencies arise if, for instance, an attempt is made by the CPU to handle data sequentially within the microprocessor's memory. Because sequential processing of data is a particularly common operation, the register containing the address (henceforth known as the index register) can be provided with an auto-incrementing feature. Whenever the register containing the data (henceforth known as the data

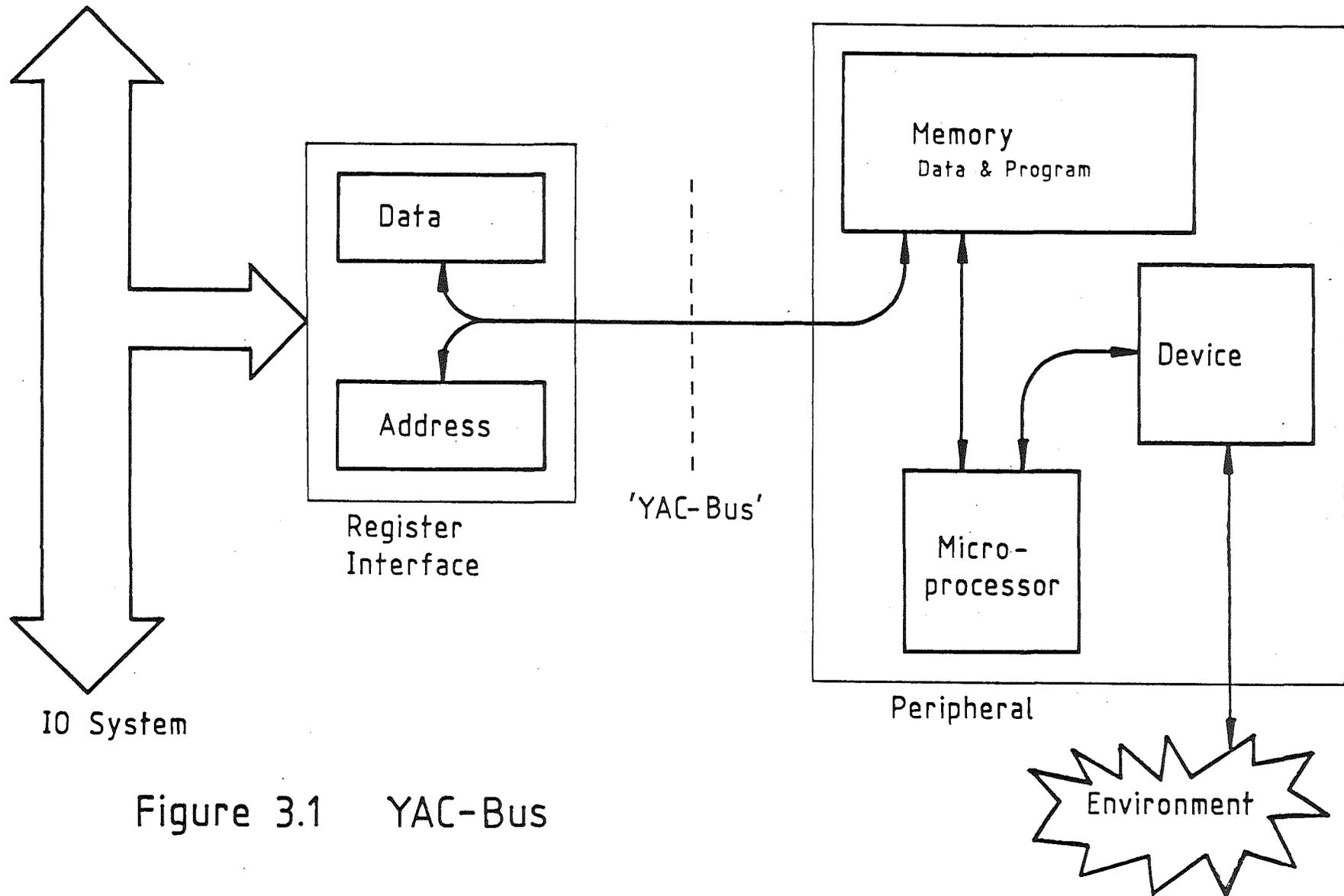


Figure 3.1 YAC-Bus

register) is accessed, the address would be incremented and so remove the need for the CPU to perform such a function.

It must be conceded that there are disadvantages in storing the data handled by the peripheral within the microprocessor's memory. For instance, since the data handled by the peripheral is stored outside of the computer's memory, it restricts the type of instructions that can be used to manipulate the data, especially where the computer has powerful addressing modes and instructions for manipulating strings of characters and decimal numbers.

Although the proposed bus as described above is usable, there are several items that are needed to make the hardware interface easier to use and provide it with the essential features of an IO system.

The first and most important item is the provision of an interrupt system. This is to allow the peripheral to interrupt the minicomputer in a similar manner to a conventional peripheral. Equally important is the ability of the minicomputer to interrupt the microprocessor. Because the microprocessor operates as a "slave" processor, it is important for the minicomputer to have control over its operation.

A feature allowing for the efficient use of the bus and interface is using the index register to address several peripherals rather than just the one. Since the CPU only needs access to that memory used to store the data handled by the peripheral and not the whole address space, the spare bits within the index register can be used to select one of several peripherals.

The address and data lines (as well as the appropriate control lines) transferring information between the minicomputer and the peripherals form a bus (henceforth called YAC-Bus). This bus provides a common interface to which one provides specialised adaptors to the minicomputers and peripherals. Some degree of portability and machine independence is possible in designing these interfaces as there are features that are essentially independent of the architecture of the minicomputers and microprocessors using the bus.

Rather than increase the complexity of the bus connecting the minicomputer and the peripherals, it was decided to use part of the address space of YAC-Bus and existing functions (read, write) of the bus to implement the interrupt system. Thus the minicomputer interrupts a microprocessor by writing to a designated address via the bus and vice versa; the interrupt system being provided as functions of the minicomputer and peripheral interfaces to YAC-Bus.

3.6 THE SUITABILITY OF OTHER BUS SYSTEMS

At this stage the reader may be wondering why yet another computer bus system (hence the name YAC-Bus) is being proposed, given the present proliferation of "standard" bus systems. The simple answer is that such bus systems provide facilities that go far beyond those required to implement YAC-Bus.

The implementation of YAC-Bus requires little more than providing addressing and data transfer capabilities. Bus systems such as the S/100 [Elmquist 1979] provide many more facilities (system power, status signals, vectored interrupts, DMA and so on) than could be hoped to be supplied through the minicomputer interface. Rather than implementing a subset of the facilities of an existing bus system, it was decided to design a new one. This had the advantage of allowing experimentation in various aspects of bus design, especially in the area of bus protocols where it was noticed that many bus systems have redundant signals and actions.

The architecture of YAC-Bus is not unique. Hirschman et. al. [Hirschman 1979] describe a bus system that is similar to YAC-Bus in several aspects and is used in the more general multi-microprocessor situation where it provides a communications link between several microprocessors. YAC-Bus may be regarded as a simplification in that the interaction has been reduced to a single minicomputer communicating with several microprocessors. Because there is usually little need for peripherals to communicate with each other, the peripherals are not provided with such facilities. This is not considered a disadvantage as inter-peripheral communication

can be provided when and as required by the CPU acting as a message switcher.

CHAPTER 4

YAC-BUS ARCHITECTURE

The purpose of this chapter is to examine the architecture of YAC-Bus, how the minicomputer and its peripherals may interface to it, and finally show the development of its signal protocols.

4.1 YAC-BUS COMPONENTS

YAC-Bus provides a means of communication between a minicomputer and several microprocessor-controlled peripherals. It has been designed to be as machine-independent as possible, so that it may be used with a variety of minicomputers and microprocessors. A block diagram of a typical YAC-Bus configuration is shown in figure 4.1.

There are four principal components:

- i) the bus itself, comprised of address, data and control lines
- ii) the minicomputer interface
- iii) the peripheral interface
- iv) the interrupt system (not shown in figure 4.1).

4.1.1 YAC-BUS

YAC-Bus consists of 29 signal lines which can be divided into two groups - 24 information lines (tri-state) and 5 control lines (open-collector). The information lines may in turn be grouped into 16 address lines and 8 data lines.

The address lines provide an address space for 64k ($65536 = 2^{16}$) bytes of memory to be used for communication between minicomputer and microprocessors. For the implementation illustrated in this thesis, the memory was split into 16 blocks of 4k (4096) bytes, each block being allocated to a peripheral. There are few reasons, other than size

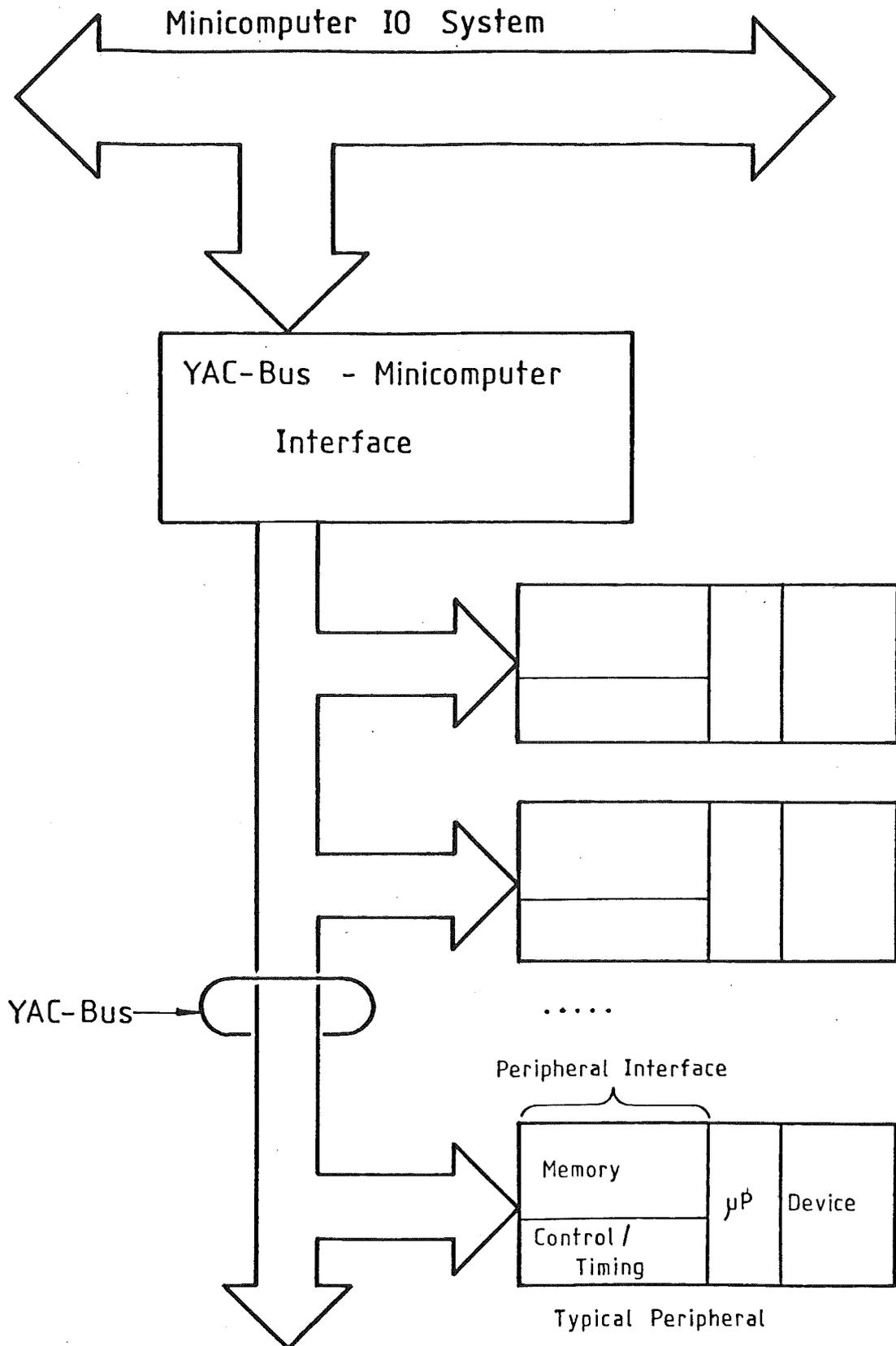


Figure 4.1 YAC-Bus Configuration

limitations of registers within the minicomputer interface, why different values of block size and number of blocks cannot be chosen. For the particular implementation shown here, the main factors determining the split were certain hardware limitations (e.g. bus loading) and the desire to reduce component count.

Since most of the common microprocessors work in bytes (8-bits), this was chosen as the word size. This means that YAC-Bus is most efficient working with bytes, a common unit of information handled by many peripherals. A 16-bit word size would have been convenient from the point of view of the minicomputer but had the disadvantage of increasing the hardware complexity.

Three of the control lines are used for bus control and synchronisation and are described in section 4.2. The other two lines are RESET, that performs a power-on reset function, and R/W (read/write) for access to the memory.

4.1.2 THE MINICOMPUTER INTERFACE

There are two registers provided within the minicomputer interface that allow the minicomputer to communicate with a peripheral via YAC-Bus. The index register addresses the peripheral and a byte within its 4k block of YAC-Bus address space (see figure 4.2). The byte selected by the index register is accessed through another register, the data register. The index register is provided with an auto-incrementing function to make the task of transferring blocks of data more efficient, the contents being incremented after every access of the data register.

The interaction between the data and index registers can be seen by considering a simple example. Suppose a section of YAC-Bus memory has the following contents:

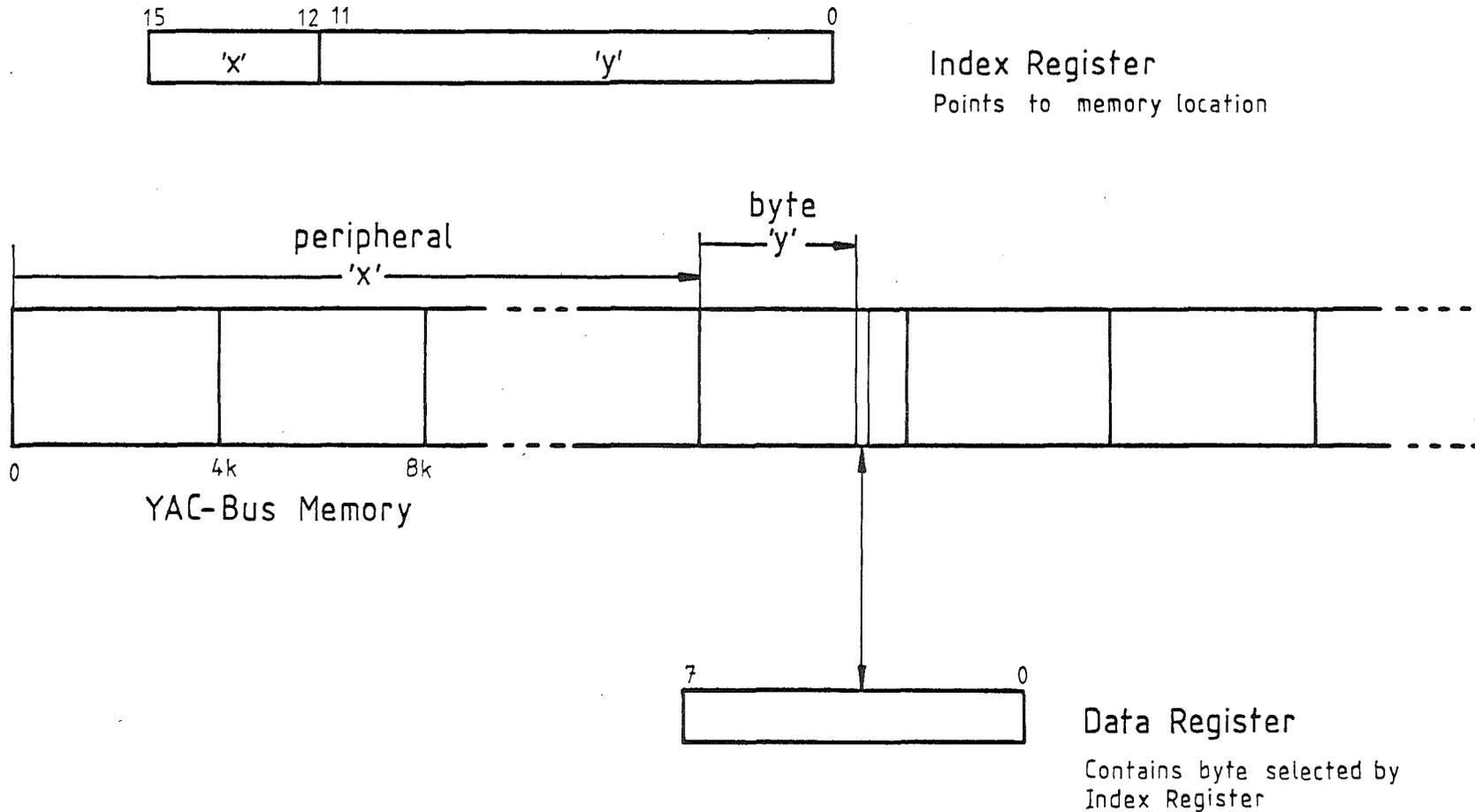


Figure 4.2 YAC-Bus Memory System

location	contents
⋮	⋮
18	34
19	45
1A	56
1B	67
1C	78
⋮	⋮

If we load the index register with 18, reading the data register will yield the value 34, and reading it again gives the value of 45. The index register now points to location 1A and if we were to read the data register again it would contain 56. Instead, if we now write FE and ED successively to the data register, our section of memory would have the following contents:

location	contents
⋮	⋮
18	34
19	45
1A	FE
1B	ED
1C	78
⋮	⋮

The contents of the data register is now 78 as the index register now points to location 1C.

Because the data register is only updated at the time the index register is altered, the programmer should be cautioned against assuming that the data register reflects the current contents of the corresponding YAC-Bus memory location.

A third register, the interrupt register, contains information about a peripheral that requests an interrupt to the minicomputer. This register and the operation of the interrupt system are described in section 4.1.4.

Certain additional status and control registers are also required - their function and implementation being very machine dependent. These include the control of interrupts from the peripherals and the indication of whether or not the contents of the data and interrupt registers are valid.

4.1.3 THE PERIPHERAL INTERFACE

The peripheral interface consists mainly of a block of dual-ported memory. The memory can be accessed by YAC-Bus or by the peripheral's microprocessor. The microprocessor can access the memory independently of YAC-Bus except when attempting to interrupt the minicomputer (see the next section).

An access via YAC-Bus has priority over the microprocessor in using this block of memory, even when the memory is being accessed by the microprocessor. This is to minimise as much as possible the time required to carry out an operation on YAC-Bus. As far as the microprocessor is concerned, during a clash over memory access, the access appears to take longer than usual and only requires of the microprocessor the ability to handle slow memory.

The remainder of the logic within the interface handles the YAC-Bus signal protocol and the interrupt system insofar as it affects the peripheral.

4.1.4 THE INTERRUPT SYSTEM

Incorporated into the design of YAC-Bus is an interrupt system. It allows the minicomputer to interrupt the microprocessor in order to supply it with a command and allows the microprocessor to interrupt the minicomputer more or less in

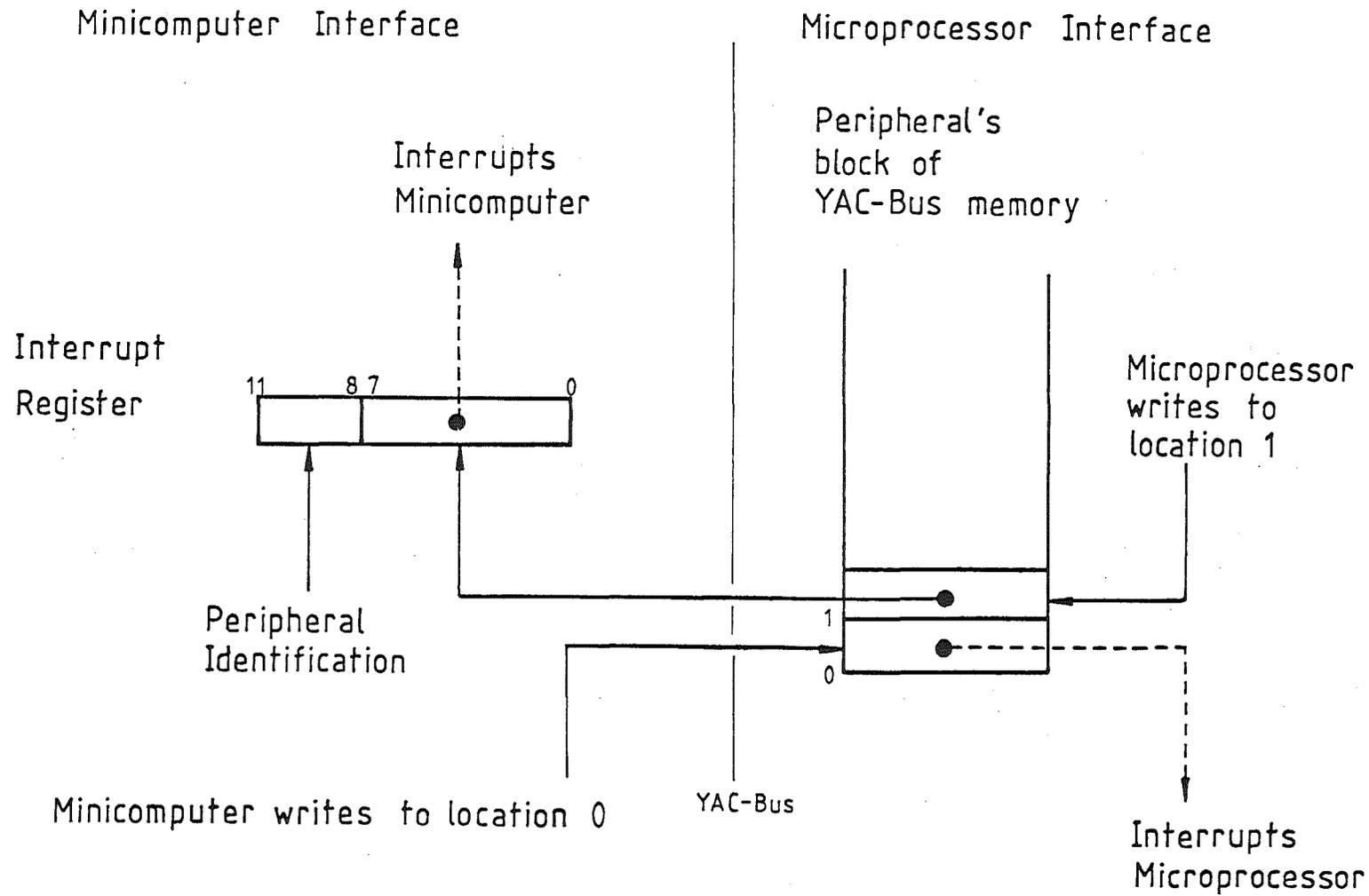


Figure 4.3 YAC-Bus Interrupt System

the same way as a conventional peripheral.

