Training Platform for the Design of a Windows Multimedia Device

G. P. O' Donnell, B.Sc.

Master of Engineering

Dublin City University

Dr. E. R. Lynch

DIT, Kevin Street

May 1995

I hereby declare that the material contained in this thesis is based on worked carried out by myself.

Signed : Gerard O'Donnell

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Engineering is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Gerard O'Donnell Date: 1-5-95

Acknowledgements

I would first like to thank my family for their support and assistance in allowing to me to complete this project and Dr. Raymond Lynch, my supervisor for his advice, assistance and supervision.

I would also like to thank the following people for their help and advice on various aspects of the project :-

The post-graduate students on the fourth floor, Jonathan, Tony, Brian, Simon, Terry, Mick and Joe.

Dr. C. Downing and P. Comiskey.

The technicians on the fourth floor Ronan, Dermot, Des, Mick, Ron and Eamon.

Mr. C. Cowley, Head of the Telecommunications and Electronics Department and the assistant head Mr. C. Bruce.

The DIT Strategic Research Development programme (SRD-2) which provided the funding for this project.

Table of Contents

Chapter 1

Introduction to the Multimedia Teaching Platform

Chapter 2

Design of the MTP's Sound Card

Chapter 3

Hardware Description of the MTP's Sound Card

Chapter 4

Introduction to Windows C Programming

Chapter 5

The MTP's Windows Application

Chapter 6

The MTP's Waveform Device Driver

Chapter 7

Debugging the Preset Faults of the MTP

Chapter 8

Summary and Future Development of the MTP

Appendix A

Appendix B

Appendix C

Appendix D

Appendix E

Appendix F

Appendix G

Appendix H

Appendix I

Appendix J

List of Figures

List of Tables

List of Listings

List of Equations

Abstract

There are many Windows Multimedia plug-in cards available for a Personal Computer (PC), but they are not suitable for a laboratory teaching platform for several reasons. Firstly, their hardware and software details are not available because of the market driven need to keep all hardware and software details from competitors. Secondly, the cards are not designed to allow faults to be introduced. Thirdly there is the inherent requirement that Windows applications be uniform to the point where application software sees the same interface, irrespective of which Windows compatible card is being used. These latter points are highly desirable from the user's point of view but not from a teaching viewpoint, where the goal is to enlighten the student in the hardware and software design techniques used to perform the stated objective.

The Multimedia Teaching Platform consists of a sound card, Windows 3.1 application and a Windows standard mode device driver. The sound card can continuously play or record audio files to the PC's hard disc in an analogue or a digital format. The digital format conforms to the consumer digital formats, IEC-958 Consumer, S/PDIF and CP-340 Type 2. Programmable logic was used on the sound card to allow hardware faults to be easily introduced. Hardware faults can be introduced by replacing the memory device which programs the logic array. Software design faults can be introduced by providing faulty source code for the device drivers and for the user interface. By introducing both hardware and software design faults, students can gain valuable experience in software and hardware debugging techniques and in the Windows environment.

Chapter 1

Introduction to the Multimedia Teaching Platform

1.1 Introduction

The purpose of this project was to develop a Multimedia Teaching Platform for a Microsoft Windows environment. In the era of VLSI and modular software where so much information is concealed behind a hardware or software interface, it is important for the engineering student to gain first-hand experience of hardware and software design and debugging techniques. Part of the philosophy of this project therefore, was to design a platform which puts the student in the position of having to detect and resolve software and hardware faults introduced to highlight the different aspects of Windows Multimedia.

This chapter provides an introduction to the Multimedia Teaching Platform and explains its three components, the Windows application, the device driver and the sound card. The structure of Microsoft Windows and Windows Multimedia are briefly explained along with the Windows Multimedia interface between the Windows application and the device driver.

1.2 Multimedia Teaching Platform

The development of the Multimedia Teaching Platform *(MTP)* consisted of writing a Windows application and device driver along with designing, building and testing a sound card. The MTP system is illustrated in Figure 1.1. The MTP system requires an analogue and digital audio amplifier to provide the source and target for the sound card's audio data. The digital audio interface supports the S/PIDF and IEC-958 consumer digital audio format which is explained in Appendix A. The Windows application controls the sound card through the device driver. The device driver directly accesses the sound card and handles the transfer of data between the sound card and the Windows application.

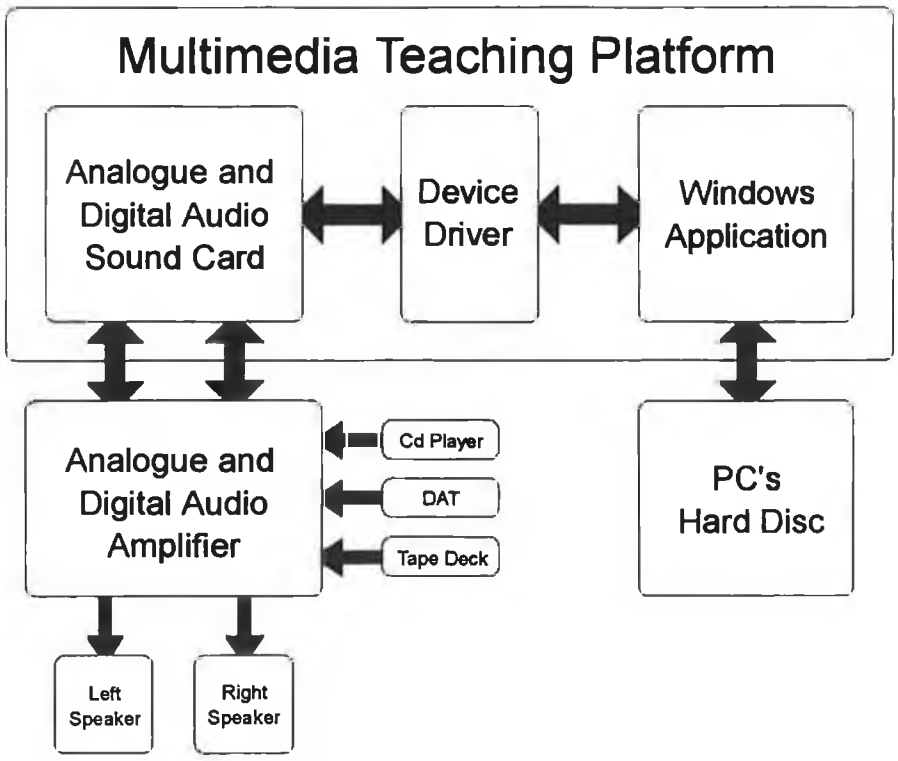


Figure 1.1 *Overview of the Multimedia Teaching Platform.*

Two designs were considered for the sound card. The first design used a Field Programmable Gate Array (FPGA) to implement the functional blocks of the sound card [16], such as Address Decoding and Clock Generation for the digital and analogue interface chips. The second used a mixture of combinational logic and a microprocessor to control the data flow through the card. This second design was discarded for its complexity when compared with the FPGA design. Furthermore the microprocessor design would have required another memory device along with a FPGA (for its combinational logic) and its memory device. The microprocessor would provide the MTP with the ability to introduce hardware faults through software by reprogramming the microprocessor but these faults would not be suitable because they could not be easily debugged and they would substantially increase the knowledge base required by the student for which the MTP is designed.

There are commercially available sound cards but their hardware schematic diagrams and their Windows application source code are not released to the public. Also, even if the sound card's schematic diagrams were available, the sound card itself uses proprietary VLSI devices and is not designed as a teaching platform. With the MTP, the user has access to the source code for its Windows application and device driver along with the schematic diagrams for its sound card.

The sound card utilises programmable logic to implement a significant part of its hardware design [16]. The hardware faults can be easily introduced into the MTP by replacing the Programmable Read Only Memory (*PROM*) device without altering any physical links or connections. This *PROM* programs the FPGA which is responsible for implementing the sound card's Address Decoding, Control Logic and Clock Generation and therefore provides a large source of possible faults. Oscilloscopes and logic analysers can be connected to test points on the card as described in the card's schematic diagrams in Appendix B. These tests points allow the student to concentrate on debugging the card rather than identifying devices and tracing connections between them. The software faults are introduced by providing faulty source code for the Windows application and device driver.

The sound card was chosen as the multimedia device because its hardware is simpler to design and understand than a video card and the device driver is easier to understand and debug. The video standard is still evolving with a suitable compression scheme yet to be accepted and, at present, sound is the most common form of Windows Multimedia in use.

1.3 MTP's Sound Card

The sound card is capable of recording and playing stereophonic analogue and digital consumer audio signals. The analogue source and target is a normal analogue audio amplifier with its associated source equipment while the digital source and target are any digital audio devices which provide digital consumer format serial outputs and inputs. For consistency the analogue audio system possesses the same sampling frequency and bit resolution as the consumer digital audio format.

Figure 1.2 shows the functional block layout of the MTP's sound card. The Swinging Buffers section consists of a pair of data buffers which maintain continuous recording or playback. The device driver can access one Swinging Buffer while the Serial Interface accesses the other. This ensures a continuous data flow between the external audio device and the Personal Computer's (PC) hard disk. The DMA \ Interrupt section generates the two interrupts which trigger the transfer of data between the Swinging Buffers and the PC's memory. The interrupt service routines *(ISRs)* for the two interrupts are contained in the device driver and they transfer data using direct memory access *(DMA)*. For more information on DMA and interrupts see Appendix C.

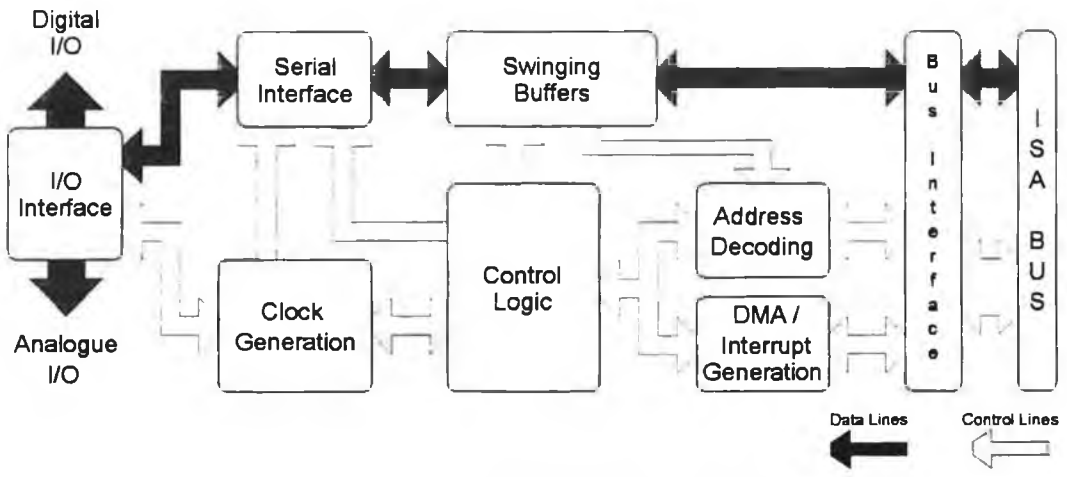


***Figure 1.2** Layout of the MTP's Sound Card.*

The Clock Generation section provides the different clocks for the I/O Interface, Serial Interface and the Control Logic sections. The Control Logic section controls access to the Swinging Buffers.

1.4 Microsoft Windows 3.1

Microsoft Windows is a graphical user interface for DOS based PCs [14]. Windows provides a multi-application environment where applications are represented by small graphical images or icons. Applications must be specifically written for Windows to take full advantage of its multi-application environment. Windows is written in C++ and runs on top of DOS which it requires for low level file access [3, 14]. The advantages of Windows can be summarised as follows;

- **Consistent User Interface**

  All applications have the same appearance and user interface principles such as pull-down menus and dialog boxes. For example, all word processing packages possess a *File* menu that contains the standard *Open* and *Save As* dialog boxes.

- **Data Transfer and Application Communications**

  There are several standard ways in which applications can communicate and interchange data. Applications can send messages to other applications and data can be transferred through the Clipboard, by Dynamic Data Exchange (*DDE*) or by Object Linking and Embedding (*OLE*).

- **Application Programming Interface**

  The Application Programming Interface *(API)* provides the functions required by a Windows application to operate in the Windows environment. The API also provides hardware independence for applications accessing the PC's hardware. These functions range from creating and displaying the application's window to the multimedia functions used to control the sound card. The multimedia devices supported by Windows are accessed through the API's multimedia functions contained in the MMSYSTEM module. These functions send standard device specific messages to the device drivers which in turn control their multimedia hardware device.

1.5 Windows Multimedia

Windows Multimedia provides the user with the ability to control media devices, audio and animation resources, audio and visual peripherals and external devices such as videodisc players, through API commands. The Windows module MMSYSTEM contains the multimedia functions of the Windows API [10]. Figure 1.3 illustrates the API

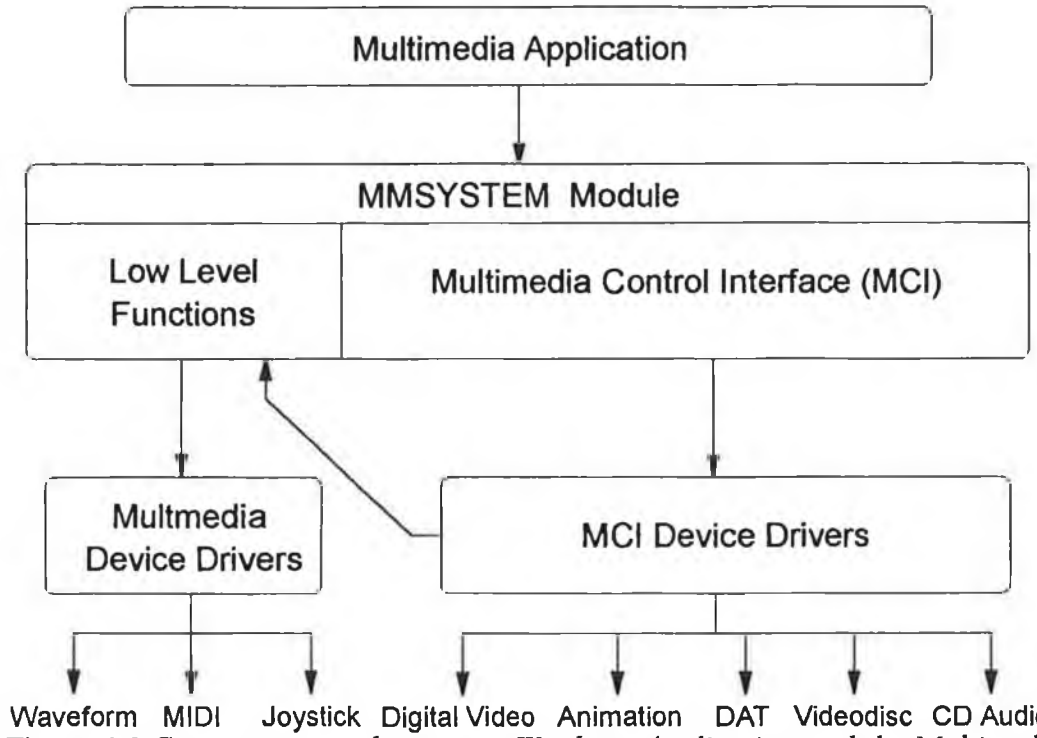


Figure 1.3 *Communication between a Windows Application and the Multimedia Device Drivers.*

multimedia interface between the Windows application and the multimedia device drivers. The MMSYSTEM module is composed of the Multimedia Control Interface *(MCI)*, and the low level functions for controlling the multimedia devices. MCI provides single high level commands for performing complete multimedia operations, such as recording an audio file. The low level functions provide direct control over the multimedia device but are more complex, requiring several steps to perform the equivalent high level command. For example playing an audio file requires at least five different low level functions compared to the MCI's single command.

The high level commands while advantageous from a programming viewpoint, hide much of the inner workings of Windows Multimedia from the programmer. For example they do not illustrate how the device driver and application communicate with each other, nor do they explain the process of playing or recording waveform audio. Consequently, the Windows application developed in this project uses the low level audio functions. The MMSYSTEM module converts these low level audio functions to waveform device driver messages which the waveform device driver processes and converts into hardware dependant code [12]. Similarly the waveform device driver communicates with the application via the MMSYSTEM module.

1.6 MTP's Windows Application

The Windows application in this thesis allows the user to record or play analogue or digital audio. The application uses the low level audio functions to record and play the analogue audio. The procedure for recording and playing back files is explained in Appendix D. The digital format is not supported in Windows Multimedia at present, but has been implemented in this project by sending special predefined messages through MMSYSTEM to instruct the waveform device driver to configure the card for digital operation. The files played and recorded are in the Windows *RIFF WAVE* format [9]. The 44.1 kHz 16 bit stereo PCM *WAVE* format, the most complex format was chosen as the one which the sound card should support. The application is written in C rather than C++, the object orientated version of C, to reduce the knowledge base requirements of the student, therefore making the MTP more accessible.

1.7 Windows Device Drivers

In any computer, device drivers provide the communication interfaces between the operating system and the hardware devices. Hardware device drivers provide hardware independence and standard software interfaces for programs which access the hardware. They relieve the programmer from needing to know any specific hardware details about the device.

In Windows, device drivers can be software or hardware oriented. For instance the driver that controls the graphics card is hardware based while the screen savers are software based. There are two types of Windows device drivers, Standard mode and 386 Enhanced mode drivers [12]. Standard mode drivers are used in the three modes of Windows. The Windows operating modes are explained in Appendix E. Enhanced mode drivers are only used in the 386 Enhanced mode and are known as virtual device drivers *(VxD)*. These VxDs make their hardware appear virtual allowing the device to be shared among many applications while maintaining the illusion to each application that it has sole ownership of the device. The Standard mode driver only allows single ownership of the hardware device it supports.

1.8 The MTP Device Driver

The device driver developed in this project is a Windows Standard mode compatable device driver. This driver replaces the waveform device driver that is responsible for audio data, if one is present in the Windows environment. Waveform audio is the Windows term for digitised analogue audio data. The device driver however, has more features than a standard waveform device driver because it also supports the consumer digital format along with an analogue format. The sound card itself can only support single ownership, so the advantages of a VxD would not be utilised.

The driver is written to accept special MMSYSTEM messages to convert the input or output medium to the analogue or digital format. The driver interprets the MMSYSTEM messages and controls the card accordingly. Two 4 kbyte ping-pong memory buffers are used by the driver to maintain data flow between the driver and the sound card. The driver is responsible for servicing the sound card's interrupts and managing the DMA transfers between the sound card and the DMA buffers. The device driver is written in a mixture of 80286 assembler language and C.

1.9 Layout of Thesis

The following chapters of the thesis can be described as follows. Chapter 2 introduces the MTP's sound card and describes the decisions taken when designing the card. The functional blocks of the card are described in detail in Chapter 3 along with the testing procedures used when verifying the card's design. Chapter 4 introduces the principles of C programming in the Windows environment and describes the files required to create a Windows application. The MTP's Windows application is described in Chapter 5 while the MTP's device driver is described in Chapter 6. Chapter 7 examines the preset faults of the MTP and describes the general debugging procedure using the recording and playback flowcharts. Chapter 8 summarises the MTP and examines areas where further development might take place.

Chapter 2

Design of the MTP's Sound Card

2.1 Introduction

This chapter introduces the Multimedia Teaching Platform's sound card and describes the design process of the card. A general description of how the card operates in the PC environment and the techniques it employs to transfer data are discussed. The hardware structure of the card is briefly discussed along with a description of the process of recording and playing audio files from the card's perspective.

2.2 Design Criteria

The design of the sound card was based on two requirements, a capability for continuous recording and playback and the ability to easily introduce hardware faults. The first involved a consideration of the PC's hard disk performance and microprocessor speed, the data transfer rate required by the analogue and digital interface devices and the environment in which the sound card operates. The second involved a method of easily modifying the sound card's logic circuits with minimal disturbance to physical connections. These design requirements are expanded in the following sections.

2.2.1 Continuous Recording and Playback

The sound card must be capable of maintaining continuous data flow to and from the PC's hard disk in the Windows and DOS environments. The PC can actually transfer data between its memory and the sound card at a faster rate than that required by the analogue or digital devices, namely at 1.4112 Megabits per second (Mbit/s) [6]. But many older hard disks are unable to sustain this data transfer rate and therefore some form of buffering on the sound card is required. For this reason two swinging buffers were included on the sound card. The PC's hard disk performance and microprocessor speed directly determine the buffer's minimum size requirements.

The operating environment of the card must also be considered when calculating the buffer size. The Windows environment dictates the use of interrupts and Direct

Memory Access (DMA) to transfer data between the PC's memory and the buffers. The microprocessor in the Windows environment usually has several applications running at once, and therefore it has more housekeeping tasks to perform than in a single application environment, such as DOS.

DMA combined with interrupts allows data to be transferred between the PC's memory and the sound card's Swinging Buffers with minimal direct microprocessor involvement. The microprocessor is interrupted only when DMA transfer sessions need to be initialised, and the transfers are then performed by the DMA controller while the microprocessor is free to perform other tasks. These DMA sessions take more than twice the time of their direct microprocessor controlled I/O counterparts, because a DMA I/O transfer cycle is longer and the DMA controller is configured to release the bus after every DMA operation [17]. Otherwise the microprocessor would be denied the bus until the DMA session finishes.

The paging hardware of the Intel microprocessors, when operating in protected mode, only guarantees 4 kbytes of contiguous physical memory [5]. This places a maximum limit of 4 kbytes on the size of DMA sessions. The size of the Swinging buffers will therefore be a multiple of 4 k. Consequently some method had to found to manage the multiple DMA sessions required to empty or fill the buffers. This was achieved by using two sources of interrupts, namely the Switching Interrupt, generated when the hardware switches from one Swinging Buffer to the other, and the DMA Interrupt generated when a DMA session on the sound card's DMA channel has finished. The Switching interrupt will trigger the initial access to the Swinging Buffers while the DMA Interrupt, generated at the end of a DMA session, manages further accesses to the buffer. In subsequent sections this transfer method will be referred to as the DMA / Interrupt transfer mode.

In the DOS environment where the microprocessor can devote most of its time to the application, microprocessor I/O read and write operations can be used to transfer data to and from the sound card. The application can poll the sound card by reading the Buffer Status Register, and determine when data need to be transferred. This transfer mode, called Polled I/O, is faster than the DMA / Interrupt mode but cannot be used in the Windows environment. At power-up the sound card is automatically configured for Polled I/O by its power-up circuitry. The DMA / Interrupt mode is enabled (and the

Polled I/O mode simultaneously disabled) by writing to specific I/O port addresses, as described in Section 2.5.

2.2.2 Introduction of Hardware Faults

For the card to be practical in a learning laboratory environment, the hardware faults must be introduced as easily as possible, without having to manually modify any physical connections. This criterion was meet by using programmable logic to implement the sound card's logic. A Field Programmable Gate Array *(FPGA)* implements the sound card's Address Decoding, Control Logic and Clock Generation sections. This *FPGA* is programmed at start-up using a Programmable Read Only Memory device *(PROM)* [16]. A hardware fault can be introduced by simply replacing the *PROM* with one which programs the sound card with a predefined fault.

2.3 Data Buffering in the MTP

The MTP was designed to operate in two different data transfer modes, Polled I/O and DMA / Interrupt in both the DOS and Windows environments. Regardless of which environment or transfer mode was used, some form of buffering is required on the sound card because the PC cannot maintain the continuous data transfer rate (one sample every 11.34 μs) required by the I/O interface of the sound card. This buffering can be in the form of a single or double (swinging) buffer arrangement as shown in Figure 2.1. There can also be single or double buffers in PC memory to further compensate for the slow data transfer rate between the hard disk and memory.

The Double Buffering System can be implemented with two first-in-first-out (FIFO) buffers [7]. A FIFO buffer allows independent access to the buffer from two separate channels. This allows simultaneous read and write requests to be processed by the FIFO's internal logic. The FIFO possesses separate internal read and write count registers which ensures that only valid locations are accessed when reading or writing to the FIFO. This system is organised so that as one buffer is being accessed by the PC the other is accessed by the I/O Interface of the sound card.

The Single Buffer System can be implemented with a single FIFO buffer. This system operates in a similar manner to the Double Buffering System in that one half of the FIFO is being accessed by the PC while the sound card's I/O Interface accesses the other half. However this system is more difficult to control than a discrete (two FIFO)

double buffering system, requiring more logic and a larger FIFO than the double buffering system. For this reason a Double Buffering System was chosen for the MTP design.

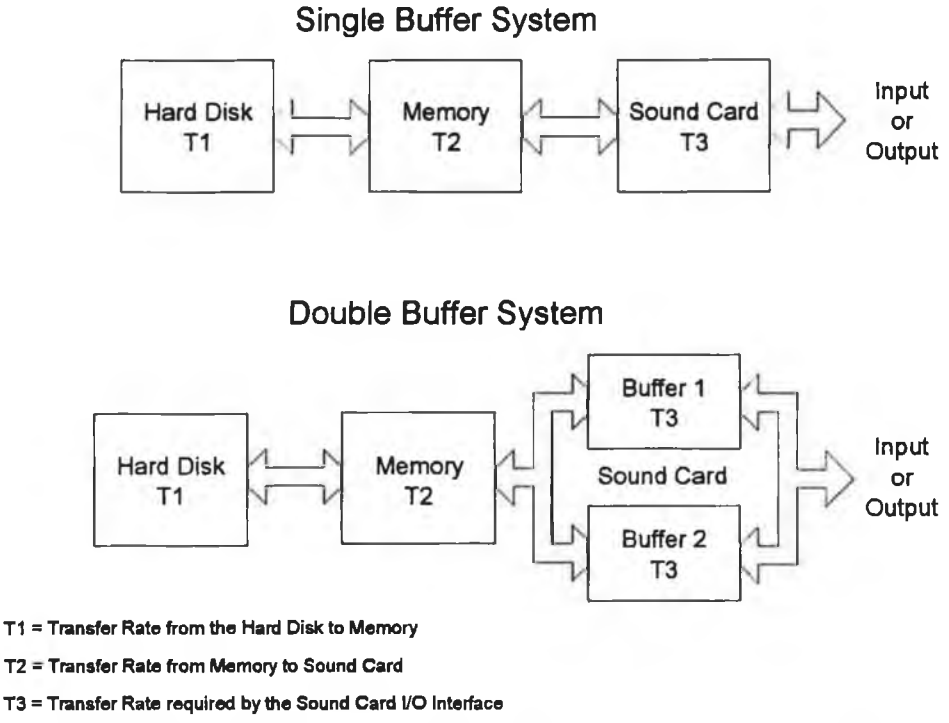


***Figure 2.1** Single and Double Buffering on the Sound Card.*

Just as double buffering can be employed on the sound card, a similar arrangement can be organised for PC memory when operating in the DMA / Interrupt transfer mode. This eases the timing demands for data transfer between the hard disk and PC memory. While the DMA controller is transferring data between the sound card and PC memory, the processor can be transferring between the hard disk and memory.

However, the Polled I/O mode does not benefit from swinging buffers in PC memory because it requires the processor to directly transfer data between memory and the sound card's buffers. For example, when recording, the processor must transfer data between the sound card's full swinging buffer and the empty PC swinging buffer and then between the full PC swinging buffer and the hard disk before the other buffer on the sound card has been filled. The same timing restrictions are imposed if only one PC memory buffer is used. For this reason the size of the sound card's buffers was calculated based on the Polled I/O mode and assuming a single memory buffer.

2.3.1 Size of the Sound Cards Swinging Buffers

The approach taken in determining the size of the Swinging Buffers was based on first finding a minimum buffer size for an unfragmented hard disk and then testing larger buffer sizes on a fragmented disk to ascertain if continuous recording or playback can be sustained. This approach was taken because of the difficulty in accurately determining the access time of a fragmented hard disk.

Referring back to Figure 2.1 the timing restrictions imposed on the double buffering system in the playback mode are that for the chosen buffer size X (in words), the time to read from the hard disk and fill the empty buffer must be less than the time for the I/O Interface to empty the other buffer. If this is achieved then continuous recording or playback can be maintained.

2.3.1.1 Physical Organisation of the Hard Disk

The hard disk used in the test machine was composed of five platters or disks stacked on top of each other as shown in Figure 2.2. Each platter possessed two recording surfaces and therefore two heads. The capacity of each platter depends on the number of tracks on it. Each track is made up of sectors which contain a fixed number of bytes. All tracks contain the same number of sectors regardless of their position on the platter.

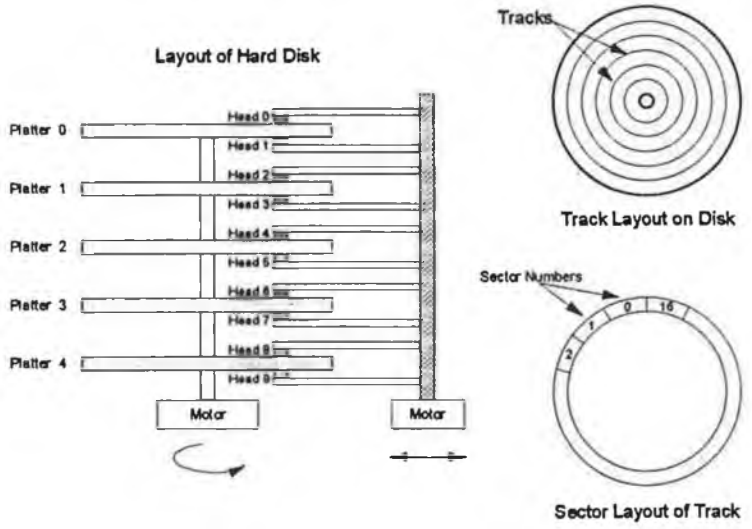


Figure 2.2 Layout of Hard Disk.

This particular hard disk possessed 512 bytes per sector and 17 sectors per track. This 80 Mbyte hard disk contains approximately 160,000 sectors which for convenience are grouped together to form clusters.

Using the physical organisation of the hard disk described above, the time to read from an unfragmented hard disk[1] and fill a memory buffer *(T1)* can be calculated as follows;

- The time for the head to move to the first sector of the file is taken as the average seek time of 31.5 ms.
- There is another transition time when the head moves to the next adjacent track after all the file's sectors have been read from this track. This movement takes 5.18 ms and the number of times this occurs when filling a buffer is given by the ratio of the buffer size X, to the number of words per track (17 sectors x 256 words per sector).
- The transfer time to fill the buffer is the buffer size multiplied by the transfer rate of the disk, 4.56 µs.

The time for the I/O Interface to empty a Swinging Buffer is X x 11.34 µs *(T3)*. The PC can perform an I/O write instruction every 1 µs, so a buffer can be filled in X µs *(T2)*. Therefore, for continuous operations the following condition must be satisfied;

$$T1 + T2 \leq T3$$

or

$$(31.5ms + 5.18ms * \frac{X}{17 * 256} + 4.56\mu s * X) + (1\mu s * X) \leq 11.34\mu s * X$$

***Equation 2.1** Minimum Buffer Size.*

Equation 2.1 yields a minimum buffer size of 6864 words. Tests were performed with 8 kword buffers which were found to be unreliable for a fragmented disk. Therefore 16 kword buffers were used which were found to be able to sustain continuous operations on a fragmented hard disk. The Swinging buffers are composed of two 8 k x 9 bit IDT7206 first in first out buffers *(FIFOs)* with 50 ns access time. The 50 ns access time was required to compensate for the propagation delays introduced by the Address Decoding and Control Logic circuits when reading data from the buffers.

[1]*Performance figures from Norton Utilities Ver. 3.0*

2.3.2 Data Buffering in the Windows Environment

There are three levels of data buffering employed by the MTP to maintain continuous recording or playback when operating in the Windows environment using the DMA / Interrupt transfer mode, as shown in Figure 2.3. The first two are implemented by the Swinging Buffers on the sound card and the ping-pong buffers in the device driver. These are required due to the inability of the hard disk to continuously transfer data to the I/O Interface at the required transfer rate (1.4112 Mbits/s), and the maximum number of DMA transfers allowed per DMA session. If the PC and hard disk were fast enough to compensate for the extra processing involved in the Windows environment, then there would be no need for any DMA buffering and the Swinging Buffers on the sound card would only be required for lossless data transfer.

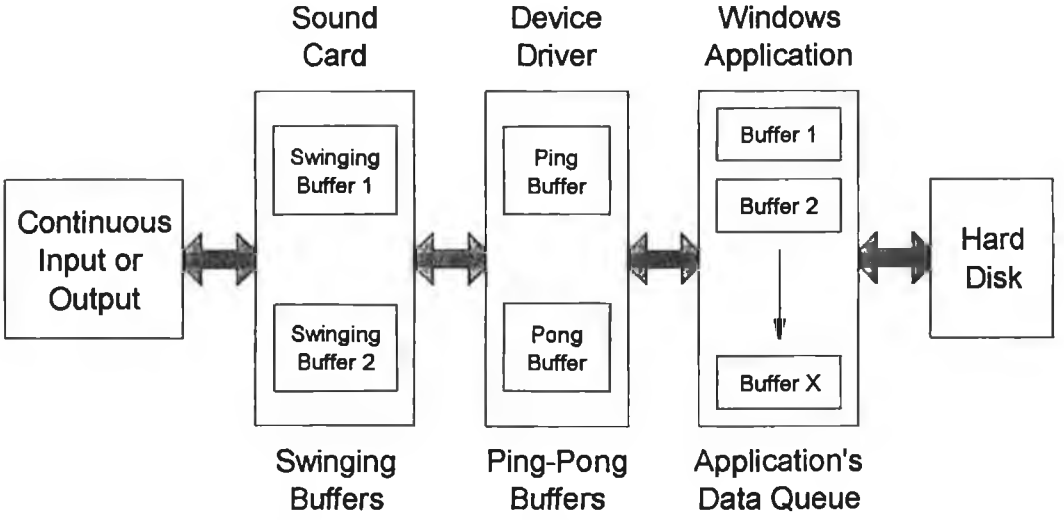


***Figure 2.3** Data Buffering used by the MTP to Maintain Continuous Recording or Playback.*

The buffering between the application and the device driver's ping-pong buffers is imposed by the Windows environment. The device driver's ping-pong buffers require another level of buffering between themselves and the hard disk for two reasons. The first reason is due to the restriction on the size of the ping-ping buffers, 4 kbytes each as explained in Section 2.2.1. These buffers are therefore too small to maintain continuous data transfer between the hard disk and the I/O Interface. The second reason is due to the device driver operating at a higher priority level than Windows applications. The application can not directly access the device driver's buffers without causing a General Protection Fault in the Windows environment. This would immediately cause the Windows environment to halt and terminate the application. Therefore the device driver must transfer data between its ping-pong buffers and the data

buffers provided by the application. This further increases the transfer time between the hard disk and the Swinging Buffers. Windows requires the application to control disk access and provide the device driver with a sufficient number of data buffers so that it can maintain continuous data transfer to the I/O Interface through the Swinging Buffers [11].

2.4 Data Transfer between Sound Card and PC's Memory

The peripheral components involved in transferring data between the sound card and the PC's memory are shown in Figure 2.4. The Swinging Buffers are accessed through I/O ports and the card supports two forms of I/O data transfer, Polled I/O and DMA / Interrupt as explained previously. The Polled I/O transfer mode is performed by the microprocessor's I/O port read and write instructions. In this case, the microprocessor is responsible for generating the memory address, enabling data buffers, and memory and I/O port control signals, such as memory read (MEMRbar) and I/O port write (IOWbar).

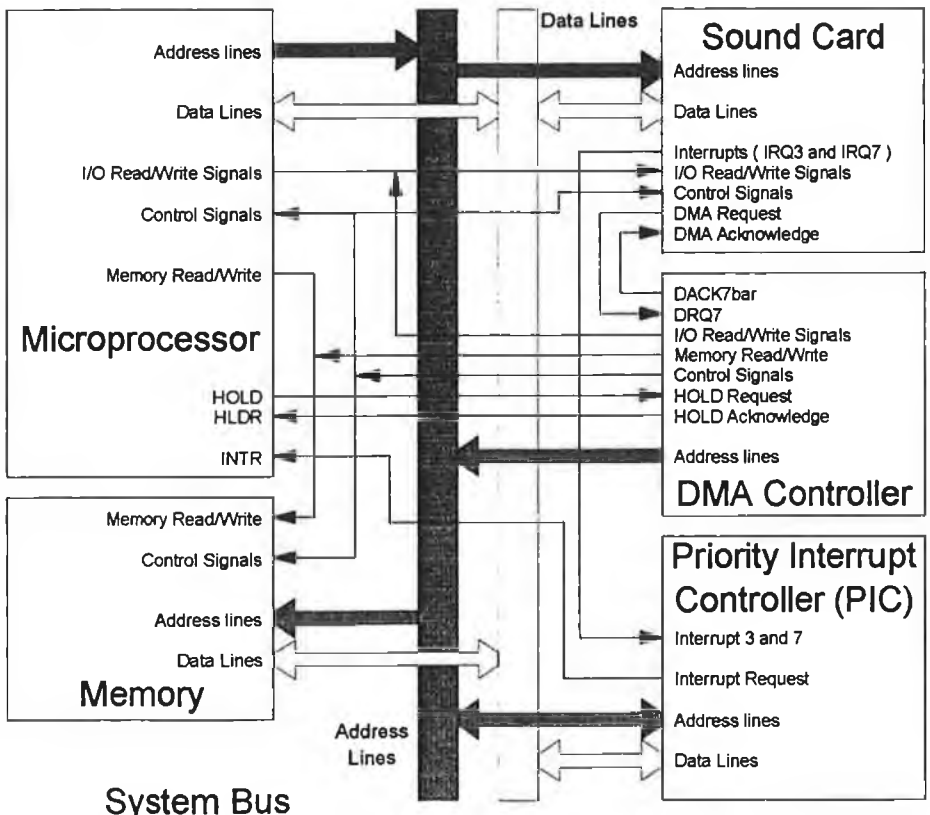


Figure 2.4 Block Diagram of System Bus and Peripheral Devices used by the Card.

The DMA / Interrupt mode is managed by the DMA controller. Once the DMA controller has control of the System Bus after a DMA request from the sound card, it will generate the signals required to transfer data between the sound card and the PC's memory. The DMA sessions are programmed by the sound card's interrupts, SwitchIRQ (IRQ3) and DMAIRQ (IRQ7). When one of these interrupt lines is pulsed low, it will be latched by the Priority Interrupt Controller (PIC). The PIC will then interrupt the microprocessor. The microprocessor will acquire from the PIC the interrupt vector of the recently activated interrupt line. This interrupt vector will point to the corresponding interrupt service routine. This interrupt service routine will then program the DMA controller. For information on programming the PIC, see Appendix C.

2.4.1 Programming the DMA Controller

The DMA controller is programmed by writing to its internal registers, accessed through I/O ports as described in Appendix C [8]. These define the number of transfers per session, the starting memory address for transfers and its operating mode, such as its type of transfer and its transfer mode. The MTP programs the DMA controller in the following configuration;

- Channel selected : Channel seven (third channel of the second DMA controller). The following configuration information refers to this channel only.
- Type of transfer : I/O port write to memory or I/O port read from memory.
- Increment Address : Current address is incremented after every transfer.
- Transfer Mode : Single mode which releases the System Bus after every transfer

Before the DMA controller is programmed, the channel being programmed must be masked by setting the appropriate bit in the DMA controller's mask register. The channel can then be programmed without DMA transfers occurring while the channel is being configured. Once programmed, the channel can be unmasked and DMA requests can now generate DMA transfers.

2.4.2 DMA Transfer I/O Cycles

The sound card requests DMA transfers on the DMA request line for channel seven, DRQ7. This signal is active high and once received by the DMA controller, the DMA controller will request control of the system bus from the microprocessor by asserting its HOLD signal. The microprocessor when it has finished its present instruction will

release the system bus by asserting its hold acknowledge signal, HLDA. Once the DMA controller receives this signal it will start a DMA transfer cycle as programmed for channel seven. The memory address generated by the DMA controller is determined by its internal page and current address registers. The DMA controller will generate the signals required to transfer the 16 bit word between the sound card and the PC's memory.

2.4.2.1 I/O Port to Memory Transfer Cycle

Figure 2.5 illustrates the DMA I/O port to memory transfer cycle [7]. The process of transferring data from the Swinging Buffers to the PC's memory can be summarised as follows:-

- DMA controller receives a DMA transfer request signal, DRQ7, on DMA channel seven from the sound card's DMA / Interrupt Generation section.
- DMA requests control of the system bus from the microprocessor.
- DMA controller is granted control of the system bus.
- DMA controller generates the memory address from its internal current address and page registers.

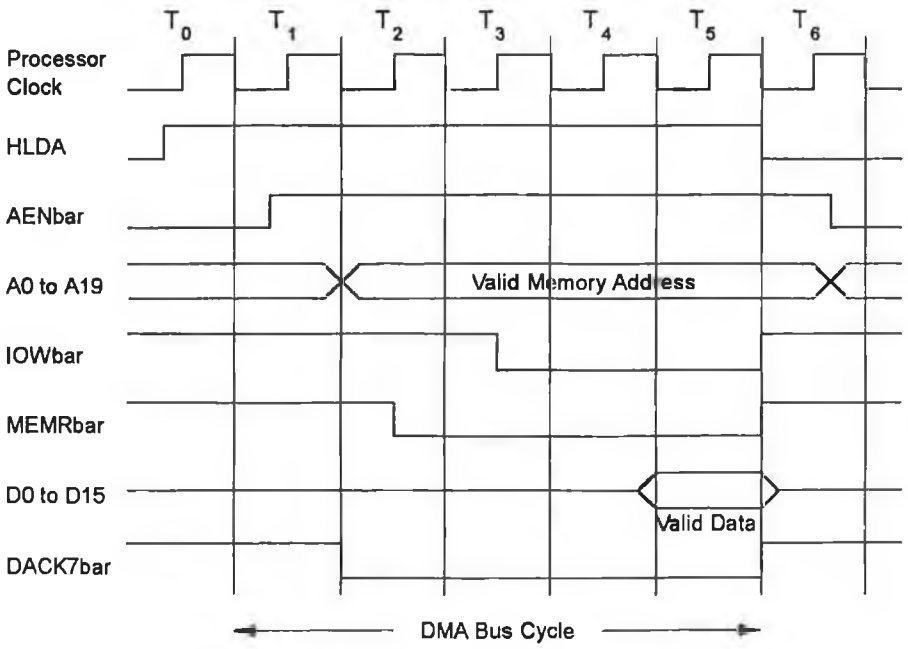


Figure 2.5 DMA transfer from an I/O Port to Memory.

- DMA controller generates its DMA transfer acknowledgement signal for channel seven, DACK7bar.
- DMA controller now generates the general I/O port read signal, IORbar. On the sound card the DMA / Interrupt Generation section generates a Swinging Buffer read request from a combination of this signal and the DACK7bar signal. The Control Logic will direct this read request signal to the first Swinging Buffer.
- Data from the Swinging Buffers drive the data lines on the system bus.
- DMA controller now generates the memory write signal MEMWbar, which latches the data on the system bus into the specified memory location.

2.4.2.2 Memory to I/O Port Transfer

Figure 2.6 illustrates the DMA memory to I/O port transfer cycle [7]. The process or transferring data from the PC's memory to the Swinging Buffers by the DMA controller can be summarised as follows:-

- DMA controller receives a DMA request signal, DRQ7, on DMA channel seven from the DMA / Interrupt Generation section.
- DMA controller requests control of the system bus from the microprocessor.

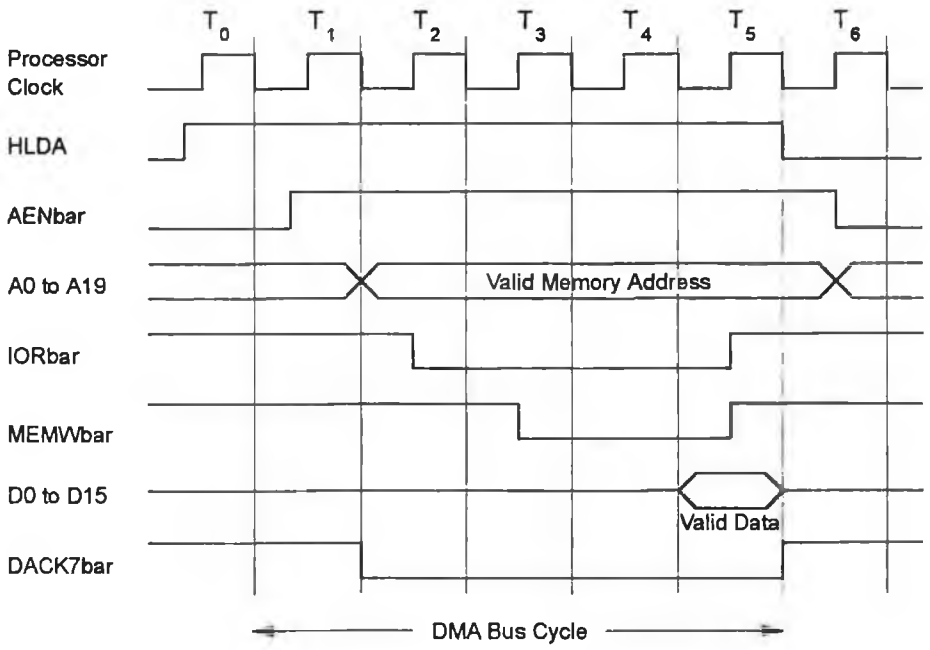


Figure 2.6 DMA transfer from Memory to an I/O Port.

- DMA controller is granted control of the system bus.
- The memory address generated by the DMA controller is determined by its current address and page registers.
- DMA controller issues a DMA transfer acknowledgement signal on channel seven, DACK7bar.
- DMA controller generates a memory read signal MEMRbar, which will place the memory location's data on the system bus.
- DMA controller now generates the I/O port write signal IOWbar. This signal in conjunction with the DACK7bar signal generates a Swinging Buffer write request. The Control Logic directs this request to the appropriate Swinging Buffer.
- The data on the system bus is now latched into the memory of the write enabled Swinging Buffer.

2.5 Capabilities of the Sound Card

The sound card is capable of recording and playing continuous audio files to and from the PC's hard disk. The audio can be in an analogue format or in the consumer digital audio format (see Appendix A). The digital and analogue formats are fully compatible. Their stored samples have the same properties in terms of sampling frequencies (44.1 kHz), sample resolutions (16 bit), and coding (PCM stereo, see Appendix F). Files recorded in one mode can also be played back in the other mode.

The digital consumer format is not supported in Windows Multimedia at present, but is supported in this card. The card can operate in the Windows or DOS environments and resides in a full length ISA slot on the system bus of the PC.

2.6 Sound Card Design

The sound card's design can be broken into several sections, each dedicated to performing a function required by the card during recording or playback. These functional blocks were illustrated previously in Chapter 1, Figure 1.2 and are now briefly described.

2.6.1 Swinging Buffers

This section is responsible for maintaining continuous data flow between the external device, amplifier or CD Player, and the sound card. The block diagram of this section is shown in Figure 2.7. The Swinging Buffers are composed of two 16 kword buffers which are in turn made up of two 9 bit first in first out *(FIFO)* buffers. The two FIFOs provide the 16 bit storage required to support the 16 bit I/O transfer operations used when transferring data between the PC's memory and the sound card. The size of the Swinging Buffers is such as to provide the PC with sufficient time to either fill or empty one Swinging Buffer while the I/O Interface is accessing the other Swinging Buffer, without breaking the continuous data flow between the Swinging Buffers and the I/O Interface. From the perspective of the CPU or the DMA controller the Swinging Buffers are seen as I/O ports. There are separate ports for reading and writing, while the same port can access both buffers. The Control Logic determines which buffer receives the request.

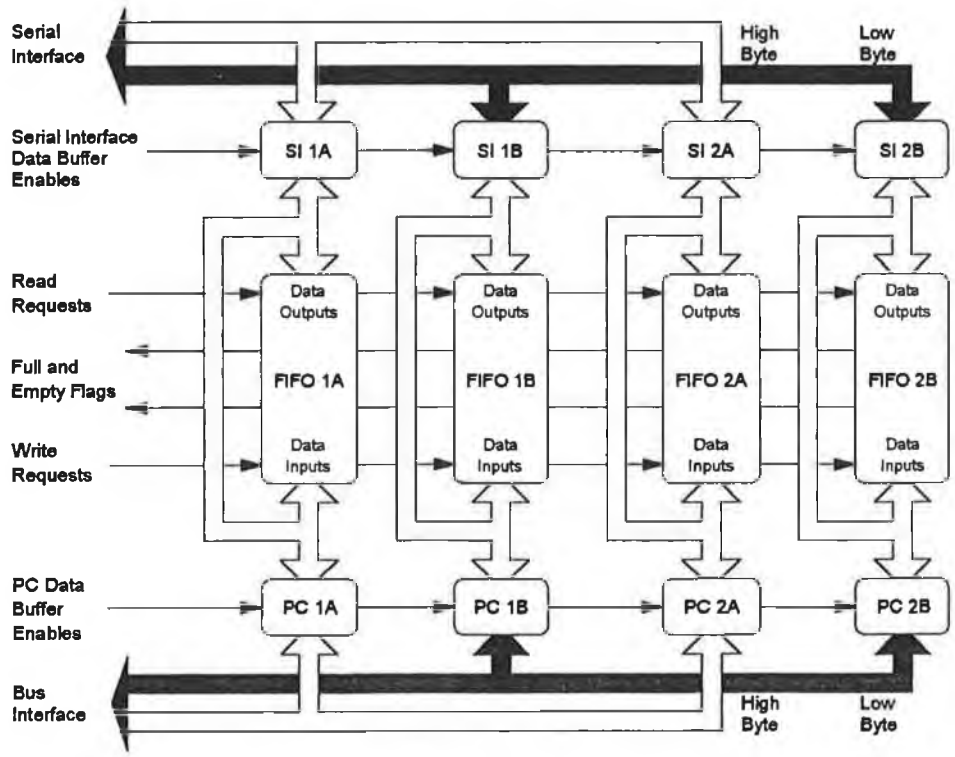


Figure 2.7 Block Diagram of the Swinging Buffer Section.

2.6.2 Address Decoding

The Address Decoding is responsible for configuring the card and for accessing the Swinging Buffers when operating in the Polled I/O transfer mode. A block diagram of the Address Decoding section and the System Bus Interface is shown in Figure 2.8. The address range of the sound card lies between 0300h to 0314h (*hexadecimal*) which is within the allowed Prototype Card range, 0300h to 031Fh. The I/O port addresses and their functions are listed in Table 2.1.

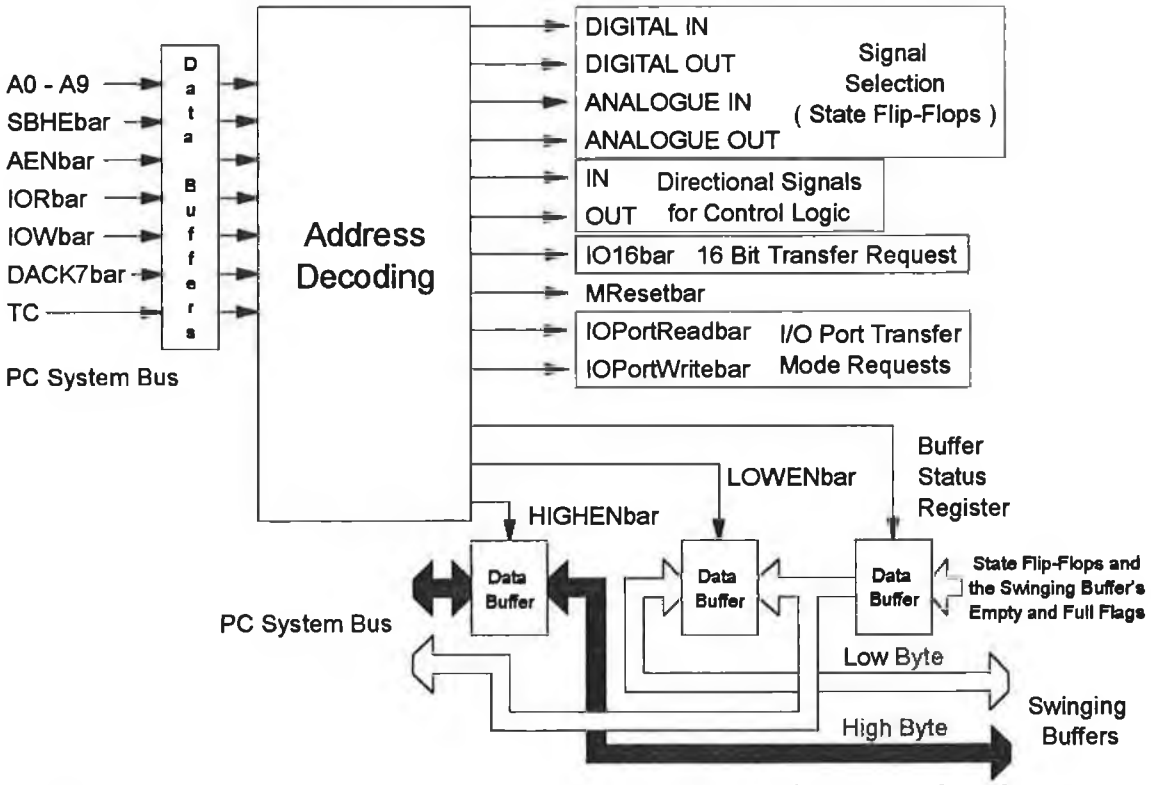


***Figure 2.8** Block Diagram of the Address Decoding and Bus Interface Sections.*

The card is reset by the Master Reset signal *(MResetbar)*. Each analogue and digital mode has a corresponding State flip-flop, DIGITAL IN, DIGITAL OUT, ANALOGUE IN and ANALOGUE OUT, which configure the card in the corresponding operating mode. Only one of these flip-flops should be in a set state at any one time, otherwise the card will not be configured correctly. At power-up they are placed in the DIGITAL IN mode by the power-up circuit as shown in Appendix B, Figure B.1. These flip-flops are accessible through their respective I/O port addresses as shown in Table 2.1. They are reset only through their I/O port addresses and not by MResetbar which resets the rest of the card's logic circuits. Therefore, their states must be known in order

to reset or set the appropriate flip-flops. These states can be determined from the Buffer Status Register, implemented as an I/O port in this section of the card.

Function	I/O Port Address (hex)
Master Reset (MResetbar)	300
Buffer Write (IOPortWritebar)	302
IRQ Enable	304
DIGITAL IN	306
DIGITAL OUT	308
ANALOGUE IN	30A
ANALOGUE OUT	30C
DMA Enable	30E
Buffer Status Register	310
Buffer Read (IOPortReadbar)	312

***Table 2.1** I/O Port Addresses.*

There are two other I/O ports implemented which allow access to the Swinging Buffers when using the Polled I/O transfer mode, Buffer Read and Buffer Write. Two mutually exclusive control signals, IN and OUT are also provided by this section for controlling the direction of data flow through the card. These signals are generated from the logical OR combination of the DIGITAL IN and ANALOGUE IN signals.

2.6.3 DMA / Interrupt Generation

The block diagram of the DMA / Interrupt Generation section is shown in Figure 2.9. This section is responsible for requesting DMA I/O transfer cycles, generating read and write requests for the Swinging Buffers from these DMA I/O transfer cycles and for generating the Switching and DMA interrupts, IRQ3 and IRQ7 respectively.

DMA cycles are requested through the DMA controller's request signal for the seventh DMA channel DRQ7. This signal is active high and is controlled by the Swinging Buffers' full or empty flags depending on the operating mode of the card.

The read and write requests, DMAIOPortReadbar for recording and DMAIOPortWritebar for playback are generated from the active low DMA acknowledge signal DACK7bar, and the IORbar or IOWbar signals. The DACK7bar signal will be accompanied by the appropriate IORbar or IOWbar depending on the DMA channel's configuration (I/O port read / DMA write to memory or I/O write / DMA read from memory) in the DMA controller.

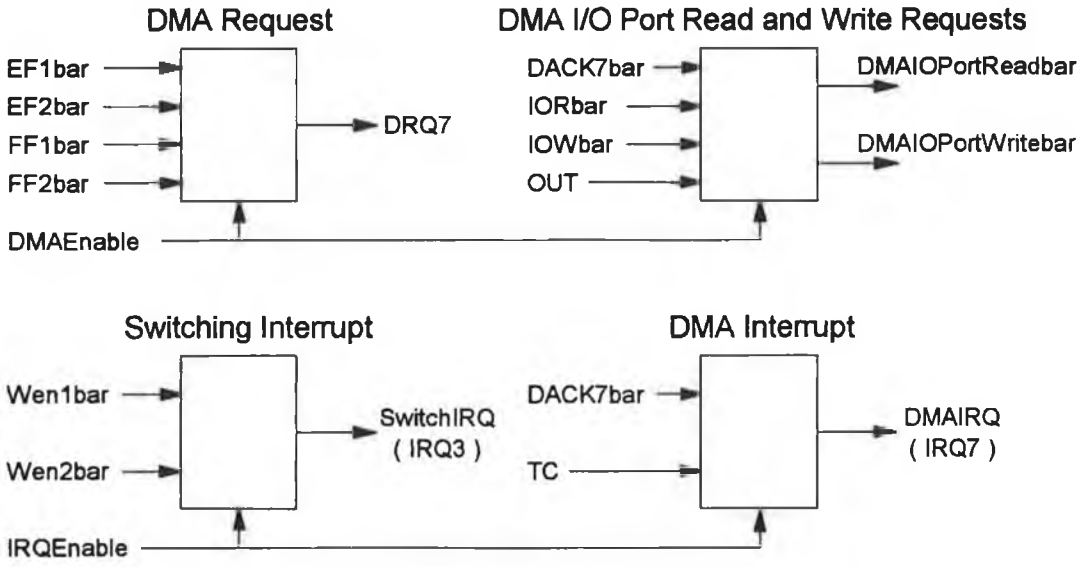


Figure 2.9 Block Diagram of the DMA / Interrupt Generation Section.

2.6.4 Control Logic

The Control Logic section controls access to the Swinging Buffers and its block diagram is shown in Figure 2.10. There are three sources of read and write requests for the Swinging Buffers. They are from the mutually exclusive Address Decoding section during Polled I/O mode and the DMA / Interrupt Generation section during DMA / Interrupt mode, and from the Serial Interface section. These requests are directed to the appropriate Swinging Buffer depending on the configuration of the card and the current state of the Swinging Buffers. Switching between the Swinging Buffers is performed by the WENbar signals. These active low signals determine which Swinging Buffer is write enabled. For example during recording, after one Swinging Buffer has been filled, the Control Logic switches these signals, write enabling the other buffer which is now filled.

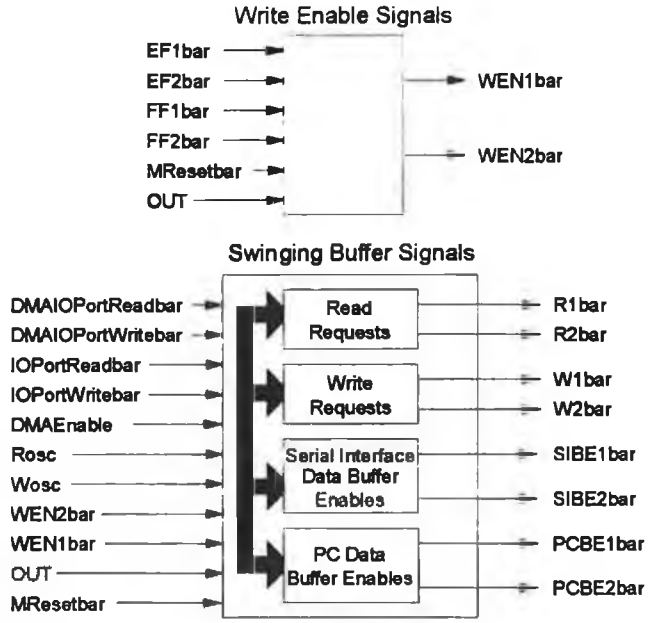


Figure 2.10 Block Diagram of the Control Logic Section.

2.6.5 I/O Interface

The I/O Interface section provides the analogue and digital interface for the sound card and its block diagram is shown in Figure 2.11. It is composed of a digital receiver *(DRX)* and digital transmitter *(DTX)* both configured for the consumer digital audio format together with a digital to analogue converter *(DAC)* and an analogue to digital converter *(ADC)* for the analogue recording and playback capabilities.

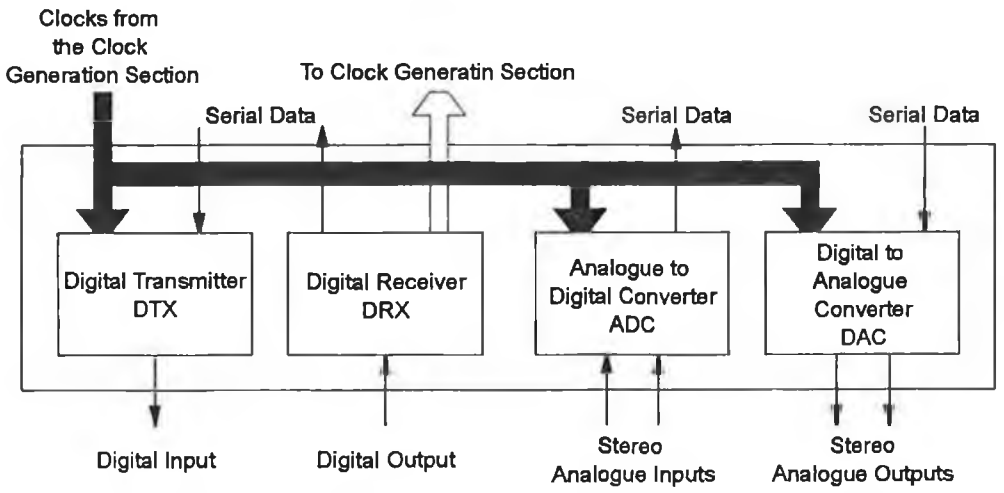


Figure 2.11 Block Diagram of the I/O Interface Section.

2.6.6 Serial Interface

The block diagram of the Serial Interface section is shown in Figure 2.12. This section converts between the parallel data of the Swinging Buffers and the serial data of the I/O Interface. The serial shift and parallel load clocks are provide by the Clock Generation section.

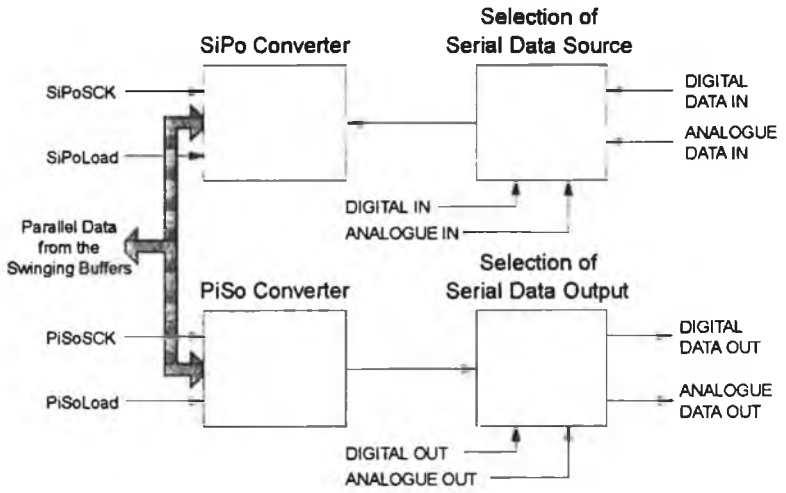


Figure 2.12 Block Diagram of the Serial Interface Section.