The key to the operation of the interrupt system is the provision of special functions associated with two memory locations in the peripheral's block of memory. For convenience, these are locations 0 and 1 (see figure 4.3). Location 0 may be regarded as a command register. If this location is loaded from YAC-Bus, it will cause the microprocessor to be interrupted as well as accepting a byte of data. This byte of information is used as a command from the minicomputer to the peripheral.

The function associated with location 1 is somewhat more complex. If written to by the microprocessor, it will cause YAC-Bus to be accessed. The lower 12 address lines from the microprocessor are just sent through to YAC-Bus ($=001_{16}$), denoting that the peripheral is attempting to interrupt the minicomputer, while the upper 4 address lines are set to the address of the peripheral. The result is to write the byte of data to the interrupt register within the minicomputer interface along with the upper 4 address bits identifying the peripheral. This byte conveys status or error information. The minicomputer is then interrupted.

A lock associated with the interrupt register prevents its contents from being overwritten until the register has been read by the minicomputer. The YAC-Bus protocol (described in the next section) informs the microprocessor if it has been locked out of the interrupt register and allows it to attempt further accesses until successful. To be sure that the minicomputer is interrupted by the peripheral, the microprocessor must keep attempting to write to location 1 until it is not interrupted.

4.2 YAC-BUS PROTOCOL

Any operation involving YAC-Bus goes through two phases:

- i) gaining control of YAC-Bus
- ii) accessing the information via YAC-Bus.

There are many signal protocols that could have been used to control these operations - the protocols being almost as prolific as computer bus systems. A summary of the basic techniques used in the control of digital bus structures can be found in Thurber et al [Thurber 72] .

Because YAC-Bus is intended to be used close to a minicomputer, problems associated with the use of long lines (noise, speed, skew etc) do not arise and simple protocols can be used. Such protocols have to be asynchronous because the system was designed to be used with a variety of minicomputers and microprocessors, some of them like the PDP-11 being basically asynchronous in nature, and there was no hope of finding common ground between them to use a synchronous protocol.

The protocols for YAC-Bus are designed to have minimal control requirements and a simple hardware implementation. To achieve some of the simplicity, it was noted that many protocols have redundant actions and signals that can be combined. The nett result is that the control of YAC-Bus is based on only three signals whose actions are interlinked to some extent with each other and other aspects of the operation of the bus as a whole.

4.2.1 BUS CONTROL

From previous discussions about the operation of YAC-Bus, the reader should be aware that the minicomputer shares the use of the bus with the peripherals. It is the function of the bus control protocol to unambiguously grant control of YAC-Bus to either the minicomputer or any of the peripherals that request its use. Once control has been granted, it cannot be preempted in any way until relinquished by whoever it was granted to.

Such a policy places certain requirements on the design of the minicomputer interface and the bus control protocol. The interaction of the minicomputer with YAC-Bus has assumed that whenever the index register of the minicomputer interface has been changed in any way (i.e. written to or incremented), the contents of the data register are

updated to reflect such a change. This requires the minicomputer interface to gain control of YAC-Bus to retrieve data for the data register.

For most minicomputers there is sufficient time (several instruction cycles) between the altering of the index register (i.e. supplying an address) and the subsequent accessing of the data register (the time by which the data has to be available) for the interface to await the completion of the current operation of YAC-Bus and then access the bus itself. In the case of the PDP-11, the timing requirements are more flexible as the Unibus can delay the CPU for up to $10\mu\text{s}$ to allow data to be processed or become available. This allows several shortcuts to be taken in the design of the minicomputer interface, the details of which will be explained in the next chapter.

The bus-granting mechanism must therefore be under some measure of control by the minicomputer interface and should also provide a priority system since this controls the relative priority of interrupts from the peripherals.

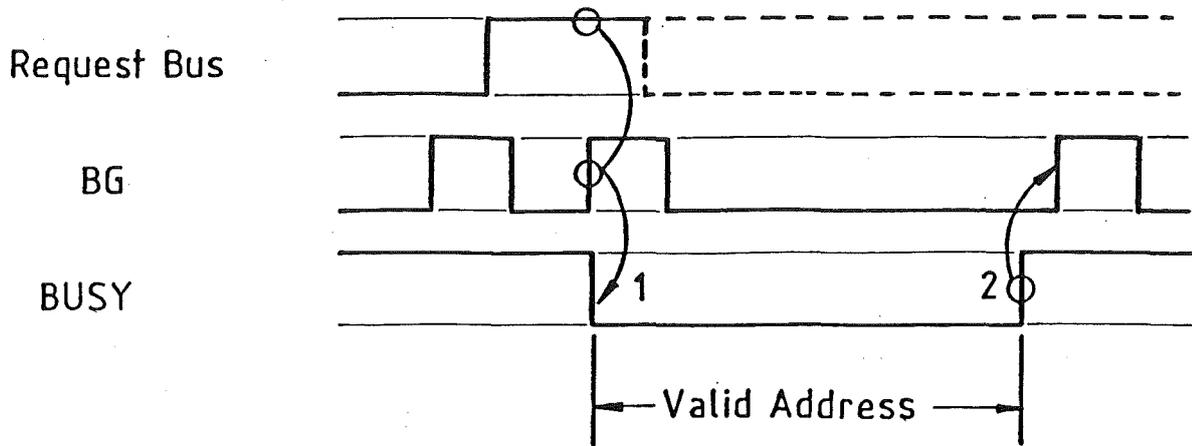
The bus control method used for YAC-Bus is a variation on that used fairly commonly for priority handling within an IO system. It makes use of two control lines, BG (Bus Grant) and BUSY. The BG line is chained through all the peripheral interfaces connected to YAC-Bus.

With reference to figure 4.4a, bus control works in the following way:

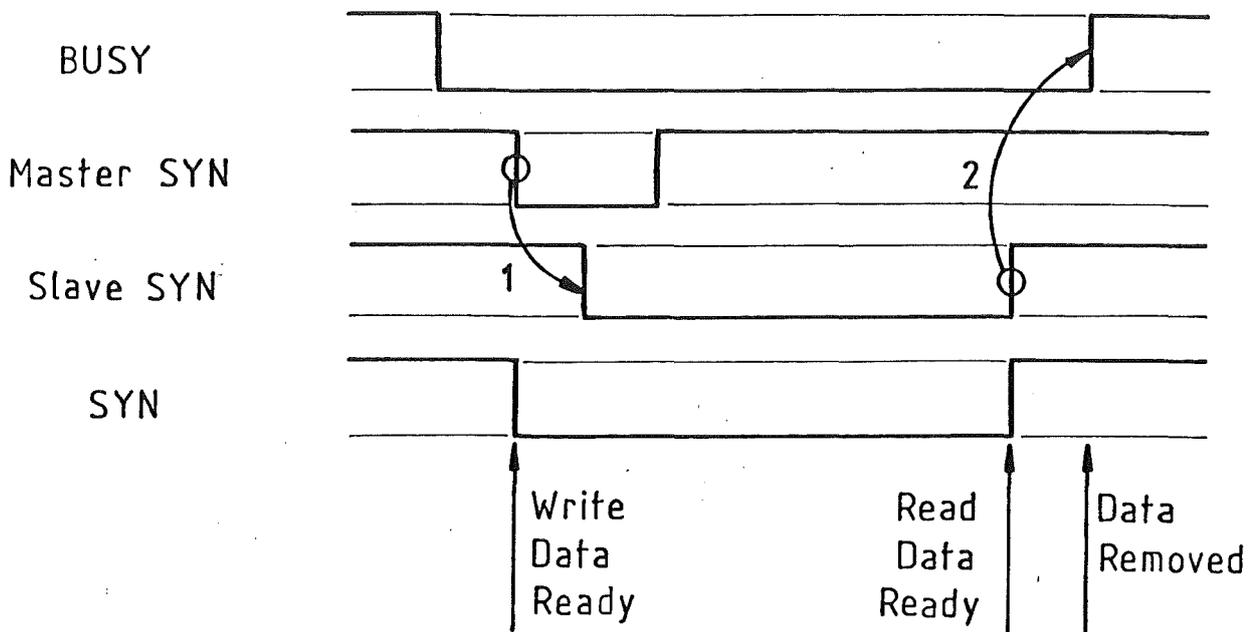
- the BG signal is a series of pulses that originates from the minicomputer interface when YAC-Bus is not being used.

- when a particular peripheral wishes to use YAC-Bus, the low-high transition of BG is blocked at its interface from propagating further along the bus and the peripheral has control of YAC-Bus when the interface receives the low-high BG signal (event 1).

- the interface then pulls BUSY low to signify that the bus is being used (the importance of this is explained in the next section).



a) Bus Acquisition



b) SYN Timing

Figure 4.4 YAC-Bus Timing

- low BUSY stops any further low-high transition of the BG line.

- when the peripheral has finished with YAC-Bus, BUSY is returned high and BG continues pulsing (event 2).

- since the BG signal originates from the minicomputer interface, that interface has highest priority access to YAC-Bus. This means that the minicomputer interface can always get control of YAC-Bus immediately the present access cycle has finished.

This bus control system has minimal requirements as regards the number of lines required but is complicated by the timing requirements of the BG signal. To reduce the latency in gaining access to YAC-Bus, the interval between successive BG signals when the bus is not in use should be as short as possible. This interval, unfortunately, will vary with the application as it depends on the length of the bus and the delay of the BG signal through the peripheral controllers under worst case conditions. The time must be sufficient to allow the last controller connected to the bus to gain control.

4.2.2 BUS SYNCHRONISATION

Once control has been granted to a minicomputer or peripheral interface, YAC-Bus is then available for a memory access cycle. This is signified by BUSY being pulled low and the memory address being placed on the memory address lines. To avoid possible side effects and spurious switching, BUSY is used as an enable signal for the address lines. Synchronisation in addition to that supplied by BUSY is needed as there is the problem of differing speeds between peripherals and the minicomputer interface and, depending on the microprocessor, there may be some delay before the memory is available for access.

In addition, there is the need to be able to detect if there was a response to the address on the address lines. This synchronisation is provided by a single control line called SYN (SYNchroniser). Its operation is a modification

of the two-line handshaking protocol used in the PDP-11 by reducing some of the redundant actions of the MSYN and SSYN signals by using a single line. The function of SYN depends on the fact that it is an open-collector line and the signal level is the AND function of all the signals placed on the line.

Calling whatever is in control of the bus the 'master' and whatever is being addressed as the 'slave', the protocol works as follows (see figure 4.4b):

- a) the master places the address on the bus and then pulses SYN low (event 1).
- b) the slave, in response to the address and SYN, pulls SYN low as well. SYN is held low by the slave until it has processed the data (carried out a write or read cycle).
- c) upon receiving high SYN, the master completes the cycle by returning BUSY high, at which time the address and data are removed from the lines (event 2). The master knows immediately after pulsing SYN whether or not a slave has responded to the address as SYN would look exactly like master SYN if there was no response, an effect that can be easily detected. This fact has been used in the design of the minicomputer and peripheral interfaces and in their interaction. The minicomputer, for instance, can determine if a peripheral exists in a system. More importantly, in the communication between peripheral and minicomputer via the interrupt register in the minicomputer interface, a lock exists on the interrupt register to prevent its contents from being overwritten. The lock on the interrupt register prevents the SYN response from being generated by the minicomputer interface, this being detected at the peripheral interface and interrupting the microprocessor to allow it to attempt to interrupt the minicomputer again.

The time required for a bus cycle can be quite fast. SYN is low for what is essentially the access time of the buffer memory. With the components typically available, this access time is of the order of 450ns and can be as short as 250ns, depending on the quality of components used.

CHAPTER 5YAC-BUS IMPLEMENTATION

The purpose of this chapter is to describe an implementation of the minicomputer and peripheral interfaces to YAC-Bus. The interfaces were implemented for a PDP-11 minicomputer and a 6800 microprocessor used as a peripheral controller. The modifications necessary in order to adapt these interfaces for use with other types of minicomputers are also considered.

5.1 THE PDP-11 INTERFACE

The design of the PDP-11 interface had a major constraint - the interface had to fit onto a single board that could be plugged into the backplane of the computer. This limited the number of components that could be used in the implementation and to some extent influenced the design of YAC-Bus as a whole.

A block diagram of the minicomputer interface is shown in figure 5.1. The interface can be considered to consist of four parts:

- i) Unibus buffering and timing providing
 - a) address decoding and buffering
 - b) data buffers and drivers
 - c) timing of the Unibus MSYN and SSYN signals
- ii) YAC-Bus Interface Registers consisting of
 - a) the index register and its YAC-Bus driver
 - b) the interrupt register and its address comparator
 - c) drivers for the data "register"
 - d) a multiplexer to present data to the Unibus data drivers.
- iii) YAC-Bus timing circuits providing timing for YAC-Bus control signals insofar as they affect the minicomputer interface - they also interact with the Unibus timing as well.
- iv) Unibus interrupt - a separate and distinct part of interface which handles the interrupt protocol for Unibus.

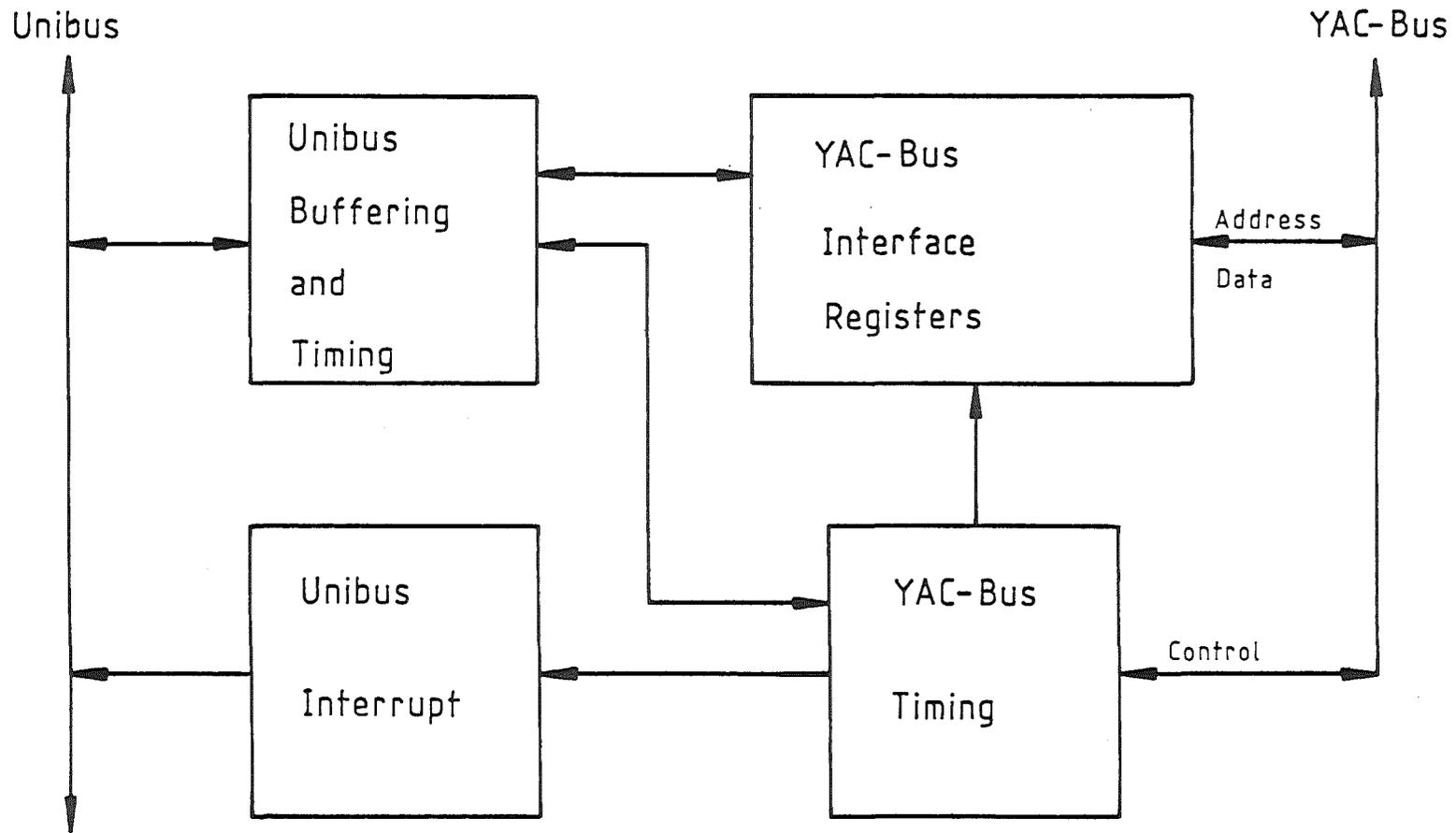


Figure 5.1 Block Diagram - Minicomputer Interface

5.1.1 UNIBUS BUFFERING AND TIMING (Figure 5.3)

The major function of this part of the interface is to provide translation between the logic levels used on the Unibus and the TTL used within the interface and YAC-Bus. The Unibus address lines and the BUS MSYN* signal are decoded to provide a signal, Start, that indicates that the interface is being accessed by the CPU. The signal BUS SSYN is asserted when the data has been processed, the signal $\overline{\text{Wait}}$ being used to delay BUS SSYN whenever the data "register" is accessed (all other data within the interface are stored in fast registers so that only a small fixed delay is needed in such circumstances). The address lines AO1 and AO2 address the individual registers within the interface; the line C1 denoting if a read or write operation is being carried out. Byte addressing is not available within the interface as it required extra circuitry and was considered unnecessary.

5.1.2 YAC-BUS INTERFACE REGISTERS (Figure 5.4)

The format of the three registers within the interface as seen by the PDP-11 CPU is shown in figure 5.2.

The index register consists of four cascaded 4-bit binary counters. The counters are used to provide the auto-incrementing function so that after each access of the data "register", the index register is incremented by one. Loading of the index register is complicated by the need to generate a clock pulse to actually load the register.

The data register does not exist as such in the interface. Instead, the data is gated between YAC-Bus and the Unibus and the CPU is stalled until the data has been processed by YAC-Bus. The reason for this implementation is that it considerably reduces the hardware needed for the data register, a particularly important point.

* To avoid confusion, all Unibus signals are prefixed by BUS and all YAC-Bus signals by YB.

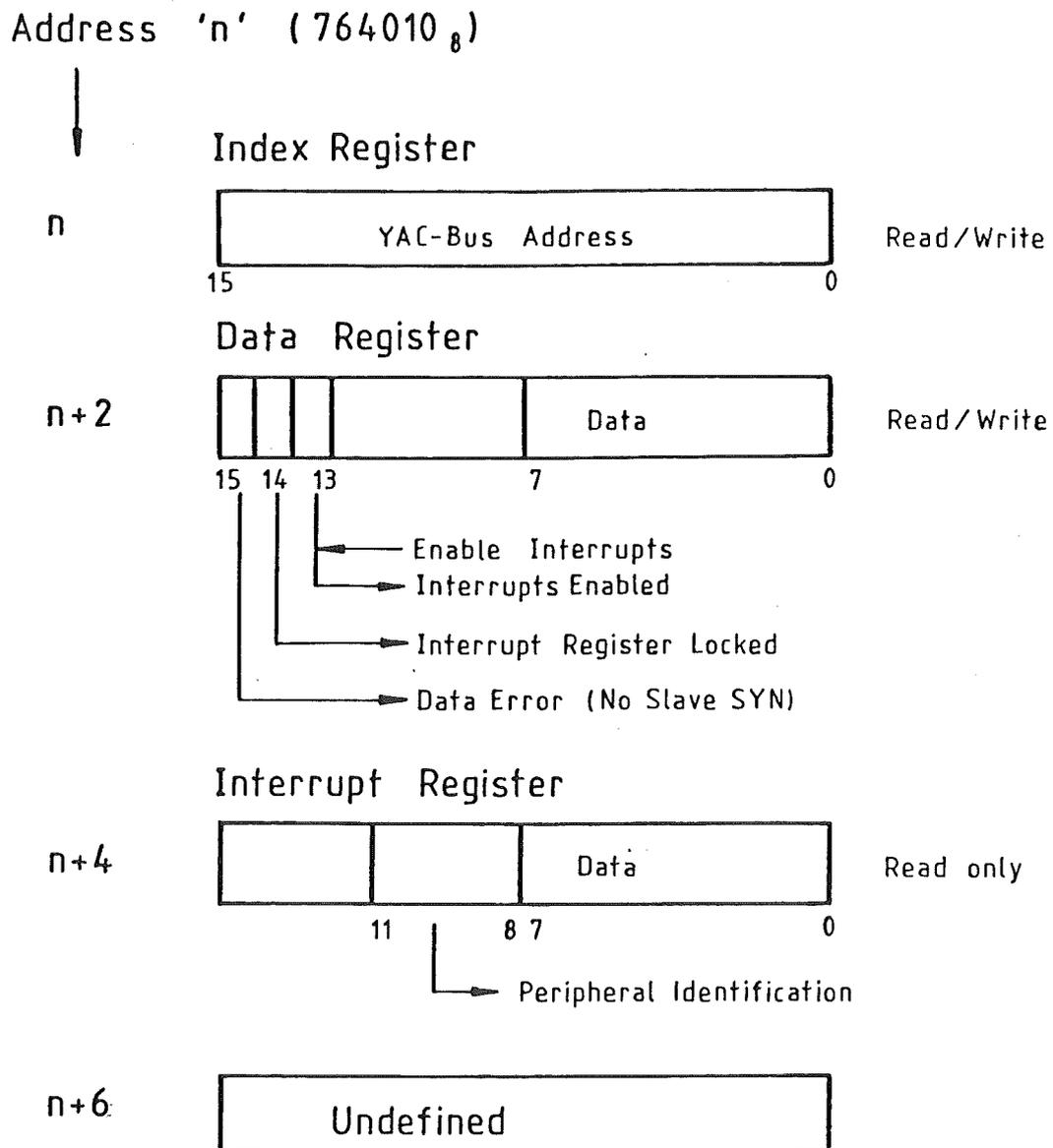


Figure 5.2 PDP-11 Interface Registers

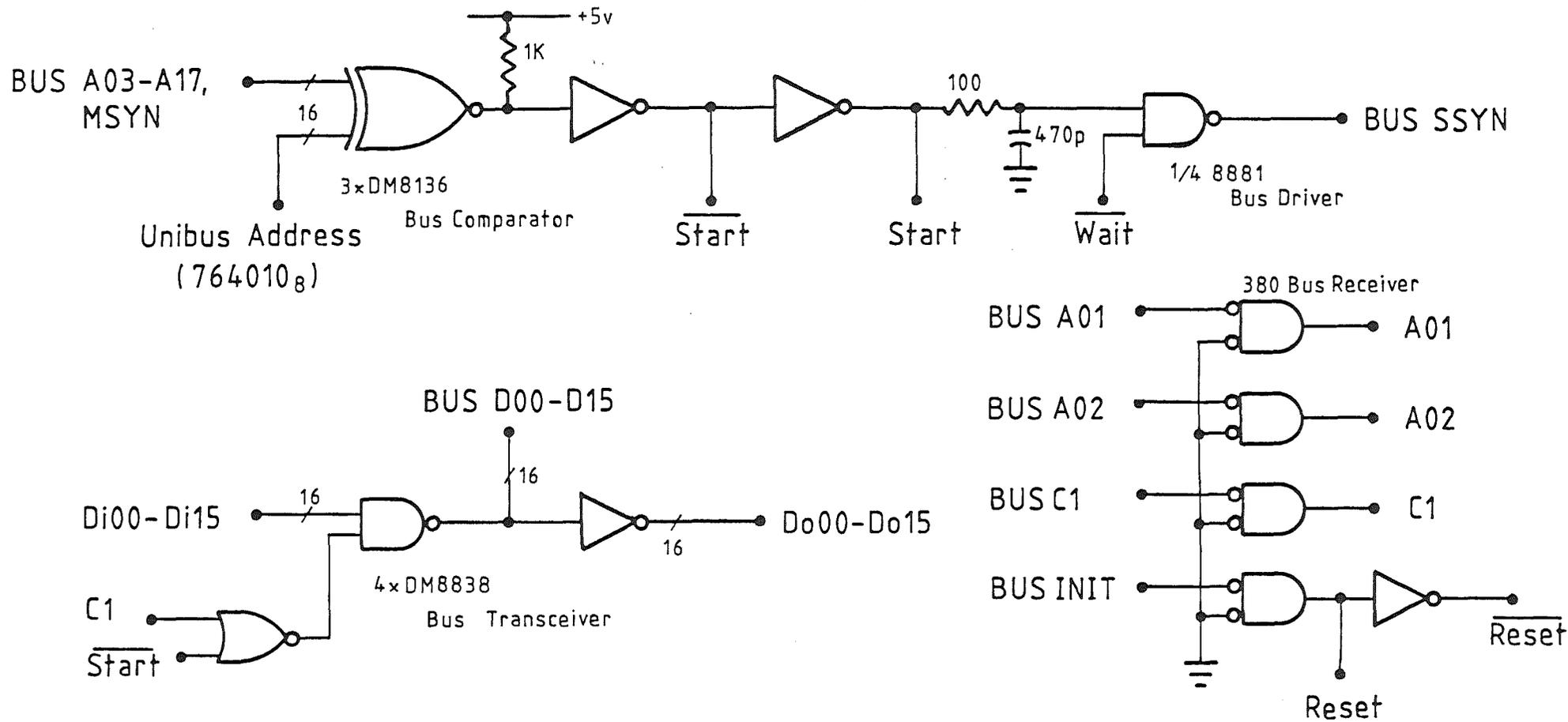


Figure 5.3 Schematic Diagram - Unibus Buffering and Timing

There would appear to be several advantages of this approach over the model presented in chapter 4, the most important of these being that the information we retrieve through the data register is up to date. A more detailed description of the timing associated with the data register is given in the next section.

The interrupt register consists of a 12-bit register. Its contents are loaded from the YAC-Bus data lines and the upper 4 bits of the address lines whenever 0 is detected on the lower 12 bits of the address lines during a write operation and the contents of the interrupt register are not locked. The 1-bit lock register is set when the interrupt register is written to from YAC-Bus and is cleared when the interrupt register is read by the CPU. The contents of the lock register are read whenever the data register is read.

To allow the CPU to read the contents of the three registers, a multiplexor is required. Because of space requirements, a 3-to-1 multiplexor catering for all 16 bits could not be used so a compromise that allowed 1 16-bit and 2 12-bit registers to be read was used. Because of the address decoding used, the interface occupies four words of the PDP-11 address space but only three have any meaning. The function of the fourth register shown in figure 5.2 is undefined and exists as the result of the address decoding used within the interface.

5.1.3 YAC-BUS TIMING (Figures 5.5 and 5.6)

The YAC-Bus timing section of the minicomputer interface deals with the interaction of the Unibus and YAC-Bus timing requirements whenever the data register is accessed by the CPU.

The cycle begins when the CPU addresses the data register. This action triggers a monostable generating the $\overline{\text{wait}}$ signal that delays the BUS SSYN signal (event 1, figure 5.6). The interface then waits until YAC-Bus

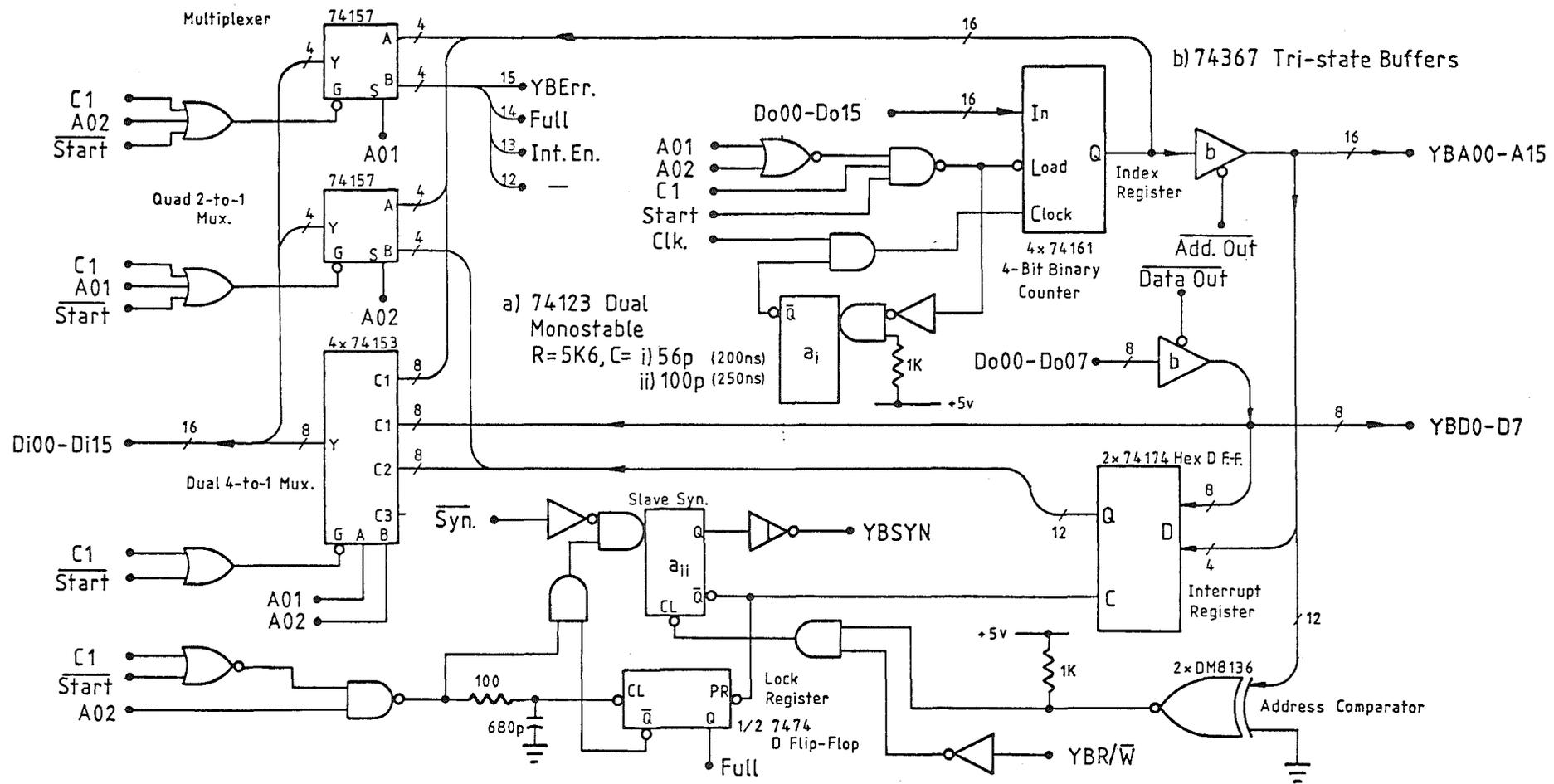
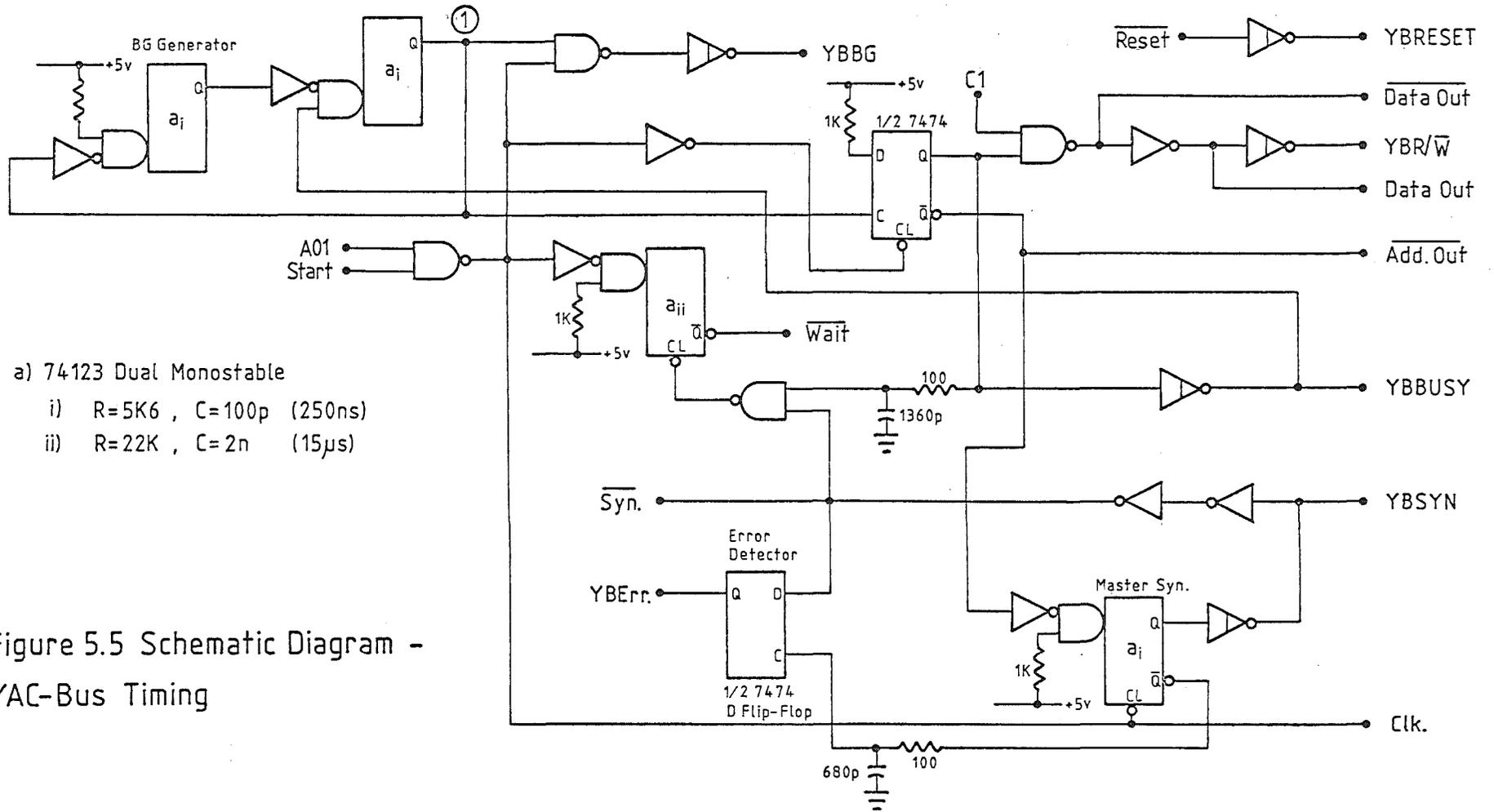


Figure 5.4 Schematic Diagram - YAC-Bus Interface Registers



- a) 74123 Dual Monostable
- i) R=5K6 , C=100p (250ns)
 - ii) R=22K , C=2n (15μs)

Figure 5.5 Schematic Diagram - YAC-Bus Timing

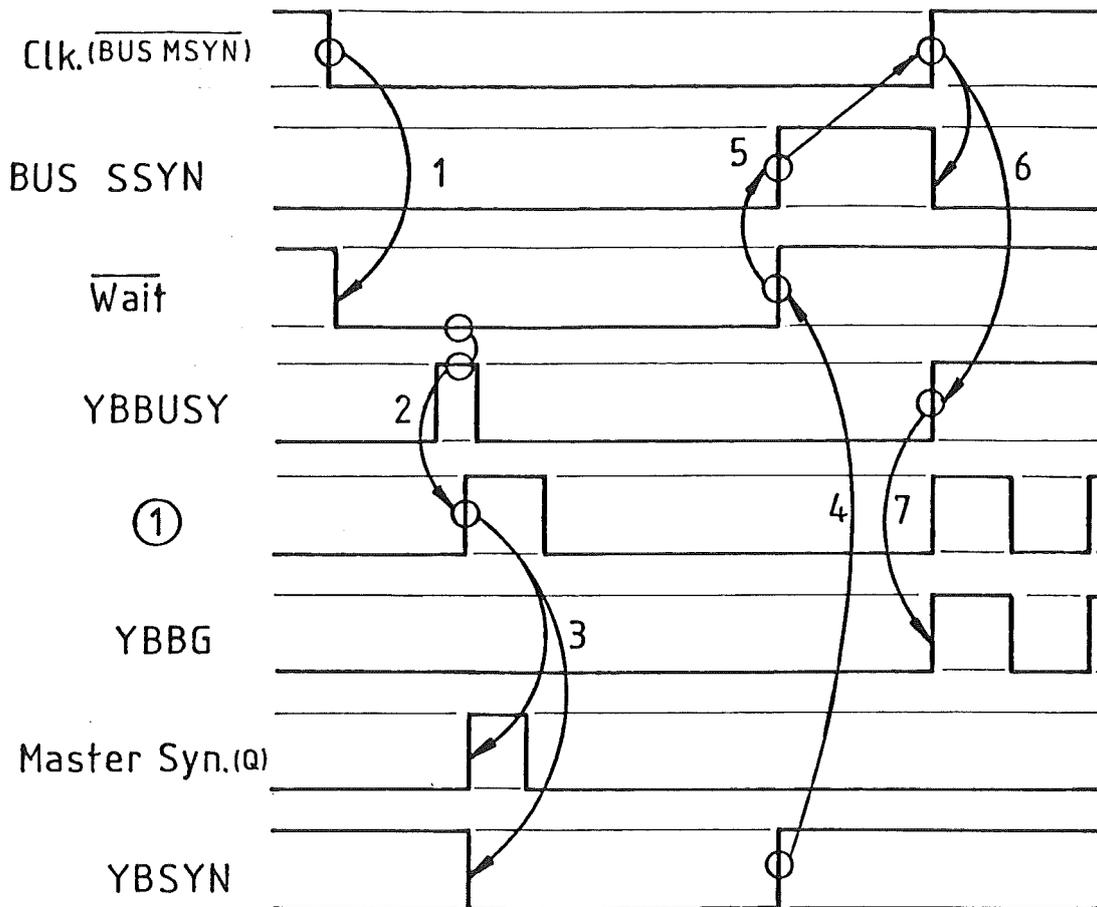


Figure 5.6 Timing Diagram -
Minicomputer Interface

becomes free. This is indicated when YBBUSY returns to a high level and point 1 on figure 5.5 goes high, this transition being blocked from propagating along YBBG (event 2).

The transition starts several actions. The signals $\overline{\text{Add. Out}}$, Data Out and $\text{YBR}/\overline{\text{W}}$ are set to enable the YAC-Bus address, data and $\text{R}/\overline{\text{W}}$ lines and Master Syn. is initiated (event 3). At this point, YAC-Bus and the Unibus are connected together through the data "register". The interface now waits until YBSYN returns high, indicating that the data has been processed and that the interface can now proceed. YBSYN returning high clears the monostable

holding the $\overline{\text{Wait}}$ signal low; BUS SSYN being asserted as the result and the CPU is notified that the data has been processed (event 4). In response, the CPU clears BUS MSYN (event 5) resulting in YBBUSY going high and indicating that YAC-Bus is no longer in use and removing information on the address and data lines of the bus (event 6). YBBUSY going high also initiates the YBBG cycles (event 7).

Detection of the non-appearance of the Slave Syn. signal on YBSYN is carried out by sampling YBSYN on the delayed low-high edge of the Master Syn. signal (refer to figure 4.4b). If there was a Slave Syn. response, YBSYN will still be low and so a 0 will be clocked into the error detector D flip-flop. In the case of no Slave Syn., YBSYN looks like Master Syn. and the delay in the clock line allows the high YBSYN signal to set the flip-flop to 1 on the clock transition. The contents of the error detector flip-flop is read whenever the data register is read.

5.1.4 UNIBUS INTERRUPT (Figure 5.7)

The Unibus Interrupt section of the minicomputer interface is responsible for handling the Unibus interrupt protocol. The operation of the Unibus Interrupt is controlled by the interrupt enable flip-flop. When set, it allows the interrupt request line to initiate the interrupt sequence. The flip-flop is accessible through the interface so that its contents may be read, set or cleared.

The circuitry shown in figure 5.7 is an adaption by the Computer Centre, University of Canterbury, of the PDP-11 M7820 Interrupt Control module used for this purpose. The full schematic is shown as a matter of record but a full description of its operation is considered to be irrelevant in this thesis. A detailed explanation of the Unibus interrupt protocol can be found in the appropriate Digital Equipment manual.