2.6.7 Clock Generation

The block diagram of the Clock Generation section is shown in Figure 2.13. The various clocks for the DTX, ADC, DAC and Serial Interface are generated here. This section also provides the read and write requests to transfer data between the Swinging Buffers and the I/O Interface. All clocks are derived from a master clock which is generated from either a crystal oscillator or by the DRX during digital recording.

2.6.8 Bus Interface

The Bus Interface section is composed of data buffers and transceiver for isolating the sound card from the PC's system bus and is included in Figure 2.8 along with the Address Decoding logic.

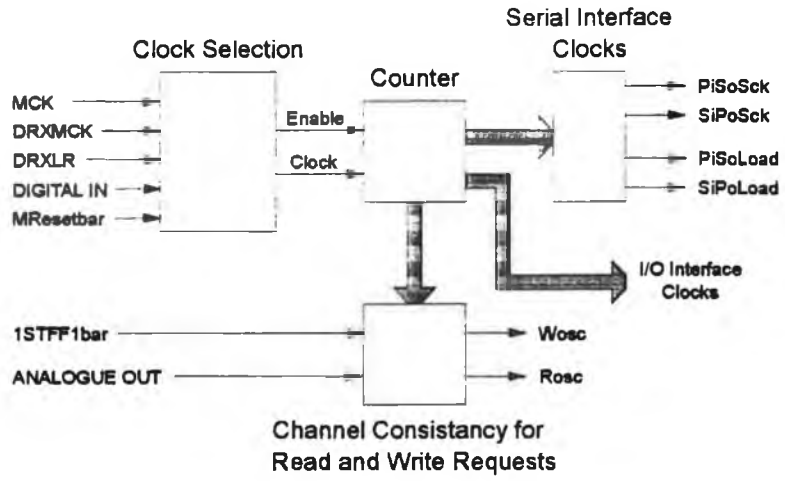


Figure 2.13 Block Diagram of the Clock Generation Section.

2.7 Operation of the Sound Card in the Playback and Recording modes

The sound card possesses four operating modes; digital recording (DIGITAL IN), digital playback (DIGITAL OUT), analogue recording (ANALOGUE IN) and analogue playback (ANALOGUE OUT). From the PC's viewpoint there is no difference between them once the card has been configured. The transfer of data between the hard disk and the Swinging Buffers is independent of the source or output format of the data. The main difference between these modes is in the area between the Swinging Buffers and the I/O Interface.

Figure 2.14 illustrates the data flow through the MTP when the DMA / Interrupt data transfer mode is used. At the user level, the Windows application sends data buffers to the Windows device driver to replay or fill with recorded data. The device driver uses DMA to transfer the data between the device driver and the Swinging Buffers. The Control Logic directs the read and write requests to the appropriate Swinging Buffers while also maintaining continuous data flow between the I/O Interface and the Swinging Buffers.

Two hardware interrupts are used by the device driver to synchronise filling or emptying of a Swinging Buffer. The Switching Interrupt indicates that a Swinging Buffer switch has occurred, which triggers a DMA session to begin emptying or filling a Swinging Buffer. Only 2 kwords are transferred during each DMA session. Each DMA interrupt, generated at the end of a DMA session, triggers a subsequent DMA session,

thereby eventually causing the Swinging Buffer to be completely filled or emptied. Along with the Switching interrupt, seven DMA interrupts are required to transfer the 16 kwords in a Swinging Buffer. A DMA interrupt count flag keeps track of the number of DMA interrupts occurring and the count flag is reset when a Swinging Buffer transfer is complete.

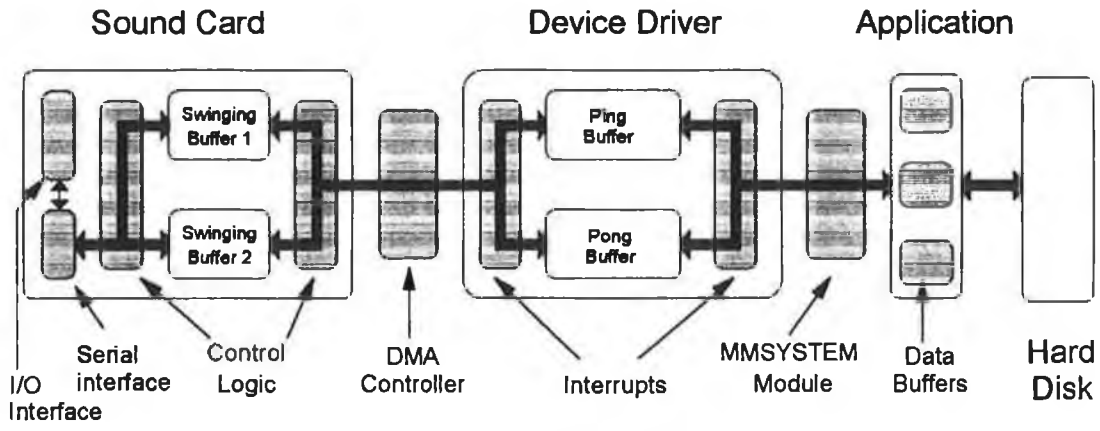


***Figure 2.14** Data flow through the Multimedia Teaching Platform.*

2.7.1 Digital Recording

The steps involved in the recording process (analogue or digital) are described in the flowchart of Figure 2.15. Looking just at the digital case, the first step is to configure the State flip-flops. This involves reading the Buffer Status Register and clearing the current State flip-flop and setting the DIGITAL IN State flip-flop. The MResetbar signal is now issued which resets the controlling logic for the FIFOs within the Control Logic section and also resets the flip-flops controlling the DMA and Interrupt enable signals.

Once the MResetbar signal has been issued, the Serial Interface's serial to parallel converter (SiPo) will convert the serial data from the DRX into the parallel data required by the Swinging Buffers. The Control Logic section will now pass the continuous write requests from the Clock Generation section through to the first Swinging Buffer, W1bar as shown in Figure 2.16. As the buffer fills the Control Logic also ensures that the Swinging Buffers never receive simultaneous read and write requests or that the same read or write request is received by both of the Swinging Buffers.

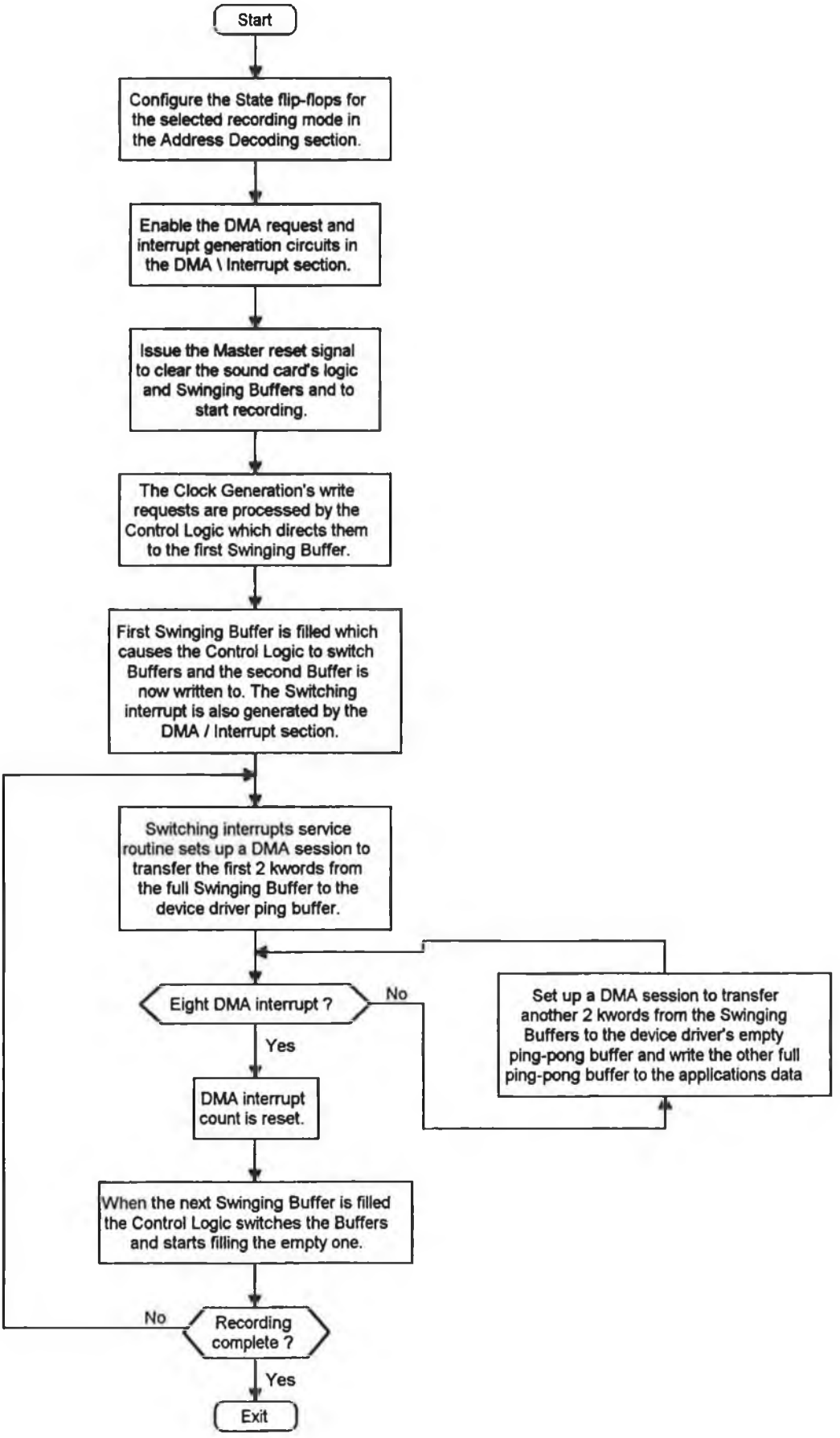


***Figure 2.15** The Recording Process.*

Once the first Swinging Buffer has been filled, the Control Logic triggered by the buffer's full flag going active, will switch the write request's destination from the first to the second buffer by toggling the WENbar signals. This switch will cause the DMA / Interrupt section to generate the Switching interrupt, SwitchIRQ. The resulting interrupt service routine, which resides in the device driver, initialises the DMA controller for a DMA session. This will read 2 kword from the card and write it to the device driver's ping memory buffer.

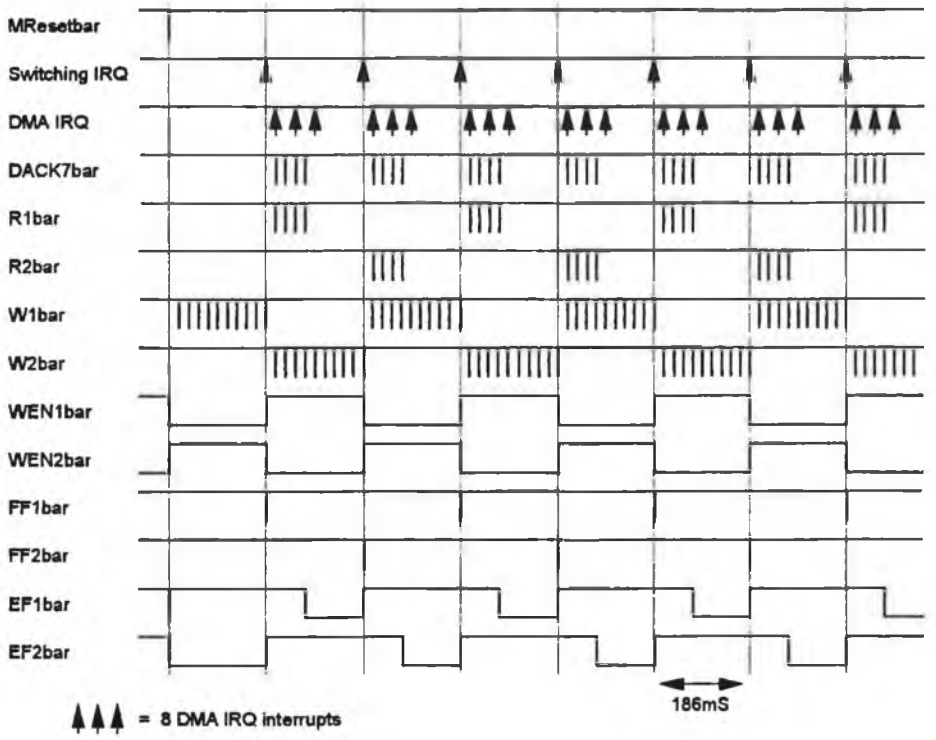


***Figure 2.16** Operation of the Sound Card during Recording.*

The DMA controller will wait for channel seven's DMA transfer request signal *(DRQ7)* to be placed high before it starts the DMA transfers. This signal is controlled by the empty flags of the Swinging Buffers and will be high only when both of the empty flags are inactive. Therefore, this signal only goes high after the first sample has been written to the second Swinging Buffer, setting its empty flag inactive. It remains high until the first Swinging Buffer has been emptied. Once the DMA controller has control of the system bus, it will perform a combined I/O read and memory write operation.

When the 2 kwords have been transferred to the DMA ping buffer an interrupt is generated from the combination of the DMA controller's terminal count signal *(TC)*, common to all its channels, and DACK7bar, to prevent other channels' TC signals triggering this interrupt. This DMA interrupt will set up a DMA session to empty the next 2 kwords from the first Swinging Buffer which will fill the DMA pong buffer. While this DMA session is taking place in the background, the device driver will transfer the previously filled ping buffer to the application's data buffer.

This process is repeated until the full Swinging Buffer has been completely emptied. Figure 2.16 shows the empty flag of the first Swinging Buffer EF1bar, active at the end of the DMA sessions. The size of each of the Swinging Buffers is 16 kwords, therefore eight 2 kword DMA sessions are required to empty one Swinging Buffer. This is triggered by one Switching and seven DMA interrupts. A count flag is used to keep track of the number of DMA sessions. The eight DMA interrupt service routine just resets the count flag.

When the second Swinging Buffer has been filled a switch will occur which will start the process of emptying it in the same manner as for the first Swinging Buffer. The sound card will continue to operate in this manner, transferring the Swinging Buffers to the empty ping-pong buffers while the full ping-pong buffer is transferred by the device driver to the application's data buffer until the device driver stops recording.

2.7.2 Differences Between the Analogue and Digital Recording Modes

The only significant difference between the analogue and digital recording modes is in the Clock Generation section. When digital recording is active, the DRX extracts its clock signals from the digital audio input signal and replaces the crystal oscillator as the source of the master clock for the Clock Generation section. All clock signals for the ADC are produced by the Clock Generation section.

2.7.3 Analogue Playback

The general playback process is described in the flowchart of Figure 2.17 while the general playback operation of the sound card is illustrated in Figure 2.18. The State flip-flops are configured for analogue playback before the MResetbar is issued as explained previously. MResetbar will reset the Swinging Buffers and the sound card's logic. The

DMA and interrupt enable signals are then activated by writing to their respective I/O port addresses.

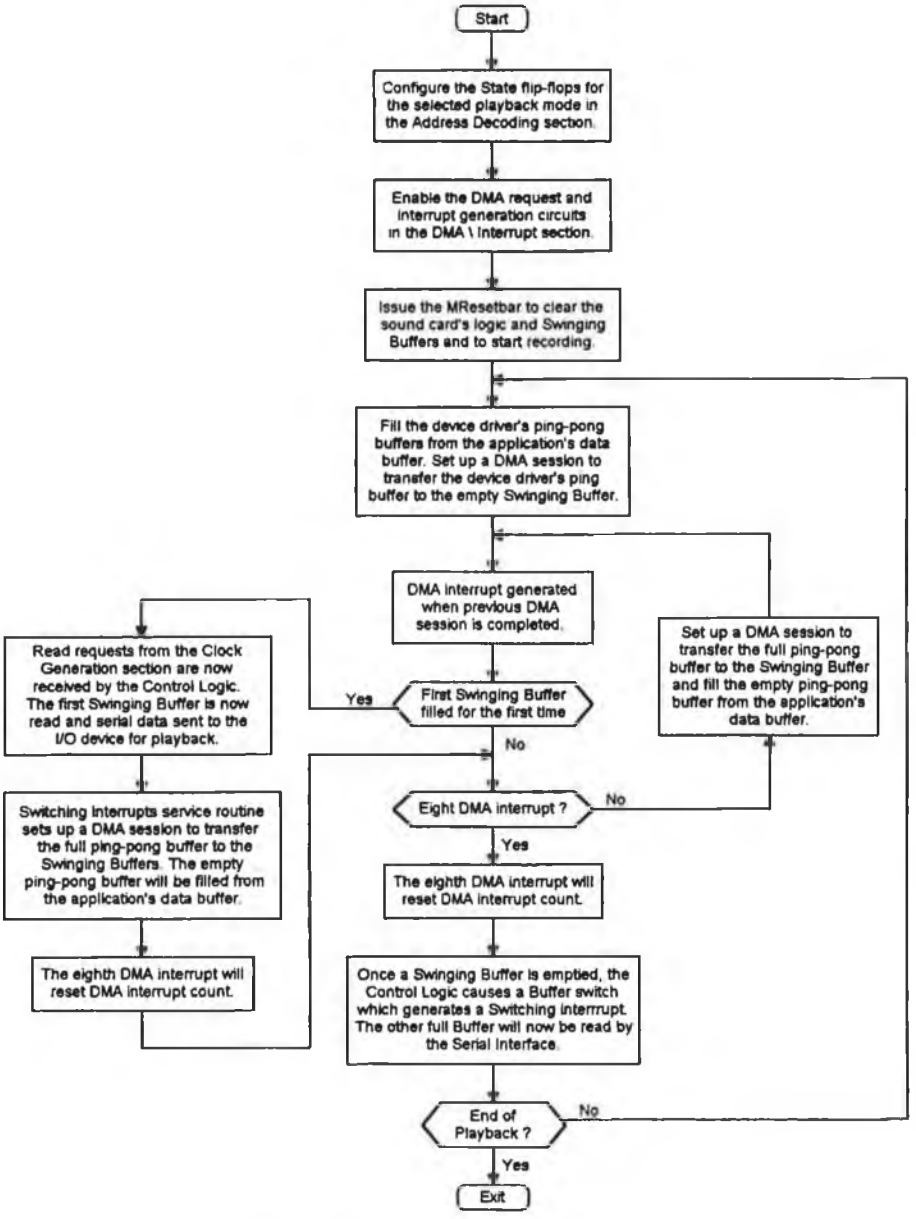


***Figure 2.17** The Playback Process.*

The ping-pong buffers are filled by the device driver from the application's data buffer before setting up a DMA session to transfer the ping buffer to the first Swinging Buffer. This DMA kickstart is required because the Switching interrupt is only generated when a Swinging Buffer has been emptied. In the playback mode, the DMA request pin for channel seven *(DRQ7)* is controlled by the full flags of the Swinging Buffers,

therefore once the DMA session has been initialised a DMA transfer is requested immediately.

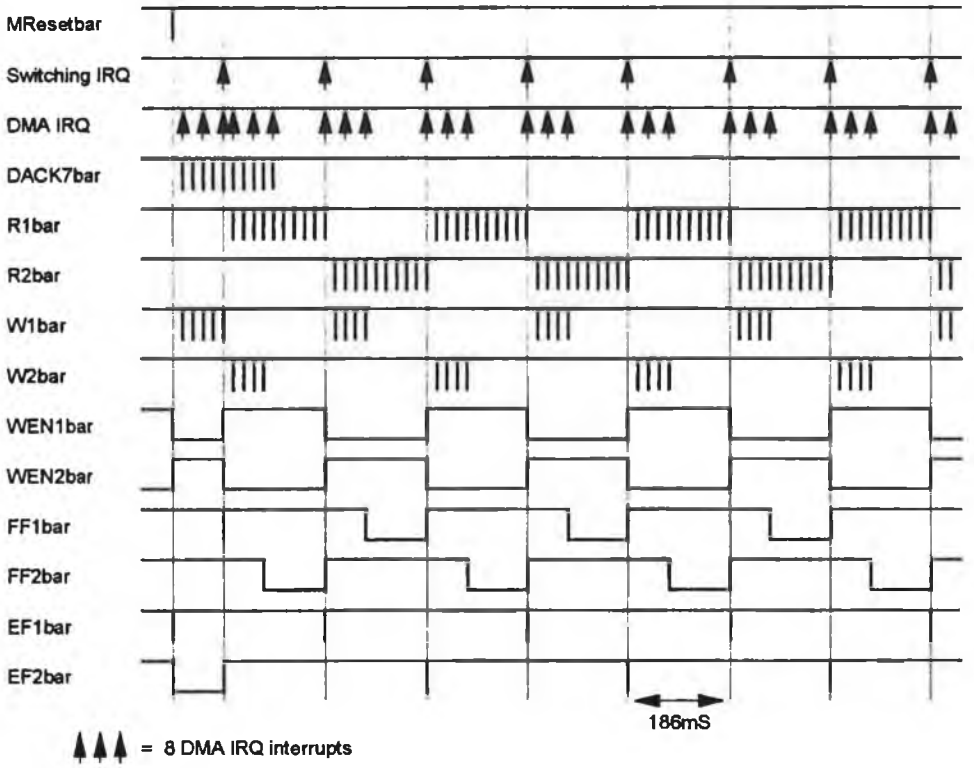


***Figure 2.18** Operation of the Sound Card during Playback.*

The first DMA interrupt will be generated when the first DMA session has finished. The DMA interrupt service routine will set up another DMA session to transfer the full pong buffer to the first Swinging Buffer while the empty ping buffer is filled by the device driver from the application's data buffer. This DMA session will in turn trigger another DMA interrupt at the end of its DMA session, and the process continues until the Swinging Buffer has been filled.

To fill a Swinging Buffer requires eight DMA sessions of 2 kwords each, which are triggered by one Switching and seven DMA interrupts except at start of playback when the DMA kickstart function replaces the Switching interrupt. As before, the eighth DMA interrupt, generated from the seventh's DMA session, must not set up another DMA session. This interrupt just resets the count flag to allow the DMA interrupts to fill the next Swinging Buffer. When the first Swinging Buffer has been filled, the Control Logic will be triggered by its full flag going active, which will toggle the

WENbar signals, thus write enabling the second Swinging Buffer as shown in Figure 2.18. This buffer switch generates a Switching interrupt which starts the process of filling the second Swinging Buffer.

This process continues until the device driver stops playback. The Swinging Buffers provide the application and device driver with sufficient time to maintain continuous playback.

2.7.4 Differences Between in the Analogue and Digital Playback Modes

The two main differences between the analogue and digital playback modes are the Serial Interface and the I/O Interface clock signals. The Serial Interface requires different serial output shift rate clocks depending on the playback mode. The DAC and DTX require different clock signals from the Clock Generation section.

2.8 Summary

This chapter describes the design process of the MTP's sound card and the method used to calculate the size of the Swinging Buffers. The layers of buffering used by the MTP when operating under the Windows environment are also explained along with the operation of the sound card during recording and playback in its DMA / Interrupt transfer mode.

Chapter 3

Hardware Description of the MTP's Sound Card

3.1 Introduction

This chapter describes the functional blocks of the sound card in detail. The unique design aspects of each functional block or section are highlighted. The testing procedures used to verify the recording and playback modes of the sound card are described at the end of this chapter.

3.2 Design of the Sound Card

This section describes the design of the functional blocks of the sound card. These sections were briefly described in Chapter 2 but their operation or design was not explained in detail. The different subsections of the sound card were illustrated previously in Figure 1.2. The signals which end in *bar* signify an active low signal and not an inversion.

3.2.1 Address Decoding

The address decoding schematic diagram is shown in Appendix B, Figure B.1. Standard address decoding techniques are used to provide access to the I/O ports, namely address-line decoders and logic gates [5, 7]. The I/O port address range is as described in Table 2.1. Two of these ports are read only, Buffer Status Register which returns the Buffer Status Byte, and the Buffer Read *(IOPortReadbar)* which returns a word from the active Swinging Buffer. The Buffer Status Byte describes the current state of the State flip-flops and the Swinging Buffers. Buffer Read is directed to the appropriate Swinging Buffer by the Control Logic as described in Section 3.2.4.2.

Table 3.1 defines the individual bits of the Buffer Status Byte. The four lowest bits of the Buffer Status Byte determine the state of the Swinging Buffers. These signals are active low while the four highest bits are the State flip-flop conditions which are active high and are explained later in this section.

There are eight addresses which can be written to. One of these, Master Reset *(MResetbar)*, resets the card and is only used when the card is being initialised. The remaining addresses set or reset t-type flip-flops which control the configuration of the card. The four State flip-flops, DIGITAL IN, DIGITAL OUT, ANALOGUE IN and ANALOGUE OUT are not cleared by MResetbar unlike the Interrupt enable *(IRQEnable)* and the DMA enable *(DMAEnable)* flip-flops.

Bit	Purpose
0	Buffer 1 Full
1	Buffer 2 Full
2	Buffer 1 Empty
3	Buffer 2 Empty
4	DIGITAL IN
5	DIGITAL OUT
6	ANALOGUE IN
7	ANALOGUE OUT

***Table 3.1** Bit definitions for the Buffer Status Byte.*

The reason for MResetbar not clearing the State flip-flops is to ensure that the state of the card is stable before and after a MResetbar signal occurs. For the recording modes, MResetbar starts recording and a change in the state of the card would cause a switch of the Swinging Buffers. This may cause invalid data to be written into the Swinging Buffers, corrupting the start of the recording.

Buffer Write *(IOPortWritebar)* writes a word to the active Swinging Buffers when the Polled I/O transfer mode is active. This write request is directed to the appropriate Swinging Buffer by the Control Logic. The I/O port read and write requests are 16 bits wide. The Control Logic circuit determines which Buffer is read or write enabled which reduces the number of I/O ports required. All the I/O ports are 16 bits wide to reduce the number of gates required to implement the address range.

The I/O Read Cycle is described in Figure 3.1. The IORbar signal is the critical signal because it is the last to go active and the first to become inactive. The active period is approximately 250ns and data must be valid at the PC buses' data buffers when IORbar goes inactive at the end of the processor's wait cycle T_W. The propagation path for the IORbar signal must be minimised. The shortest propagation path through the address decoding circuits was achieved by using IORbar only at the end of the address decoding sequence. This is used to enable the last address line decoder which generates

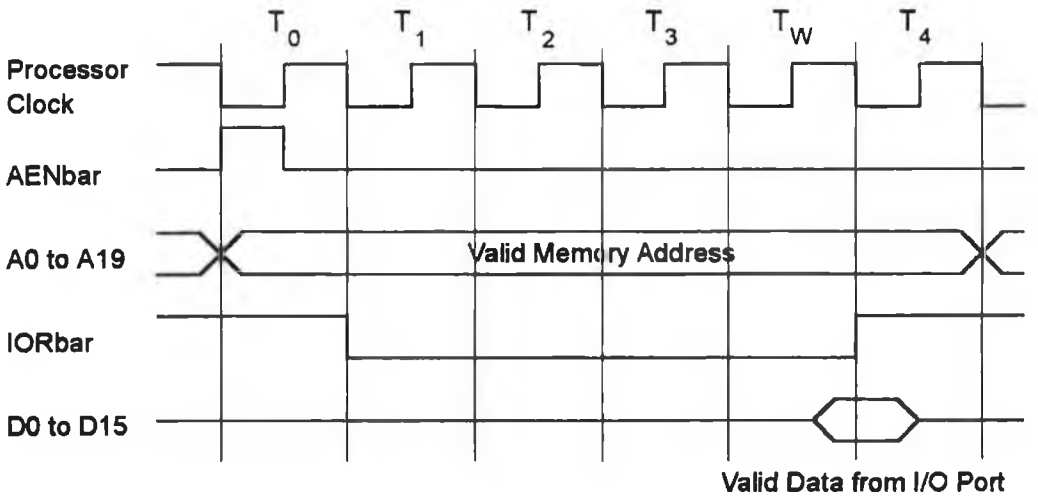


Figure 3.1 *General I/O Port Read Cycle.*

the read request. Because the path through the Address Decoding and Control Logic, although minimised, was still over 120 ns, this dictated the use of fast FIFOs for the Swinging Buffers.

The I/O Port Write Cycle is shown in Figure 3.2. Propagation delays are not as critical as in the I/O Port Read Cycle because now the FIFOs are being written to. However the same technique was carried out as for the IORbar case. The address decoding section also generates the IN and OUT directional signals. The signal IN is the logical OR operation of DIGITAL IN and ANALOGUE IN states while the OUT signal is the inverse of the IN signal.

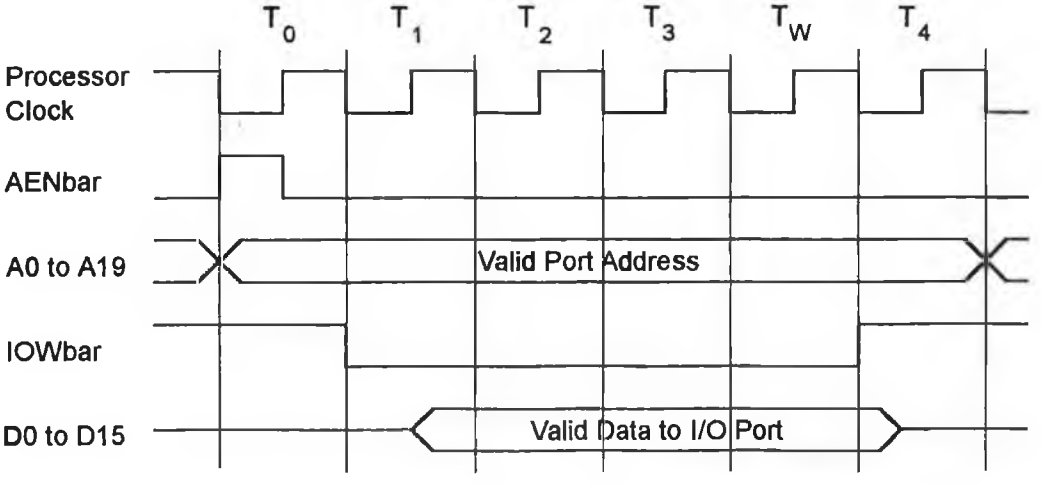


Figure 3.2 *General I/O Port Write Cycle.*

3.2.2 Swinging Buffers

The Swinging Buffers consist of four FIFOs and eight 74F245 bidirectional data buffers. The block diagram of this section is shown in Figure 3.3 while its schematic diagram is shown in Appendix B, Figure B.3. The FIFOs are 16 k x 9 bits wide and there are two per Swinging Buffer to provide the 16 bit data width. There are also two 74F245 bidirectional data buffers per FIFO. The data buffers ensure that there is no data contention between the Serial Interface and the PC's data buffers, and they are enabled by the Control Logic.