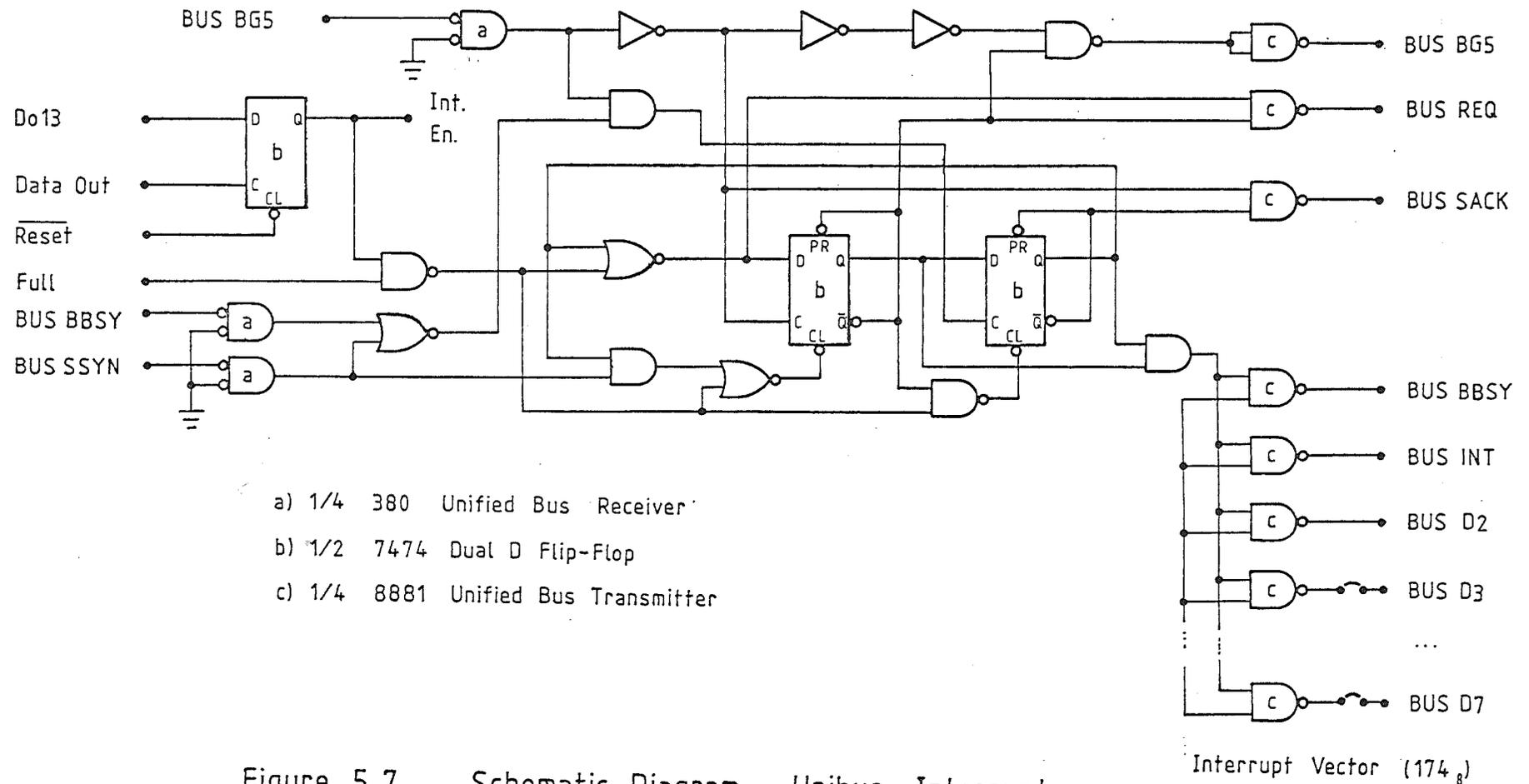


Figure 5.7 Schematic Diagram - Unibus Interrupt

5.2 OTHER MINICOMPUTER INTERFACES

There are several aspects of the PDP-11 - YAC-Bus interface that should be considered in designing an interface for a different minicomputer.

Obviously, circuitry that is machine dependent such as the interrupt logic and logic to handle the signal protocol of the IO system will differ from minicomputer to minicomputer. However, the PDP-11 interface achieved most of its simplification by making use of the fact that the PDP-11 is basically an asynchronous machine. This allowed the data register to be removed, freeing much needed space on the prototyping board.

Other minicomputers, typified by the Data General Nova, are synchronous in nature. That is to say that the timing of the IO system in particular is rigidly controlled by the CPU and cannot be varied as is possible with the PDP-11. In designing a YAC-Bus interface for such minicomputers, there are several points to watch relating to the implementation and operation of the data register. Because the data register contains a copy of the byte addressed by the index register, whenever the index register is changed the interface must access YAC-Bus to update the data register.

Consider the case of a write operation to the data register. This would cause two accesses to YAC-Bus - the first being the actual write to YAC-Bus and the second the read operation bringing the next byte into the data register (this latter operation being caused by the auto-incrementing of the index register). Under some circumstances the read operation is unnecessary (such as when writing an array of bytes to YAC-Bus) and the interface designer should consider giving the programmer more control of the operation of the interface.

For instance, the Nova (or Eclipse) minicomputer has as part of its IO instructions a set of options that can be used to perform special functions within a peripheral (namely the Start, Clear and Pulse controls). These can be used in the operation of the interface to good effect i.e.

a write operation that increments the index register but does not read the next byte. A case could also be made for an operation that does not increment the index register - useful when reading a byte, altering and then returning it to the same location.

Finally, one must consider the timing requirements of the interface. Whenever the index register is altered, it takes some time before the data register reflects this change as a byte has to be retrieved from YAC-Bus memory. If we assume a YAC-Bus cycle time of 500ns (this depending on the speed of the buffer memories within the peripherals), it may take between 1 to 3 cycles (0.5 to 1.5 μ s) to carry out a particular YAC-Bus operation. This time may be a significant number of CPU instruction cycles and so with minicomputers such as the Nova, one cannot alter the address register in one instruction (by either loading it or causing it to auto-increment) and then expect to retrieve valid data from the data register on the next instruction.

Such timing constraints are not serious as it is usually possible to alter the address register several instructions before the data is required. For instance, where an array of bytes is being moved, sufficient delay may be obtained through the need to execute loop control instructions and those instructions used to unpack or pack the bytes in the computer's memory. To be on the safe side, it is suggested that a "valid data" bit be provided to allow the programmer to make absolutely certain that the data register really contains the data addressed by the index register.

5.3 THE 6800 INTERFACE

There are fewer constraints in designing the peripheral interface compared to the minicomputer interface although there was still the need to keep it as simple as possible in order to reduce costs. A block diagram of the peripheral interface is shown in figure 5.8.

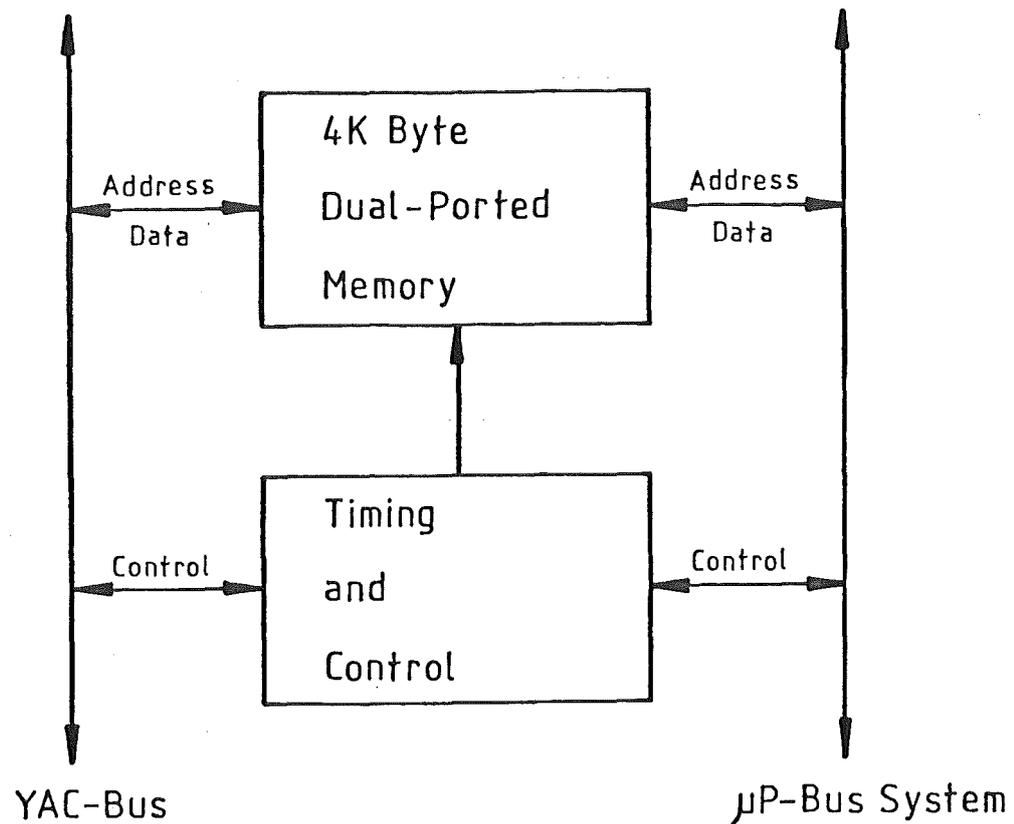


Figure 5.8 Block Diagram - Peripheral Interface

The interface can be considered to consist of two main parts:

- i) a 4k byte dual-ported memory;
- ii) peripheral control and timing logic that provides YAC-Bus control signals insofar as they effect the peripheral interface - the logic also interfaces with the microprocessor timing signals as well.

5.3.1 PERIPHERAL CONTROL AND TIMING (Figures 5.9 and 5.10)

In order to discuss the operation of the peripheral control and timing logic, we shall consider the events that occur when the microprocessor writes to location 1 of the peripheral's buffer memory (i.e. attempts to interrupt the minicomputer).

The cycle begins in the ϕ_1 cycle of the 6800's clock. $\overline{\text{Vma}}$ (valid memory address) goes low enabling the upper 4 bits of the microprocessor's address lines to be compared with the address of the buffer memory set in the B switch register. After gating with additional signals to validate the fact that the microprocessor is actually writing to location 1, the signal first blocks the YBBG signal from propagating to other peripheral interfaces along YAC-Bus and then allows the interface to gain control of YAC-Bus on the next low-high YBBG transition (event 1 on figure 5.10). Upon getting control of YAC-Bus, the interface signals that the bus is busy by pulling YBBUSY low.

The next phase of the operation begins when the ϕ_2 clock goes high. This signifies that the 6800 has valid information on the data lines and triggers the Master Syn. monostable (event 2). When YBSYN returns high signifying that the interrupt register has been accessed (event 3), the microprocessor continues its ϕ_2 cycle until completion. On the high-low transition of ϕ_2 , the interface has finished with YAC-Bus and release YBBUSY (event 4).

Additional logic includes the Syn. error detector, the YAC-Bus address comparator (using the A 4-bit switch register) and the ϕ_2 extend logic. This last logic is needed to stretch the ϕ_2 clock of the microprocessor under various conditions. These conditions arise when

- 1) the minicomputer and the microprocessor attempt to access the buffer memory at the same time. The microprocessor is held off by extending the ϕ_2 clock until the minicomputer has completed its access.

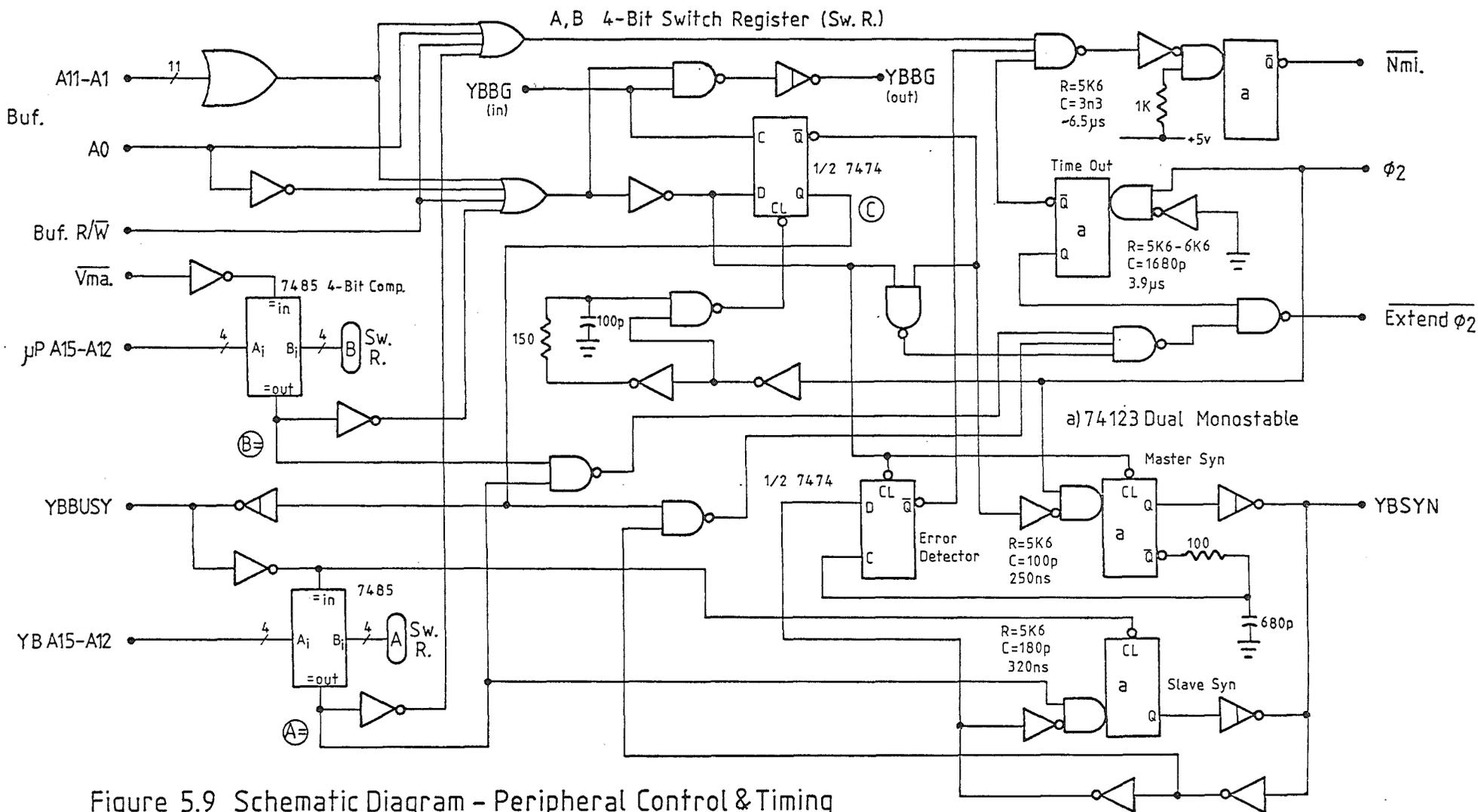


Figure 5.9 Schematic Diagram - Peripheral Control & Timing

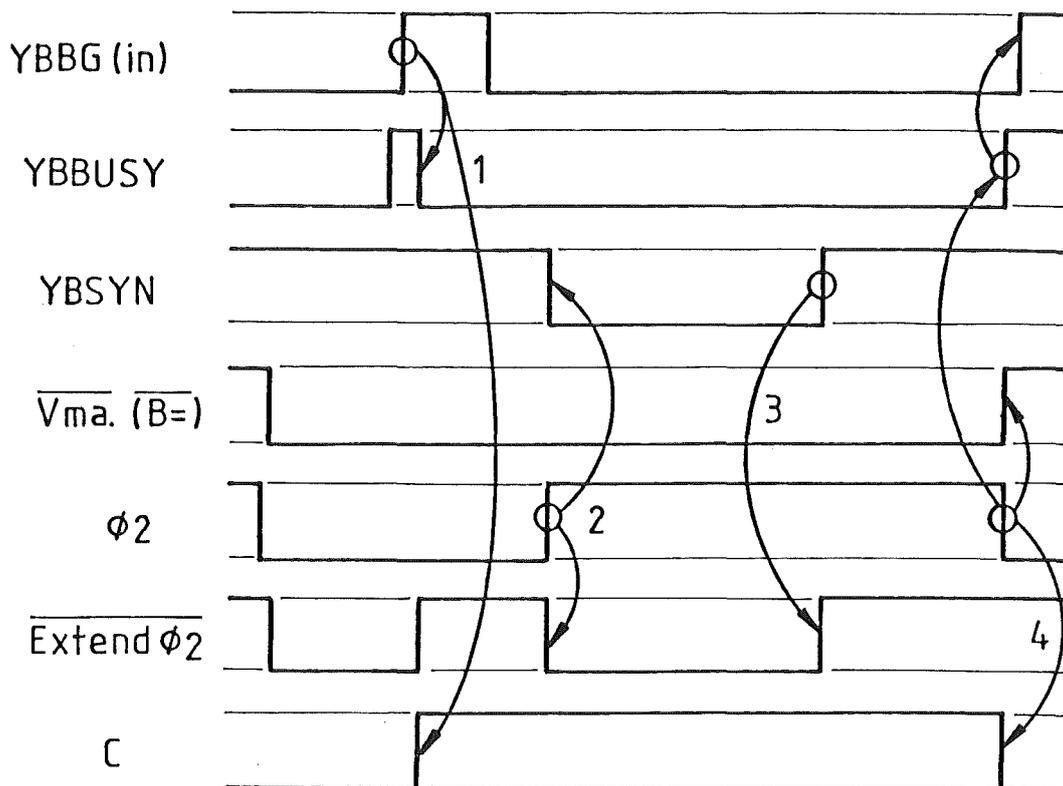


Figure 5.10 Timing Diagram -
Peripheral Interface

2) the microprocessor wishes to use YAC-Bus but has not yet been granted control.

3) the microprocessor is using YAC-Bus and is waiting for YBSYN to return high indicating that the interrupt register has received the data.

Some of these conditions may occur during the ϕ_1 part of the clock cycle; however the $\overline{Extend \phi_2}$ is only operable when ϕ_2 is high. A precaution that is required with the use of the $\overline{Extend \phi_2}$ signal is to limit the length of the ϕ_2 clock cycle. If an attempt is made to extend ϕ_2 for more than 3.9 μ s, the time-out monostable releases $\overline{Extend \phi_2}$ to allow the microprocessor to complete the cycle without the danger of losing the contents of its registers. To indicate to the microprocessor that a time-out occurred and to allow it to take the appropriate recovery action, the time-out monostable triggers the microprocessor's non-maskable interrupt (NMI).

Other conditions under which the NMI is triggered are when a YAC-Bus SYN error occurs and when location 0 of the buffer memory is written to. These conditions are such that it is possible to determine the cause of the NMI and make the appropriate action or response.

5.3.2 DUAL-PORTED MEMORY (Figure 5.11)

The dual-ported memory consists of 4k bytes of memory and associated circuitry to provide access by the microprocessor and from YAC-Bus.

The circuitry makes use of three signals derived in the control and timing logic:

- 1) A= indicating that the buffer memory is being accessed from YAC-Bus;
- 2) B= indicating that the buffer memory is being accessed by the microprocessor;
- 3) C indicating that the microprocessor is accessing YAC-Bus.

5.4 USING OTHER MICROPROCESSORS

Adapting the peripheral interface for use with other types of microprocessors should present few difficulties. Any problems that are likely to occur relate to the use of the microprocessor clock timing signals ϕ_2 and $\overline{\text{Extend } \phi_2}$.

Microprocessors such as the 8080 and the Z80 do not generally provide for the interfacing of slow memory by stretching the clock signals as is typically done with the 6800. Instead, the microprocessor can be forced into a wait state at various parts of the instruction execution cycle (the memory access parts of the cycle). This introduces the complication that a memory access requiring the use of wait states must anticipate such a need to allow the necessary control signals to be set up - invariably this must be done before the access is initiated. Note that this is in contrast with the 6800 which allows either ϕ_1 or ϕ_2 clock signals to be stretched (achieving the same effect as a wait cycle) at any phase of its clock cycle. However,

there is a limit to how long the clock signals on the 6800 can be stretched - the use of the wait state on the Z80 and 8080 allow them to be "stalled" indefinitely.

The implications of the wait state mean that the operation of YAC-Bus has to be slightly modified when the minicomputer is accessing the bus. Simultaneous accesses to the buffer memory by the microprocessor and the minicomputer can be resolved by the simple but slightly more complicated expedient of providing a "first-come, first-served" strategy. Once the memory is being accessed by either the minicomputer or the microprocessor, it completes a memory access cycle without interruption. The result is that a YAC-Bus access by the minicomputer may take longer to carry out.

Suitable equivalents of the other microprocessor dependent signals such as $\overline{Vma.}$, R/\overline{W} , $\overline{Nmi.}$ etc can easily be provided, regardless of the type of the microprocessor.

CHAPTER 6

USING YAC-BUS

In this chapter the software required to support YAC-Bus is considered along with important issues relating to the design of this software. The design of a monitor is studied with regard to some of the issues raised.

6.1 SOFTWARE FOR YAC-BUS

The operation of a peripheral interfaced through YAC-Bus depends to a large extent on software, much more than a conventional peripheral. The software allows the microprocessor to control the peripheral, defines the communication protocol between the CPU and the peripheral, and defines the structure of data accessible via YAC-Bus. The design of this software is considered to be centred on three issues:

- i) the allocation of tasks between the microprocessor and the minicomputer;
- ii) the interfacing of the peripheral to the applications program;
- iii) the functions to be performed by the software.