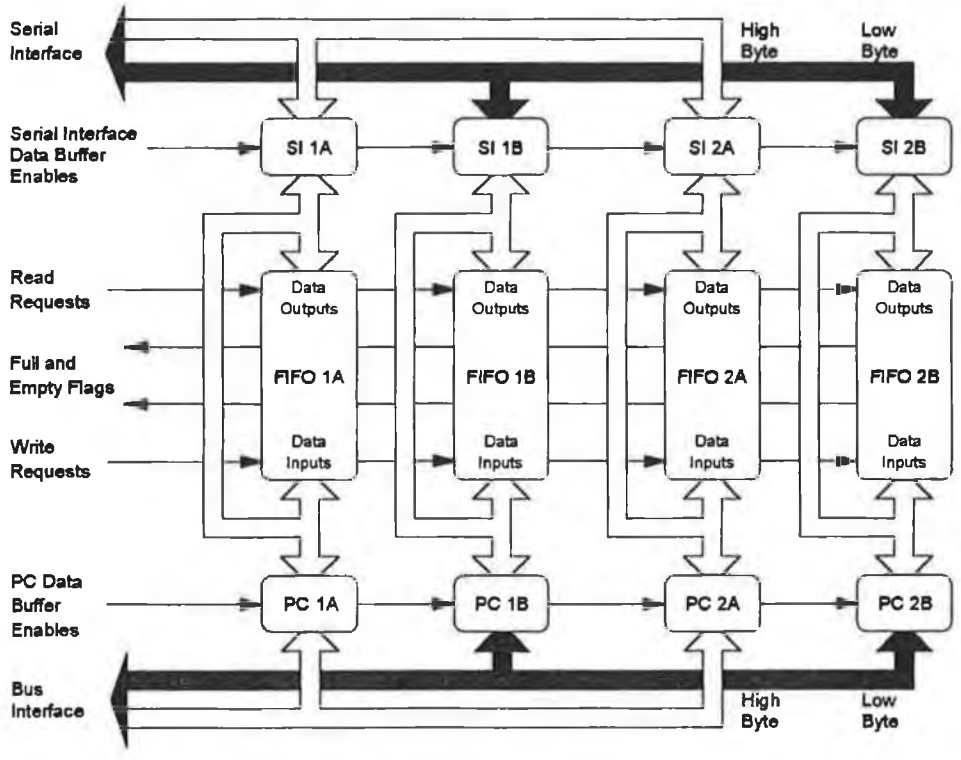


Figure 3.3 Block Diagram of the Swinging Buffers.

Both the Serial Interface and the PC must have access to the inputs and outputs of the FIFOs. Therefore, the FIFOs inputs and outputs are tied together, to allow each data buffer access to both. In this situation data contention would be a problem between the inputs and outputs, if read and write requests occurred simultaneously but the Control Logic ensures that this never occurs. Read and write requests are active low signals. The data are clocked into the FIFO inputs on the rising edge of the write requests. Data are available on the outputs, 50ns after the falling edge of the read

request signal. The empty and full flags are active low and are taken from each Swinging Buffer's high byte FIFO.

3.2.3 DMA / Interrupt Generation

The schematic diagram for the DMA / Interrupt Generation section is shown in Appendix B, Figure B.2. This section of the card consists of a DMA request circuit, two interrupt generators, and a DMA read/write pulse generator circuit. These different components are enabled by the DMAEnable and IRQEnable signals which are provided by the Address Decoding section. The interrupts are the DMA and Switching interrupts, DMAIRQ and SwitchIRQ respectively. The DMA interrupt line is IRQ7 *(LPT1)* while the Switching interrupt is IRQ3 *(COMM2)*.

The DMA controller's channel seven request line (*DRQ7*) is active high, and is generated from the Swinging Buffer's empty and full flags. A DMA request is required immediately on entering playback mode, to fill the first Swinging Buffer. The logical AND combination of the full flags produces this request. Initially both buffers' full flags will be high as the buffers have just been reset and are now empty. Therefore DRQ7 remains high until the first buffer is filled. The DMA request for the second Swinging Buffer will not be generated until the first buffer's full flag has gone inactive, after the first read has taken place. For the recording modes the buffers' empty flags control DRQ7.

The DMA interrupt is generated at the end of the each DMA session and triggers a subsequent DMA session. It is produced from a logical combination of the DMA controller's Terminal Count signal (*TC*), and the DMA controller's channel seven acknowledgement signal, (*DACK7bar*). TC is a general DMA channel signal which indicates that the present data transfer completes a DMA session (4 kbytes). DACK7bar instructs the sound card that this present I/O cycle is a DMA I/O transfer cycle on channel seven. Therefore the combination of TC and DACK7bar is required to prevent interrupts from being called by other channels' TCs.

The PC's PIC requires its interrupt lines *(IRQ#)* to be pulsed low to signal an interrupt [5]. The low to high transition of this interrupt pulse latches the interrupt into the PIC. Therefore, TC signal is inverted and logically ANDed with DACK7bar to produce this low pulse. IRQ7 is the logical OR combination of this pulse and

IRQEnablebar as shown in Figure 3.4. IRQEnablebar is required to place the interrupt in a high state when IRQEnablebar is inactive, high.

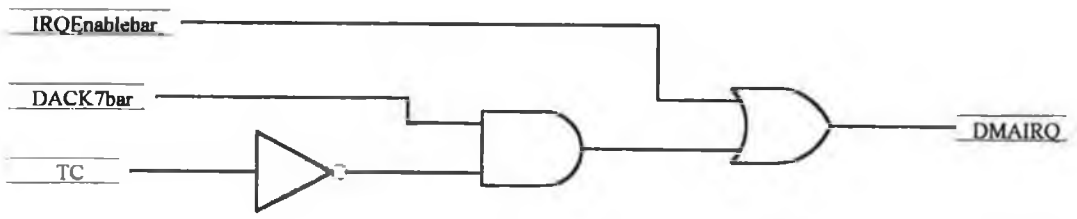


***Figure 3.4** Generation of the DMA Interrupt.*

The Switching interrupt must be generated for every switch of the Swinging Buffers, to trigger a DMA session which begins emptying or filling a Swinging Buffer. The circuit shown in Figure 3.5 is used to trigger this interrupt. The t-type flip-flop is cleared by the MResetbar signal which also sets WEN2bar low and WEN1bar high. These are the write enable signals for the Swinging Buffers provided by the Control Logic and their generation is explained in Section 3.2.4.1. The IRQEnablebar signal is then set (low) which will allow the WENbar signals to control the interrupt line. The OR gate is required to place the interrupt line high when IRQEnablebar is inactive (high).

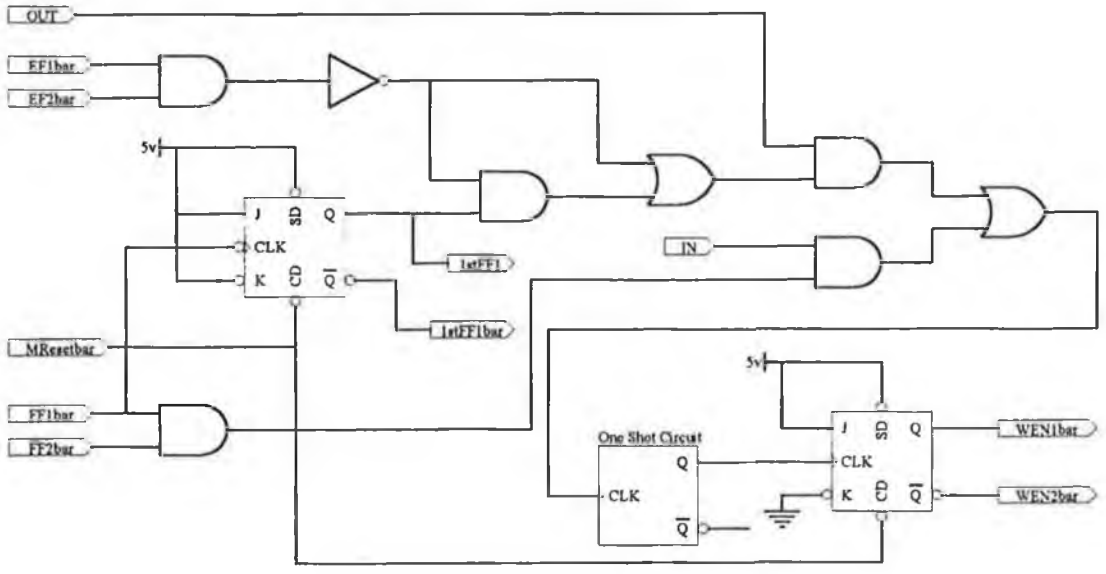


***Figure 3.5** Generation of the Switching Interrupt.*

WEN2bar now controls the interrupt line because the flip-flop's inverted output enables its AND gate.

When the first Swinging Buffer switch occurs, the high to low transition of WEN2bar will pull the interrupt line low. The low transition of WEN2bar will trigger the one shot circuit. The one shot circuit is configured to produce a low pulse of 300 ns duration that toggles the t-type flip-flop on its rising edge. WEN1bar will now control the interrupt line because its AND gate is enabled by the flip-flop's Q output, which pulls the line high again. This rising edge clocks the interrupt into the PIC. On the next Swinging Buffer switch, WEN1bar will start the interrupt generation sequence by pulling the line low while WEN2bar pulls it high. The WENbar signals will continue to generate this Switching interrupt until the sound card ceases operations.

$$DMAIOPortReadbar = (IORbar \bullet IOWbar) + DACK7bar + OUT \qquad 3.1.1$$

$$\overline{DMAIOPortWritebar} = (\overline{IORbar} + \overline{IOWbar}) \bullet (\overline{DACK7bar} \bullet \overline{IN}) \qquad 3.1.2$$

Equation 3.1 *DMA Read and Write Request's Boolean Expressions.*

The read and write cycles for DMA I/O read and write operations are similar to the I/O Port operations which were explained in Section 2.4.2. Read and write request signals for the Swinging Buffers are generated from DACK7bar and IORbar or IOWbar. The PC system bus's data buffers are enabled by DACK7bar. DACK7bar is active longer than the IORbar or IOWbar signals, therefore the data are always stable when the PC latches it into memory. The boolean expressions for the DMA read and write requests are listed in Equations 3.1. The first equation, 3.1.1 is the logic expression as implemented in the sound card while the second 3.1.2 is the general active low expression.

3.2.4 Control Logic

The schematic diagram for the Control Logic is shown in Appendix B, Figure B.4. The Control Logic manages the Swinging Buffers. WEN1bar and WEN2bar are mutually exclusive signals which avoid any read-write conflicts. These signals are produced by the two outputs of a t-type flip-flop and therefore they cannot be in the same state at the

same time. The Address Decoding section provides two mutually exclusive signals, IN and OUT which determine the directional strategy of the Control Logic.

3.2.4.1 Swinging Buffer Switch

The Swinging Buffers must be regularly switched to maintain continuous data flow between the I/O Interface and the Swinging Buffers. The full and empty flags of the Swinging Buffers control this switching. Figure 3.6 shows the Swinging Buffer's switching circuit.

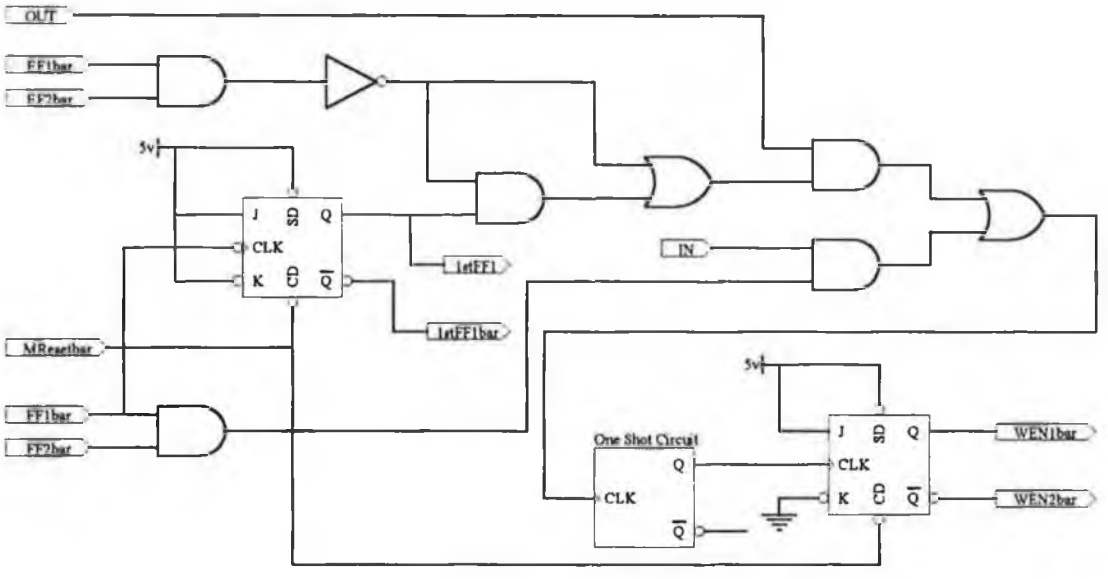


***Figure 3.6** Switching Circuit for the Swinging Buffers.*

When in playback mode, switching is controlled by the Swinging Buffers empty flags EF1bar and EF2bar. The waveforms for a general switch from the first to the second Swinging Buffer are illustrated in Figure 3.7. When the first Swinging Buffer has been emptied by the read request oscillator pulse *(Rosc)*, the buffers must switch so that Rosc begins emptying the full second buffer. I/O write requests from the microprocessor or the DMA controller are now diverted to the first buffer. Data must be continually read from the Swinging Buffers by Rosc to maintain continuous playback.

The second Swinging Buffer will be full with its empty flag inactive, when this switch occurs. When the first Swinging Buffers's empty flag goes low after its last read, the logical AND combination of the two empty flags will move from high to low. This falling edge triggers a one shot circuit that produces a low pulse on its inverted output.

The rising edge of this pulse clocks the WENbar's t-type flip-flop which toggles the WENbar signals, read enabling the second Swinging Buffer and write enabling the first.

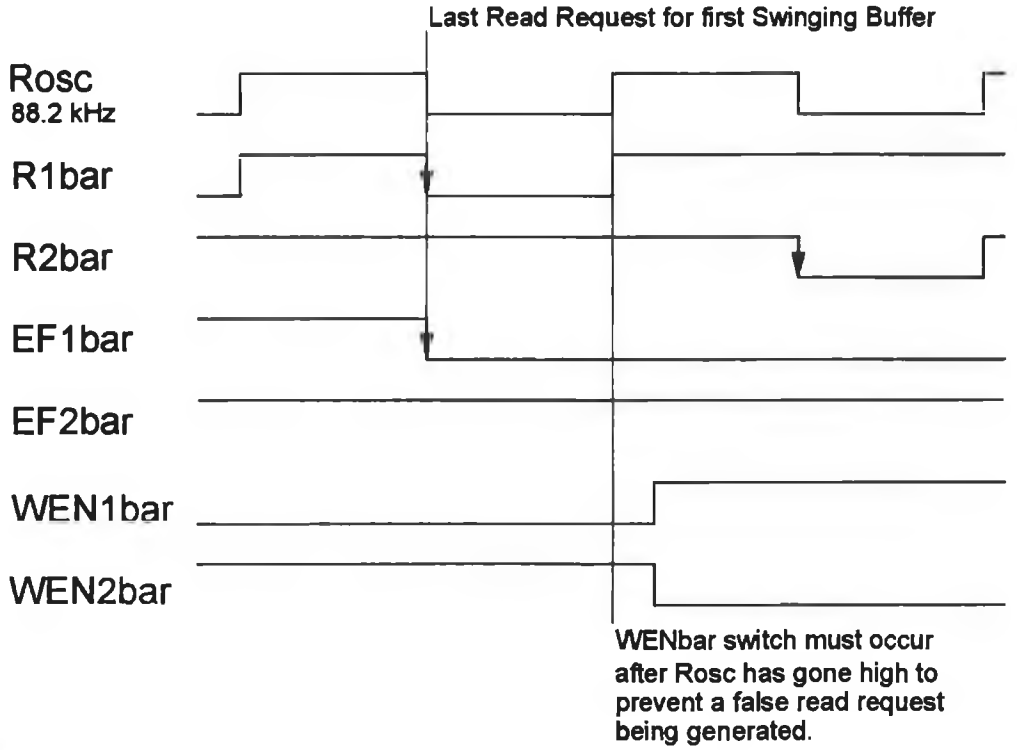


Figure 3.7 Waveforms during a switch from the first Swinging Buffer to the second.

The one shot circuit is required to allow sufficient time for the last read from the first Swinging Buffer to be loaded into the Serial Interface and subsequently shifted into the I/O Interface's active output device. The Swinging Buffer's empty flag is active immediately after the last read request has been received. If the switch occurred immediately, data from the last read would be corrupted by the other Swinging Buffer receiving a false read request caused by the generic Rosc signal still being low. To prevent this data loss and subsequent exchange of channels, the switch must be delayed for more than half a read request cycle, which is 5.7 μs (1/88200 Hz). The one shot circuit provides a 7 μs delay.

There is a difficulty at the start of playback when the first Swinging Buffer has been filled and the other is still empty. The logical AND combination of the empty flags in this situation will not provide the low to high transition required by the one shot circuit to toggle the WENbar signals. The process of generating this switch is as follows, the Swinging Buffer's first empty flag sets a d-type flip-flop which was initially cleared by the MResetbar signal. This flip-flop's output provides the low to high transition

which will trigger the one shot circuit and subsequently toggle the WENbar t-type flip-flop, switching the buffers.

In the recording modes, the empty flags are replaced by the full flags. The switching operation is identical to the play mode with no special circuitry required at the start of recording.

3.2.4.2 Swinging Buffers' Read and Write Requests

The Swinging Buffers read and write requests, R1bar, R2bar, W1bar and W2bar are generated from three possible sources, the DMA controller, the Clock Generation section and the microprocessor I/O instructions. Equations 3.2 illustrate the logic steps in generating the separate read and write requests for the individual Swinging Buffers. The equations are written in two different forms. The write requests are the active low Boolean expressions, while the read requests describe the logic implementation on the sound card as can be seen from Appendix B, Figure B.4.

The PC Swinging Buffer request signals, *RPCbar* and *WPCbar* are generated as shown in Equations 3.2.1 and 3.2.2. These signals are never active simultaneously. The WENbar signals are then used to generate separate read and write requests for the individual Swinging Buffers as shown in Equations 3.2.3 to 3.2.6. The Clock Generation section's single read and write requests *(Rosc and Wosc)*, are combined with the WENbar signals to produce another set of read and write requests for the Swinging Buffers, *Rosc1bar*, *Rosc2bar*, *Wosc1bar* and *Wosc2bar*. These are then combined with *RPC1bar*, *RPC2bar*, *WPC1bar* and *WPC2bar* to produce the Swinging Buffer read and write requests as shown in Equations 3.2.10 to 3.2.14.

3.2.4.3 Swinging Buffers' Serial Interface and PC Data Buffer Enables

The Control Logic also controls enabling and disabling of the Serial Interface buffers and the PC buffers within the Swinging Buffer section. These signals are active low and their Boolean expressions are listed in Equations 3.3. Again there is a distinction between the active low Boolean expressions and the logic as implemented on the sound card. The Serial Interface buffers are controlled by either the Wosc write requests or the Rosc read requests depending on the data direction through the card. The PC data buffers are enabled from either of the separate PC read or write requests. These separate

$$RPCbar = (DMAIOPortReadbar \bullet DMAEnable) + (IOPortReadbar \bullet DMAEnablebar) \quad 3.2.1$$

$$\overline{WPCbar} = (\overline{DMAIOPortWritebar} + \overline{DMAEnable}) \bullet (\overline{IOPortWritebar} + \overline{DMAEnablebar}) \quad 3.2.2$$

$$RPC1bar = RPCbar + WEN2bar \quad 3.2.3$$

$$RPC2bar = RPCbar + WEN1bar \quad 3.2.4$$

$$\overline{WPC1bar} = \overline{WPCbar} \bullet \overline{WEN1bar} \quad 3.2.5$$

$$\overline{WPC2bar} = \overline{WPCbar} \bullet \overline{WEN2bar} \quad 3.2.6$$

$$Rosc1bar = Rosc + IN + WEN2bar \quad 3.2.7$$

$$Rosc2bar = Rosc + IN + EN1bar \quad 3.2.8$$

$$\overline{Wosc1bar} = \overline{Wosc} \bullet \overline{OUT} \bullet \overline{WEN1bar} \quad 3.2.9$$

$$\overline{Wosc2bar} = \overline{Wosc} \bullet \overline{OUT} \bullet \overline{WEN2bar} \quad 3.2.10$$

$$R1bar = Rosc1bar \bullet RPC1bar \quad 3.2.11$$

$$R2bar = Rosc2bar \bullet RPC2bar \quad 3.2.12$$

$$\overline{W1bar} = \overline{Wosc1bar} + \overline{WPC1bar} \quad 3.2.13$$

$$\overline{W2bar} = \overline{Wosc2bar} + \overline{WPC2bar} \quad 3.2.14$$

Equation 3.2 *Swinging Buffer's Read and Write Requests.*

read and write requests are composed of DMA and Polled I/O requests as illustrated previously in Equations 3.2. These data buffers are oriented so that their directional pins are all controlled by the IN signal.

$$SIBE1bar = (Wosc + OUT + WEN1bar) \bullet (Rosc + IN + WEN2bar) \quad 3.3.1.$$

$$SIBE2bar = (Wosc + IN + WEN2bar) \bullet (Rosc + OUT + WEN1bar + 1stFF1bar) \quad 3.3.2.$$

$$\overline{PCBE1bar} = \overline{RPC1bar} + \overline{WPC1bar} \quad 3.3.3.$$

$$\overline{PCBE2bar} = \overline{RPC2bar} + \overline{WPC2bar} \quad 3.3.4.$$

Equation 3.3 *Serial Interface and PC Data Buffers' Enables.*

3.2.5 I/O Interface

The analogue and digital I/O interface chips are manufactured by Crystal Semiconductor Corporation [6] and are listed in Table 3.2. The I/O Interface is divided into an Analogue and a Digital I/O Interface and their schematic diagrams are shown in Appendix B, Figures B.7 and B.8 respectively. All devices use a 44.1 kHz L/R clock to delineate the samples into left and right, left being the clock's high half. The most significant bit is the first bit of every channel. The local clocks are generated from a 22.5792 MHz crystal oscillator or from the DRX in the digital recording mode.

Analogue to Digital Converter (*ADC*)	CS5338KP
Digital to Analogue Converter (*DAC*)	CS4328KP
Digital Receiver (*DRX*)	C8412A
Digital Transmitter (*TDX*)	CS8402A

***Table 3.2** I/O Interface Chips.*

3.2.5.1 Digital Receiver

Unless it is configured to extracts its master clock signal from its input signal, the DRX will lose synchronisation with its input signal, resulting in loss of data. The locally generated master clock possesses too much jitter to provide lossless data reception [6]. Two other clock outputs are also provided by the DRX for synchronisation with the serial data. The DRX's clocks are a 11.2896 MHz master clock, a 2.8224 MHz bit clock and a 44.1 kHz L/R clock. This master clock replaces the master clock in the Clock Generation section. Therefore in this mode, all the sound card's clocks are derived from the DRX. The DRX's L/R clock synchronises the other clocks to the raw audio data from the DRX. The raw data are clocked out from the DRX on the rising edges of the first 16 bit clock cycles after every L/R clock transition as shown in Figure 3.8. Only the raw audio data are stored and all the control and user information is discarded in this present design.

3.2.5.2 Digital Transmitter

The digital transmitter (*DTX*) is configured so that each one of its three clocks are externally generated. These clocks are generated by the clock generation section and consist of a 11.2896 MHz master clock, a 2.8224 MHz bit clock and a 44.1 kHz L/R

clock. The DTX samples the raw audio data from the Serial Interface on the first 16 falling edges of its bit clock at the start of each channel. The DTX can accept limited control and user information on dedicated pins. The validity bit is set via its dedicated pin to indicate that the data being transmitted are suitable for conversion to analogue.

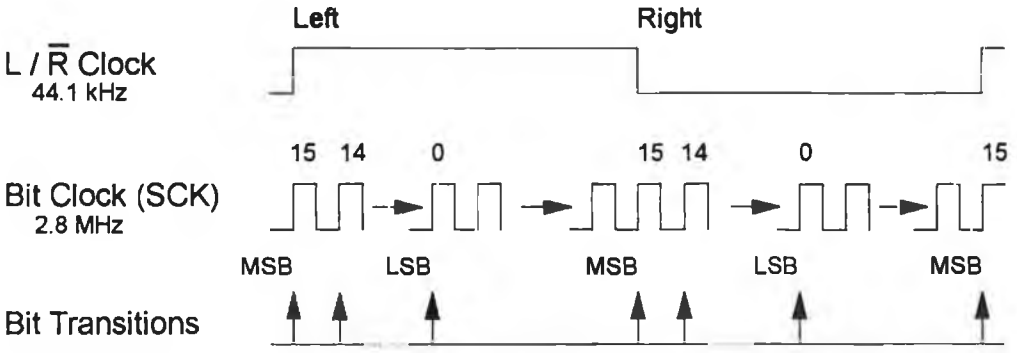


Figure 3.8 Serial Data from the Digital Receiver.

3.2.5.3 Analogue to Digital Converter

The Analogue to Digital Converter System (*ADC*) is the CS5338KP. Contained on this chip are a sample and hold, analogue to digital converter and anti-aliasing filters for each channel. The ADC uses 64 times over sampling delta sigma modulators with digital filtering and decimation which removes the need for an external anti-aliasing filter. The filter has a band pass range of 0 to 24 kHz with 0.01 dB pass band ripple, linear phase and greater than 80 dB stop band attenuation. The input range is ±3.68 volts. Three locally generated clocks are required, a 5.6448 MHz master clock, a 2.8224 MHz bit clock and a 44.1 kHz L/R clock. The data are clocked out in the same format as the DRX (see Figure 3.8). For input protection, a NC5532 chip, containing two low noise operational amplifiers, buffers the input signals.

3.2.5.4 Digital to Analogue Converter

The Digital to Analogue System (*DAC*), CS4328KP, is contained on one chip. This chip has an 8 times over-sampling interpolation filter followed by a 64 times over sampling delta sigma modulator for each channel. The system has 120 dB signal to noise ratio, linear phase, and zero phase error between the channels. Three locally generated clocks are required, a 5.6448 MHz master clock, a 1.4112 MHz bit clock and a 44.1

kHz L/R clock. Data are sampled on the falling edge of each bit clock cycle as shown in Figure 3.9.

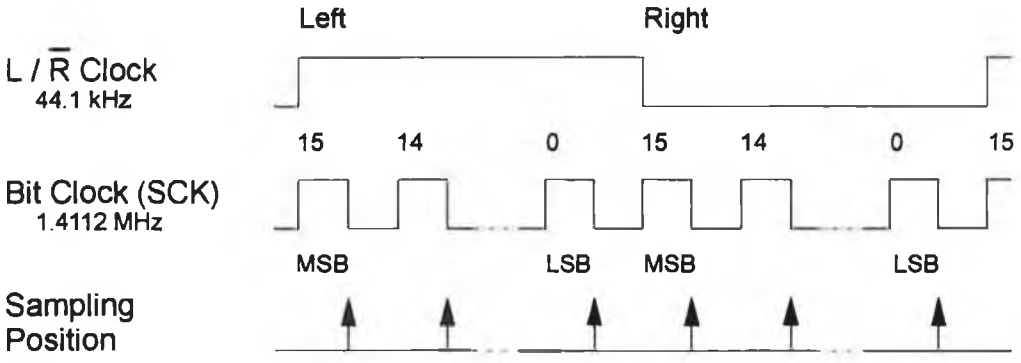


***Figure 3.9** Serial Data required by the Digital to Analogue Converter.*

3.2.6 Clock Generation

The Clock Generation section consists of a crystal oscillator, an 8 bit counter and channel consistency circuits. The schematic diagram for this section is in Appendix B, Figure B.6. The local clocks are generated from a 22.5792 MHz crystal in a series resonant circuit with the crystal operating in its fundamental mode. A divide by two circuit produces the 11.2896 MHz local master clock that clocks the 8 bit counter. This 8 bit counter produces the clocks required by the I/O Interface, Serial Interface and Control Logic circuits. All subsequent clocks derived from the master clock change on the rising edge of the previous clock.

The 11.2896 MHz local clock is masked in the digital recording mode. As explained previously, the DRX extracts its own 11.2896 MHz clock from the incoming digital audio signal which then clocks the 8 bit counter, producing the required clock signals. The circuit shown in Figure 3.10 is required to synchronise the 8 bit counter's L/R clock with the L/R clock from the DRX, otherwise synchronisation between the counter's and the DRX's L/R clocks cannot be guaranteed and data will be lost at the Serial Interface. The MResetbar signal will clear the d-type flip-flop, which in turn clears the 8 bit counter. The next rising edge of the DRX's L/R clock will permanently set this flip-flip, enabling the counter. The counter is now synchronised with the DRX's L/R clock.

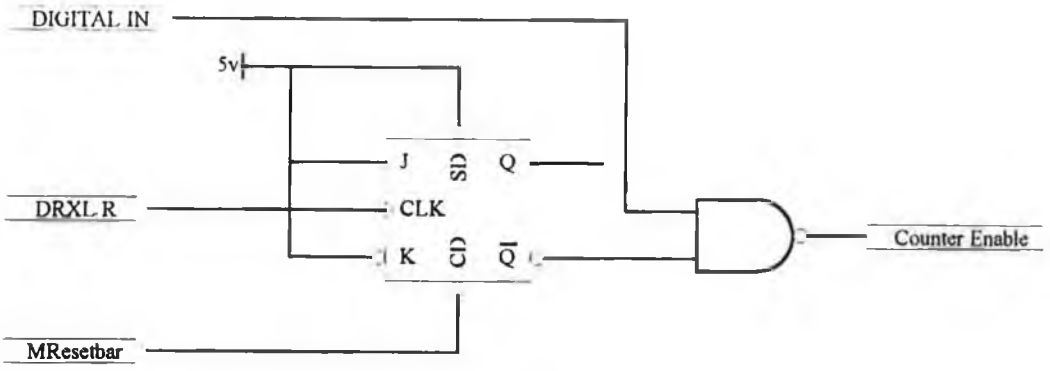


Figure 3.10 *Synchronisation of the Digital Receiver and Counter.*

The L/Rx2 clock from the 8 bit counter provides the source of the read and write requests (Rosc and Wosc), that the Control Logic delivers to the Swinging Buffers for transferring data between them and the Serial Interface.

3.2.6.1 Channel Consistency

The circuit in Figure 3.11 is required to maintain channel consistency in the playback and recording of stereo WAVE files. When recording data, the d-type flip-flop is reset by the MResetbar signal which masks the oscillator generated write request *(Wosc)* signal. When the first L/R clock's rising edge occurs after the MResetbar signal, it sets the flip-flop and allows the L/Rx2 clock to generate the Wosc signal by unmasking its OR gate. Therefore the first Wosc signal occurs on the left channel or high half of the L/R clock, and is the first sample to be written to the Swinging Buffers.

A similar arrangement is provided for playing the left channel first. Rosc's OR gate is disabled until the first Buffer has been filled by using the first Swinging Buffer's Full flag (*1st FF1*) to clear the flip-flop. When this flag goes active the next falling edge of the L/R clock will set the d-type flip-flop, enabling the L/Rx2 clock to generate the Rosc signal. Rosc is generated on the right or low half of the L/R clock, because the data must be first loaded into the Serial Interface before being shifted out to the I/O Interface on the next half of the L/R clock. Therefore, the I/O Interface receives its first sample on the left (high) half of the L/R clock.

The Clock Generation circuit also provides the parallel loading and bit shifting clocks for the Serial Interface which are described in the next section.

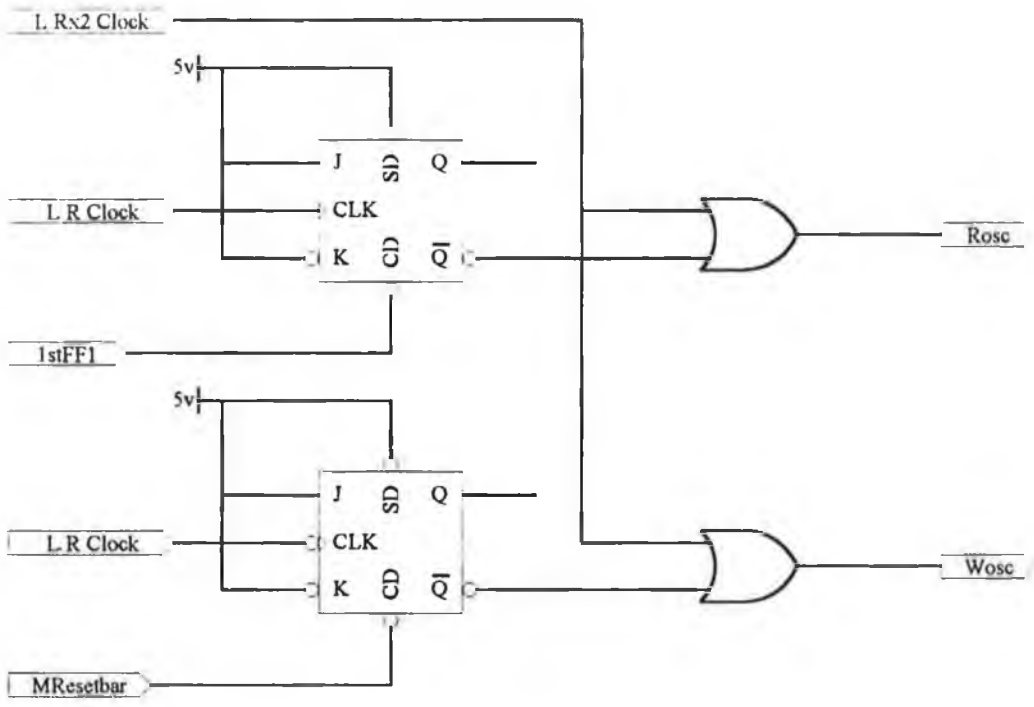


Figure 3.11 Channel Consistency Circuit.

3.2.7 Serial Interface

The Serial Interface converts the parallel data of the Swinging Buffers to and from the serial data of the I/O Interface. It consists of 16-bit serial to parallel and parallel to serial converters, SiPo and PiSo respectively and its schematic diagram is shown in Appendix B, Figure B.5.

The SiPo is composed of two 8 bit 74F595 shift registers while the PiSo is composed of two 8 bit 74F165 shift registers. For PiSo operation, the serial output of the low byte shift registers is feed into the serial input of the high byte shift registers, allowing the whole sample to be shifted through the high byte's serial output pin. Similarly for SiPo operation the serial output of the low byte shift registers is feed into the serial input of the high byte shift registers, allowing the whole sample to be shifted in through a single serial input pin. Both the SiPo and PiSo clocks for parallel loading and serial shifting are provided by the Clock Generation section.

3.2.7.1 Serial to Parallel Converter Clocks

The serial data from the DRX or the ADC are shifted on the first 16 rising edges of their bit clocks after every L/R clock transition. Therefore the serial data are stable on the corresponding falling edges of the bit clocks. The serial shift clock signal is

generated from the logical AND combination of the inverted bit clock and the L/Rx2 clock as is shown in Figure 3.12.

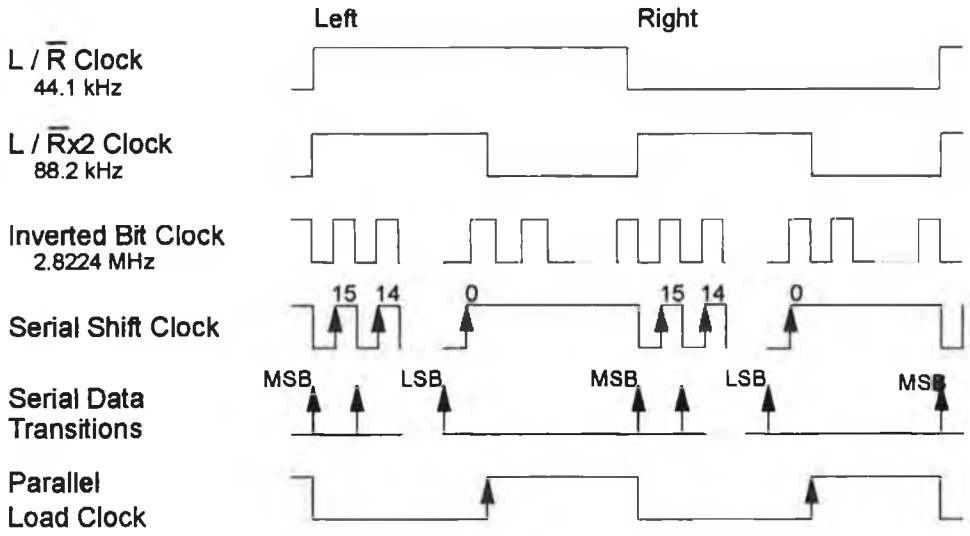


Figure 3.12 *SiPo's Serial Shift and Parallel Load Clock Signals, SiPoSCK and SiPoLOAD.*

The Swinging Buffers sample data on the rising edge of their write requests which are generated from the L/Rx2 clock. The SiPo's parallel outputs are also loaded on the rising edge of its parallel load signals. Therefore, for valid data to be sampled by the Swinging Buffers, its write request and the parallel load signal must be the inverse of each other. The inverted L/Rx2 clock loads the parallel outputs of the SiPo.

3.2.7.2 Parallel to Serial Converter Clocks

The PiSo requires different serial shift clocks for shifting data into the DAC and DTX and is illustrated in Figure 3.13. The DAC requires a slower bit clock than the DTX. The DAC and DTX sample their serial data on the falling edges of their bit clocks so the data must be stable at these transitions. To guarantee the correct bit clocks, ANALOGUE OUT is used to mask or unmask the two bit clocks, thus allowing only one clock to generate the PiSo's serial shift clock. When the DAC is the target, the ANALOGUE OUT signal allows the 1.4112 MHz clock to supply the serial shift clock. The 2.8224 MHz DTX bit clock is enabled by the inverse of the ANALOGUE OUT signal. It is then logically ANDed with the L/Rx2 clock to generate the PiSo's serial shift clock. This logical AND combination is required because the data must be shifted

out during the first 16 bit clock cycles after every L/R clock transition as shown in Figure 3.13.

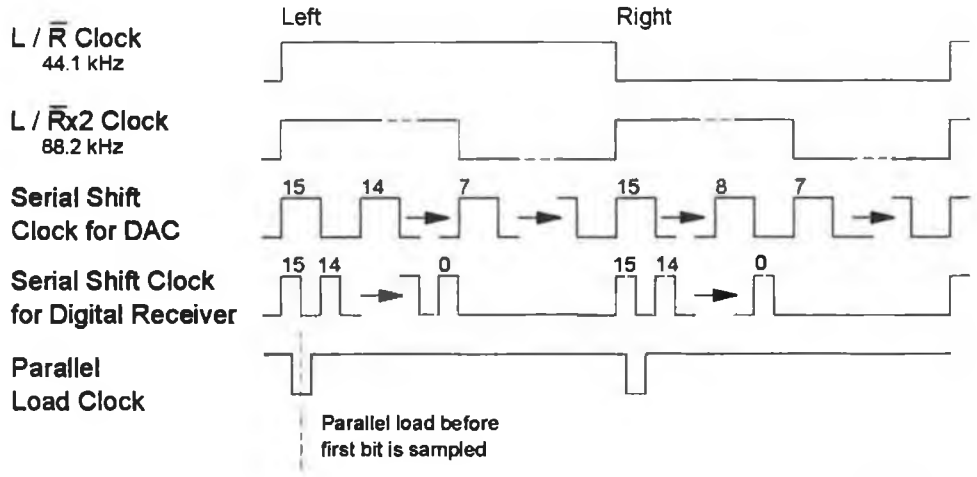


***Figure 3.13** PiSo's Serial Shift and Parallel Load Clock Signals, PiSoSCK and PiSoLOAD.*

The internal registers of the PiSo must be loaded from their parallel inputs before the first bit of each sample is shifted out. This parallel load signal is not edge triggered but is an active low signal, therefore a one shot circuit is used to produces this low pulse. The waveforms in Figure 3.13 also illustrate the location of the parallel load pulse in relation to the first sampling intervals of the two serial shift clocks. The DTX latches its samples earlier than the DAC, therefore the DTX's 2.8224 MHz serial shift clock dictates the timing restrictions. The parallel load must be active within half a bit clock cycle to guarantee that the first bit is stable when sampled by the DTX. It must be inactive before the second data bit is shifted out by the second rising edge of the serial shift clock. The one shot circuit produces a low pulse of 200 ns duration, to load the parallel inputs and be inactive before the second shift of the serial shift clock.

3.3 Field Programmable Gate Array

The XILINX XC4003 Field Programmable Gate Array (*FPGA*) [16] was chosen to implement the Address Decoding, Control Logic and Clock Generation circuits for the sound card. This FPGA which can operate at up to 60 MHz is sufficient to meet the timing requirements which occur when writing to the Swinging Buffers from the PC. The FPGA is a volatile device which must be programmed at power-up. This program

is contained in a PROM which is connected to the FPGA in its master parallel mode. At power-up the FPGA will read the PROM and configure itself.

3.3.1 FPGA Logic Implementation

The FPGA is divided into discrete logic blocks, each capable of implementing a three variable Boolean equation. The Boolean equation is implemented in random access memory *(RAM)* look up tables. Each logic block also possesses a flip-flop on its output which can be used if required. These logic blocks are called control logic blocks *(CBLs)* and are connected together using:-

- local links which only connect adjacent CBLs
- the global network which can connect all CBLs
- fast global network which only connects certain CBLs.

3.4 Design Implementation

The sound card's design was first implemented with discrete logic devices to quickly and efficiently test the design of the sound card. The prototype design was built from the 74F and 74LS series logic families.

After the sound card's design was finalised it was then transferred to the XILINX FPGA. This required some modifications to the design due to the manner in which the logic circuits are implemented in the XILINX environment. These modifications were required to meet the various timing specifications which were automatically met by the discrete logic but were not met by the FPGA. The software package for programming the FPGA was responsible for converting the schematic diagrams to CBL functions and connecting these CBL blocks together.

3.5 Testing of the Sound Card

The sound card in both discrete and FPGA forms was thoroughly tested using the logic analyser to examine the signals which possessed critical timing characteristics and with test files and test signals as described in Section 3.7.4. The following sections describe the critical timing signals and the steps taken to ensure they were maintained. Due to restrictions on the size of labels for the leads of the logic analyser, some names were shortened, such as MResetbar to MRST.

3.5.1 Address Decoding

The address decoding section guarantees valid data transfer between the PC's memory and the FIFOs. The FIFOs latch data on the rising edge of their write requests and therefore the PC data buffer enable signals must remain active for at least the minimum time interval stated in the FIFOs specifications *(5ns)* to guarantee that the FIFOs latch the data correctly. This was achieved using a one shot circuit to produce the PC bus's data buffer enable signals. This guarantees that the PC bus's data buffers will be enabled long enough for the data to be latched into the Swinging Buffers. The one shot circuit consists of a counter, driven by one of the FPGA's internal clocks generated from its internal oscillator.

Figure 3.14 illustrate the simulated waveforms for a DMA write request that generates the PC bus's data buffer enable signals while the actual waveforms sampled by the logic analyser are shown in Figure 3.15. Figure 3.16 shows the simulated waveforms for a variety of Address Decoding operations ranging from resetting the sound card to I/O port read requests.

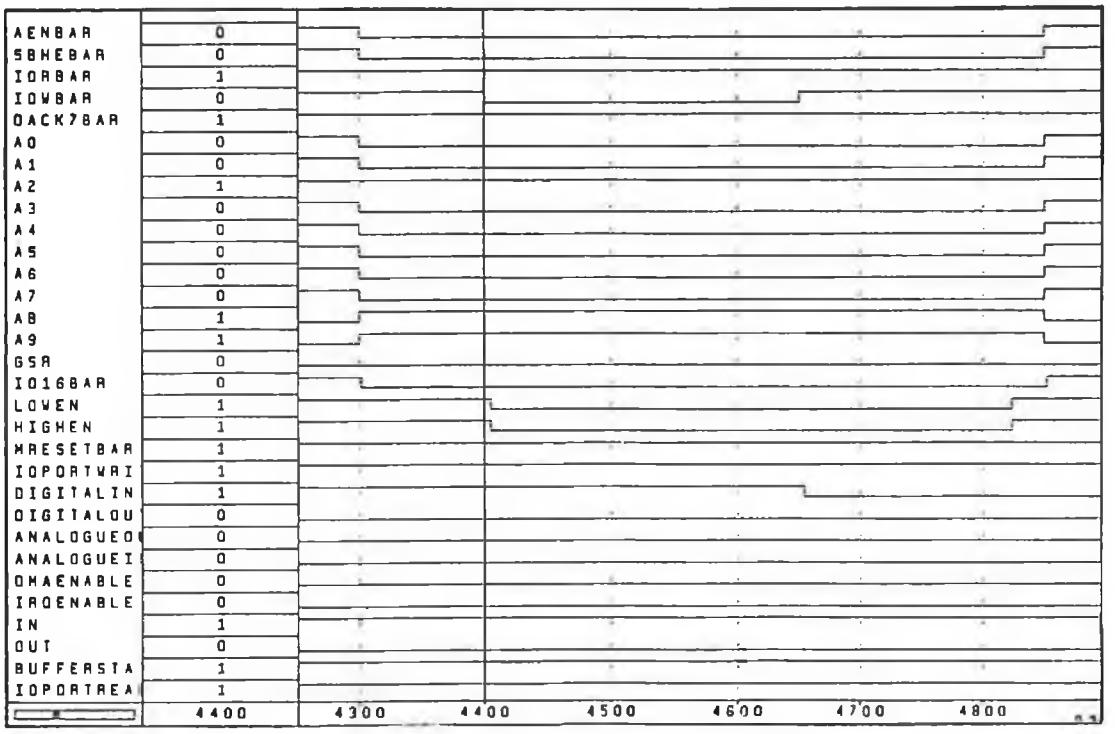


Figure 3.14 Simulated DMA Write Request.

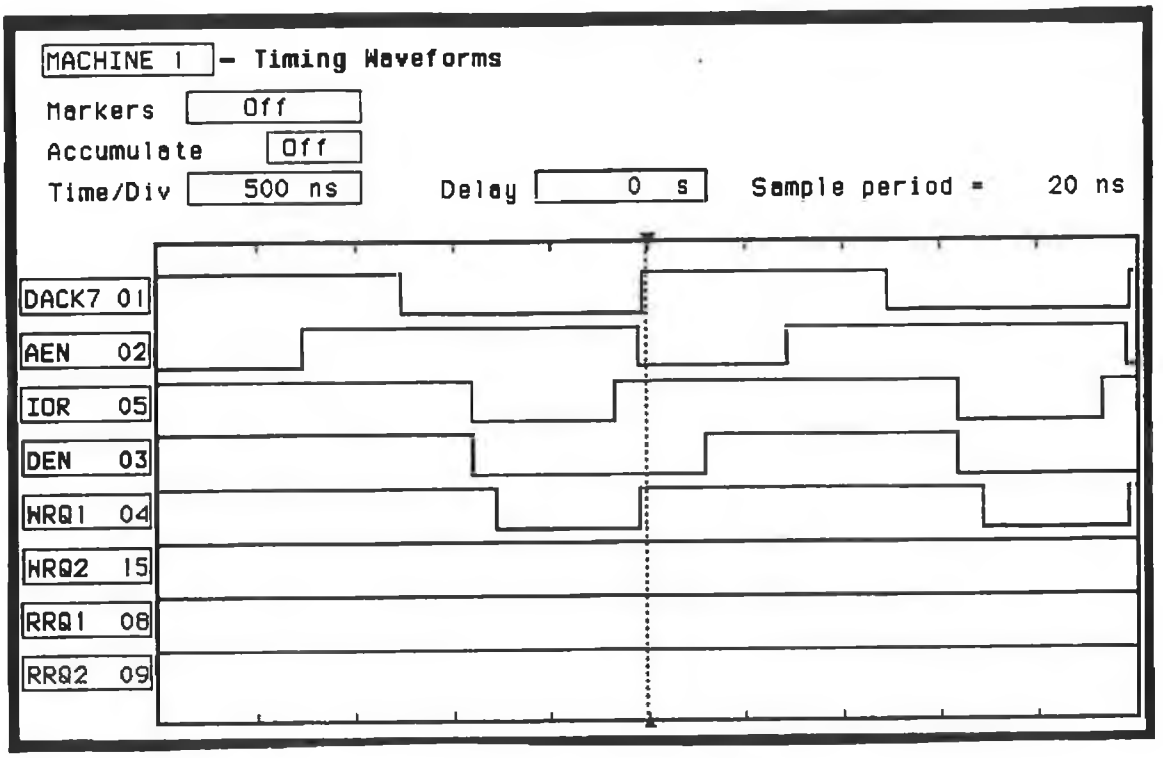


Figure 3.15 DMA Write Request.

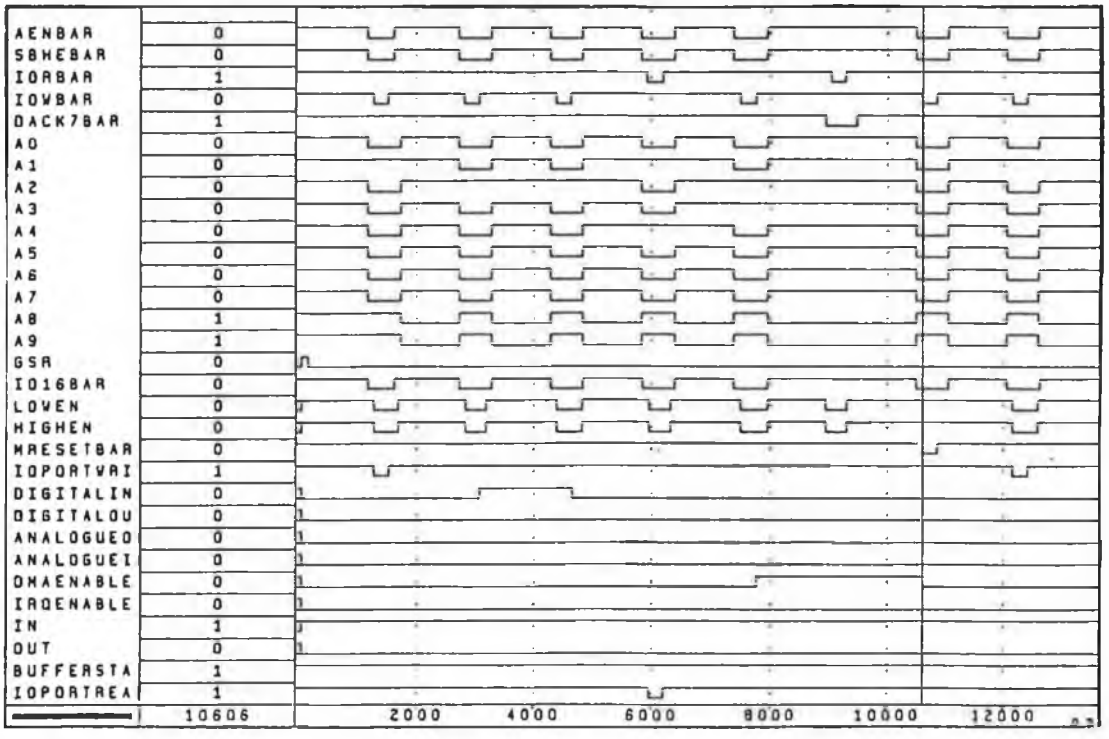


Figure 3.16 Simulated Address Decoding Waveforms.

3.5.2 Control Logic

The simulated waveforms that confirm the bidirectional operation of the Control Logic are shown in Figure 3.17. The enable signals for the Swinging Buffer Serial Interface and the PC data buffers were required to be active longer than the read and write requests to guarantee correct data transfer. This was accomplished with several one shot circuits which produce the required buffer enable signals. The simulated waveforms for the sound card's write requests, generated from the sound card's oscillator, are shown in Figure 3.18 while the actual waveforms from the logic analyser for the Rosc and Wosc signals are shown in Figures 3.19 and 3.20 respectively.

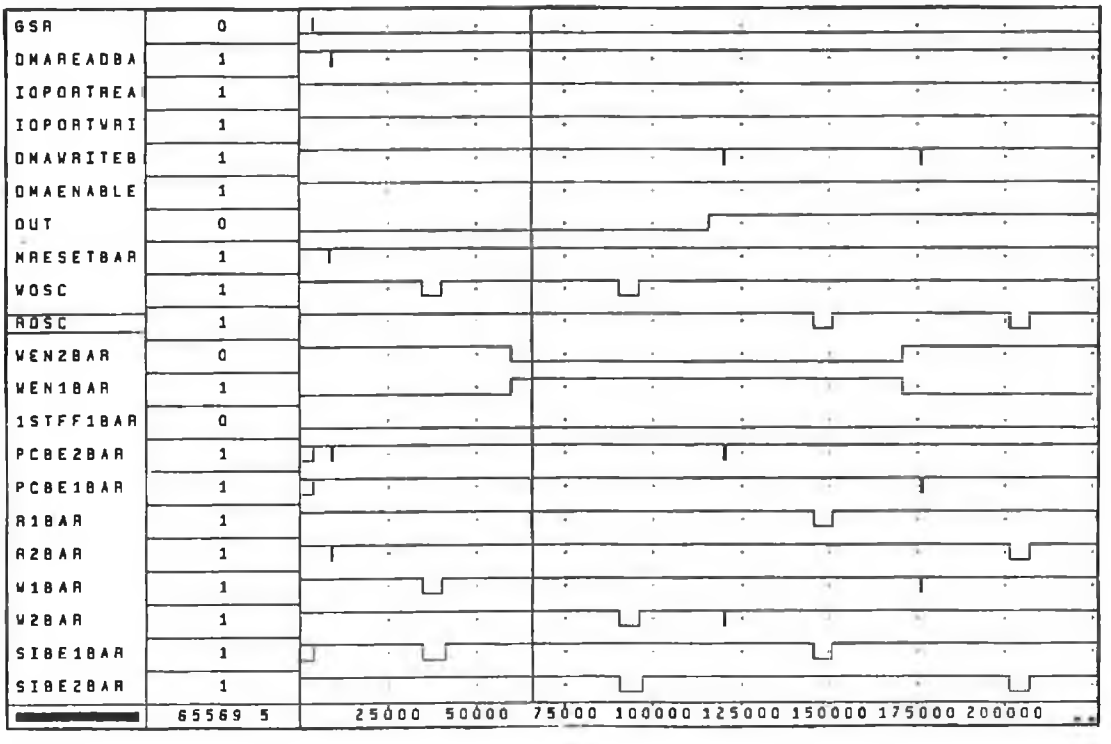


Figure 3.17 Simulated Swinging Buffer Read and Write Requests and Data Buffer Enable Signals.

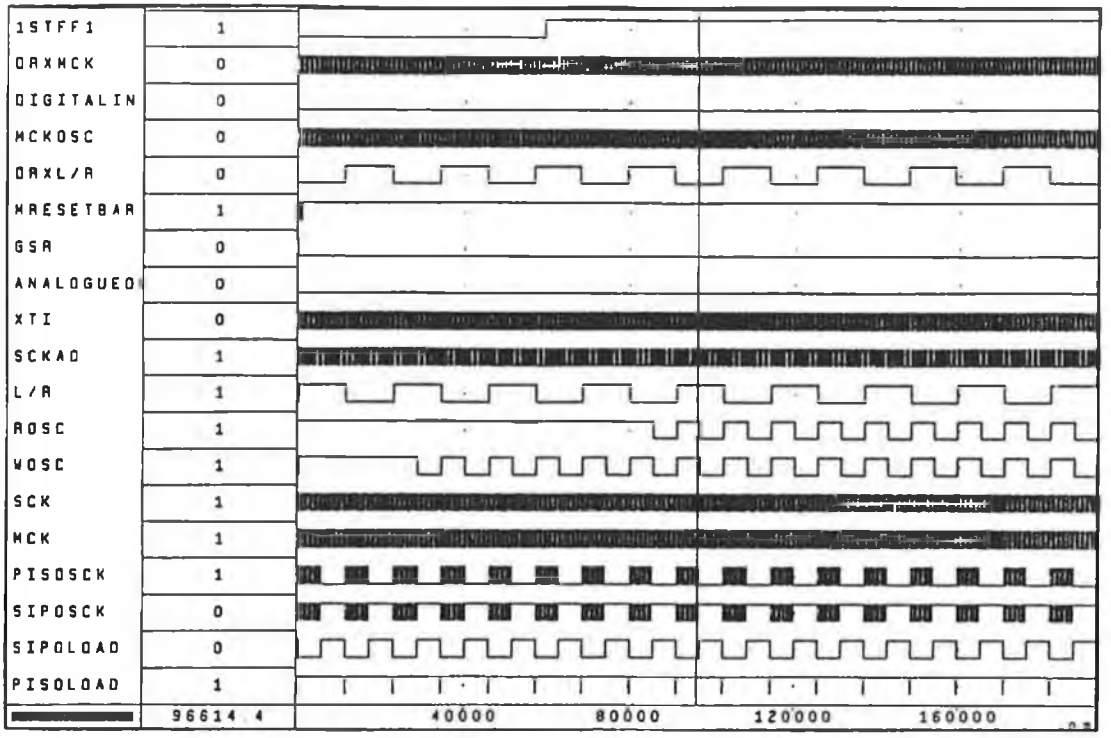


Figure 3.19 Simulated Wosc and Rosc Signals.

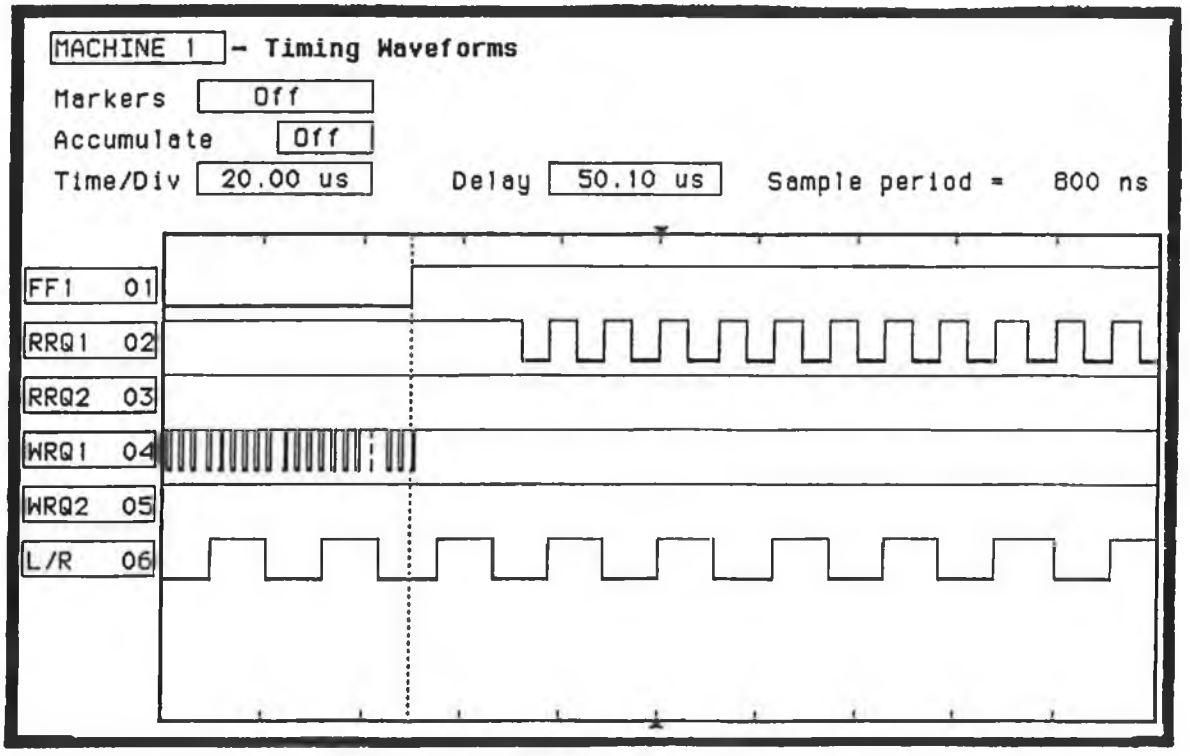


Figure 3.18 Rosc Requests.

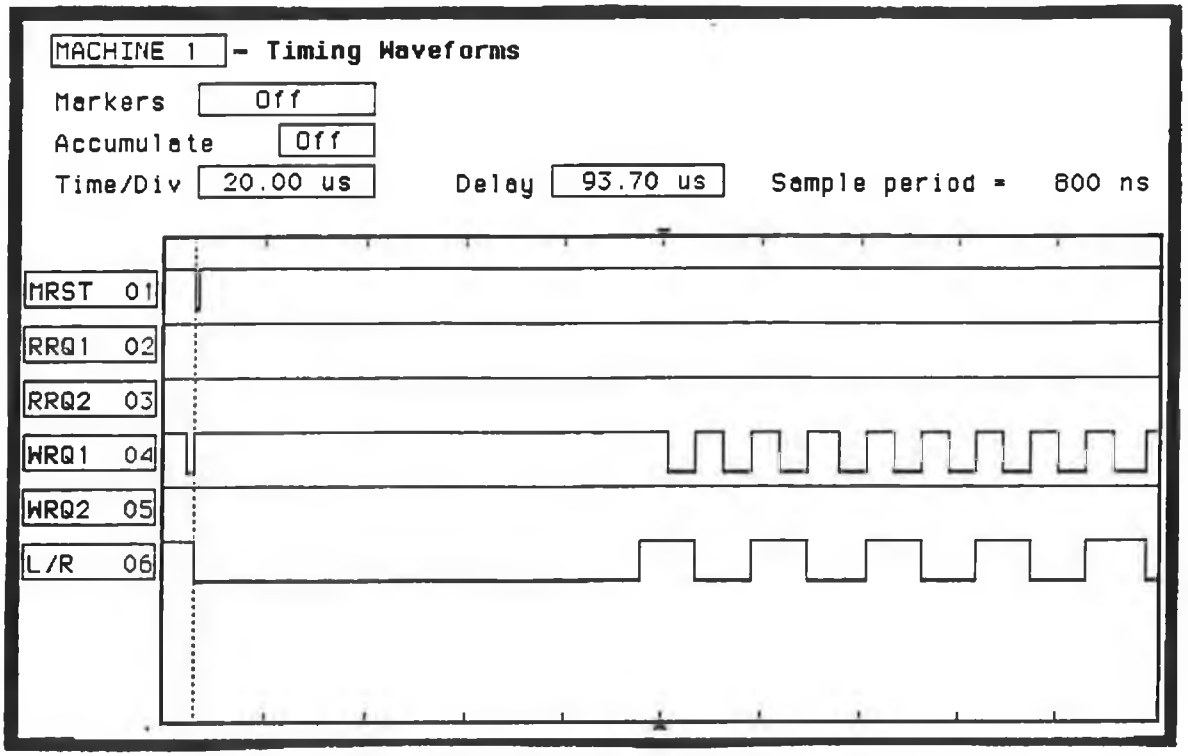


Figure 3.20 Wosc Requests.

3.5.3 Clock Generation

The clock signals required by the Serial Interface are generated in this section. The PiSo load clock is the most difficult to produce because it requires a low pulse for every sample with a maximum duration of 354 ns. This is generated using a one shot circuit triggered by the L/Rx2 clock (88.2 kHz) signal. The one shot circuit in the FPGA was composed of several logic gates and a digital counter run from an internal clock. The simulated waveforms showing the clocks required by the Serial Interface for the recording and playback modes are shown in Figure 3.21, while the actual waveforms are shown in Figure 3.22. Figure 3.18 illustrates channel consistency between recording and playback modes. The first read request occurs on the left channel while the first write request occurs on the right channel before it is shifted in the DTX or DAC on the left channel.