6.1.1 ALLOCATION OF TASKS

The tasks performed by a computer in controlling a peripheral can be placed into three categories.

- i) Control of the peripheral itself. This function is carried out by the software that issues the commands at the hardware interface to the peripheral. The task involves the correct handling of the sequence of commands, status interrogations etc to ensure the proper operation of the peripheral.
- ii) Control of the peripheral in relation to the computer. There may be several strategies used within the operating system to ensure the efficient use and operation of the computer as a whole. Some of these strategies may directly involve a peripheral. For instance, the performance of disk storage depends to a large extent on how files are stored on disk. Other strategies such as resource management relate

the peripheral to the performance of the computer as a whole and are of less immediate concern to the peripheral itself.

iii) Control of the peripheral in relation to the user. The user sees the peripheral in terms of an abstract model. It is the task of the device driver software to reconcile the differences between this model and the actual peripheral. This may require considerable processing to convert information from a form convenient to an applications program to that convenient to a peripheral.

By using a microprocessor-controlled peripheral, it should be possible for the microprocessor to take over some of these tasks from the CPU. The extent to which this might be possible depends on the relative processing capabilities of the microprocessor and the CPU. However, it must be remembered that the primary role of the microprocessor is the control of the peripheral. Thus the microprocessor's ability to perform any other tasks result from any capacity in excess of that required to control the peripheral.

To allocate the various tasks between the minicomputer and the microprocessor, a "bottom-up" allocation strategy is suggested. Given that the microprocessor has some surplus capacity after the basic control functions of the peripheral have been implemented, one "uses up" this capacity by moving functions from the minicomputer's device driver software to the microprocessor's software. The shifting of functions across to the microprocessor ceases when either the microprocessor has no more surplus capacity or the applications program interfaces directly to the peripheral.

With regard to the type of task best performed by the microprocessor, the minicomputer should be left with those tasks suited to its capabilities and resources. For instance, tasks such as resource control and management involving the peripheral (e.g. the allocation of a peripheral to a process) generally cannot be handled by the microprocessor. Although such tasks effect the operation of the peripherals, it would be impractical to give the microprocessor access to the data upon which decisions are made. As the collection and generation of such data requires knowledge of what the entire computer system is doing, resource management is best left to the minicomputer.

6.1.2 THE USER INTERFACE

Having determined which processor should be responsible for the various tasks associated with the operation of a peripheral, it is now pertinent to discuss how one might provide an interface to the peripheral for an applications program. This interface is usually in terms of some abstract model of the peripheral assumed in the design of the programming language. Different languages assume different models. For instance, Fortran IO operations are based on records while IO operations involving the console in Basic are based on strings of characters. For some applications, these models may prove inadequate and so special purpose IO routines are needed. One might supply routines to the Fortran user to provide console IO operations on a character-by-character basis in real-time (important in interactive programs), or provide Basic with record oriented IO for accessing disk files (required for any serious work).

These observations have several implications for the design of the interface to a peripheral accessed through YAC-Bus. They identify two levels of involvement by users of a peripheral. There is the casual user whose main interest in using a computer is to solve a particular problem with as little inconvenience to himself as possible. Such a user does not want to get involved in the finer details of the operation of the peripheral and so can be satisfied with a more general model of IO operations (such as Fortran IO operations based on records).

The other type of user is usually involved in a more serious manner with the operation of the computer. For such a user, the performance of a peripheral can be important and so close control of the peripheral is required. Such control demands little, if any, abstract modelling of the operation of the peripheral. Usually IO operations are synchronous in that the user's program continues execution only after an IO operation has been completed. In a real-time environment (and other environments where it is important to keep the CPU in use for reasons of efficiency) it is important for IO operations to be asynchronous; i.e. the user's program starts

an IO operation and execution continues while the peripheral carries out the operation. This creates the difficulty of informing the user's program when the IO operation is completed. Usually the completion of an IO operation interrupts the CPU and a typical method of allowing the user to service certain interrupts is by providing "hooks" to allow user-defined interrupt service routines to be patched into the operating system.

These two levels of involvement in the operation have to be reflected in the design of the interface to the user and the software controlling the peripheral. Because different types of user require varying levels of accessibility to the operations of a peripheral, we can either provide software that will cater for most types of user or load specially tailored software along with the user's program. This latter idea is made easier by the fact that programs for the microprocessor can be very easily down-line loaded using YAC-Bus. Thus if a user is not happy with the software facilities provided, he can write his own.....

Presenting the user with an abstract model of a peripheral has its difficulties. For instance, does one base the model on the existing IO facilities provided in a language or attempt to provide a completely different model? Basing the model on existing facilities would certainly make the peripheral easier to use for existing users of the language in spite of the fact that the model may be lacking facilities to give the user the desired control of the peripheral.

However, if the existing facilities are rejected in favour of implementing a different model, there are problems associated with the implementation for most of the common computer languages. As an example, Fortran's IO statements (READ, WRITE etc) are part of the language's definition; if these statements are to be changed or added to, the definition of the language is changed and requires alterations of the Fortran compiler to carry out the implementation (an exercise fraught with difficulties).

Consequently, it is usual to find that special IO operations are implemented as subroutines and libraries of such subroutines are provided as part of the library of a computer system.

6.1.3 USER FACILITIES

What software facilities does the user need to allow him to use YAC-Bus? The facilities are centered on two basic needs:

- i) the need to handle interrupts from YAC-Bus
- ii) the need to get access to the data handled by the peripheral. Although it is usual for the operating system to handle interrupts, the need often arises to give the user access to certain interrupts (examples being the actions to be carried out upon power failure and the servicing of interrupts from a "non-standard" peripheral). Consequently, it is usual to find routines within the operating system of a minicomputer to provide this access although in general direct access by the user to the IO system (by executing IO instructions etc) is prevented by the protection system that may be available on the computer; this access being the sole prerogative of the operating system. Thus, to provide the user with access to YAC-Bus requires that we endow the operating system with facilities to handle YAC-Bus.

This support for the user can be at many levels; one may provide the bare minimal facilities and expect the user to take care of everything else, including the handling of interrupts, or provide the user with an abstract model with facilities that interface with the user's favourite high level language (HLL). It is the bare minimal facilities that are needed that are of concern here.

In the particular implementation described in this thesis, the situation was very simple. The particular PDP-11 minicomputer used for experimentation did not have a protection system. Therefore, it was very easy to set up an interrupt handler and to gain access to the registers of the YAC-Bus interface.

As examples of the type of facilities that one might provide the user, consider figure 6.1. The subroutines listed in this figure illustrate some of the more obvious problems in providing the user with access to YAC-Bus.

There is on occasion the need to go beyond the capability of the user's HLL (illustrated here by Fortran only for convenience), especially when handling peripherals. Listing a) is of a PDP-11 assembler language subroutine, SETINT, callable from Fortran that sets up and calls a Fortran subroutine to service interrupts.

The reason for wanting to use a Fortran subroutine (or any HLL subroutine) to service interrupts is that it provides a convenient means of accessing a user's data structures during an interrupt. Fortran is perhaps a poor example in this respect since its language data types are not overly complicated. However, with languages such as Pascal and Algol 68 where the user can define his own data types upon which to build data structures, the convenience of an interrupt service routine written in the same language as the rest of the program is more apparent. The difficulty with a lot of HLLs is illustrated in the latter part of the subroutine in the actual code executed upon an interrupt. Because the HLL user is shielded from the details of the computer's architecture, the actions needed to be carried out upon an interrupt (such as saving the state of the computer) are impossible in a language that is not designed to handle such events.

The subroutine SETINT was used in work with YAC-Bus instead of the subroutine INTSET supplied with the RT-11 operating system under which the PDP-11 operated. SETINT was needed so as to run the interrupt service routine at a priority level that prevented further interrupts from the YAC-Bus interface, a feature that INTSET did not provide. With different operating systems and different computers, the principle behind the design and operation of SETINT might not be usable or appropriate.

Listings b) and c) of figure 6.1 are of Fortran subroutines to provide access via YAC-Bus to data handled by the peripheral. There are two types of subroutines provided that reflect how YAC-Bus is likely to be accessed. The first type provides single byte read and write functions, suitable for

```

1          .TITLE  INTERRUPT SERVICE DIRECTOR
2          .GLOBL SETINT
3          .MCALL  ..V2...REODEF
4 000000   .V2..
5 000000   .REGDEF
6
7          ; THIS SUBROUTINE SETS UP A FORTRAN SUBROUTINE TO ACT
8          ; AS AN INTERRUPT SERVICE ROUTINE, RUNNING AT A
9          ; STATED PRIORITY.
10         ; PARAMETERS ARE AS FOLLOWS:
11         ; FIRST ARGUMENT IS THE ADDRESS OF THE INTERRUPT VECTOR
12         ; SECOND ARGUMENT IS THE PRIORITY LEVEL OF THE INTERRUPT
13         ; THIRD ARGUMENT IS THE ADDRESS OF THE SERVICE SUBROUTINE
13 000000   005725   SETINT: TST      (R5)+      ; SKIP OVER NUMBER OF PARAMETERS
14 000002   010046   MOV      RO,-(SP)      ; SAVE
15 000004   010146   MOV      R1,-(SP)      ; WORKING
16 000006   010246   MOV      R2,-(SP)      ; REGISTER
17 000010   013500   MOV      @(R5)+,R0     ; RETRIEVE FIRST PARAMETER
18 000012   012720   000054'  MOV      #SERINT,(R0)+ ; STORE ADDRESS OF SERVICE ROUTINE
19                                     ; AT LOCATION
20 000016   013501   MOV      @(R5)+,R1     ; RETRIEVE SECOND PARAMETER
21 000020   042701   177770   BIC      #177770,R1    ; MASK OUT INSIGNIFICANT BITS
22 000024   012702   000005   MOV      #5,R2        ; SET UP LOOP COUNTER
23 000030   006301   LOOP:   ASL      R1      ; SHIFT LEFT UP TO PRIORITY POSITION
24 000032   005302   DEC      R2
25 000034   003375   BGT     LOOP
26 000036   010110   MOV      R1,(R0)      ; STORE INTERRUPT PS WORD
27 000040   011567   000046   MOV      (R5),RLRTN   ; RETRIEVE AND STORE THIRD PARAMETER
28 000044   012602   MOV      (SP)+,R2     ; RESTORE
29 000046   012601   MOV      (SP)+,R1     ; WORKING
30 000050   012600   MOV      (SP)+,R0     ; REGISTER
31 000052   000207   RTS      PC
32
33         ; THE INTERRUPTS WILL JUMP TO THE FOLLOWING LOCATION
34         ; WHICH THEN CALLS THE SUBROUTINE
34 000054   010046   SERINT: MOV     RO,-(SP)
35 000056   010146   MOV     R1,-(SP)
36 000060   010246   MOV     R2,-(SP)
37 000062   010346   MOV     R3,-(SP)
38 000064   010446   MOV     R4,-(SP)
39 000066   010546   MOV     R5,-(SP)
40 000070   004777   000016   JSR     PC,@RLRTN
41 000074   012605   MOV     (SP)+,R5
42 000076   012604   MOV     (SP)+,R4
43 000100   012603   MOV     (SP)+,R3
44 000102   012602   MOV     (SP)+,R2
45 000104   012601   MOV     (SP)+,R1
46 000106   012600   MOV     (SP)+,R0
47 000110   000002   RTI
48                                     .EVEN
49 000112   000000   RLRTN: .WORD
50 000000'  000000'   .END   SETINT

```

a) Interrupt access

0001	INTEGER FUNCTION YREAD*2(WHERE)	0001	SUBROUTINE YWRITE(WHAT,WHERE)
C		C	
C	CORE SINGLE BYTE YAC-BUS READ	C	CORE SINGLE BYTE YAC-BUS WRITE
C		C	
0002	IMPLICIT INTEGER*2(A-Z)	0002	IMPLICIT INTEGER*2(A-Z)
0003	LOGICAL*1 ERF	0003	EXTERNAL IPOKE
0004	COMMON /IO/ERF	0004	CALL IPOKE("164010,WHERE)
0005	EXTERNAL IPEEK, IPOKE	0005	CALL IPOKE("164012,WHAT,DR. "20000)
0006	CALL IPOKE("164010,WHERE)	0006	RETURN
0007	I=IPEEK("164012)	0007	END
0008	ERF=I.LT.0		
0009	YREAD=I.AND."377		
0010	RETURN		
0011	END		

b) Single byte access

0001	SUBROUTINE YBREAD(WHAT,NUM,WHERE)	0001	SUBROUTINE YBRITE(WHAT,NUM,WHERE)
C		C	
C	CORE ARRAY OF BYTES YAC-BUS READ	C	CORE ARRAY OF BYTES YAC-BUS WRITE
C		C	
0002	IMPLICIT INTEGER*2(A-Z)	0002	IMPLICIT INTEGER*2(A-Z)
0003	LOGICAL*1 WHAT(NUM),ERF	0003	LOGICAL*1 WHAT(NUM)
0004	COMMON /IO/ERF	0004	EXTERNAL IPOKE
0005	EXTERNAL IPOKE, IPEEK	0005	CALL IPOKE("164010,WHERE)
0006	ERF=.FALSE.	0006	DO 10 I=1,NUM
0007	CALL IPOKE("164010,WHERE)	0007 10	CALL IPOKE("164012,WHAT(I),DR. "20000)
0008	DO 10 J=1,NUM	0008	RETURN
0009	I=IPEEK("164012)	0009	END
0010	ERF=(I.LT.0).OR.ERF		
0011 10	WHAT(J)=I.AND."377		
0012	RETURN		
0013	END		

c) Array of bytes access

Figure 6.1 YAC-Bus Access

sending commands to the microprocessor. The second type provides multi-byte read and write functions and are different from the single byte functions in that the auto-incrementing function of the index register can be used to advantage.

The four subroutines are shown as written in Fortran. This was done for illustration purposes to show how it is possible in some HLLs and computers to get access to parts of the basic structure of the computer, in this case the IO system. The subroutines IPEEK and IPOKE provide access to any part of the PDP-11's address space and are used in this case to access the registers in the YAC-Bus interface.

IPEEK and IPOKE are convenient for providing access to memory-mapped IO systems. For other types of computers (e.g. the Nova) where special IO instructions are used, access is generally not quite so convenient and it would be more useful to provide subroutines that give access to the individual parts of the minicomputer interface (the index, data or interrupt registers).

Finally, there is an important point to note when using YAC-Bus. The operation of YAC-Bus does not allow for indivisible functions - i.e. a memory cycle in which data is read, altered and then written back to the same memory location and which cannot be interrupted or interfered with in any way. Such functions are useful for implementing semaphores that are used for the communication between and synchronization of processes and processors, such as might be found in a typical system using YAC-Bus. Without the availability of an indivisible function on YAC-Bus, care must be taken, for instance, when allocating resources (such as buffers) between the CPU and the microprocessor to avoid difficulties like deadlocks and corruption of data areas.

6.2 YAC-BUS EFFICIENCY

Although the PDP-11 Fortran compiler used to compile the subroutines in figure 6.1 produces reasonably efficient code, in actual use the subroutines would be coded in assembler routines shown in figure 6.2, it is possible to get some idea of the efficiency of accessing data via YAC-Bus.

```

; INDEX REGISTER (164010) ASSUMED TO BE INITIALISED
; R1 - DESTINATION POINTER
; R2 - BLOCK SIZE
      ⋮
      MOV      #164012,R3
LOOP:  MOV      (R3),RO
      MOVB    RO,(R1)+
      DEC     R2
      BGT     LOOP      ; LOOP IS 4 WORDS
      ⋮

```

a) YAC-Bus Block Read

```

; RO - SOURCE POINTER
; R1 - DESTINATION POINTER
; R2 - BLOCK SIZE
      ⋮
LOOP:  MOVB    (RO)+,(R1)+
      DEC     R2
      BGT     LOOP      ; LOOP IS 3 WORDS
      ⋮

```

b) Block Move

Figure 6.2 Accessing YAC-Bus with the PDP-11 - a comparison

The code sequence a) is essentially equivalent to the DO loop of the subroutine YBREAD in figure 6.1 without the test for invalid data. Sequence b) is a block move routine for comparison. There appears to be very little difference between the two routines but a little caution is needed in accessing this example. Corresponding code for the YWRITE subroutine requires an extra instruction within the loop to set the interrupt enable bit (bit 13 of the data register).

For large amounts of data, the code in figure 6.2 B) can be changed to move a word (2 bytes) at a time which increases its efficiency quite considerably (the PDP-11 requires extra time to process byte data in spite of having a byte addressable memory). However, for operations on bytes of data (such as strings of characters) there is little difference in accessing data via YAC-Bus or in the PDP-11's own memory.

For other types of minicomputer, the efficiency in using YAC-Bus may vary depending on the capability of the minicomputer. For instance, the Nova minicomputer has no instructions for directly accessing a byte in its memory. The Eclipse (the enhanced version of the Nova) has this capability but in order to maintain hardware compatibility the execution times of the IO instructions are relatively long in comparison with its more commonly used instructions. Such factors make it difficult to assess how efficiently YAC-Bus can be used in general.

6.3 A MONITOR FOR YAC-BUS

This example illustrates some techniques in using YAC-Bus from the programmer's point of view. It consists of two programs, one running on a PDP-11 under RT-11 and the other on a 6800 development system, that provide most of the facilities needed in a monitor system that handles the operation of YAC-Bus for a user's programs running on both processors. The examples are merely illustrative, in a practical system various parts of the software would need to be redesigned to fit in with the conventions of the particular operating system

running the minicomputer.

6.3.1 OVERVIEW

The main objective of the monitor was to provide debugging and diagnostic facilities to aid in the use and operation of YAC-Bus. The diagnostic aids were to catch common programming blunders (e.g. trying to use non-existent memory and peripherals) and to provide trace information about the events that occur during the operation of YAC-Bus (e.g. interrupts). Such diagnostic aids are suitable for programs running on the minicomputer where there are usually facilities for interfacing with the user during the running of his program. Finding out what the microprocessor is doing is more difficult as it will be built into a peripheral and obtaining diagnostic information is much more difficult. Thus the debugging facilities need to provide the user with access to the basic operations of YAC-Bus and the control of the microprocessor. A monitor program for the microprocessor is needed that is in continuous control so that if any error occurs, the minicomputer can force the microprocessor into a known state and allow remedial action to be taken. For obvious reasons this monitor program should be stored in read-only-memory (ROM).

6.3.2 THE 6800 MONITOR (see appendix A for listing)

The function of the monitor program for the 6800 microprocessor is to:

- i) provide the minicomputer with complete control over the operation of the microprocessor
- ii) assist the minicomputer during debugging operations
- iii) handle the details of the YAC-Bus communication protocol.

The monitor also provides for the user who is programming the microprocessor by allowing him to "plug into" various parts of the monitor for the purpose of executing his program as well as providing useful routines for handling YAC-Bus.

The operation of the monitor can be seen by following the flow control upon the receipt of a command from the minicomputer. The monitor is assumed to have been entered

at MAIN (line 101 of the listing) at power-on or after a system reset. Having initialised the monitor jump table and various flags and variables, the microprocessor waits for an interrupt (line 119).

The type of interrupt expected at this stage is the non-maskable interrupt (NMI) caused by the minicomputer writing to the command register of the peripheral's interface to YAC-Bus. The NMI has the advantage of allowing control to be able to be returned to the monitor regardless of any programming disaster that might befall the microprocessor in the course of executing a program. When the interrupt occurs, control transfers to NMI (line 31), the NMI service routine.

Because there are two distinct causes of NMI, some care is needed within the service routine to distinguish between them. The scheme used here is that the minicomputer writes a value other than zero to the command register. It is assumed that if the value of the command register is zero then the NMI was the result of a bus error caused by the microprocessor being locked out of the interrupt register of the minicomputer interface in the course of the microprocessor's attempt to interrupt the minicomputer (the other cause of NMI).

The communications protocol assumed by the microprocessor requires that the minicomputer waits for an interrupt in reply to the command before sending it another command. Such a protocol prevents the NMI service routine itself from being interrupted by another NMI: as a precaution against such an event all commands other than a halt request (command #1) will be ignored while the microprocessor is still executing a command.

Having received a command, the microprocessor now decodes and acts upon it. The sequence of code from NMIINT (line 123) onwards carries this out by the use of a table of addresses (jump table) pointing to the routine associated with each command. The command is checked for validity and is used as an index into the jump table to direct the execution of the microprocessor to the appropriate routine (line 138 is where the microprocessor jumps off to the appropriate routine).

At present there are 6 commands implemented:

Command	Action
1	Halt the microprocessor from its current task and return to the monitor. As the microprocessor's registers (except the stack pointer) have been pushed onto the stack as the result of the interrupt caused by this command and the value of the stack pointer is known (the location SAVSTK), the microprocessor's registers are accessible for inspection and alteration.
2	Return the amount of buffer memory implemented (in multiples of 128 bytes) in location 2 of the buffer memory. Although the maximum size of the memory is 4k bytes, it is possible that only a fraction of this is actually implemented.
3	<div style="display: flex; align-items: center;"> } Read </div> <div style="display: flex; align-items: center;"> } Write </div> a block of the microprocessor's memory.
4	
	These two commands give the minicomputer access to the whole of the microprocessor's memory. The address of the block is at locations 2 and 3 and the number of bytes is at location 4 of the buffer memory. The block of memory is moved to location 5 onwards of the buffer memory in the case of the read command; in the case of the write command the block of memory is moved from location 5 onwards.
5	Start a user's program. The address of the program is locations 2 and 3 of the buffer memory and the stack is re-initialised.
6	Continue execution (used after command #1 or a software interrupt instruction (SWI)) after having popped the registers off the stack.