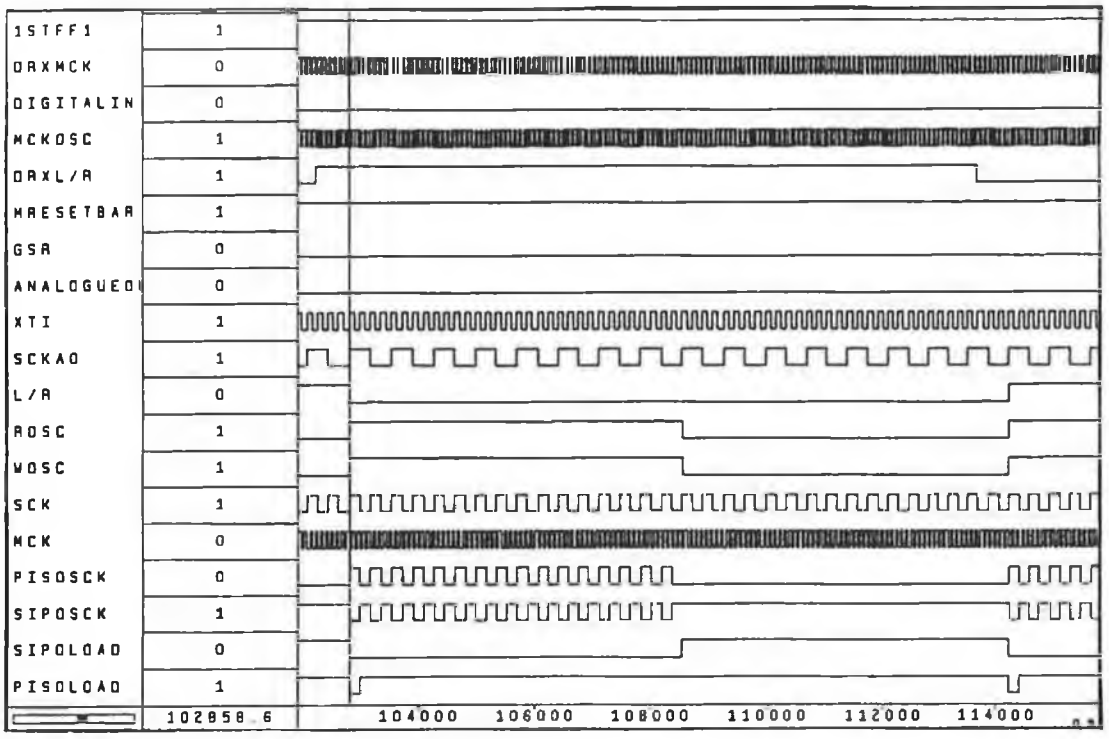


Figure 3.21 Simulated Parallel Load and Serial Shift Clocks for the Serial Interface.

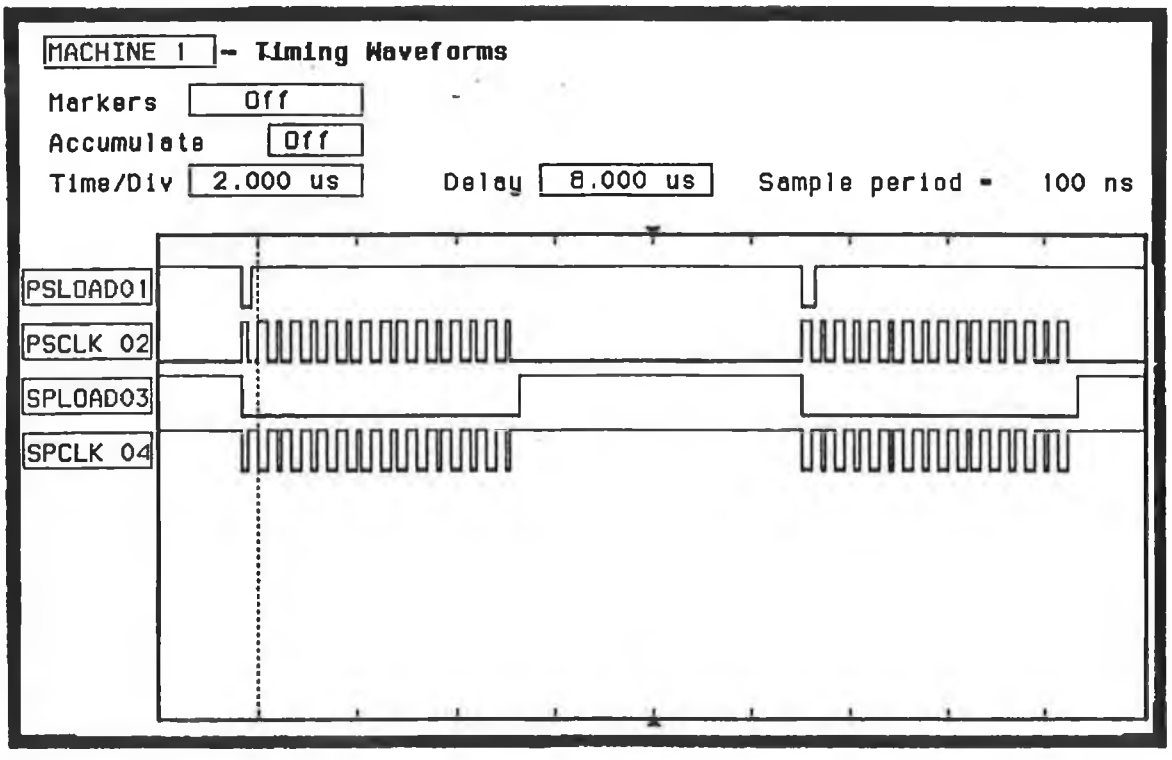


Figure 3.22 Parallel Load and Serial Shift Clocks for the Serial Interface.

3.5.4 Recording and Playback Tests

Once the functional blocks of the sound card had been individually tested, the recording and playback modes of the card were tested. The playback and recording tests described in the following sections are also required when identifying the predefined faults introduced in Chapter 7.

The hardware functionality was tested in the DOS environment for several reasons. Firstly, testing the card in the Windows environment would involve testing and developing the hardware and software simultaneously. This would be undesirable because this would greatly increase possible sources of errors. Secondly, hardware errors would be more difficult to identify due to the fact that the software cannot be guaranteed to be error free. Thirdly, a DOS program which records or plays back through the sound card is not tied to the restrictions imposed by Windows, such as the maximum size and priority levels required to access the DMA buffers. These factors make it undesirable to develop and test the sound card in the Windows environment.

3.5.4.1 Playback Tests

The files used in the playback tests were generated to provide a reference, against which the playback modes can be tested. The playback tests can be divided into three categories:-

- Predefined Playback : Involves playing back a test file and observing the outputs of the sound card. For the digital mode the output of the digital amplifier is observed. These test files allow the output to be compared against the known waveform of the test files. This test can easily verify channel consistency and data loss. If a test file has one channel muted, the loss of one sample will cause the waveform to switch channels.
- Data Transfer : Tracing individual samples through the data paths of the sound card, verifying data transfer.
- Audio : Playing back audio files, music or voice. These are preliminary tests carried out before the Predefined Playback tests to provide general information on the quality of the playback modes.

3.5.4.2 Recording Tests

Recording tests have an advantage over playback tests in that the recorded samples can be examined. Recording tests can be divided into similar categories as the playback tests:-

- Predefined Recording : This can only be performed with the analogue recording mode. It involves placing a waveform, generated by a signal generator, on one or both of the analogue inputs. The recorded waveforms can be compared against the waveforms on the analogue inputs. As explained previously, channel consistency and data loss can be tested with one input channel muted.
- Comparison Recording : This is restricted to the digital recording mode, because it requires the samples to be identical for every recording, a CD for example. The same piece of audio signal from a CD was recorded several times. Each time recording was started before the start of the audio signal. This guaranteed that the same reference point was captured for each recording. The samples before the start of the audio, whose values are zero, were removed from the recorded files. These modified files were then compared against each other for missing samples and different sample values.
- Data Transfer : Involves tracing samples through the data paths of the sound card, verifying data transfer.
- Audio : This involves playing back recorded files to give a general indication of the state of the recording modes of the sound card. This relies on the playback mode being correct because otherwise inaccurate conclusions could be drawn.

3.5.4.3 Analogue Quality Tests

The tests carried out in Sections 3.5.4.1 and 3.5.4.2 are functional tests and do not provide any information regarding the overall quality of recording and reproduction. These are primarily determined by the ADC and DAC chips, whose performance measurements are available from their data sheets. Although the sound card is a fully functioning 16 bit analogue and digital I/O card it is primarily intended for use as a teaching aid in a laboratory environment. Consequently quantitative paramemters such as Signal to Noise Ratio *(SNR)*, Frequency Response or Harmonic Distortion are not of major importance to the card function and are left as an exercise for the MTP's student audience.

3.6 Summary

This chapter describes in detail the functional blocks of the sound card and highlights the unique aspects of each. The testing procedures used in verifying the operating modes of the card are also described. The sound card's operation is verified with the selected timing diagrams shown in this chapter.

Chapter 4

Introduction to Windows C Programming

4.1 Introduction

This chapter provides an brief introduction to writing Windows applications in C source code. The major Windows programming issues are explained along with the various elements of a Windows application. This chapter assumes that the reader is familiar with the C programming language and with using Windows in general. In the software chapters, Chapter 4, 5 and 6, the following convention is used to describe functions:-

- Windows API functions are in *italics*
- functions called by Windows which the application must supply are in **bold** are
- functions specially written for the MTP are in ***bold and italics***.

4.2 The Windows Operating Environment

Microsoft Windows is a message based graphical environment where applications are represented by small graphical images. The user is no longer required to enter a command line to run these applications, instead applications are run from these graphical images. The Windows operating environment runs on top of DOS which it requires for its file I/O functions. The Windows environment controls all application access to the PC's hardware [14]. The Windows environment offers many advantages over the command line DOS environment, such as device independence for applications and a multi-application environment.

4.2.1 Non Preemptive Multi-tasking

Windows is not a true multi-tasking environment where tasks are allocated processor time depending on the tick of a hardware clock and are suspended when their time slice has elapsed. It is more accurate to call Windows a non-preemptive multi-application environment because it does not forcefully remove applications once their allocated time slice has elapsed. Windows relies on applications cooperating with the system and releasing control when inactive [14]. Windows applications can run in two states,

foreground and background. The application running in the foreground has the system focus and therefore receives most of the CPU's time and most of the keyboard input. Only one application can run in the foreground at a time while there may be several running in the background.

The *386 Enhanced* application in the *CONTROL PANEL* of the *MAIN* group in Windows, provides the ability to set the priorities of the foreground and background applications. The time slice is also determined here but these figures are only estimates as to the exact scheduling which occurs [12]. For a single PC, Windows offers a huge advantage over DOS in its ability to run more than one application simultaneously, providing fast switching between them along with a uniform application interface.

4.2.2 Application's Message Queue

Windows communicates with applications by sending messages to them. These messages are placed in the application's message queue waiting to be processed by the application [14]. The messages range from mouse and keyboard events to messages from other applications. The application can also send messages to Windows and to other applications using Windows API functions.

4.2.3 Application Programming Interfaces

Windows provides the Application Programming Interface (API) for applications to communicate and control the various hardware and software elements of the PC [11, 14]. The application can use the API functions for hardware independence. The API functions send standard messages to the device drivers which then access the hardware. These API functions also provide interfaces for the COM ports and for Windows Multimedia devices.

4.2.4 Dynamic Link Libraries

Dynamic Link Libraries (*DLLs*) are used extensively in Windows. The Windows operating system itself is composed of several DLLs. DLLs can contain functions, Windows resources and data which can be loaded, removed and reloaded into memory when required [14]. They are program modules which are only accessed by other modules at run time and therefore reduce the memory requirements for the system. Device drivers are also DLL modules.

4.3 Differences between DOS and Windows Applications

The C program for DOS in Listing 4.1 displays the text *Hello* on the PC's screen. The whole program is contained in a single file consisting of six lines. To display this text in the Windows environment requires more than sixty lines of code and three separate files. This may appear to be elaborate but the program created is substantially more versatile than the DOS program. The Windows program possesses all the usual properties of a Windows application, such as being capable of being moved and resized with the keyboard or mouse and having multiple copies active simultaneously. Most of the implementation of this functionality is hidden from the programmer in the Windows environment.

```
#include <stdio.h>
int main(void)
{
 printf("Hello");
 return (0);
}
```

***Listing 4.1** Program listing of HELLO.C.*

The fundamental component of a Windows application is where it processes its messages. This section is called the message processing loop [14]. Along with system messages, applications can send messages to each other. This concept of an application waiting for the system to send it messages is foreign to a DOS application programmer, but is typical of a graphical user interface such as Windows. A DOS program usually follows a procedural format whereas a Windows program uses an event based structure where it responds to user and system messages.

4.4 Windows Programming Conventions

One of the problems of programming in Windows is the wealth of new terminology, functions and structures in the Windows environment. Windows provides the programmer with over one thousand API functions, many of which are totally new to DOS based C programmers. An example is the standard Windows *SaveAs* dialog box common to all text editing packages. This function requires the programmer to initialise a Windows *OF* data structure and pass it to the *GetSaveFileName* function. Windows

will then manage the dialog box and return with *TRUE* if a filename was selected or *FALSE* if the operation was cancelled. If *TRUE* was returned then the *lpstrFile* field of the *OF* structure contains the selected file. These functions are powerful because of the control Windows possesses over the application's environment.

4.4.1 Main Window of Application

Windows is an object orientated environment and as such, the term *window* can have two meanings. Firstly, in the object oriented sense it refers to both the data and the code for the window. Secondly, in the visual sense it refers to the graphical representation on the PC's screen of a window. The term application's windows in this chapter, refers to the second graphical definition. The window for the MTP's application is shown in Figure 4.1.

***Figure 4.1** Main window of Application.*

The application can possess other windows, known as *CHILD* windows which are objects themselves because they contain data and code and are self contained. Examples of these windows are dialog boxes, scroll bars, edit windows and push buttons. The *PUSHBUTTON* child window for example changes its bitmap whenever the left mouse button is depressed while positioned over the push button without any direct intervention from the application.

4.4.2 Handles

Handles are used extensively in Windows to provide indirect access to Windows resources and objects. For instance, handles are initialised when resources are opened, such as when the waveform output device is opened and they are used to access the device or resource. For *C* DOS-based developers the only time handles are encountered is when accessing files using *C*'s file I/O functions or the *DOS INT 21h* interface functions.

Memory is also accessed through handles. The memory block must first be requested from the system with the *GlobalAlloc* function which if successful will return a handle to the memory block. The memory locations can only be directly accessed by locking the block in memory using the *GlobalLock* function with the memory block's handle passed as one of its arguments. This function will then return a pointer which can be used as a normal memory pointer. The pointer locates the beginning of the memory block. When the block is not locked in memory, the handle allows Windows to move the block around or swap it to disk, without having to continually update pointers [14].

4.4.3 Functions and Variables

Windows introduces a new naming convention and some new variables for programming in Windows. The naming convention is called *HUNGARIAN* notation where the prefix of the variable name describes the variable type. For instance a variable called *dwDuration* is a 'double word' *(dw)* variable while *lpszString* is a 'long pointer to a string terminated by a zero' *(lpsz)*. Appendix G contains a list of the most common hungarian notation prefixes and the new Windows variables. Function and variable names in Windows can be quite large, and capital letters are strategically introduced to make them more easily readable. For example the function *waveOutUnprepareHeader* is easier to read than *waveoutunpreparehead er.*

All Windows functions use the *PASCAL* calling convention for parameter passing, therefore functions called by Windows must also use this convention and are declared with the *PASCAL* keyword. The *PASCAL* calling convention is different to the normal *C* convention in that the order in which parameters are passed on the stack differs. The *PASCAL* convention pushes the parameters on to the stack in order, from left to right, which is opposite to the *C* convention. Furthermore the *PASCAL* convention relies on the called function to remove the parameters from the stack before

returning, whereas with the C convention the function which issued the call cleans up the stack after the called function returns.

Functions which Windows calls must be declared with *the EXTERN* keyword or be declared in the *EXPORTS* section of the application's module definition file [3, 14]. These functions must also use the *FAR* keyword in their declaration because they reside in a different code segment. Functions which reside in DLLs must also follow these guidelines.

4.5 The MTP's Windows Application

A Windows application is composed of several files whose responsibilities range from creating the application's window to determining how its various code and data segments are dealt with in memory. The files required by most applications are the C source code files for the application window and dialog boxes, the resource script file, the module definition file and the project file which links all these different files together [3]. The files which contain the source code for the applications window and dialog boxes are explained in the first two sections while the other files required to build the application are explained later. Table 4.1 describes the files required to build a Windows application.

4.5.1 Source Code for Main Window of the MTP's Application

The application window's source code is responsible for creating the application's window and processing the application's messages and for this project is contained in the file, *MMAPP.C*. The source code for the application in the Borland (TurboC++) and Microsoft (Visual C++) environments are on the two disks which accompany this thesis. The Borland source code is on the application's installation disk in the BSOURCE directory while the Microsoft code is on the device driver's installation disk in the MSOURCE directory. The application's two principal functions are **WinMain** and **MainWndProc**, which create the application's window and process its messages respectively [14].

The **WinMain** function is the Windows equivalent of the *MAIN* function in C for DOS and is the first function called by the system when the application is started. This function creates, initialises and displays the application's window and specifies the procedure which will process its messages.

Type	File Description
SOURCE.C	Contains source code for processing the application's messages.
HEADER.H	Contains the application's identifiers for its resource elements.
RESOURCE.RC	Windows resources use by the application are defined or referenced in this file.
MODULE.DEF	Defines how Windows handles the application's code and data segments in memory .
HELP.HLP	Help file for the application.
PROJECT.PRJ	Project file for compiling and linking the application's files in the Borland TurboC++ development environment.
MAKEFILE.MAK	Project file for compiling and linking the application's files in the Microsoft Visual C++ development environment.

***Table 4.1** Application's File Types.*

There are several steps involved in setting up the application's window which involves calling Windows functions and initialising Window structures required by these functions. The first step involved in displaying the application's window is to register the window class with the function *RegisterClass*, but only if this application is the first instance of the application. The *hInstance* variable is examined to determine if there are other instances of the application active. There can be several instances of the same applications simultaneously active within Windows. This function takes the Windows structure *WNDCLASS* as its sole argument. This structure is defined in the *WINDOWS.H* file and defined within this structure is the application's icon, mouse cursor, the window's background pattern and its message processing function, **MainWndProc**. The different elements of this structure are illustrated in Listing 4.2. The *RegisterClass* function returns the value *TRUE* (one) when successful and *FALSE* (zero) otherwise.

Secondly, the window must be created using the *CreateWindow* function. This function has eleven parameters passed to it ranging from the application's window title and style, to the position and size of the window and it returns the handle of the

```
wndclass.style          = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc    = (LPVOID)MainWndProc;    // Applications.
wndclass.cbClsExtra     = 0;              // Processing function.
wndclass.cbWndExtra     = 0;
wndclass.hInstance      = hInstance;
wndclass.hCursor        = LoadCursor(hInst,IDC_ARROW );
                                    // Application's cursor.
wndclass.hIcon          = LoadIcon( hInstance,
                            MAKEINTRESOURCE(MMAPPICON) );
                                      // Application's icon.
wndclass.hbrBackground  = GetStockObject(WHITE_BRUSH);
                          // Application's background colour.
wndclass.lpszMenuName   = "MMAPP";      // Application's menu bar.
wndclass.lpszClassName  = szAppName;
```

Listing 4.2. *Windows window class structure, WNDCLASS.*

window. Thirdly, the window must be displayed using the *ShowWindow* function. This function only displays the window and it does not display the window's *client area.* The *client area* is the area enclosed by the windows border and menu bar. The *ShowWindow* function takes the handle of the window along with a command string describing how the window is to be displayed, normally or minimised. The *client area* is displayed by the application posting a *WM_PAINT* message to itself using the *UpdateWindow* function. The applications message processing routine will then display the *MMAPP.BMP* bitmap in the window's client area as shown in Figure 4.1.

The next part of the **WinMain** function is known as the application's message loop. This loop consists of four functions, *GetMessage*, *TranslateAccelerator*, *TranslateMessage*, *DispatchMessage*. This loop constantly polls the application's message queue using the *GetMessage* function. Once a message has been received, the *TranslateAccelerator* function translates the keyboard's *VM_KEYUP* and *WM_KEYDOWN* message sequence into the appropriate *WM_SYSCOMAND* or *WM_COMMAND* messages. The *TranslateMessage* function provides keyboard translation from the physical keyboard to the virtual keyboard which the application employs. The *DispatchMessage* returns the message back to Windows to be sent to the application's **MainWndProc** procedure for processing.

These functions all use the Window's message structure as one of their parameters. This structure is shown in Listing 4.3. The first element of the message structure is the handle to the target window while the second is the message number. The message itself is determined by the next three elements. The last two elements relate

```
typedef struct tag MSG {
        HWND            hwnd;           // Handle of target window.
        UINT            message;        // The message identifier.
        WPARAM          wParam;         // Message parameters.
        LPARAM          lParam;
        DWORD           time;           // Time of message.
        POINT           pt;             // Mouse position at time
}                                       // of message.
MSG ;
```

Listing 4.3. *Windows message structure, MSG.*

to the time the message was posted and the mouse cursor position in screen coordinates at that particular time.

The **MainWndProc** function is responsible for processing the application's messages. It is specified as the message processing procedure in the application's *WNDCLASS* structure defined in the **WinMain** function. For this project the message processing structure of **MainWndProc** is shown in Listing 4.4. Four parameters are passed to this function, the window's handle, the message number and the *wParam* (16 bit) and *lParam* (32 bit) integers. Messages can be from a number of different sources, mouse, keyboard, timer or from the low level waveform device driver which have the *MM_* prefix, *MM_WIM_CLOSE* for example. The *WM_COMMAND* message is sent whenever a menu item is selected or a control item is accessed, for example an edit window in a dialog box. The low word of the 32 bit *lParam* integer is examined to determine the menu item selected. This value must be non-zero. The macro *LOWORD* extracts the low order word.

```
long FAR PASCAL MainWndProc(HWND hWnd, WORD wMessage,
                                        WORD wParam, LONG lParam)
{
   switch(message)                              // Message processing.
   {
       case WM_INITDIALOG :
       break;
       case MM_WIM_OPEN :           // Waveform input device opened.
       break;
       case MM_WIM_DATA :
       break; // Waveheader returned from the waveform input device.
       case MM_WIM_CLOSE :          // Waveform input device closed.
       break;
       case MM_WOM_OPEN :           // Waveform output device opened.
       break;
       case MM_WOM_DONE :
       break; // Waveheader returned from the waveform output device.
       case MM_WOM_CLOSE :          // Waveform output device closed.
       break;
       case WM_COMMAND :                        // Pull-down messages.
       if(!LOWORD(lParam)) {   // Identify which pull-down menu item
                                                    was selected.
           switch(wParam) {
           case  IDM_OPEN :
           break;
           case  IDM_SAVEAS :   // Select default file for recording.
           break;
           case IDM_WAVEIN :   // Waveform input device capabilities.
           break;
           case IDM_WAVEOUT : // Waveform output device capabilities.
           break;
           case IDM_APPLICATION :        // Application information.
           break;
           case IDM_PLAYANALOGUE :   // Analogue playback dialog box.
           break;
           case IDM_PLAYDIGITAL :    // Digital playback dialog box.
           break;
           case IDM_RECORDANALOGUE :// Analogue recording dialog box.
           break;
           case IDM_RECORDDIGITAL :  // Digital recording dialog box.
           break;
           case IDM_YES :                       // Exit application?
           break;
           case IDM_NO :                    // Cancel exit command.
           break;

           default:
           break;
           }
       }
       break;
       case WM_SIZE  :  // Application's window size has been changed.
       return 0;
       case WM_PAINT :                // Client area needs redrawing.
       break;
       case WM_DESTROY :        // Sent to the application after its
                          // window has been removed from the screen.
       break;
       default :                    // Message is processed anyway to
                                    // Maintain good house keeping.
       return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0L;
}
```

Listing 4.4. *Message processing procedure, WndMainProc, for the Application.*

Each menu item is identified by a unique number defined in the application's header file, *MMAPP.H* and this identifier is also used in the resource script when declaring the menu items. The *lParam* of the *MSG* structure will be set to this number when the menu item has been selected by the user. The case switch structure identifies the particular menu item and the message is then processed. This usually involves calling a dialog box. The *WM_SYSCOMMAND* messages relate to control box messages, such as *client area* repaint, minimising and maximising the application's window. The control box is situated in the top left hand corner of the windows or dialog box's caption bar. This box controls the position and state of the window. In the case of a *WM_PAINT* message being received, the window's *client area* is redrawn.

For the *WM_DESTROY* message, the function *PostQuitMessage* sends a message to Windows requesting termination of the application. After each message has been processed, control is returned to the **WinMain** message loop. If there is no match within the switch case statements, the message is returned to Windows for processing by the *DefWindowProc* procedure to ensure that all message are processed..

4.5.2 Source Code for Dialog Boxes

The source code for a dialog box is very similar to the application's window code and is composed of an initialisation and display function, and a message processing function [14]. The first function, ***fnRecdDigt***, is called by the application window's message processing procedure, **MainWndProc**. Listing 4.5 shows the dialog box's main function, ***fnRecdDigt***. The dialog box's message processing routine is ***fnRecdDigtDlgProc***, which is passed as a parameter to the Windows function, *MakeProcInstance*. This function returns the address of the ***fnRecdDigtDlgProc*** function's *PROLOG* code. This code is required by Windows to establish the function's *DATA* segment when it is called by Windows to process the dialog box's messages.

The Windows function *DialogBox* displays the dialog box and links the message processing procedure to this dialog box. The message processing function's *PROLOG* code pointer *lpfnRecDigtDlgProc* is passed along with the application's main window handle, application's instance handle and the dialog box's resource name. This function does not return until the dialog box has been closed.

The structure of the message processing loop for the digital recording dialog box is shown in Listing 4.6. and consists of a *CASE SWITCH* structure to identify the

```
int fnRecdDigt(HWND hParentWnd)
{
   int   RetCode;
   FARPROC  lpfnRecdDigtDlgProc;

   hPrtWnd=hParentWnd;     // Pointer to dialog box's processing function.
   lpfnRecdDigtDlgProc = MakeProcInstance((FARPROC)fnRecdDigtDlgProc,
                                                              hInst);
   if((RetCode = DialogBox(hInst, "RecordDigital", hParentWnd,
                                        lpfnRecdDigtDlgProc)) == -1) {
        MessageBox(NULL, "Unable to display dialog", "System Error",
                   MB_SYSTEMMODAL | MB_ICONHAND | MB_OK);
        return FALSE;
   }// Display the dialog box. Function returns when dialog box is closed.
   FreeProcInstance((FARPROC)lpfnRecdDigtDlgProc);
   return(RetCode);
}
```

Listing 4.5. *Main dialog box function fnRecdDigt, for the Digital Recording dialog box.*

message. The *WM_COMMAND* message is sent when one of the dialog boxes' control elements is accessed. To determine the sender of the message the *wParam* is examined using another *CASE SWITCH* structure. The dialog box controls are described in the resource script and are assigned designators in the application's header file, *MMAPP.H* and the message is compared with these designators to identify the message.

4.5.3 The Resource Script

The Windows resources used by the application are defined in the resource script. Window resources are the application's icons, bitmaps, menu bar, pull-down menus and dialog boxes. The resource script must have the *.RC* file extension. The different resources defined in the application's resource file can be produced by the visual resource editing packages from Borland or Microsoft which accompany their Windows programming packages [1]. These editors produce the application's resource script and the different resources are described in the following sections.

```
BOOL FAR PASCAL _export fnRecdDigtDlgProc(HWND hDlg, WORD message,
                                          WORD wParam, LONG lParam)
{
    switch(message)
    {
       case WM_INITDIALOG :
       break;                      // Initialisation of dialog box.
       case WM_HSCROLL :
       break;                    // Processing scroll bar messages.
       case WM_COMMAND :
           switch(wParam) {
               case RecdDigt_Record :
               break;               // Record push button pressed.
               case RecdDigt_Cancel :
               break;               // Cancel push button pressed.
               case RecdDigt_FileName :
               break;             // Filename push button pressed.
               case RecdDigt_Duration :
               break;             // Duration EDIT window accessed.
               default :
               return FALSE;
           }
       case WM_SYSCOMMAND :
           switch(wParam & 0xFFF0) {
               case SC_CLOSE :
              break;   // Dialog box closed from its Control Box.
           }
   }
   return FALSE;
}
```

Listing 4.6. *Message processing structure for the Digital Recording Dialog Box.*

4.5.3.1 The Dialog Box

The dialog box's appearance is determined by the resources attributed to it in the resource file. The dialog box's physical dimensions, push buttons, bitmaps, scroll bars, edit windows and any other resources are declared in the resource file [1]. The dialog box can also be defined in a separate file by including it in the resource file using the *C #INCLUDE* statement. External dialog box definition files can usually be identified by their *.DLG* file extensions.

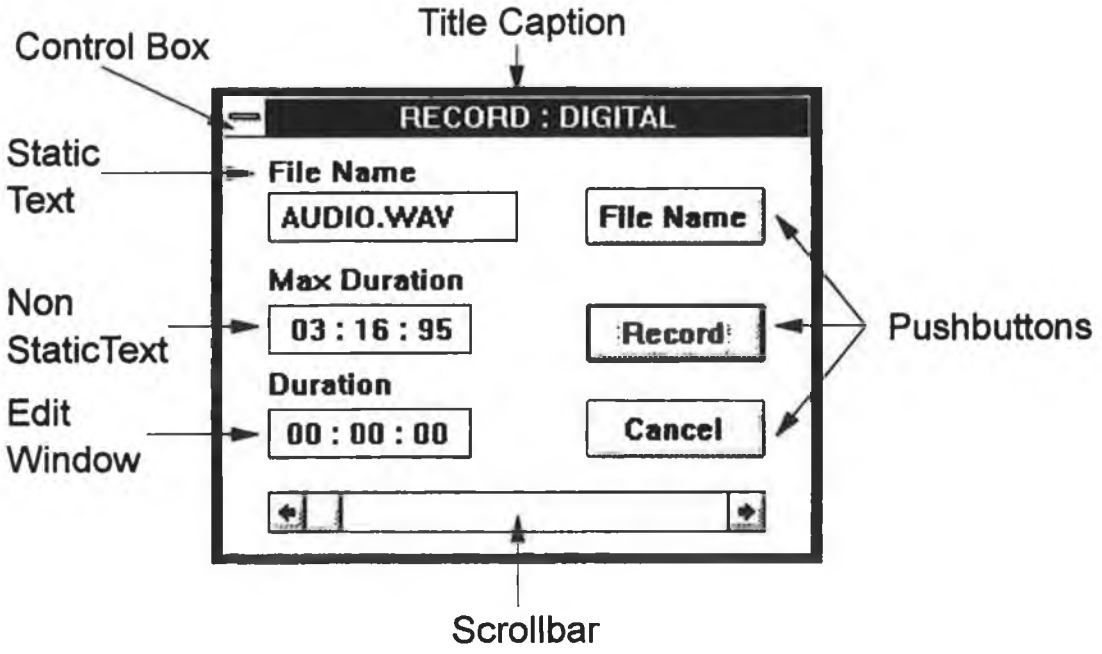


Figure 4.2. The Digital Recording Dialog Box.

```
RecordDigital DIALOG 18, 18, 130, 100
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "RECORD : DIGITAL"
BEGIN                                                  // Default push button.
      DEFPUSHBUTTON "Record", RecdDigt_Record, 80, 40, 40, 14
      PUSHBUTTON "File Name", RecdDigt_FileName 80, 12, 40, 14
      PUSHBUTTON "Cancel", RecdDigt_Cancel, 80, 63, 40, 14
      CONTROL " 00 : 00 : 00 ", RecdDigt_Duration, "EDIT",
            E_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP
            10, 65, 45, 12
                              // Edit window for entering record duration.
      LTEXT "File Name", -1, 10, 4, 30, 8     // Static text, left aligned.
      LTEXT "Duration", -1, 10, 55, 30, 8
      LTEXT "Max Duration",-1,10,30,50,8
                                          // To be updated at display time.
      LTEXT "", RecdDigt_FileText, 13, 15, 50 , 8
      LTEXT "", RecdDigt_DiskSpace, 15, 42, 39 , 8

      CONTROL "", -1, "static", SS_BLACKFRAME | WS_CHILD |
            WS_VISIBLE, 10, 13, 55, 12
      CONTROL "", -1, "static", SS_BLACKFRAME | WS_CHILD |
            WS_VISIBLE, 10, 40, 45, 12

      SCROLLBAR RecdDigt_ScrollBar,10,85,110,10
END
```

***Listing 4.7.** Resource Script definition of the Digital Recording Dialog Box.*

The digital recording dialog box is illustrated in Figure 4.2 while its definition is shown in Listing 4.7. The resource is defined as a dialog box by the ***DIALOG*** statement in the first line of Listing 4.7 and is called ***RecordDigital***. ***RecordDigital*** is used by the Windows *DialogBox* function to display this dialog box [14]. The position and size of the dialog box are defined by the sequence of numbers at the end of the first line of the listing. The first pair of dimensions refer to the display position of the dialog box's top left hand corner in relation to the screen's top left-hand corner, while the second pair refer to the subsequent width and height of the dialog box. All elements of the dialog box requiring position and size use this format. These units refer to Dialog Box Units (*DLUs*), which are derived from the standard 8-bit font. A standard character will be 8 DLUs wide and high. The style of the dialog box is defined in the second line. For example the ***WS_SYSMENU*** flag declares that the dialog box has a control box in its top left hand corner for closing or moving the dialog box. The caption for the dialog box's caption bar is defined in the next line.