The possible replies to a command (where expected) are:

Value	Meaning
1	Invalid command (attempting to index outside of the jump table).

- 2 The command was executed without error.
- 3 Execution of the command was attempted but an error occurred (an example would be when attempting to execute command #4 (a block write) to ROM).
- 4 Halt acknowledge (reply to command #1 or the SWI instruction).

Finally, the means by which a user can link his program into the monitor are considered. The microprocessor, when executing the monitor program, is assumed to be in one of three states. There is the interrupt state where the microprocessor is servicing an interrupt (such as NMI), the idle state where the microprocessor is waiting for something to happen (mainly an interrupt), and finally the command state where the microprocessor is executing a command. Of particular interest to the user are the last two states, the idle state and the command state.

At present the idle state is implemented as simply a wait for an interrupt and the user is assumed to use the 6800's IRQ interrupt. After an interrupt, the microprocessor either finds itself with a command to execute (NMI) or there might be something to do for the user as a result of an IRQ. Having the microprocessor wait and do nothing until an interrupt may not be convenient for the user. It may well be that the microprocessor should be put to work performing some type of polling process. Unfortunately, what the user would want the microprocessor to do is very application dependent and would suggest that the idle state be removed from the control of the monitor. Care must be taken since part of the processing done by the microprocessor in the idle state determines if it should enter the command state (i.e. do we have a command to execute?) and this process should not be interfered with.

The monitor sets up its jump table and table parameters for entering the command state in RAM by copying them from the ROM storing the monitor (lines 18-23 and lines 107-112). The table being set up in RAM makes it convenient for the user to add to or redefine the commands obeyed by the microprocessor; all that is needed is to add the address of the routine to execute the command to the appropriate place in

the table and adjust the table size accordingly. Because of the vulnerability of the jump table, it would be pertinent if a "software reset" command was added to the microprocessor's repertoire to re-initialise the table on command.

6.3.3 THE PDP-11 MONITOR (see appendix B for listings)

Having discussed the operation of and requirements for the 6800's monitor program running on the PDP-11 minicomputer and the facilities it provides to the user. The function of the PDP-11 monitor is twofold. Firstly it provides the user with access to YAC-Bus. Secondly it provides debugging facilities for the microprocessors and the software interface through which the user accesses YAC-Bus.

The monitor consists of a main program and a set of subroutines. The main program provides some of the interactive debugging facilities and the user is assumed to interface his applications program to the monitor through a subroutine called USER.

The user accesses the data within each microprocessor's buffer memory through 4 subroutines. YBRD1 and YBWR1 provide single byte access while YBRDA and YBWRA provide access for arrays of bytes (see page B-8 of the listings). Validity checks ensure that non-existent peripherals and memory are not being accessed; the information used in the checks come from the microprocessors themselves as a result of the monitor's initialisation (lines 20-40, page B-2).

The subroutine YBCOM (page B-7) is provided for the user to send commands to the microprocessors. Again, validity checks ensure that non-existent peripherals are not being used as well as ensuring that the microprocessor is still not executing a command (as far as the user's program is concerned). The status information held by the monitor about each peripheral is

- i) the size of memory accessible from YAC-Bus - this is held in the array STATUS and also indicates if a peripheral exists on the system (zero if non-existent);

- ii) what command the microprocessor is currently executing - this is held in the array CURCMD and is zero if the microprocessor is doing nothing;
- iii) the reply received from the microprocessor in response to the command (REPLY);
- iv) any errors detected by the interrupt service routine YBINT when the reply was received (ERROR).

The arrays STATUS, CURCMD, REPLY and ERROR make up the COMMON area YBSTS (line 8, page B-2).

The interrupt service routine YBINT (page B-7) is set up by the subroutine SETINT shown in figure 6.1 to service the interrupts from YAC-Bus (line 8, page B-2). YBINT does not do very much in servicing interrupts - it merely extracts the microprocessor identification and the reply from the interrupt register and places the microprocessor's identification in a queue (SERQ) for later action by the user's program. Some error checking is performed, especially for the case where the interrupt is unexpected. Because of the relative interrupt priorities of YAC-Bus and the console device, debugging statements (such as extra WRITES) cannot be inserted within YBINT without upsetting its operation and so YBINT must record information for later examination and perusal. Specifically, the information recorded is the contents of the interrupt register for the last 5 interrupts (the array IQ is used for this function).

The user deals with the interrupts outside of YBINT by extracting information from SERQ (pointer QS) about the interrupting microprocessor. When the interrupt has been dealt with, the user calls NEXT (page B-6) to adjust QS and the status information in YBSTS. The function QLEN (page B-7) provides the user with the number of interrupts still to be serviced in SERQ.

The debugging facilities provided within the monitor operate at two levels. Errors caused by the user interface subroutines (YBCOM, YBRD1 etc) invoke the subroutine ERROR (page B-5). This subroutine provides the user with access to the various monitor data structures as well as providing some elementary facilities for accessing YAC-Bus and its interrupt register. The facilities include the ability to

dump the interrupt service queue (SERQ) and the status tables (YBSTS), and doing an interrupt trace (IQ). ERROR will allow the user to continue in spite of the error or allow the execution of the monitor to be aborted.

The main program provides the user with debugging access to the microprocessors and works in conjunction with them to implement some of the debugging commands. The same elementary facilities for accessing YAC-Bus and its interrupt register as provided in subroutine ERROR are also available. The facilities available for debugging the operation of the microprocessors are:

- i) display any part of the microprocessor's memory;
- ii) alter any part of the microprocessor's memory (where possible;
- iii) start a program executing;
- iv) tell the microprocessor to continue execution after a breakpoint (SWI) or a halt request.

The main program can also invoke ERROR and, of course, the user's program. Because the PDP-11 Fortran compiler does not provide facilities for hexadecimal input/output formats and all work with microprocessors is done in hexadecimal, a set of hexadecimal conversion routines are provided as part of the monitor.

The monitor program listed in appendix B has many limitations. The user interface is not particularly convenient as it would be more useful to provide the monitor as a set of subroutines to be called by a main program written by the user.

There is no provision for down-line loading of programs to the microprocessors - there would be little difficulty in implementing such a facility but the exact method would depend on the cross-assemblers and other tools that are available.

Finally, the interaction between YAC-Bus and the user program requires further attention. At present a somewhat elementary approach is assumed in that interrupts from the microprocessor generate entries in the service queue SERQ for servicing by the user's program at a later and more convenient time and in the order in which the interrupts

occur. Devoting YAC-Bus to a single user at a time may not be convenient and where the minicomputer's operating system permits multi-user operation (i.e. several users sharing the minicomputer simultaneously), a more general approach would be needed to allow YAC-Bus to be shared amongst several users. This would require the operating system to become more involved in the operation of YAC-Bus.

CHAPTER 7

COMMENTS AND CONCLUSIONS

In this chapter, conclusions and suggestions for future work are presented.

7.1 CONCLUSIONS

Chapter 3 described a series of design aims for an interface to a micro-processor-controlled peripheral. Also mentioned in chapter 3 was a limitation under which the interface had to be designed - namely a limit on the number of integrated circuits to be used in its construction. It is now pertinent to assess the effect that this limitation had on the design of the interface and to assess how well the design aims were achieved.

The effect of the component restriction upon the design of YAC-Bus is difficult to assess as it was designed with the limitation very much in mind. The design was perhaps too ambitious: the PDP-11 interface is somewhat contrived as liberties were taken with the Unibus timing to achieve a working interface. It is considered that the programmer does not have sufficient control over the operation of the index register and a design error in the layout of the interface registers made the interface more difficult and less efficient to use than it need have been. The component restriction meant that alterations and additions to correct errors and improve the interface were difficult, if not impossible to carry out.

The component restriction is particularly sensitive to developments in the design of integrated circuits. For instance, most of the Unibus interrupt logic shown in figure 5.7 has now been placed in a single integrated circuit. Another factor influencing the component restriction was the layout of the prototyping board (the board was a standard Digital Equipment board). Had the

layout been somewhat different (as is the case with later designs of the board), considerably more integrated circuits could have been used in the minicomputer interface. Given that the component restriction influenced the design of YAC-Bus and that this restriction could have been altered by several seemingly minor details (such as the two examples given above), what would be the effect of lifting the restriction?

With the present implementation of YAC-Bus, shortcomings of the minicomputer interface could be rectified. For instance, the interrupt enable bit (bit 13) of the data register in the minicomputer interface has to be set whenever the data register is written to otherwise the microprocessors could not interrupt the PDP-11. At present the minicomputer interface loads the YAC-Bus data lines unnecessarily and it is unlikely that the present open-collector bus transmitters for the 5 control lines would be able to handle the load if 16 peripherals were to be interfaced through YAC-Bus.

During the early design stages of YAC-Bus, the use of DMA was considered. There are particular advantages in using it (e.g. data transfers without the direct intervention of the CPU) and in the absence of component restrictions it is possible that YAC-Bus would have centred around some type of DMA facility.

The decision made in section 3.4.2 to design an interface that attempted to do without DMA for the transfer of data remains to be justified. The essential difference between the facilities provided by YAC-Bus and DMA lies in how the data handled by a peripheral is accessed by the computer. DMA provides a peripheral with access to the computer's own memory. YAC-Bus allows the data handled by a peripheral to be stored outside of the computer's memory. Since minicomputers are noted for their lack of memory address space (the use of mapping techniques for extending the physical address space is common practice), the extra memory provided by YAC-Bus must be seen as an

advantage.

This advantage must be qualified. Firstly, it is acknowledged that there are instances where data must be transferred to and from the computer's memory by a peripheral using DMA. For instance, in a virtual memory system for a multi-programmed computer, DMA is particularly suited to the swapping of blocks of data to and from the peripheral providing the secondary memory (usually disk) as it operates independently of the CPU and can access any part of the computer's address space.

Secondly, in comparing YAC-Bus with DMA it was assumed that YAC-Bus provides the CPU with efficient access to the data handled by the peripheral. At present the access is restricted to bytes and this places YAC-Bus at a disadvantage in any comparison with DMA. However, it would not be difficult to provide YAC-Bus with the capability to access data in terms of the minicomputer's word size (the problem of component restrictions prevented it from being considered in the present design), in which case the data within the microprocessor's memory could be accessed with an efficiency that is comparable to accessing data in the computer's own memory. However, what must ultimately count against YAC-Bus is that a computer usually has special instructions for manipulating data stored in its memory and a programmer's access to YAC-Bus is limited by the capabilities of the IO instructions that access the registers in the minicomputer's interface.

In considering the design aims for the system, it should be noted that the aims are not consistent in that they involve different and conflicting criteria. Therefore the success of YAC-Bus must be judged by considering the aims as a whole and not each in isolation. The four aims to be achieved were:

- i) efficiency
- ii) transformation of the hardware interface
- iii) flexibility
- iv) portability.

In designing YAC-Bus, the main objective was the transformation of the hardware interface of a peripheral. YAC-Bus places few requirements on the information that can be transferred across the interface. Although one is restricted to dealing with data in bytes, this is not considered to be a particularly critical shortcoming of the interface since most peripherals use the byte as a unit of information. YAC-Bus also removes format restrictions from the command and status information.

Although it is apparent that the use of a microprocessor to control a peripheral permits considerable flexibility, in the case of YAC-Bus this flexibility is overshadowed by the limitations of the minicomputer interface and the access it provides for the programmer to the data handled by the peripheral. As mentioned previously, because YAC-Bus is interfaced through the minicomputer's IO system the capabilities of the computer's IO instructions limit the access to YAC-Bus by its users.

Here it is seen that the requirements of flexibility conflict with the requirements of portability where it is essential that the interface be implemented using those facilities commonly provided by a minicomputer's IO system (such as the register interface) in order to ensure that YAC-Bus can be used on different types of minicomputer. In the attempt to implement the PDP-11 interface to YAC-Bus, the various measures taken in its design meant that parts of the implementation became very machine dependent. As pointed out in section 5.2, there is a fundamental difference between the PDP-11 and most other types of minicomputer and this was reflected in the design of the interface, especially the data register. If YAC-Bus were to be implemented on another minicomputer then another set of design problems would have to be faced. However, these problems only centre on the implementation of the data register and YAC-Bus is simple enough to allow it to be implemented on different minicomputers without the need to undergo extensive modifications to its basic functions. When implemented on another minicomputer, the user is likely

to see YAC-Bus very much as it was implemented on the PDP-11 but reflecting the differences that may exist between the two computers.

Along with the above-mentioned design aims is the consideration of efficient communication. Admittedly there are inefficiencies that arise from the fact that YAC-Bus is only capable of handling data in terms of bytes whereas it is quicker and easier for the minicomputer to handle data in terms of words. However, if YAC-Bus is considered as providing access to a type of secondary memory, this memory can be used (as data buffers, for instance) to release the computer's own memory for more important work.

7.2 FUTURE WORK AND IMPROVEMENTS

The most effective improvements that could be made to YAC-Bus would be the improvement of the facilities available at the minicomputer interface. By improving the control the programmer has over the operation of the index register, it would be possible to give him access to the data within the microprocessor's memory with facilities that match those available for accessing the computer's own memory (such as addressing modes and so on). Because we are limited to accessing YAC-Bus through the minicomputer's IO system there is a limit to how far one can go in this respect.

Another possible improvement is provision for inter-peripheral communication. By providing such a facility we eliminate the need for the computer to play an active role in the process of transferring data between peripherals. Admittedly, the need for such a facility is not common (spooling between disk storage and some input or output devices would be an example) but it could be supplied with very few modifications needed to the peripheral interface.

Can YAC-Bus be used as an IO system for a minicomputer? The reason for this question is that YAC-Bus was designed as a system to augment existing minicomputers. However, if

there was the opportunity of being able to design a mini-computer system using YAC-Bus as its IO system, how well would YAC-Bus be able to undertake the role?

It is doubtful if YAC-Bus would survive in its present form in such an application because of the need to provide such facilities as word addressing and perhaps DMA (although it may be beneficial to provide DMA as a distinctly separate system). The present minicomputer interface must be seen as merely an artifice to provide the minicomputer with access to the microprocessors' memories via YAC-Bus. In designing a new minicomputer involving YAC-Bus, the functions performed by the present interface could be built into the computer. For instance, the data and index registers could be eliminated by regarding YAC-Bus as being an extension of the computer's memory bus. This would allow the CPU to address directly the data handled by a peripheral and overcome the disadvantage mentioned in section 3.5, namely the restrictions inherent in accessing the data handled by a peripheral through the facilities provided by a conventional IO system.

Finally, we must consider the question of software since the operation of YAC-Bus is very dependent on it. In section 6.1.2, two types of YAC-Bus user were identified in terms of the software support they require. There is the casual user who is interested not so much in YAC-Bus itself but more in the facilities provided by the peripherals connected to it.

The other type of user (such as the programmer who provides the software facilities for the casual user) is more concerned with the operation of YAC-Bus and the microprocessors. This user requires tools for producing programs for the microprocessors (assemblers, loaders etc). His task is not made any simpler by the fact that the microprocessor software is very device dependent since the microprocessor has to take care of every little idiosyncrasy of the device it is controlling. This requires the generation of sequences of signals, each correctly timed and in its correct logical relationship with other signals.

Satisfying the needs of these two types of users raise questions that go beyond the scope of this thesis. For the casual user, the question of providing an abstract model that takes advantage of the capabilities of YAC-Bus and maximises the benefits to the user requires further investigation.

Because this thesis was more concerned with the interface between the microprocessor and the minicomputer, little attention has been given to the most important task of the microprocessor - that of controlling the peripheral. The software for this task is of particular concern to the more serious user of YAC-Bus and the design of this software, the allocation of tasks between the microprocessors and the minicomputer, and the design of software tools and methodologies to aid the programmer in producing software for the microprocessors are suggested as topics for future investigation.

BIBLIOGRAPHY

[Burroughs 72]

BURROUGHS CORPORATION
Burroughs B6700 Information Processing Systems
Reference Manual, 1972.

[CDC 66]

CONTROL DATA CORPORATION
Control Data 6400/6500/6600 Reference Manual, 1966.

[DEC 75]

DIGITAL EQUIPMENT CORPORATION
PDP-11/04/34/45/55 Processor Handbook, 1976.

[DG 75]

DATA GENERAL CORPORATION
Programmer's Reference Manual
Eclipse Line Computers, 1975.

[DG 78]

DATA GENERAL CORPORATION
User's Manual Programmer's Reference
Nova Line Computers, 1978.

[Elmqvist 1979]

Elmqvist, K.A.; Fullmer, H.; Gustovson, D.B.; and
Morrow, G.

"Standard Specification for S100 Bus Interface
Devices", COMPUTER, IEEE, July 1979, pp. 28-52.

[Hirschman 1979]

Hirschman, A.D.; Ali, G.; and Swan, R.

"Standard Modules offer Flexible Multiprocessor Design",
COMPUTER DESIGN, May 1979, pp. 181-189.

[IBM 74]

IBM SYSTEM PRODUCTS DIVISION
System/370 Principles of Operation, 1974.

[IBM 78]

IBM GENERAL SYSTEMS DIVISION
System/38 Technical Developments 1978

[Thurber 72]

Thurber, K.J. et. al.

"A Systematic Approach to the Design of Digital
Bussing Structures",

Proceedings AFIPS Fall Joint Computer Conference,
1972, pp. 719-740.

Other bibliographical material not explicitly referenced in
the text:

Adam Osbourne and Associates Inc.,

An Introduction to Microcomputers:

Volume 2 Some Real Microprocessors

Volume 3 Some Real Support Devices

1978

American Microsystems, Inc.

6800 Prototyping Board Manual, 1976.

MOTOROLA Inc.

M6800 Microcomputer System Design Data, 1976.