Two push buttons are defined next using the *PUSHBUTTON* statement with the record push button declared as the default one (***DEFPUSHBUTTON***). This default push button is the push button which has the input focus when the dialog box is initially displayed. The *PUSHBUTTON* statement defines the push button's text and position in relation to the dialog box's top left hand corner. The components of the dialog box are also known as controls, which can be static or non static. Push buttons and scroll bars are examples of non static controls while the rectangle around the file name is a static control.

All non-static dialog components are referenced with an identifier which is assigned a unique number defined in the application's header file. For example the *RECORD* push button's identifier ***RecdDigt_Record*** is assigned the value 150 in the header file. This value will be one of the parameters of the message sent to the dialog box's message processing function whenever this button is pressed.

The text displayed in the dialog box is defined using the ***LTEXT*** statement. This statement aligns the text to its left horizontal coordinate while the statements RTEXT and CTEXT, right align and centre the text respectively. The text which is not assigned the value *-1* as its identifier, is text which can be changed by the application. For example the available recording duration cannot be determined in advance so it must be updated when the dialog box is initially displayed and when a recording has taken place.

The Windows functions *GetDlgItem* and *SetWindowText* are used to update the dialog box's text items.

The ***CONTROL*** and ***"EDIT"*** keywords define a text entry field or edit window. This allows the user to enter text into this window through the keyboard when the window has the input focus. An edit window is most commonly encountered in word processing packages. This edit window allows the user to enter the desired recording length more accurately than the scroll bar will allow. This edit window is also a *CHILD* window because it can receive and send messages to other windows. The dialog box has no control over this window. The edit window receives keyboard messages if it has the input focus and will display what the user has entered. Each time it is accessed it informs the dialog box by sending messages to it. The flags accompanying the *CONTROL* statement define the type and the appearance of the edit window. The *CONTROL* statement is also used to define two black rectangles which are non changeable (***-1***) and static, ***"static"***. These rectangles are below the *FILE NAME* and the *MAX DURATION* text as illustrated in Figure 4.2.

4.5.3.2 Menu Bar and Accelerator Table

The application's menu bar is composed of the following items, *FILE*, *PLAY*, *RECORD*, *INFO* (information), *HELP* and *EXIT*. These items possess a pull-down menu with two or more elements. These pull-down menu items will in turn call a dialog box. The definition of the ***PLAY*** menu bar item is shown in Listing 4.8 along with the section of the accelerator table which defines the *PLAY* menu items short cut key or accelerator keys [1].

The ***POPUP "&Play"*** statement highlighted in Listing 4.8. defines a menu bar item while the pull-down elements of this menu item are declared between the following ***BEGIN*** and ***END*** statements. The ampersand '**&**' preceding the letter in the menu bar item's name determines which letter in conjunction with the ALT key selects this pull-down menu, *ALT+P* in this case. The ***MENUITEM*** statement defines the pull-down menu items and their identifiers. These identifiers are declared in the application's header file and are used by the application's *Message Loop* to determine which pull-down menu element was selected. When the pull-down menu is active the ampersand in the pull-down menu name determines which letter selects the pull-down menu item. The three dots after the name of the pull-down item is a standard way of notifying the user

```
MMAPP          MENU                    // Menu Bar and pull-down menu definitions.
BEGIN
       POPUP "&Play"                                  // Definition of menu bar item.
       BEGIN                                 // Definitions of pull-down menu items.
              MENUITEM "&Analogue...\tAlt+A", IDM_PLAYANALOGUE
              MENUITEM SEPARATOR
              MENUITEM "&Digital...\tAlt+D", IDM_PLAYDIGITAL
       END
END

MMAPP ACCELERATORS                                          // Accelerator Table.
BEGIN
     // Definition of accelerator keys for analogue play pull-down menu item.
       "A",                  IDM_PLAYANALOGUE, VIRTKEY , ALT
       "D",                  IDM_PLAYDIGITAL, VIRTKEY , ALT
END
```

Listing 4.8. *Resource Script definition for the Application's 'Play' Pull-Down Menu.*

that the item calls a dialog box for requesting further information from the user. The ***MENUITEM SEPARATOR*** statement introduces a line dividing the previous and subsequent pull-down menu items. The menu definition can also reside in an external file which is included in the resource script using the *#INCLUDE* statement. This file usually has the file extension *.MNU*.

Accelerator keys are key combinations which select the pull-down menu element whether the pull-down menu is active or not. They are defined in the *Accelerator Table*, ***MMAPP ACCELERATORS*** section of the resource script, part of which is shown in Listing 4.8. Accelerator keys are usually a simultaneous combination of a letter of the alphabet and one or more of the *SHIFT*, *CONTROL* and *ALT* keys. They are usually included in the pull-down menu item's name, for example ***ALT+A*** tells the user the key combination for the analogue playback dialog box.

4.5.3.3 Icons and Bitmaps

The icons and bitmaps employed by the application are stored in external files which are declared with the *ICON* or *BITMAP* statements respectively in the resource script as follows. *COMPACTSYM ICON COMPACT.ICO*

MMAPPBMP BITMAP MMAPP.BMP

These files are in the Windows standard icon or bitmap formats and can be created by many graphics packages. The icon's files have the *.ICO* file extensions while the bitmaps have the *.BMP* extensions.

4.5.3.4 String Table

This is an array of character strings which the application can use for displaying information. The strings have a unique identifier defined in the application's header file which the application requires to retrieves the string from the resource script with the *LoadResource* function. The application does not possess a *String Table* but the device driver, which is a Windows program, does contain a String Table in its resource script.

4.5.4 The Module Definition File

The applications module definition file [14] determines the manner in which the application's code and data segments are handled in memory. The different code and data segments can be defined as combinations of one or more of the following modifiers, *PRELOAD, MOVEABLE, DISCARDABLE, MULTIPLE* and *FIXED*. The *PRELOAD* modifier specifies that the segment is loaded into memory once the application is run. With the *MOVEABLE* modifier selected, the segment can be moved about in memory by Windows when necessary. This may be due to the segment not being used for a long time if the application is running in the background or to release memory for use by other applications.

The *DISCARDABLE* modifier specifies that the segment does not have to be saved to disk if it is removed from memory, it can simply be reloaded from the application when required again. This is reserved for segments which are never modified such as code and constant data segments. The *MULTIPLE* modifier specifies that more than one copy of the segment can exist in memory. This is reserved for data segments where each must be exclusive to the particular instance of the application.

When the *FIXED* modifier is used, once the segment is loaded, it cannot be moved by Windows until the application is closed. The application's module definition file is shown in Listing 4.9. with the data segment described as *FIXED* and the code segment as *DISCARDABLE PRELOAD.*

The sizes of the *LOCAL HEAP* and *STACK* are also defined in this file. The *STACK* size for a Window's application is much larger than for a DOS based

```
NAME                MMAPP
DESCRIPTION         'Multimedia Teaching Platform'
EXETYPE             WINDOWS
STUB                'WINSTUB.EXE'
CODE                DISCARDABLE PRELOAD
DATA                FIXED
HEAPSIZE            8192
STACKSIZE           40000
EXPORTS
        MAINWNDPROC                     @1
        FNPLAYANLGDLGPROC               @2
        FNRECDDIGTDLGPROC               @3
        FNPLAYDIGTDLGPROC               @4
        FNRECDANLGDLGPROC               @5
        FNAPPLICATIONINFODLGPROC        @6
```

Listing 4.9 Application's Module Definition file, MMAPP.DEF.

application due to the re-entrant nature of the message structure used in Windows. For this reason the *STACK* size is set at 40000 bytes. The *LOCAL HEAP* is the memory that the application can allocate from its local memory while the application is running. Its size depends very much on the memory requirements of the application.

The modal definition file also contains the line *STUB 'Windows Stub'*. Whenever an attempt is made to run the application from the DOS environment, this stub stops the application from running and displays a text message that the application requires the Windows environment to run [14]. Defined in the *EXPORTS* section are the functions that can be called by other applications, including Windows. These are the message processing functions for the application's windows and it's dialog boxes.

4.5.5 Header File

The application's header file MMAPP.H, contains the identifiers for the dialog box's control elements, menu bar items, pull-down menu items and any other definitions required by the application. This file is included in the C source files along with any other header files required, *WINDOWS.H* for example.

4.5.6 Project File

This file is used to compile and link all the separate files together within the chosen programming environment. The Borland TurboC++ project file has a .PRJ extension

while the Microsoft Visual C++ project file has the .MAK extension. The TurboC++ version of the application's project file is illustrated in Figure 4.3. The files can be compiled separately using the different command line switches and then linked together again using command line switches to produce the Windows executable file but the scope for errors is much greater than if the application is compiled and linked within the Windows environment [3]. All that is involved in creating the application's executable file is selecting the correct compiler and linker options and then compiling and linking the application. The compiler options range from the memory model that the application uses to the default parameter passing conventions used.

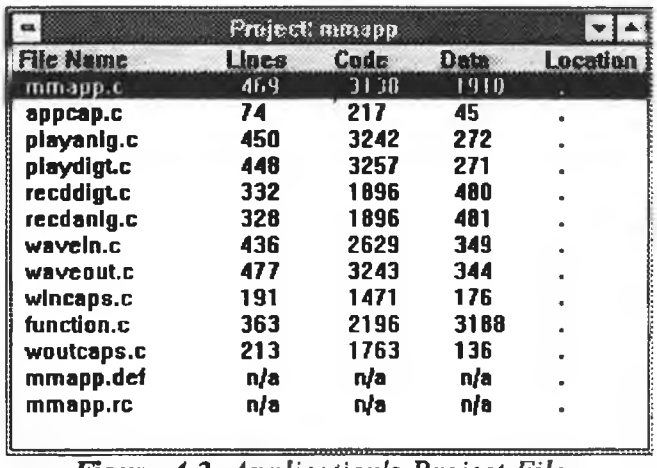


Figure 4.3. *Application's Project File.*

4.5.7 Help File

This file while not required to generate a Windows application, is nevertheless an essential part of every application's user interface. The Windows environment also introduces a standard format for help files. These files usually have the *.HLP* extension. There is also a standard application which allows the user to navigate through the help file, *WINHELP.EXE* situated in the main Windows directory [15]. The process of generating the help file is described in Appendix H. The application can call this help file application using the Windows *WinHelp* function and request which help file to load. The contents page of the help file, *MPTHELP.HLP* can be called as follows;

WinHelp(hWnd, "mtphelp.hlp", HELP_CONTENTS, 0);

In this thesis, the application's help file contains bitmap images of the recording and playback dialog boxes with pop-up boxes explaining the purpose of each of their controls. The contents page also provides pop-up boxes to explain the application's menu bar.

4.6 Windows Multimedia

The Windows API multimedia functions were first introduced as extensions to the Windows 3.0 environment in 1991. They were integrated into the Windows environment with the introduction of Windows 3.1 [11]. The multimedia functions are contained in the Windows module MMSYSTEM, which is contained in the DLL, *MMSYSTEM.DLL* situated in the *SYSTEM* subdirectory of the main Windows directory. The MMSYSTEM module is composed of the Multimedia Control Interface *(MCI)*, and the low level functions for audio and MIDI as illustrated in Figure 1.3.

The MCI provides the high level multimedia functions which perform complete multimedia operations with one function call, such as playing an audio file. The multimedia device drivers are contained in the SYSTEM subdirectory of the Windows directory and are listed in Table 4.2. The Windows file *SYSTEM.INI* determines which drivers are loaded into the Windows operating environment at Windows start-up time. The multimedia device drivers are declared in the *[mci]* section of the *SYSTEM.INI* file, while the audio and MIDI device drivers are declared in the *[drivers]* as illustrated in Listing 4.10 [11].

The low level waveform (analogue) audio and MIDI functions allow more control of the waveform and MIDI device because they access the device driver directly and not indirectly through the MCI device driver. They require several different functions to perform the same operation as one of the high level commands.

Media Type	Device Driver
CD Audio	mcicda.drv
Sequence	mciseq.drv
Waveform Audio	mciwave.drv
Audio and Video	mciavi.drv

***Table 4.2** Multimedia Device Drivers.*

```
[mci]
CDAudio=mcicda.drv                    // Compact Disc driver.
Sequencer=mciseq.drv
WaveAudio=mciwave.drv          // Multimedia Waveform driver.
AVIVideo=mciavi.drv  // Multimedia Audio and Video driver.

[drivers]
midimapper=midimap.drv
timer=timer.drv                                  // System Timer.
WAVE=mmteach.drv // Multimedia Teaching Platform's driver.
```

***Listing 4.10** Declaration of the Multimedia Device Drivers in SYSTEM.INI.*

4.7 Summary

This chapter introduces the basic programming and operating principles of Windows. Fundamental examples of functions that a Windows application must provide are explained along with the files required by a Windows application and the Multimedia section of the Windows operating environment are also explained.

Chapter 5

The MTP's Windows Application

5.1 Introduction

This chapter describes the Windows application developed to play and record *RIFF WAVE* files through the sound card. The application allows the user to record and play analogue and digital audio files to and from the hard disk of the PC using the multimedia low level audio functions provided by the Windows API [11]. The application is written specifically to take advantage of the sound card's digital audio capabilities.

The analogue audio mode of the sound card is known as waveform audio in Windows Multimedia and the hardware device which produces waveform audio is known as the waveform device. The waveform device can also be split into waveform input and output devices.

5.2 The Windows Application

The Windows application allows recording and playback of Windows' *RIFF WAVE* (*.WAV*) formatted files. The structure of WAVE files is explained in Appendix F. The menu structure of the application is shown in Figure 5.1. All pull-down menu items invoke dialog boxes except for the *NO* pull-down menu item of the *EXIT* menu item which cancels the exit command and returns control to the application's window. The

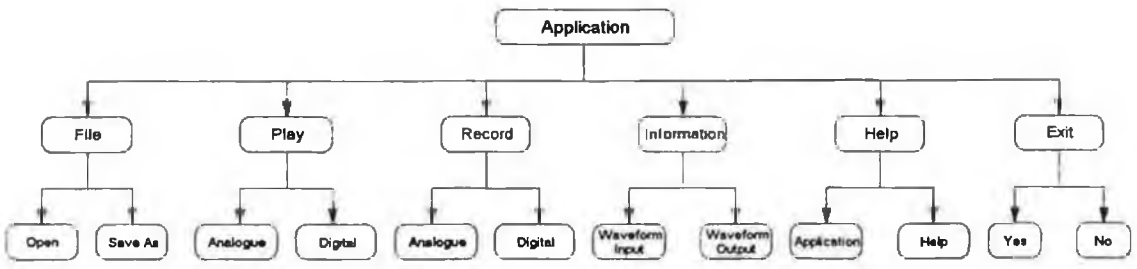


Figure 5.1. Structure of Application.

dialog boxes range from waveform device information to recording and playing files. The files which compose the application are listed in Table 5.1 along with a brief description of each. The application's source code for the Borland and Microsoft environments can be found on the two disks which accompany this thesis. The application communicates with the waveform device driver through the multimedia API functions provided by the Windows MMSYSTEM module, while the waveform device driver will control the sound card in response to these messages.

File	Description
MMAPP.PRJ	Project file (Borland TurboC++ environment).
MMAPP.C	Source code for application's main window.
MMAPP.RC	Resource script.
MMAPP.DEF	Module definition file.
MMAPP.H	Header file.
VARIABLE.H	MMAPP.C header file for global variable definitions.
APPCAP.C	Source file for the information dialog box.
WINCAPS.C	Source code for the waveform device's input capabilities dialog box.
WOUTCAPS.C	Source code for the waveform device's output capabilities dialog box.
WAVEIN.C	Recording functions.
WAVEOUT.C	Playback functions.
RECDDIGT.C	Source code for the digital recording dialog box.
RECDANLG.C	Source code for the analogue recording dialog box.
PLAYDIGT.C	Source code for the digital playback dialog box.
PLAYANLG.C	Source code for the analogue playback dialog box.
FUNCTION.C	Other functions required by the application.

Table 5.1. *Description of Application's File.*

5.3 The API's Low Level Audio Functions

The WAVE files can be played or recorded using the MMSYSTEM's high or low level audio functions. The high level commands [11] can accomplish recording or playback with a single function whereas the low level commands [11] require several functions as illustrated in Figure 5.2. These high level audio commands use the standard Windows Multimedia waveform driver, *MCIWAVE.DRV*, to control the waveform device via the MTP's device driver, *MMTEACH.DRV*. This hides the actual process of playing and recording files. The low level commands directly access the MTP's device driver, therefore providing more control over playback and recording than the higher level functions.

The low level commands place the responsibility on the application to control the waveform device. This involves opening the waveform device, closing the device and maintaining a continuous data flow to the device. Using the low level functions provides a greater insight into Windows Multimedia than the high level functions. The following subsections describe the responsibilities imposed on the application when using the low level audio functions.

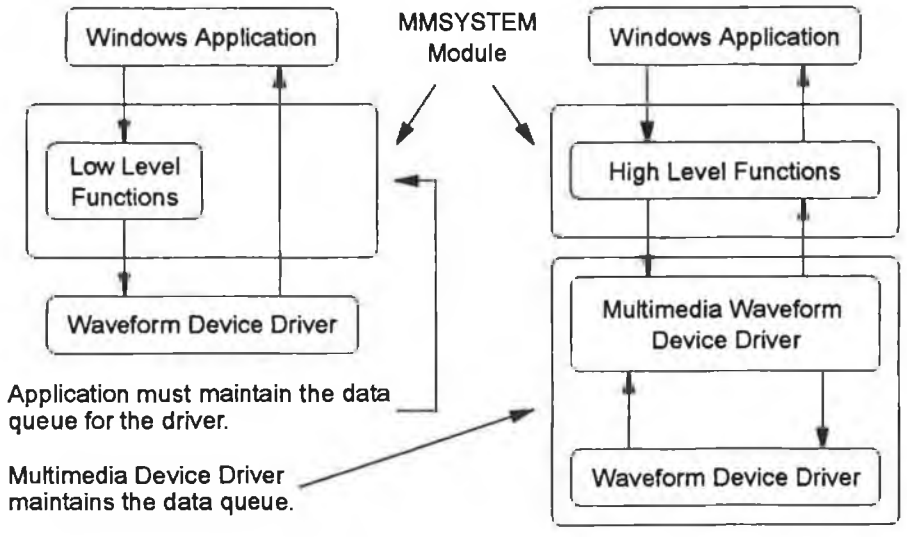


Figure 5.2 Control over the Waveform Device Driver during Recording or Playback.

5.3.1 Data Queue

The recording and playback processes are sustained by the application sending data memory buffers to the waveform device via its device driver. A single buffer cannot be used to maintain continuous recording or playback because the data transfer rates between the PC's hard disk and memory and between memory and the waveform device over the ISA bus are not fast enough. If two buffers were used it could result in playback or recording skips due to the application not being able to access the hard disk quickly enough. Therefore to guard against these skips or data losses, three buffers were used.

The buffers are returned to the application when the device driver has finished filling or emptying them. The application must empty \ fill and prepare them before sending them back to the device driver. The application's three data memory buffers guarantee a data queue of one buffer for the device driver. One buffer is in the data queue while the device driver is accessing another buffer and the third has been returned to the application. The application processes the returned buffer and accesses the WAVE file before the data queue is emptied by the device driver.

5.3.2 Sending data to the Waveform Device

The data are sent to the device driver using the *waveInAddBuffer* and *waveOutWrite* functions for recording and playback respectively [11]. These functions require three parameters, the device handle, a pointer to the *WAVEHEADER* structure and the size of the passed structure pointer. The *WAVEHEADER* structure describes the data memory buffer and contains the pointer to the data memory buffer, size of the buffer, number of bytes used by the device driver along with flags describing the status and properties of the buffer. The buffer must be locked in memory as it is accessed at interrupt time by the device driver. This is accomplished by the *waveInPrepareHeader* or *waveOutPrepareHeader* functions [11]. When the buffers are returned they must be released from their memory locked state using the *waveInUnprepareHeader* or *waveOutUnprepareHeader* functions [11].

5.3.3 Opening and Closing the Waveform Device

The low level audio functions for opening the input or output waveform device are *waveInOpen* and *waveOutOpen*, while the functions *waveInClose* and *waveOutClose* close the waveform devices [11]. The *waveInOpen* function is defined as follows,

waveInOpen (LPHANDLE &hWaveIn,
WORD wDeviceID,
LPPCMWAVEFORMAT lppcmWaveFormat,
DWORD Callback, DWORD dwCallbackInstance,
DWORD dwFlags);

where *&hWaveIn* is a pointer to a device handle, *wDeviceID* is the waveform device identification number, *lppcmWaveFormat* is a pointer to a structure describing the data format, *dwCallback* is the handle of the callback window or the pointer to the callback function, *dwCallbackInstance* is the callback instance data and *dwFlags* is the flags for opening the waveform device. The *waveOutOpen* function is identical except for the handle pointer being the address of a *WaveOut* handle.

The device handle will be initialised by the device driver and provides proof of ownership of the device when communicating with the waveform device. The device identification number can be any number between zero and one less than the number returned by the wave capability functions, *waveInCaps* and *waveOutCaps*. This identification number must be zero because the MTP waveform device is only capable of supporting one device at a time.

The *PCMWAVEFORMAT* structure describes the format of the WAVE file's data and for the sound card must be the PCM 44.1 kHz 16 bit stereo format [11]. The application uses *windowed callback* for waveform communications and therefore *dwCallback* is the application's window handle. The *dwCallbackInstance* parameter is not used here because the waveform device does not support multiple ownership.

The last parameter determines how the waveform device is opened and several flags can be used together. If the device is only being asked for its capabilities then the flags parameter is set to *WAVE_FORMAT_QUERY* while for *windowed callback* it is set to *CALLBACK_WINDOW*.

A waveform device must be closed as soon as the application is finished using it. The *waveInClose* and *waveOutClose* functions only require the respective device handle to close the particular device.

5.3.4 Application Callback Methods

The device driver must communicate with the application during the process of recording or playback to maintain continuous data flow. The methods for receiving waveform device driver messages [11] are as follows:-

- **Windowed Callback** : handle to a window.
- **Function Callback** : pointer to a function which resides in a DLL module locked in memory. This function is limited because it is called at interrupt time which restricts the data and functions it can call.
- **Task Unlocking** : pointer to a task which will be unblocked.

In this project the Windowed Callback method was used because it does not require an extra program module (DLL) to hold the callback processing algorithms and it demonstrates the principle of a dialog box and a window being separate objects. The dialog box is the application's active window while its main window processes the waveform device driver messages.

5.3.5 Waveform Messages supported by the Application

The waveform device driver is opened for windowed callback and therefore the window specified as the callback window must process the device driver's waveform messages. These messages are sent by the device driver through the MMSYSTEM module and have the *MM* prefix marking them as multimedia messages [11] and they are illustrated in to Figure 5.3. The messages correspond to opening and closing the waveform device and to the return of data buffers from the device driver. The algorithm for processing the data buffer return messages, *MM_WOM_DONE* and *MM_WIM_DATA*, are explained in Sections 5.5.2 and 5.6.2.

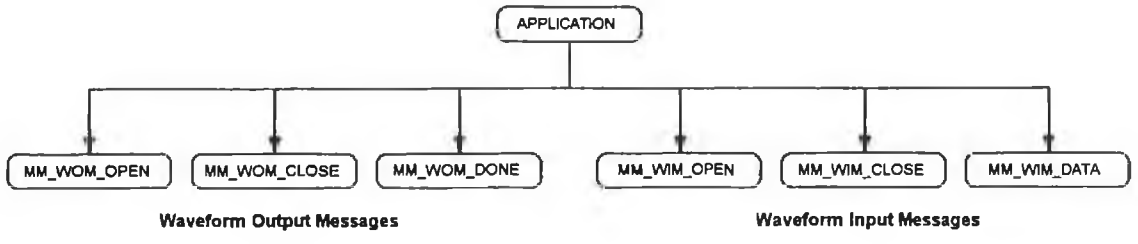


Figure 5.3 Waveform messages processed by the Application.

5.3.6 Accessing WAVE files

The multimedia file I/O functions are used by the application to access the RIFF WAVE files. These functions are specially designed for general RIFF files and are part of the Windows system software [11]. Therefore, they allow easier management of multimedia files and do not increase the programs size. The functions are explained in Appendix F.

5.4 Playback and Recording Process

In this section, the general process of recording or playing a WAVE file is described in general terms before explaining the steps in greater detail. The processes are illustrated in Figure 5.4. and can be summarised as follows:-

- **Acquiring Resources**

  Acquire system resources needed to perform the waveform operation. The first step involves acquiring memory for the data buffers which the application uses to send data to the device driver by the Windows application. The waveform input or output device is then opened in the desired format. The data buffers are then prepared and sent to the waveform device. Playback starts once a data buffer is sent to the device driver while recording must be started directly by the application. The multimedia file I/O functions are used to access the RIFF WAVE format file.

- **Maintaining the Data Queue**

  Maintain the data queue to produce continuous playback or recording. The data buffers are returned to the application after the device driver has finished with them. The application must access the WAVE file to fill or empty the data buffer before sending them back to the device driver to maintain continuous data flow.

- **Releasing Resources**

  Release the data buffers back to the system and close the waveform device and the WAVE file. The waveform input and output devices stop once they run out of data from the application.

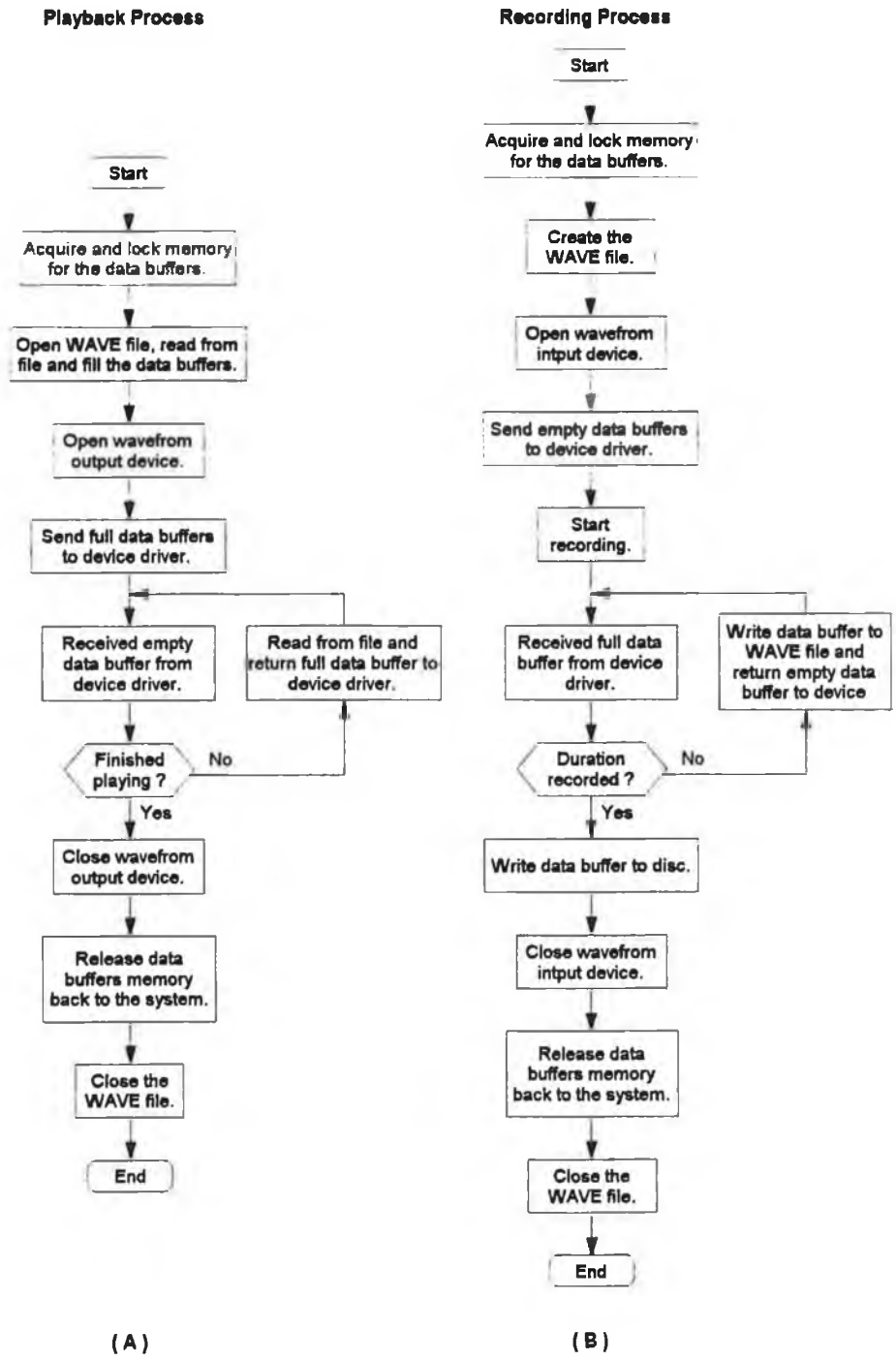


Figure 5.4 Playback and Recording Process from Application's Perspective.

5.4.1 Application's Analogue and Digital Modes

The only difference between the analogue and digital modes from the application's view point is the extra message sent to the sound card when the digital modes are chosen by the user. For digital playback the message *WODM_SETDIGITAL* is sent to the waveform device driver using the *waveOutMessage* function while for digital recording *WIDM_SETDIGITAL* is sent using *waveInMessage*. These messages are defined in both

the Windows application's and device driver's header files as 21 and 22 respectively. The default modes of the waveform device are the analogue modes.

5.5 Starting and Maintaining Continuous Playback

The ***PlayProc*** function is responsible for initialising the waveform output device and starting playback of the chosen WAVE file. Initialising the device involves acquiring memory, opening the device, reading from the WAVE file and preparing the data memory buffers and sending them to the device driver. The device driver automatically starts playback once it receives a buffer so as to respond as quickly as possible to the users playback request.

The *MM_WOM_DONE* message is sent by the device driver to the application whenever a data memory buffer has been played [11]. The application's main window message processing function, **MainWndProc**, processes this message and maintains continuous playback. This involves reading from the WAVE file, filling the returned buffer before preparing the buffer and sending it back to the device driver.