Appendix A

6800 Monitor Listing

```

00001 NAM YAC-BUS 6800 ROUTINES
00003 * PAGE ZERO LOCATIONS
00004 ORG 0
0000 00 00005 INTFLG FCB 0 SET IF NMI INTERRUPT
0001 00 00006 NCMDFG FCB 0 SET IF WE HAVE A NEW COMMAND
0002 00 00007 COMMND FCB 0 THE CURRENT COMMAND
0003 0000 00008 SAVSTK FDB 0 STACK POINTER ON A SWI
0005 0000 00009 CMSTR FDB 0 POINTER TO YAC-BUS MASTER ADDRESS
0007 0000 00010 T1 FDB 0
0009 0000 00011 SRCE FDB 0 POINTERS FOR
000B 0000 00012 DEST FDB 0 SUBROUTINE BMOVE
000D 00 00013 SMEM FCB 0 SIZE OF BUFFER IN TERMS OF 128 BYTES
000E 00 00014 TBLSZ FCB 0 NUMBER OF SYSTEM COMMANDS
000F 0000 00015 TBLPTR FDB 0 POINTER TO TABLE OF SYSTEM COMMANDS
0011 00016 TABLE EQU * TABLE IS ADDRESSES OF ROUTINES FOR EACH COMMAND
00017 ORG $FE00
00018 * START OF SYSTEM ROM
00019 * BASIC SYSTEM PARAMETERS AND JUMP TABLE CONTENTS
FE00 08 00020 TBLBS FCB 8 MEMORY SIZE
FE01 06 00021 FCB 6 MAX NUMBER OF SYSTEM COMMANDS
FE02 0011 00022 FDB TABLE
FE04 FEC3 00023 FDB INQMEM,MREAD,MWRITE,START,CONT
FE06 FED2
FE08 FEEF
FE0A FEFB
FE0C FFO3
000E 00024 NMBYTS EQU *-TBLBS
00025 * NMI SERVICE ROUTINE
00026 * COMMAND #0 IS NULL
00027 * COMMAND #1 IS HALT CURRENT USER PROGRAM
00028 * ALL COMMANDS EXCEPT RESET IGNORED IF NCMDFG IS STILL SET
00029 * OTHERWISE PUT IN COMMND AND NCMDFG SET
00030 * INTFLG, NCMDFG ASSUMED TO BE ZERO
FE0E B6 8000 00031 NMI LDAA BMEM GET COMMAND
FE11 7F 8000 00032 CLR BMEM AND CLEAR
FE14 81 00 00033 CMPA #0 IF IT WAS NOTHING,
FE16 26 04 00034 BNE NEWCMD THEN INDICATE
FE18 7C 0000 00035 INC INTFLG INTERRUPT HAS HAPPENED
FE1B 3B 00036 RTI AND RETURN
FE1C 81 01 00037 NEWCMD CMPA #1 HALT
FE1E 27 0B 00038 BEQ HALT REQUEST ?
FE20 7D 0001 00039 TST NCMDFG CHECK IF STILL DOING
FE23 26 05 00040 BNE REPEAT CURRENT COMMAND
FE25 97 02 00041 STAA COMMND FINALLY WE
FE27 7C 0001 00042 INC NCMDFG HAVE A NEW
FE2A 3B 00043 REPEAT RTI COMMAND TO DO
FE2B FE2B 00044 HALT EQU *
00045 * SWI INTERRUPT HANDLER
00046 * SWI IS USED IN THIS SYSTEM
00047 * AS A BREAKPOINT FOR TRACING
00048 * THE CODE IS ALSO SHARED BY THE NMI HANDLER FOR HALTS
FE2B 9F 03 00049 SWIHDR STS SAVSTK SAVE THE STACK POINTER
FE2D CE 8001 00050 LDX #BMEM+1 INDICATE
FE30 86 04 00051 LDAA #4 BREAKPOINT TRAP
FE32 8D 0E 00052 BSR YWRITE
FE34 20 59 00053 BRA WTLOOP AND WAIT FURTHER ACTION
00054 * YAC-BUS READ AND WRITE ROUTINES
00055 * USED IN CONJUNCTION WITH THE NMI INTERRUPT SERVICE
00056 * ROUTINE TO ACCESS YAC-BUS
00057 * CALLING METHOD;
00058 * INDEX REGISTER CONTAINS ADDRESS
00059 * ACCUMULATOR A WILL RECIEVE DATA IN READ
00060 * CONTAINS DATA IN WRITE
00061 * INTFLG IS SET IF INTERRUPT (IE NON-ACCESS OR PROBLEMS)
FE36 7F 0000 00062 YREAD CLR INTFLG
FE39 A6 00 00063 LDAA 0,X
FE3B 01 00064 NOP LET THE INTERRUPT OCCUR
FE3C 7D 0000 00065 TST INTFLG BEFORE TESTING IF IT DID
FE3F 26 F5 00066 BNE YREAD
FE41 39 00067 RTS
FE42 7F 0000 00068 YWRITE CLR INTFLG
FE45 A7 00 00069 STAA 0,X
FE47 01 00070 NOP
FE48 7D 0000 00071 TST INTFLG
FE4B 26 F5 00072 BNE YWRITE
FE4D 39 00073 RTS
00074 * SUBROUTINE BMOVE
00075 * MOVES A BLOCK OF BYTES:
00076 * - FROM THE POINTER AT SRCE
00077 * - TO THE POINTER AT DEST
00078 * THE NUMBER OF BYTES TO BE SHIFTED (1-255)
00079 * IS ASSUMED TO BE IN ACCUMULATOR B.
00080 * ACCUMULATOR A AND INDEX REGISTER ARE LOST.
00081 * USES YREAD AND YWRITE.
00082 * ON AN ATTEMPT TO WRITE TO AN INVALID ADDRESS
00083 * ACC. B WILL CONTAIN RESIDUAL COUNT AND
00084 * DEST THE INVALID ADDRESS.
FE4E DE 09 00085 BMOVE LDX SRCE PICK UP BYTE AT
FE50 8D E4 00086 BSR YREAD SOURCE ADDRESS
FE52 08 00087 INX INCREMENT THE
FE53 DF 09 00088 STX SRCE SOURCE POINTER
FE55 97 07 00089 STAA T1 SAVE COPY OF BYTE
FE57 DE 0B 00090 LDX DEST DEPOSIT BYTE AT
FE59 8D E7 00091 BSR YWRITE DESTINATION ADDRESS (HOPEFULLY)
FE5B 8D D9 00092 BSR YREAD RE-READ AND
FE5D 91 07 00093 CMPA T1 COMPARE AGAINST THE ORIGINAL
FE5F 26 06 00094 BNE EMOVE AND FINISH IF NOT THE SAME
FE61 08 00095 INX INCREMENT THE
FE62 DF 0B 00096 STX DEST DESTINATION POINTER
FE64 5A 00097 DECB DECREMENT THE COUNTER
FE65 26 E7 00098 BNE BMOVE IN ACCUMULATOR B UNITL ZERO
FE67 39 00099 EMOVE RTS

```

```

00100 * MAIN LINE OF PROGRAM
FE68 0F 00101 MAIN SEI REFUSE INTERRUPTS
FE69 8E FFF0 00102 LDS #STKBS RESET STACK
FE6C CE FE0E 00103 LDX #NMI LOAD UP VECTORS
FE6F FF FFFC 00104 STX $FFFC NMI
FE72 CE FE2B 00105 LDX #SWIHDR
FE75 FF FFFA 00106 STX $FFFA SWI
FE78 CE 000D 00107 LDX #TBLPTR-2 MOVE
FE7B DF 0B 00108 STX DEST JUMP
FE7D CE FE00 00109 LDX #TBLBS TABLE
FE80 DF 09 00110 STX SRCE DOWN INTO
FE82 C6 0E 00111 LDAB #NMBYTES WORKING
FE84 8D C8 00112 BSR BMOVE STORAGE
FE86 7F 8000 00113 CLR BMEM RESET COMMAND INPUT
FE89 CE 8001 00114 LDX #BMEM+1 RESET OUR CURRENT MASTER
FE8C DF 05 00115 STX CMSTR TO FOLLOW HIM ABOVE
FE8E 0E 00116 CLI LET'S GO
FE8F 7F 0001 00117 WTLOOP CLR NCMDFG RESET NEW COMMAND FLAG
FE92 7F 0002 00118 CLR COMMND COMMAND ITSELF
FE95 3E 00119 WAI WAIT IF THERE IS NOTHING ELSE
FE96 7D 0001 00120 TST NCMDFG ANYTHING TO DO ?
FE99 26 03 00121 BNE NMIINT PERHAPS -
FE9B 7E 0200 00122 JMP MSKINT OTHER TYPE OF INTERRUPT ?
FE9E 96 02 00123 NMIINT LDAA COMMND MUST BE A NMI:
FEA0 91 0E 00124 CMPA TBLSZ COMMND > 1 (WE ALREADY KNOW)
FEA2 22 17 00125 BHI ILLCMD COMMND <= TBLSZ (MAX. AVAILABLE COMMANDS)
FEA4 D6 0F 00126 LDAB TBLPTR B TAKES UPPER BYTE OF JMP TABLE
FEA6 80 02 00127 SUBA #2
FEA8 48 00128 ASLA MULTIPLY REG 2 BY 2
FEA9 24 01 00129 BCC NINCB1 WAS THERE CARRY ?
FEAB 5C 00130 INCB YES - ADD 1 TO B
FEAC 99 10 00131 NINCB1 ADCA TBLPTR+1 ADD LOWER BYTE INTO
FEAE 24 01 00132 BCC NINCB2 REG A AND AGAIN
FEB0 5C 00133 INCB CHECK FOR CARRY
FEB1 D7 07 00134 NINCB2 STAB T1 STORE VALUES IN
FEB3 97 08 00135 STAA T1+1 TEMPORARY
FEB5 DE 07 00136 LDX T1 TO LOAD INTO
FEB7 EE 00 00137 LDX 0,X INDEX REG
FEB9 6E 00 00138 JMP 0,X SO ONE CAN JUMP
FEBB DE 05 00139 ILLCMD LDX CMSTR TELL WHOEVER GAVE US
FEBD 86 01 00140 LDAA #1 SUCH RIDICULOUS THINGS -
FEBF 8D 81 00141 BSR YWRITE WE WON'T
FEC1 20 CC 00142 BRA WTLOOP SO THERE
00143 * SYSTEM FUNCTIONS
00144 * COMMAND #2 -INQUIRE AS TO BUFFER MEMORY SIZE
FEC3 96 0D 00145 INQMEM LDAA SMEM PICK UP CONSTANT
FEC5 B7 8002 00146 STAA BMEM+2 AND DEPOSIT
FEC8 86 02 00147 DONE LDAA #2 SAY
FECA DE 05 00148 LDX CMSTR WE'VE
FECC BD FE42 00149 JSR YWRITE DONE
FECF 7E FEBF 00150 JMP WTLOOP
00151 * COMMAND #3 -READ BLOCK
FED2 FE 8002 00152 MREAD LDX BMEM+2 LOAD
FED5 DF 09 00153 STX SRCE SOURCE POINTER
FED7 CE 8005 00154 LDX #BMEM+5 LOAD
FEDA DF 0B 00155 STX DEST DESTINATION POINTER
FEDC F6 8004 00156 DOIT LDAB BMEM+4 LOAD BYTE COUNTER
FEDF BD FE4E 00157 JSR BMOVE MOVE IT ALL
FEE2 F7 8004 00158 STAB BMEM+4 RETURN RESIDUAL COUNT
FEE5 FF 8002 00159 STX BMEM+2 WHERE WE STOPPED
FEE8 5D 00160 TSTB SEE IF WE
FEE9 27 DD 00161 BEQ DONE FINISHED THE MOVE
FEEB 86 03 00162 NOTDNE LDAA #3 NO - ERROR RETURN
FEED 20 DB 00163 BRA DONE+2
00164 * COMMAND #4 -WRITE BLOCK
FEFF FE 8002 00165 MWRITE LDX BMEM+2 LOAD
FEF2 DF 0B 00166 STX DEST DESTINATION POINTER
FEF4 CE 8005 00167 LDX #BMEM+5 LOAD
FEF7 DF 09 00168 STX SRCE SOURCE POINTER
FEF9 20 E1 00169 BRA DOIT CONTINUE AS THE SAME AS BLOCK READ
00170 * COMMAND #5 -START UP A USER PROGRAM
FEFB 8E FFF0 00171 START LDS #STKBS RESET STACK
FEFE FE 8002 00172 LDX BMEM+2 PICK UP ADDRESS
FF01 6E 00 00173 JMP 0,X AND JUMP TO IT
00174 * COMMAND #6 -CONTINUE EXECUTION
FF03 9E 03 00175 CONT LDS SAVSTK
FF05 3B 00176 RTI
00177 * MISC. CONSTANTS ETC.
0200 00178 MSKINT EQU $200 LOCATION OF USER IRQ HANDLER
FFF0 00179 STKBS EQU $FFF0 STACKBASE
8000 00180 BMEM EQU $8000 LOCATION OF YAC-BUS BUFFER AREA
0000 00181 END

```

Appendix B

PDP-11 Monitor Listing

Main Program

B-2

Subroutines

ERROR

B-5

CONHEX, FHEXD, GDVADR,
GCMD, INC, NEXT, QADD1

B-6

QLEN, USER, VALADR,
VALHEX, YBCOM, YBINT

B-7

YBRD1, YBW1, YBRDA, YBWRA,
ISHFT

B-8

SETINT, YREAD, YWRITE,
YBREAD, YBRITE

Figure 6.1

```

C
C THE YAC-BUS RUN-TIME MONITOR SYSTEM
C
0001 IMPLICIT INTEGER*2(A-Z)
C
C THE FOLLOWING ARE THE SUBROUTINES AND FUNCTIONS
C USED WITHIN THE MONITOR
C
0002 EXTERNAL IPEEK, IPOKE, SETINT, ISHFT, INC,
1 QLEN, QADD1,
2 VALHEX, FHEXD, CONHEX,
3 VALADR, YBCOM, NEXT,
4 YBRDA, YBWRA, YBRD1, YBWR1,
5 YREAD, YWRITE, YBREAD, YBRITE,
6 USER, ERROR, GDVADR, QCMD, YBINT
C
C YAC-BUS INTERRUPT-SERVICE QUEUE AND SUPPORT
C
0003 COMMON /Q/SERG(17), QS, QE, IQ(5), QI
C
C INPUT BUFFER AREA
C
0004 LOGICAL*1 INBUF(72)
0005 COMMON /BUFFER/INBUF
C
C VALID INPUT COMMANDS
C
0006 LOGICAL*1 CMD(9)
0007 DATA CMD/'D', 'S', 'G', 'I', 'C', 'U', 'E', 'R', 'W'/
C
C CURRENT STATUS INFORMATION
C
0008 COMMON /YBSTS/CURCMD(16), REPLY(16), STATUS(16), ERR(16)
C
C INTERRUPT STATUS INFORMATION
C
0009 LOGICAL*1 SINT
0010 COMMON /INT/WHO, WHY, SINT
C
C YAC-BUS READ ERROR FLAG
C
0011 LOGICAL*1 ERF
0012 COMMON /IQ/ERF
C
C TEMPORARY AND WORKING STORAGE
C
0013 LOGICAL*1 BUF1(4), BUF2(4), VALADR
0014 LOGICAL*1 SVAR(256), TEMP(20), TEMP2(52), TEMP3(4), FLAG, FLAG2
C
C S T A R T O F P R O G R A M
C
0015 TYPE 5000
0016 5000 FORMAT(1X, 'YAC-BUS RUN-TIME SYSTEM')
0017 PAUSE 'TYPE <CR> TO CONTINUE'
C
C SET UP INTERFACE AND INTERRUPT SERVICE ROUTINE
C
0018 I=IPEEK("164014)
0019 CALL SETINT("174, 3, YBINT)
C
C DO A CENSUS ON WHO IS WORKING ON YAC-BUS
C
0020 QS=1
0021 QE=1
0022 QI=1
0023 DO 40 I=1, 16
C
C CHECK WHO IS OUT THERE BY SENDING COMMAND #2
C TO EACH IN TURN
C (THE COMMAND ASKS FOR THE BUFFER SIZE)
C IF IT DOES NOT REPLY WITHIN A REASONABLE TIME
C THEN IT IS ASSUMED TO NON-EXISTANT
C
0024 CURCMD(I)=2
0025 SINT=.FALSE.
C
C GIVE COMMAND
C
0026 CALL YWRITE(2, ISHFT(I-1, 12))
C
C NOW WAIT FOR REPLY
C
0027 DO 10 J=1, 1000
0028 IF(SINT)GOTO 20
0030 CONTINUE
0031 STATUS(I)=0
0032 GOTO 30
C
C BEWARE OF INTRUDERS
C
0033 20 IF(SERG(QS), NE. I+1. AND. REPLY(SERG(QS)), NE. 2)CALL ERROR(2, SERG(QS)-1)
C
C VALUE OBTAINED IS THE NUMBER OF 128 BYTE BLOCKS OF BUFFER MEMORY
C
0035 STATUS(I)=YREAD(ISHFT(I-1, 12)+2)*128
C
C HAVE GENERATED AN INTERRUPT THEREFORE DELETE FROM QUEUE
C
0036 CALL QADD1(QS)
0037 30 CURCMD(I)=0
0038 ERR(I)=0
0039 REPLY(I)=0
0040 40 CONTINUE
C
C WE ARE NOW READY TO ACCEPT COMMANDS FROM THE USER
C

```

```

0041 50 TYPE 5010
0042 5010 FORMAT(1X, '>', $)
0043 ACCEPT 5020, INBUF
0044 5020 FORMAT(72A1)
C
C FIND THE COMMAND
C
0045 I=CCMD(CMD, 9)
0046 IF(I. LE. 0)GOTO 2000
0048 IF(I. LE. 3)GOTO 70
C
C JUMP TO THE APPROPRIATE CODE TO FURTHER DECODE THE COMMAND
C AND EXECUTE IT
C
0050 60 GOTO(100, 200, 300, 400, 500, 600, 700, 800, 900) I
C
C COMMON CODE FOR THE FIRST THREE COMMANDS
C FIRST PARAMETER (II) IS THE DEVICE ADDRESS (0-F)
C SECOND PARAMETER (ADDR) IS AN ADDRESS (0000-FFFF)
C
0051 70 II=CDVADR(ADDR, FLAG, FLAG2)
0052 IF(FLAG)GOTO 2100
0054 IF(FLAG2)GOTO 2200
0056 JJ=3
0057 GOTO 60
C
C C O M M A N D " D " (DISPLAY)
C THIRD PARAMETER IS NUMBER OF BYTES TO DISPLAY
C
0058 100 NUMBR=FHEXD(2, FLAG)
0059 IF(FLAG)GOTO 2300
0061 CNTR=3
0062 GOTO 1400
C
C C O M M A N D " S " (SET)
C THIRD PARAMETER IS A LIST OF BYTES
C
0063 200 DO 210 NUMBR=1, 40
0064 T1=FHEXD(2, FLAG)
0065 IF(FLAG)GOTO 220
0067 210 TEMP(NUMBR+3)=T1
0068 GOTO 2500
0069 220 NUMBR=NUMBR-1
0070 JJ=4
0071 CNTR=NUMBR+3
0072 GOTO 1400
C
C C O M M A N D " G " (GO)
C THE MICROPROCESSOR IS TOLD TO START EXECUTING
C A PROGRAM AT ADDR
C
0073 300 TEMP(1)=ISHFT(ADDR, -8)
0074 TEMP(2)=ADDR. AND. "377
0075 CALL YBWRA(II, 2, TEMP, 2)
0076 CALL YBCOM(II, 5)
0077 GOTO 50
C
C C O M M A N D " I " (INSPECT INTERRUPT REGISTER)
C
0078 400 II=IPEEK("164014)
0079 CALL CONHEX(TEMP, 3, II)
0080 TYPE 5030, (TEMP(JJ), JJ=1, 3)
0081 5030 FORMAT(1X, 'INTERRUPT VECTOR: DEVICE ', A1, ' WITH ', 2A1)
0082 GOTO 50
C
C C O M M A N D " C " (CONTINUE EXECUTION)
C TELL THE MICROPROCESSOR TO KEEP GOING.....
C
0083 500 II=FHEXD(1, FLAG)
0084 IF(FLAG)GOTO 2100
0086 CURCMD(II+1)=0
0087 REPLY(II+1)=0
0088 CALL YBCOM(II, 6)
0089 GOTO 50
C
C C O M M A N D " U " (USER ROUTINE)
C
0090 600 CALL USER
0091 GOTO 50
C
C C O M M A N D " E " (ERROR ROUTINE)
C GIVES ACCESS TO FURTHER DEBUGGING INFORMATION
C
0092 700 CALL ERROR(0, 0)
0093 GOTO 50
C
C C O M M A N D " R " (READ FROM YAC-BUS)
C ARGUMENT IS THE YAC-BUS ADDRESS
C
0094 800 ADDR=FHEXD(4, FLAG)
0095 IF(FLAG)GOTO 2200
0097 II=YREAD(ADDR)
0098 CALL CONHEX(BUF1, 4, ADDR)
0099 CALL CONHEX(BUF2, 2, II)
0100 TYPE 5040, BUF2(1), BUF2(2), BUF1
0101 5040 FORMAT(1X, 'YACBUS READ: VALUE ', 2A1, ' AT ADDRESS ', 4A1)
0102 GOTO 50
C
C C O M M A N D " W " (WRITE TO YAC-BUS)
C FIRST PARAMETER IS YAC-BUS ADDRESS
C SECOND PARAMETER IS THE BYTE
C
0103 900 ADDR=FHEXD(4, FLAG)
0104 IF(FLAG)GOTO 2200
0106 II=FHEXD(2, FLAG)
0107 IF(FLAG)GOTO 2500

```

```

C     THE REST OF THE CODE FOR THE "D" AND "S" COMMANDS
C     FIRSTLY, SAVE WORKING AREA
0111 1400 CALL YBRDA(II,2,SVAR,NUMBR+3)
C
C     SET UP THE PARAMETERS FOR THE BLOCK TRANSFER
C     BY THE MICROPROCESSOR
0112     TEMP(1)=ISHFT(ADDR,-8)
0113     TEMP(2)=ADDR.AND."377
0114     TEMP(3)=NUMBR
0115     CALL YBWRA(II,2,TEMP,CNTR)
0116     CALL YBCOM(II,JJ)
C
C     WAIT FOR THE REPLY FROM THE MICROPROCESSOR
0117 1410 IF(QLEN().LE.0)GOTO 1410
C
C     CHECK IF IT IS THE ONE WE ARE CURRENTLY USING
0119     IF(SERG(QS)-1.EQ.II)GOTO 1440
C
C     IT'S NOT... BUT WE'LL TELL HIM JUST IN CASE.
0121     T2=SERG(QS)
0122     CALL CONHEX(TEMP,3,ISHFT(T2-1,8).OR.REPLY(T2))
0123     CALL CONHEX(TEMP2,4,ISHFT(CURCMD(T2),8).OR.ERR(T2))
0124     TYPE 5055,(TEMP(JJ),JJ=1,3),(TEMP2(JJ),JJ=1,4)
0125 5055 FORMAT(1X,'DEVICE ',A1,' REPLY ',2A1,' COMMAND ',2A1,' ERROR ',2A1)
0126     CALL NEXT
0127     GOTO 1410
C
C     RECOVER FINISHING INFO FROM THE BLOCK MOVE
0128 1440 CALL YBRDA(II,2,TEMP,3)
0129     T2=TEMP(1)
0130     T2=ISHFT(T2,8).OR.TEMP(2)
C
C     THUS ENDS THE COMMON CODE
0131     GOTO(1450,1490)I
C
C     C O M M A N D " D "
C     WE ARE NOW GOING TO PRINT OUT THE BIT OF
C     MEMORY WE WERE ASKED FOR
0132 1450 BASE=5
0133     T3=TEMP(3)
0134     T3=T3.AND."377
0135     CNTR=NUMBR-T3
0136     T3=3
0137 1460 CALL CONHEX(TEMP2,4,ADDR)
0138     T1=ADDR.OR."17
0139     TEMP2(5)="40
0140     DD 1470 JJ=1,16
0141     IF(CNTR.LE.0)GOTO 1480
0143     CALL CONHEX(TEMP3,2,YBRD1(II,BASE))
0144     BASE=BASE+1
0145     T3=3*(JJ+1)
0146     TEMP2(T3)=TEMP3(1)
0147     TEMP2(T3+1)=TEMP3(2)
0148     TEMP2(T3+2)="40
0149     IF(ADDR.EQ.T1)GOTO 1480
0151     CNTR=CNTR-1
0152 1470 ADDR=INC(ADDR)
0153 1480 T3=T3+2
0154     TYPE 5060,(TEMP2(JJ),JJ=1,T3)
0155 5060 FORMAT(1X,52A1)
0156     ADDR=INC(ADDR)
0157     CNTR=CNTR-1
0158     IF(CNTR.GT.0)GOTO 1460
0160     GOTO 1550
C
C     C O M M A N D " S "
0161 1490 IF (REPLY(SERG(QS)).EQ.2)GOTO 1550
0163 1500 CALL CONHEX(TEMP3,4,T2)
0164     TYPE 5070,TEMP3
0165 5070 FORMAT(1X,'INVALID ADDRESS AT ',4A1)
C
C     RESTORE WORKING AREA
0166 1550 CALL YBWRA(II,2,SVAR,NUMBR+3)
0167     CALL NEXT
0168     GOTO 50
C
C     SUITABLE ERROR MESSAGES:
0169 2000 TYPE 5200,CMD
0170 5200 FORMAT(1X,'PRESENT USER COMMANDS ARE:',/,16(3X,A1))
0171     GOTO 50
0172 2100 TYPE 5300
0173 5300 FORMAT(1X,'NO DEVICE NUMBER')
0174     GOTO 50
0175 2200 TYPE 5400
0176 5400 FORMAT(1X,'NO ADDRESS')
0177     GOTO 50
0178 2300 TYPE 5500
0179 5500 FORMAT(1X,'NO NUMBER OF BYTES')
0180     GOTO 50
0181 2400 TYPE 5600
0182 5600 FORMAT(1X,'INVALID ADDRESS RANGE')
0183     GOTO 50
0184 2500 TYPE 5700
0185 5700 FORMAT(1X,'NO BYTE')
0186     GOTO 50

```

```

0001      SUBROUTINE ERROR(WHAT,CLUE)
          C
          C      ERROR HANDLER AND DEBUGGER
          C      THE TWO PARAMETERS HAVE NO PRE-DEFINED MEANING...
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      EXTERNAL CONHEX, NEXT, FHEXD, YREAD, YWRITE, IPEEK
0004      COMMON /YBSTS/CURCMD(16), REPLY(16), STATUS(16), ERR(16),
          1      /Q/SERQ(17), QS, QE, IQ(5), QI,
          2      /BUFFER/INBUF(72)
0005      LOGICAL*1 ECMD(9), INBUF, FLAG, BUF1(4), BUF2(2)
0006      DATA ECMD/'C', 'S', 'Q', 'D', 'R', 'W', 'I', 'N', 'X'/
0007      IF(WHAT.EQ.0)GOTO 2000
          C
          C      SAY WHERE WE ARE...
          C
0009      TYPE 5000,WHAT,CLUE
0010      5000  FORMAT(1X,'ERROR: ',I4,' CLUE: ',I6)
0011      2000  TYPE 5200,CLUE
0012      5200  FORMAT(1X,I4,'E=>',S)
          C
          C      GET A COMMAND
          C
0013      ACCEPT 5205,INBUF
0014      5205  FORMAT(72A1)
0015      II=QCMD(ECMD,9)
          C
          C      AND DO SOMETHING ABOUT IT
          C
0016      IF(II.LE.0)GOTO 2010
0018      GOTO (2010,2020,2030,2040,3000,3100,3200,3300,3400)II
          C
          C      C O M M A N D " C " (CONTINUE)
          C
0019      2010  RETURN
          C
          C      C O M M A N D " S " (STOP)
          C
0020      2020  STOP 'ERROR HALT'
          C
          C      C O M M A N D " Q " (QUEUE DUMP)
          C
0021      2030  TYPE 5210, QS, QE, SERQ
0022      5210  FORMAT(1X,'QSTART = ',I2,4X,'QEND = ',I2/,
          1 1X,'SERVICE QUEUE : ',/,17(2X,I2))
0023      GOTO 2000
          C
          C      C O M M A N D " D " (DUMP STATUS)
          C
0024      2040  TYPE 5220
0025      5220  FORMAT(1X,'STATUS DUMP',/,
          1 4X,'DEVICE STATUS COMMAND REPLY ERROR')
          C
0026      DD 2045 I=1,16
0027      IF(STATUS(I).EQ.0.AND.CURCMD(I).EQ.0.AND.
          1 REPLY(I).EQ.0.AND.ERR(I).EQ.0)GOTO 2045
          C
0029      TYPE 5230,I-1,STATUS(I),CURCMD(I),REPLY(I),ERR(I)
0030      5230  FORMAT(5(4X,I4))
0031      2045  CONTINUE
0032      GOTO 2000
          C
          C      C O M M A N D " R " (READ YAC-BUS)
          C
0033      3000  ADDR=FHEXD(4,FLAG)
0034      IF(FLAG)GOTO 4000
0036      JJ=YREAD(ADDR)
0037      CALL CONHEX(BUF1,4,ADDR)
0038      CALL CONHEX(BUF2,2,JJ)
0039      TYPE 5500,BUF2,BUF1
0040      5500  FDRMAT(1X,'YACBUS READ: VALUE ',2A1,' AT ADDRESS ',4A1)
0041      GOTO 2000
          C
          C      C O M M A N D " W " (WRITE YAC-BUS)
          C
0042      3100  ADDR=FHEXD(4,FLAG)
0043      IF(FLAG)GOTO 4000
0045      JJ=FHEXD(2,FLAG)
0046      IF(FLAG)GOTO 4010
0048      CALL YWRITE(II,ADDR)
0049      GOTO 2000
          C
          C      C O M M A N D " I " ( INTERRUPT REGISTER)
          C
0050      3200  CALL CONHEX(BUF1,3,IPEEK("164014"))
0051      TYPE 5600,(BUF1(I),I=1,3)
0052      5600  FORMAT(1X,'INTERRUPT VECTOR: DEVICE ',A1,' WITH ',2A1)
0053      GOTO 2000
          C
          C      C O M M A N D " N " (NEXT)
          C
0054      3300  CALL NEXT
0055      GOTO 2000
          C
          C      C O M M A N D " X " (INTERRUPT TRACE)
          C
0056      3400  TYPE 5700
0057      5700  FORMAT(1X,'PREVIOUS INTERRUPTS:')
0058      DD 3420 JJ=1,5
0059      CALL CONHEX(BUF1,3,IQ(JJ))
0060      IF(JJ.EQ.QI)GOTO 3410
0062      TYPE 5800,(BUF1(I),I=1,3)
0063      5800  FORMAT(3X,A1,2X,2A1)
0064      GOTO 3420
0065      3410  TYPE 5900,(BUF1(I),I=1,3)
0066      5900  FORMAT(1X,'>>',A1,'**',2A1,'<<')
0067      3420  CONTINUE
          C
          C      SOME APPROPRIATE ERROR MESSAGES
          C
0069      4000  TYPE 6000
0070      6000  FORMAT(1X,'NO ADDRESS')
0071      GOTO 2000
0072      4010  TYPE 6010
0073      6010  FORMAT(1X,'NO BYTE')
0074      GOTO 2000
0075      END

```

```

0001      SUBROUTINE CONHEX(J,K,L)
          C
          C      CONVERTS L INTO ASCII HEXADEcimal DIGITS
          C      STORED IN ARRAY J.
          C      THE CHARACTERS ARE THEN SHIFTED DEPENDING
          C      ON HOW MANY WE WANT (K)
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      LOGICAL*1 J(4), ZERO, AA
0004      EXTERNAL ISHFT
0005      DATA ZERO/'0'//, AA/'A'//
0006      IF(K.GT.4)K=4
0008      DO 10 I=1,4
0009          T1=ISHFT(ISHFT(L.(I-1)*4),-12)
0010          J(I)=T1+ZERO
0011      10  IF(T1.GT.9)J(I)=T1+AA-10
0013          DO 20 I=1,K
0014      20  J(I)=J(4-K+I)
0015          RETURN
0016          END

0001      INTEGER FUNCTION FHEXD*2(I,FLAG)
          C
          C      FUNCTION RETURNS THE VALUE OF UP TO I (MAX 4)
          C      CONSECUTIVE HEXADEcimal DIGITS IN INBUF
          C      TERMINATED BY A NON-HEXADEcimal CHARACTER
          C      FLAG SET TO .TRUE. IF NONE FOUND
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      LOGICAL*1 INBUF,FLAG
0004      COMMON /BUFFER/INBUF(72)
0005      EXTERNAL VALHEX,ISHFT
0006      FLAG=.FALSE.
0007      T1=0
0008      DO 10 J=1,72
0009          T2=VALHEX(INBUF(J))
0010          INBUF(J)=0
0011          IF(T2.GE.0)GOTO 20
0013      10  CONTINUE
0014          FLAG=.TRUE.
0015          RETURN
0016      20  DO 30 K=J,71
0017          IF(T2.LT.0)GOTO 40
0019          T1=ISHFT(T1,4).OR.T2
0020          T2=VALHEX(INBUF(K+1))
0021      30  INBUF(K+1)=0
0022      40  T2=(4-I)*4
0023          FHEXD=ISHFT(ISHFT(T1,T2),-T2)
0024          RETURN
0025          END

0001      INTEGER FUNCTION GDVADR*2(ADDR,F1,F2)
          C
          C      GETS THE FIRST TWO PARAMETERS FOR THE D,S AND Q COMMANDS
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      LOGICAL*1 F1,F2
0004      EXTERNAL FHEXD
0005      GDVADR=FHEXD(1,F1)
0006      ADDR=FHEXD(4,F2)
0007      RETURN
0008      END

```

```

0001      INTEGER FUNCTION GCMD*2(WTODD,MAX)
          C
          C      THIS FUNCTION SEES IF ANY COMMAND
          C      CHARACTER IN WTODD OCCURS IN THE
          C      INPUT STRING INBUF
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      COMMON /BUFFER/INBUF(72)
0004      LOGICAL*1 INBUF,WTODD(MAX)
0005      DO 20 I=1,72
0006      DO 10 J=1,MAX
0007          IF(WTODD(J).EQ.INBUF(I))GOTO 30
0009      10  CONTINUE
0010          INBUF(I)=0
0011          GCMD=-1
0012          RETURN
0013      30  INBUF(I)=0
0014          GCMD=J
0015          RETURN
0016          END

0001      INTEGER FUNCTION INC*2(VAL)
          C
          C      FUNCTION TO AVOID OVERFLOW PROBLEMS WITH
          C      INTEGER ARITHMETIC ON UNSIGNED QUANTITIES
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      IF(VAL.EQ."077777")GOTO 10
0005      INC=VAL+1
0006      RETURN
0007      10  INC="100000"
0008          RETURN
0009          END

0001      SUBROUTINE NEXT
          C
          C      REMOVE ENTRY FROM THE SERVICE QUEUE
          C      (USED WHEN WE HAVE FINISHED WITH THAT INTERRUPT)
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      COMMON /YBSTS/CURCMD(16),REPLY(16),STATUS(16),ERR(16),
          1  /Q/SERG(17),QS,QE,IQ(5),QI
0004      EXTERNAL GADD1
0005      CURCMD(SERG(QS))=0
0006      REPLY(SERG(QS))=0
0007      ERR(SERG(QS))=0
0008      CALL GADD1(QS)
0009      RETURN
0010      END

0001      SUBROUTINE GADD1(PTR)
          C
          C      THIS ROUTINE INCREMENTS A QUEUE POINTER
          C
0002      INTEGER*2 PTR
0003      PTR=PTR+1
0004      IF(PTR.GE.18)PTR=1
0006      RETURN
0007      END

```

```

0001      INTEGER FUNCTION GLEN*2
          C
          C      RETURNS THE NUMBER OF WAITING ENTERIES IN SERQ
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      COMMON /G/SERG(17),GS,GE,IG(5),GI
0004      IF(GE.GE.GS)GOTO 10
0006      GLEN=17-(GS-GE)
0007      RETURN
0008 10    GLEN=GE-GS
0009      RETURN
0010      END

0001      SUBROUTINE USER
          C
          C      GUESS WHAT.....
          C
0002      RETURN
0003      END

0001      LOGICAL FUNCTION VALADR*1(WHO,WHERE,NUMBR)
          C
          C      VALIDATES A YAC-BUS ADDRESS
          C      CHECKS IF A PERIPHERAL EXISTS AND THAT
          C      WE ARE ACCESSING VALID MEMORY
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      COMMON /YBSTS/CURCMD(16),REPLY(16),STATUS(16),ERR(16)
0004      T1=WHO.GE.0.AND.WHO.LE.15
0005      IF(T1)T1=WHERE.GE.0.AND.WHERE.LT.STATUS(WHO+1)
0007      IF(T1)T1=WHERE+NUMBR.LT.STATUS(WHO+1).AND.NUMBR.GT.0
0009      VALADR=T1
0010      RETURN
0011      END

0001      INTEGER FUNCTION VALHEX*2(I)
          C
          C      HEXADECMAL CHARACTER INPUT CONVERSION ROUTINE
          C
0002      IMPLICIT LOGICAL*1(A-Z)
0003      DATA ZERO/'0'/,NINE/'9'/,AA/'A'/,FF/'F'/
0004      VALHEX=-1
0005      IF(I.GE.ZERO.AND.I.LE.NINE)VALHEX=I-ZERO
0007      IF(I.GE.AA.AND.I.LE.FF)VALHEX=I-AA+10
0009      RETURN
0010      END

0001      SUBROUTINE YBCOM(WHO,WHAT)
          C
          C      PERIPHERAL COMMAND ROUTINE
          C      ERRORS: 3) NOT-EXISTENT PERIPHERAL
          C              4) PREVIOUS COMMAND NOT PROCESSED
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      COMMON /YBSTS/CURCMD(16),REPLY(16),STATUS(16),ERR(16)
0004      LOGICAL*1 VALADR
0005      EXTERNAL VALADR,ERROR,ISHFT
0006      IF(.NOT.VALADR(WHO,1,1))CALL ERROR(3,WHO)
0008      IF(CURCMD(WHO+1).NE.0.OR.REPLY(WHO+1).NE.0)CALL ERROR(4,WHO)
0010      CURCMD(WHO+1)=WHAT
0011      ERR(WHO+1)=0
0012      CALL YWRITE(WHAT,ISHFT(WHO,12))
0013      RETURN
0014      END

```

```

0001      SUBROUTINE YBINT
          C
          C      INTERRUPT SERVICE ROUTINE
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      LOGICAL*1 SINT
0004      COMMON /YBSTS/CURCMD(16),REPLY(16),STATUS(16),ERR(16),
1          /G/SERG(17),GS,GE,IG(5),GI,
2          /INT/WHO,WHY,SINT
0005      EXTERNAL ISHFT,GADD1,IPEEK
0006      SINT=.TRUE.
          C
          C      GET INFO FROM THE INTERRUPT REGISTER
          C
0007      I=IPEEK("164014)
0008      WHY=I.AND."377
0009      WHO=ISHFT(I,-8)
          C
          C      STORE THE VALUE IN THE IG QUEUE
          C      FOR INTERRUPT TRACING JUST IN CASE...
          C
0010      IG(QI)=I
0011      QI=1+QI
0012      IF(QI.GT.5)QI=1
0014      I=WHO+1
          C
          C      IS THIS WHAT WE EXPECT ?
          C
          C      UNKNOWN PERIPHERAL
0015      IF(STATUS(I).EQ.0)ERR(I)=1
          C      ILLEGAL COMMAND
0017      IF(WHY.EQ.1)ERR(I)=2
          C      ERROR (NOT DONE)
0019      IF(WHY.EQ.4)ERR(I)=3
          C      UNSOLICITATED
0021      IF(CURCMD(I).EQ.0)ERR(I)=4
          C
          C      SET UP STATUS TABLE
          C      AND ENTER INTO THE SERVICE QUEUE
          C
0023      REPLY(I)=WHY
0024      SERG(GE)=I
0025      CALL GADD1(GE)
0026      RETURN
0027      END

```

```

0001      INTEGER FUNCTION YBRD1*2(WHO,WHERE)
          C
          C      PROTECTED YAC-BUS SINGLE BYTE WRITE
          C      ERRORS: 10) ILLEGAL ADDRESS
          C      11) YAC-BUS ERROR
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      EXTERNAL VALADR, ISHFT, ERROR, YREAD
0004      LOGICAL*1 VALADR,ERF
0005      COMMON /IO/ERF
0006      IF(.NOT.VALADR(WHO,WHERE,1))CALL ERROR(10,WHO)
0008      YBRD1=YREAD(ISHFT(WHO,12).OR.WHERE)
0009      IF(ERF)CALL ERROR(11,WHO)
0011      RETURN
0012      END
    
```

```

0001      SUBROUTINE YBWR1(WHO,WHERE,WHAT)
          C
          C      PROTECTED YAC-BUS SINGLE BYTE WRITE
          C      ERROR: 9) ILLEGAL ADDRESS
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      EXTERNAL VALADR, ISHFT, ERROR, YWRITE
0004      LOGICAL*1 VALADR
0005      IF(.NOT.VALADR(WHO,WHERE,1))CALL ERROR(9,WHO)
0007      CALL YWRITE(WHAT,ISHFT(WHO,12).OR.WHERE)
0008      RETURN
0009      END
    
```

```

0001      SUBROUTINE YBRDA(WHO,WHERE,WHAT,NUM)
          C
          C      PROTECTED YAC-BUS ARRAY OF BYTES READ
          C      ERRORS: 6) ILLEGAL ADDRESS
          C      7) YAC-BUS ERROR
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      LOGICAL*1 WHAT(NUM),ERF,VALADR
0004      COMMON /IO/ERF
0005      EXTERNAL VALADR, ERROR, YBREAD, ISHFT
0006      IF(.NOT.VALADR(WHO,WHERE,NUM))CALL ERROR(6,WHO)
0008      CALL YBREAD(WHAT,NUM,ISHFT(WHO,12).OR.WHERE)
0009      IF(ERF)CALL ERROR(7,WHO)
0011      RETURN
0012      END
    
```

```

0001      SUBROUTINE YBWRA(WHO,WHERE,WHAT,NUM)
          C
          C      PROTECTED YAC-BUS ARRAY OF BYTES WRITE
          C      ERROR: 8) ILLEGAL ADDRESS
          C
0002      IMPLICIT INTEGER*2(A-Z)
0003      LOGICAL*1 WHAT(NUM),VALADR
0004      EXTERNAL VALADR, YWRITE, ERROR, ISHFT
0005      IF(.NOT.VALADR(WHO,WHERE,NUM))CALL ERROR(8,WHO)
0007      CALL YWRITE(WHAT,NUM,ISHFT(WHO,12).OR.WHERE)
0008      RETURN
0009      END
    
```

```

1          .TITLE  SHIFTER
2          .GLOBL  ISHFT
3          .MCALL  ..V2...REGDEF
4 000000    ..V2..
5 000000    .REGDEF
6          ; THIS FUNCTION IS FOR SHIFTING FORTRAN INTEGER*2
7          ; VARIABLES EITHER LEFT OR RIGHT UP TO 15 PLACES
8          ; FIRST ARGUMENT IS THE VALUE TO SHIFTED
9          ; SECOND ARGUMENT IS THE NUMBER OF BITS TO SHIFT
10         ;
11         ;      +VE FOR LEFT SHIFT
12         ;      -VE FOR RIGHT SHIFT
13         ;
13 000000    005723    ISHFT:  TST   (R5)+      ;SKIP OVER 1ST PARAMTER
14 000002    010146    MOV    R1,-(SP)      ;SAVE WORKING
15 000004    010246    MOV    R2,-(SP)      ; REGISTERS
16 000006    013500    MOV    @R5)+,R0     ;RETRIEVE VALUE
17 000010    017501    000000    MOV    @R5),R1      ; DITTO # OF PLACES TO SHIFT
18 000014    005002    CLR    R2
19 000016    005701    TST    R1
20 000020    100002    BPL    STRT      ; BRANCH IF LEFT
21 000022    005401    NEG    R1
22 000024    005202    INC    R2
23 000026    042701    177760    STRT:  BIC    #177760,R1   ; SET R2 AS FLAG
24 000032    005301    LOOP:  DEC    R1          ; MASK OFF BITS NOT NEEDED
25 000034    100406    BMI    EXIT      ; DECREMENT SHIFT COUNTER
26 000036    005702    TST    R2          ; AND GET OUT IF DONE
27 000040    001002    BNE    RIGHT     ; TEST FLAG TO SHIFT LEFT
28 000042    006100    RDL    R0          ; OR RIGHT, THEN SHIFT
29 000044    000772    BR    LOOP        ; (TST ALSO CLEARS THE
30 000046    006000    RIGHT:  ROR    R0          ; CARRY BIT TO TURN THE
31 000050    000770    BR    LOOP        ; ROTATES INTO SHIFTS)
32 000052    012602    EXIT:  MOV    (SP)+,R2   ; RESTORE
33 000054    012601    MOV    (SP)+,R1     ; REGISTERS
34 000056    000207    RTS    PC
35 000000    .END     ISHFT
    
```