5.5.1 The *PlayProc* Function

The ***PlayProc*** function is described in Figure 5.5. Once it has acquired the memory for the data buffers, ***PlayProc*** will open the specified WAVE file using the procedure **OpenWAVFile**. This procedure will open the file and confirm that it is in the required format for the waveform output device. The file pointer is moved to the data field of the data subchunk of the WAVE file using the Multimedia File I/O functions [11]. ***PlayProc*** then moves the file pointer to the requested starting location along the data subchunk using the *mmioSeek* function. The data memory buffers are then filled with the data from the WAVE file using the ***ReadWAVDataProc*** function and their *WAVEHEADER* structures are initialised and prepared using the *waveOutPrepare* function.

The waveform output device is now opened. If the digital output mode is selected then the special *WIDM_SETDIGITAL* message is sent to the device driver. Playback is ready to commence and it is started when the first *WAVEHEADER* is sent to the waveform output device using the *waveOutWrite* function.

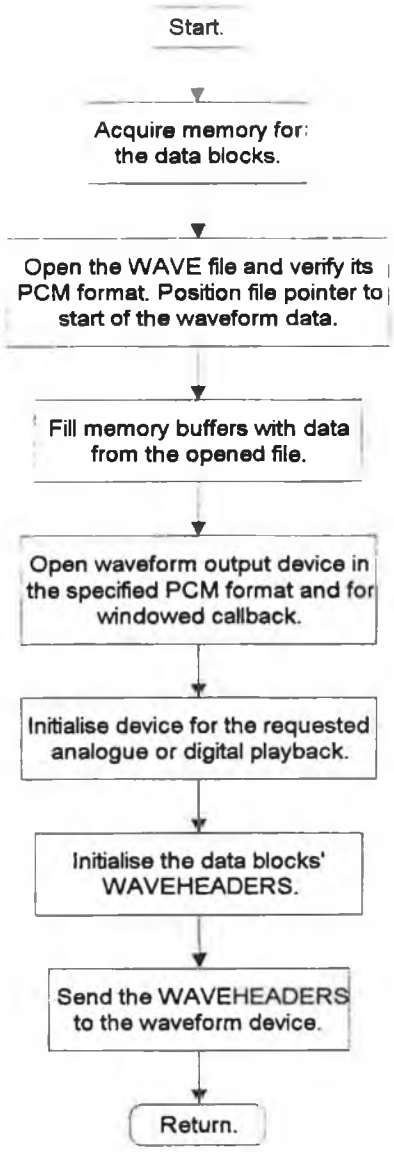


***Figure 5.5** The **PlayProc** Function.*

5.5.2 Processing *MM_WOM_DONE* Messages

The device driver will send a *MM_WOM_DONE* message to the requested callback window when it has played a data memory buffer. The callback window is the application's main window as specified when the waveform output device was opened. The processing of this message is described in Figure 5.6.

The processing of these *MM_WOM_DONE* messages can be summarised as follows. The returned *WAVEHEADER* is unprepared using the *waveOutUnPrepareHeader* function and the header queue variable *bHeaderQueue* is decremented. This variable keeps track of the number of buffers in the data queue. The global play duration variable *dwPlayDuration*, is now reduced by the *WAVEHEADER*'s data size field. The *bHeaderQueue* variable is compared with zero to determine if this

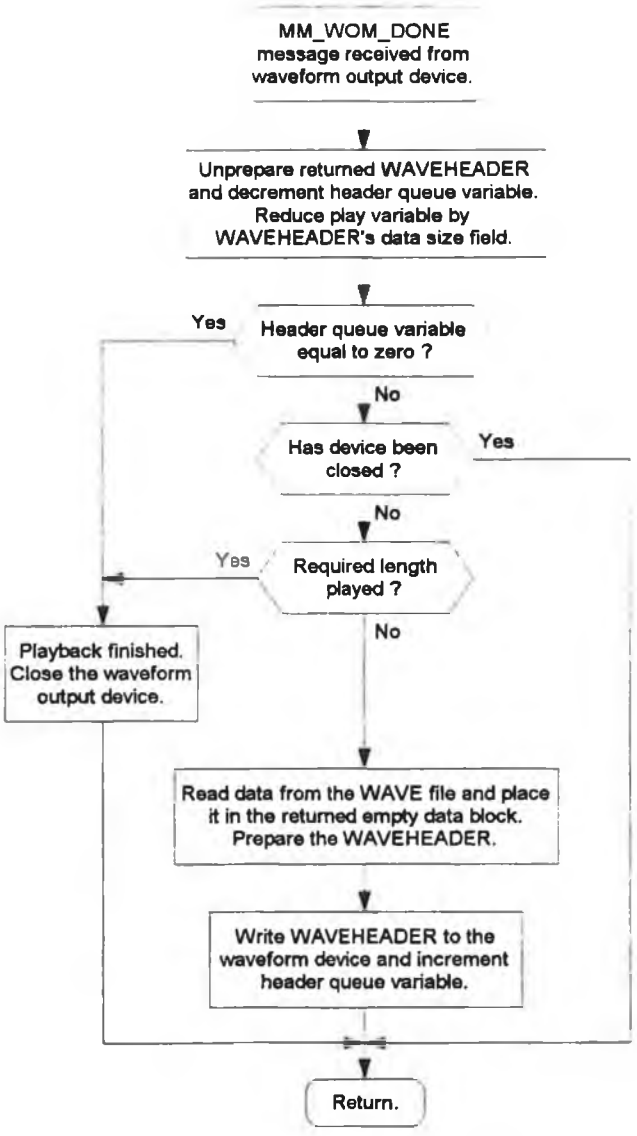


Figure 5.6. *Processing a MM_WOM_DONE message.*

is the last *WAVEHEADER* in the queue. If it is the last then the waveform output device is closed using the *waveOutClose* function.

If it is not, then the *dwPlayDuration* variable is examined to determine if the remaining *WAVEHEADER*s in the data queue will complete playback. If they will complete playback then the *WAVEHEADER* is not returned to the device driver and no further processing of this particular message occurs. Alternatively if another *WAVEHEADER* is required in the data queue, the close flag *bClose*, is examined to determine if the user has stopped playback by pressing the *Cancel (Stop)* push button or by closing the playback dialog box. If this flag is set then the waveform output device is immediately closed. If the waveform device is still open, then the ***ReadWAVDataProc*** function will fill the data memory buffer from the WAVE file. The *WAVEHEADER* is

now prepared and sent to the device driver. The *bHeaderQueue* variable is also incremented.

If the device was closed then the *WAVEHEADER* is not sent to the device driver and the *bHeaderQueue* is not incremented. Therefore, when the remaining *WAVEHEADER*s in the queue are returned, the *bHeaderQueue* will be decremented to zero and the waveform output device will be closed. The data buffers' memory is released when the application receives the *MM_WOM_CLOSE* from the device driver after the waveform output device has been successfully closed.

5.6 Starting and Maintaining Continuous Recording

Recording involves the ***RecordProc*** function which initialises the resources required for recording and starts recording. Recording must be started using the *waveInStart* function. The applications window's message processing function **MainWndProc** processes the *MM_WIM_DATA* messages and maintains continuous recording [11].

5.6.1 The *RecordProc* Function

The ***RecordProc*** function is described in Figure 5.7. Firstly, it acquires the memory buffers from global memory, before creating the WAVE file with the ***CreateFile*** function. This function creates a WAVE file in the PCM 44.1 kHz 16 bit stereo format and places the file pointer at the data field of the file's data subchunk. The *WAVEHEADER*s are then initialised and prepared before being sent to the waveform input device with the *waveInAddBuffer* function. Recording is now started using the *waveInStart* function.

5.6.2 Processing *MM_WIM_DATA* messages

The device driver will send *MM_WIM_DATA* messages to the application main window's processing procedure, **MainWndProc**, when it has filled a *WAVEHEADER*'s data field. Figure 5.8 describes the processing of this message. The record duration variable *dwRecordDuration*, is reduced by the returned *WAVEHEADER*'s bytes recorded field *dwBytesRecorded*, before the *WAVEHEADER* is unprepared, reinitialised and prepared. The *WAVEHEADER*'s data is written to the WAVE file using the ***WriteWAVData*** function and the queue variable *bHeaderQueue* is decemented. This variable keeps track of the number of *WAVEHEADER*s in the data queue. The

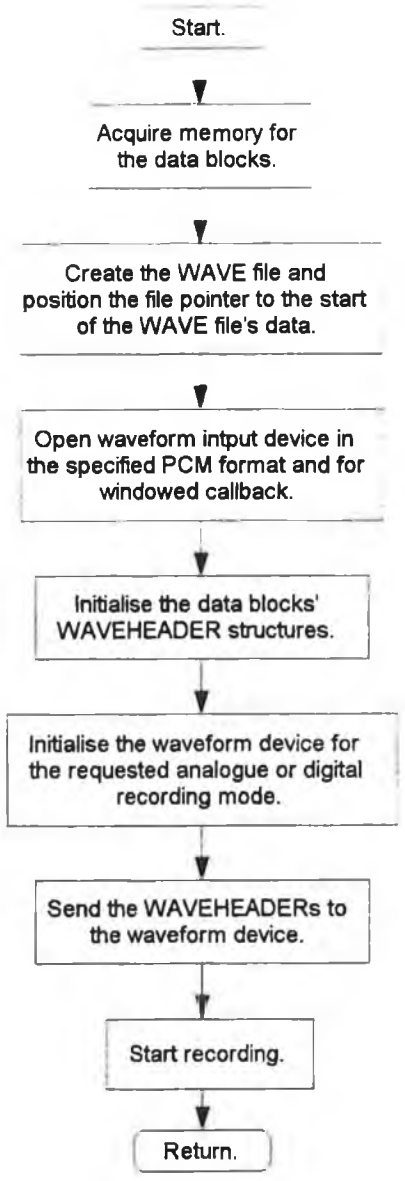


Figure 5.7. RecordProc function for starting recording.

*WAVEHEADER*s *bytes recorded* field is examined to determine if any recording took place. If this field is zero then recording has been stopped and the *WAVEHEADER* is not sent to the device driver.

The *dwRecordDuration* variable is examined to determine if the *WAVEHEADER*s in the data queue will complete recording. If they will not, then the *WAVEHEADER* is sent to the device driver and *bHeaderQueue* is incremented. The *bHeaderQueue* is compared with zero to determine if there is still a queue. If this variable is zero then recording has been completed and the waveform input device is closed with the *waveInClose* function.

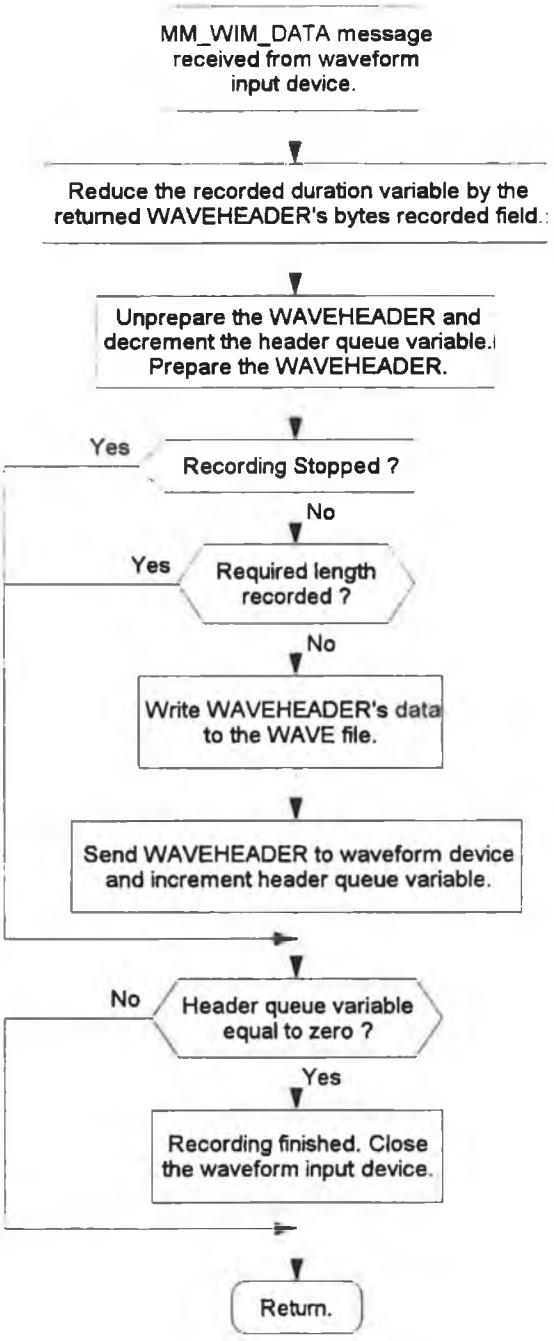


Figure 5.8. *Processing of a MM_WIM_DATA message.*

When the waveform input device has been closed, the *MM_WIM_CLOSE* message is sent to the application by the device driver. This message will release the memory back to the system and close the WAVE file.

5.7 Summary

This chapter describes the MTP's Windows application and the data queue maintenance algorithms used to maintain continuous recording or playback of the WAVE formatted

files. The Windows API functions used by the application are also described along with the functions specially written for the application.

Chapter 6

The MTP's Waveform Device Driver

6.1 Introduction

This chapter describes the device driver written to meet the requirements for a Windows waveform device driver. The processing of MMSYSTEM messages by the MTP's driver is explained in detail along with the driver algorithms for playing and recording WAVE files with the sound card.

The waveform device driver developed in this project to control the sound card, or waveform device in Windows terminology, conforms to the standards required by Windows Multimedia [11]. This driver supports the PCM 44.1 kHz 16 bit stereo waveform (analogue) audio recording and playback formats. At present, the consumer digital format is not a standard Windows Multimedia format, therefore the MTP's application sends messages specific to this project to the MTP's driver, which in turn configures the sound card for the selected digital operations.

6.2 Waveform Device Driver

The waveform device driver is responsible for controlling the waveform device. This device driver is written specially for the waveform device because it must control the waveform device in response to MMSYSTEM messages [11]. These messages are generated by the MMSYSTEM's multimedia functions which are called by the application in response to particular user actions, such as a request to play a WAVE file.

The responsibilities of the driver are follows:-

- Managing the transfer of data between the waveform device and the PC's memory
- Only allowing single ownership of the device
- Communicating with the application which owns the device
- Loading and removing itself from memory at the start and end of a Windows session

The waveform device dictates that the device driver only supports single ownership of the sound card, because it possesses no hardware facilities to support multiple ownership. The device driver can be divided into several sections as illustrated in Figure

6.1. The source code for the device driver can be found in the *DSOURCE* directory on the driver's installation disk.

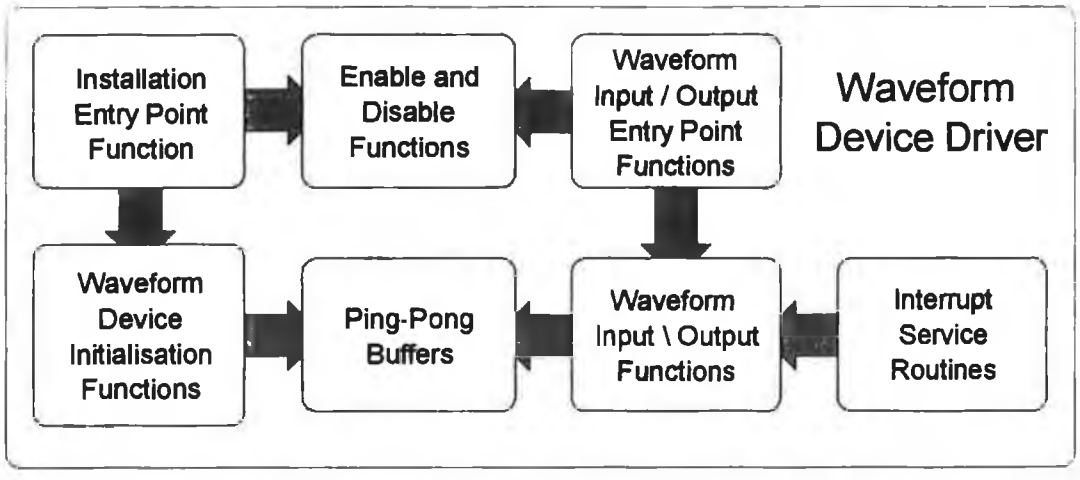


***Figure 6.1** Layout of Waveform Device Driver.*

6.2.1 Managing Data Transfer

The device driver receives from the application, *WAVEHEADERs* which contain pointers to data memory buffers. These buffers are then sent to or filled from the Swinging Buffers on the waveform device, depending on the mode of operation. These *WAVEHEADER* structures are passed to the device driver using the MMSYSTEM's low level audio functions [11]. The device driver will be sent several *WAVEHEADERs* from the application and these must be accessed in the order in which they were sent by the application. Otherwise the playback or recording process will be corrupted.

The device driver therefore uses a linked queue of *WAVEHEADERs*, each one pointing to the next one in the queue. The *WAVEHEADER* structure possesses a field which can be used to point to another *WAVEHEADER*. This field is appropriately known as the *WAVEHEADER's next_waveheader_pointer*. The device driver will update this field of the last *WAVEHEADER* in the queue to the next *WAVEHEADER* it receives from the application. The queue can also be referred to as the *data queue* or the *WAVEHEADER queue*. The *WAVEHEADER* at the top of the queue is stored in the *glpWOQueue* or *glpWIQueue* variables in the device driver, depending on the operating mode of the waveform device .

6.3 Device Driver Communications

The MMSYSTEM module provides the interface between the application and the device driver. The application sends messages to the device driver via MMSYSTEM's low level audio functions and the device driver sends messages to the application through MMSYSTEM's *DriverCallback* function [11]. Figure 6.2 illustrates this relationship between the application, MMSYSTEM and the device driver when the applications uses the low level functions to communicate with the driver.

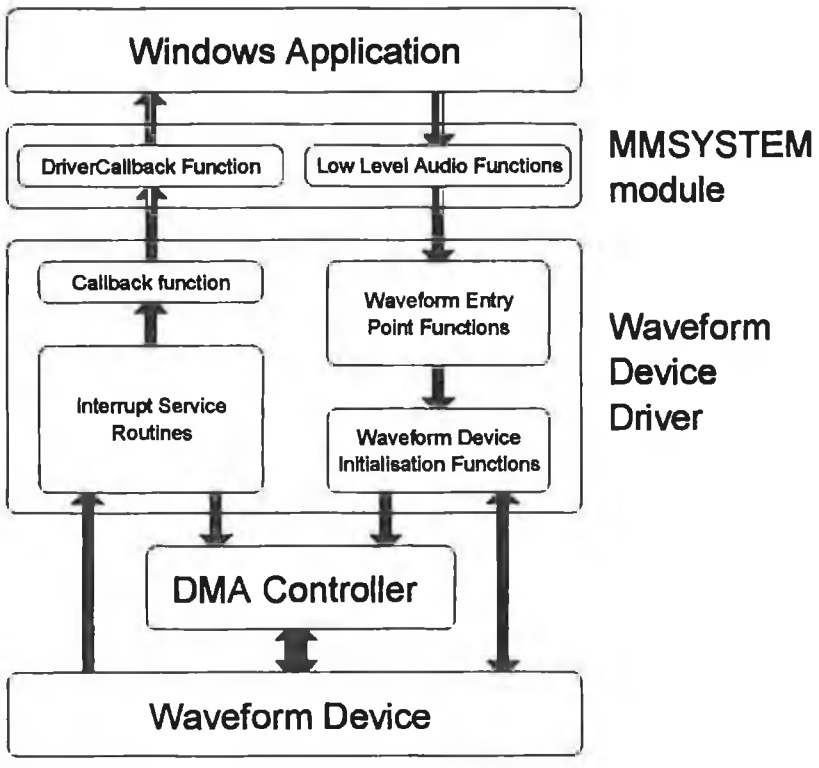


Figure 6.2 *Communication between Application, Device Driver and the Waveform Device.*

6.3.1 Communicating with the Device Driver

The waveform device driver can only be called from five functions. Two of these are the standard DLL functions **LibEntry** (or **LibMain** if there is no **LibEntry** present) and **WEP** which every DLL must possess. **LibEntry** is called by Windows whenever a DLL is being loaded into memory while **WEP** is called when removing a DLL from memory [11]. The remaining three functions are known as entry point functions. There must be entry point functions for initialising the device driver at Windows start-up time

DriverProc, along with one each for controlling waveform input and output, **widMessage** and **wodMessage** respectively [11]. These three functions are explained in detail in Sections 6.5.1, 6.6.1 and 6.6.2.

The **DriverProc**, **widMessage** and **wodMessage** entry point functions are passed a number of parameters with one parameter describing the message sent by MMSYSTEM. In this thesis a *C CASE SWITCH* structure has been used to identify these messages and process them accordingly. All the messages passed are standard Windows messages with the exception of the messages for the sound card's digital modes which are unique to this project and waveform device driver.

6.3.2 Communicating with the Windows Application

The device driver is opened for *Windowed Callback* as described previously in Chapter 5, Section 5.3.4. The MMSYSTEM's *DriverCallback* function [11] is used to send messages to the application's main window for processing. The format of this function is shown in Listing 6.1. This function will send the application its messages in the method chosen when the application opened the waveform device. The device driver must send standard multimedia messages when it has been opened or closed in either of its input or output modes and when it has finished with a *WAVEHEADER* sent to it by the application. The messages sent to the application were illustrated previously in Chapter 5, Figure 5.3.

```
DriverCallback(pWave->dwCallback,        // User's callback DWORD.
      HIWORD(pWave->dwFlags) | DCB_NOSWITCH,            // Flags.
      pWave->hWave,              // Handle to the waveform device.
      msg,                                              // The message.
      pWave->dwInstance,                 // User's instance data.
      dw1,                                              // First DWORD.
      0L);                                             // Second DWORD.
```

***Listing 6.1** DriverCallback function.*

6.4 Structure of the Waveform Device Driver

The waveform device driver in this project can be considered to be composed of three functional sections, responsible for driver installation, waveform input and waveform output. Each of these sections contains an entry point function along with other functions and interrupt service routines. These sections can be summarised as follows:-

- **Driver Installation**

  The installation and removal of the device driver from memory at the start and end of a Windows session.

- **Waveform Input**

  Controls the waveform input device and transfers data from the waveform device, maintaining continuous recording.

- **Waveform Output**

  Controls the waveform output device and transfers data to the waveform device, maintaining continuous playback.

6.4.1 Program Module

The device driver is composed of one data and several code segments. It is split into several different segments to provide as much memory flexibility as possible. The code and data segments which contain the interrupt service routines and the driver's data must reside in fixed memory segments because they are accessed at interrupt time and must therefore be present in memory. Therefore, to keep the size of fixed memory to a minimum the device driver is split into several code segments.

The *_TEXT* code segment and the data segment are assigned the *FIXED PRELOAD* descriptors [14] in the driver's module definition file, *MMTEACH.DEF*. This module definition file differs slightly from the applications file because the device driver is a Windows DLL. Therefore the *PROGRAM* identifier is replaced by the *LIBRARY* identifier and the driver does not contain a stack size because it uses the stack of the application which called it. The other code segments, *COMMON*, *INIT* and *WAVE* are assigned the *MOVEABLE DISCARDABLE PRELOAD* descriptors [14]. These three code segments contain functions which are accessed at different times and can be discarded by Windows from memory when memory is scarce. For example the *INIT* code segment is only required at the start and end of Windows. The files which make up the device driver are described in Table 6.1.

File Name	File Description
MMTEACH.H	Header file for driver's C programs.
MMTEACH.INC	Include file with definitions for the driver's assembler language programs.
MMTEACH.DEF	Module definition for constructing the driver as a Windows DLL module.
LIBINIT.ASM	Functions which initialises the driver.
INITA.ASM	Contains functions called when enabling and disabling the driver.
COMMONA.ASM	Functions common to the waveform input and output devices.
WAVEA.ASM	Contains functions used to initiate and control the waveform input and output devices.
MMTEACH.ASM	Functions required by the device to maintain recording or playback.
DRVPROC.C	Driver Installation Entry Point function.
INITC.C	Functions called to detect the waveform input and output devices.
CONFIG.C	Configuration information dialog box function.
WAVEFIX.C	Contains functions for transferring data between the waveheaders and the DMA buffers.
WAVEIN.C	Waveform input device's Message Entry Point function.

***Table 6.1** Waveform Device Driver Files.*

6.4.2 Interrupt Service Routines and Flags

The device driver provides the interrupt service routines for the waveform device's two hardware interrupts. These interrupts are initialised when the driver is enabled and removed when the driver is disabled. The interrupts are common to the recording and playback modes. An address table contains the recording and playback functions. The *_gbIntUsed* flag is used to produce the table index which in turn is used to call the appropriate function.

Flags are used extensively in this project to determine the state of the waveform device. There are flags describing the state and ownership of the hardware interrupts

(_gbIntUsed) and whether a waveform device is free *(_gbWaveInFlags)*. These flags are examined whenever an attempt is made to open the waveform device. There are also flags for determining if DMA in occurring *(_gfDMABusy)*, for stopping playback loops *(bBreakLoop)* and for counting the DMA interrupts *(gbIntCount7)*.

6.5 Device Driver Initialisation

Device drivers are loaded into memory at Windows start-up time. The drivers are loaded in the order they appear in the *SYSTEM.INI* file [11]. This file may also contain configuration information for the driver if necessary. This information is usually contained in a section named after the driver. In the case of this project the name *[MMTEACH]* would be used as the section name if the driver needed initialisation information.

The steps involved in loading the driver are described in Figure 6.3 and can be summarised as follows (see Section 6.3.1). The function **LibEntry** *(LIBINIT.ASM)*, which must be included in all Windows device drivers, is called by Windows to begin loading the device driver [11]. This function in turn calls the Windows Kernel procedure *LibInit* to initialise the device driver's local memory heap if one is declared in the driver's module definition file. The DLL initialisation function ***LibMain*** (*INITC.C*) is then called by **LibEntry** to perform any other initialisation required by the DLL before being loaded into memory.

The ***LibMain*** function saves the driver's handle in the global variable *hModule*. This is the handle passed to the Windows application by the device driver whenever it successfully opens one of the waveform devices. The sound card's (waveform device) *BUFFER STATUS REGISTER* is used to test if the sound card is present. This test involves attempting to place the card in its digital playback mode. If it succeeds then the driver is loaded, otherwise a message box is displayed informing the user that the driver cannot be loaded due to the waveform device not being detected.

Windows now sends messages to the device driver's installation entry point function **DriverProc** [11], to determine if the driver is to be enabled and if not whether it is to remain in the system or be discarded. This function is described in detail in the next section.

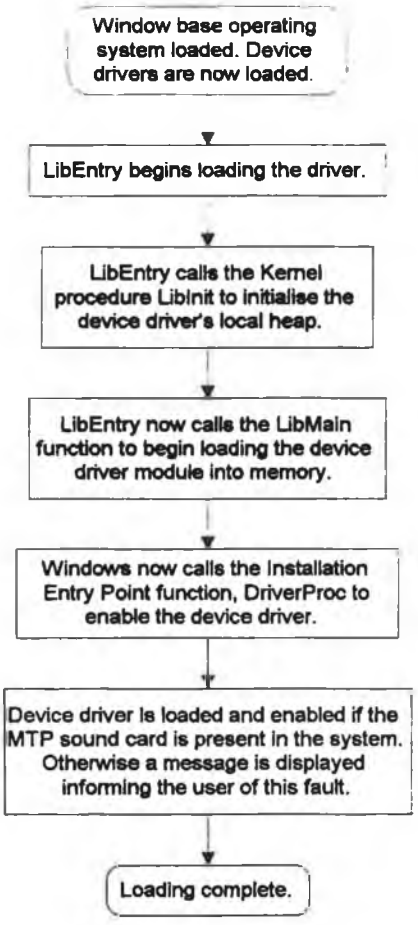


***Figure 6.3** Waveform Device Driver Initialisation.*

6.5.1 The Driver Installation Entry Point Function

The Driver Installation Entry Point function is called **DriverProc** (DRVPROV.C) and is responsible for enabling, disabling, removing and installing the waveform device driver in response to standard messages sent to it by Windows [11]. The format of this function is as follows:

FAR PASCAL **DriverProc***(DWORD dwDriverID, HANDLE hDriver, UINT uiMessage, LPARAM lParam1, LPARAM lParam2)*

The double word *dwDriverID* is the identification number for the driver while *hDriver* is the driver handle. The parameter *uiMessage* is the message identifier and the *lParam1* and *lParam2* parameters are message dependant. The function is implemented in a *C CASE-SWITCH* structure to identify and process the received messages. Figure 6.4 illustrates the messages which the function must support and they are described as follows.

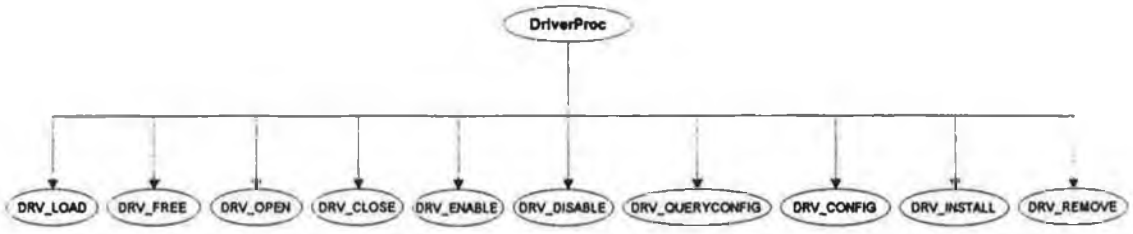


***Figure 6.4** Messages supported by the* **DriverProc** *function.*

DRV_LOAD : Sent to the driver when it is loaded, always the first message received by a driver.

DRV_FREE : Sent to the driver when it is about to be discarded, always be the last message received by a driver before it is freed.

DRV_OPEN : Sent to the driver when it is opened.

DRV_CLOSE : Sent to the driver when it is closed.

DRV_ENABLE : Sent to the driver when the driver is loaded or reloaded and whenever Windows is enabled. The drivers should only hook interrupts or expect any part of the driver to be in memory between enable and disable messages.

The ***Enable*** (*INITA.ASM*) function is called first and will attempt to allocate memory for the two DMA buffers using the ***dmaAllocateBuffer*** and ***dmaAllocateBufferPong*** (both in *INITA.ASM*) functions. These functions are identical except for their address storage variables. They attempt to acquire a 4 kbyte buffer from memory which does not cross a page boundary. If a page boundary is crossed then the buffer cannot be guaranteed to be contiguous in memory, due to the processors protected mode paging hardware which can map linear memory to any location in physical memory. First, the algorithm acquires a 4 kbytes memory block from global memory using the Windows function *GlobalAlloc* with the *GMEM_FIXED* and *GMEM_SHARED* flags. The *GMEM_FIXED* flag locks the block in memory because it is accessed at interrupt time. The block is checked for a page boundary. If a page boundary is

present, the block is returned to system memory using the Windows function *GlobalFree* and an 8 kbytes memory block is acquired. If the lower half of this block contains a page boundary then the starting address of the DMA buffer is moved to the next page within this 8 kbyte memory block.

The interrupt vectors for the two interrupts are set to the addresses of the interrupt service routine functions using the ***InitSetInterruptVector*** (*INITA.ASM*) function. This function uses the DOS 21h interrupt functions, 35h and 25h to retrieve and replace the interrupt vectors respectively. The function ***InitSetIntMask*** (*INITA.ASM*) will enable the waveform device's two interrupts lines by modifying the PIC's mask register. These interrupts will be inactive due to the power-up circuitry of the waveform device.

DRV_DISABLE : Sent to the driver before the driver is freed and whenever Windows is disabled. The ***Disable*** (*INITA.ASM*) function is called which will call the ***widSuspend*** and ***wodSuspend*** (both in *INITA.ASM*) functions to examine the waveform devices active flags and suspend the waveform devices if they are active. The interrupt mask is restored to its original state using the ***InitSetIntMask*** (*INITA.ASM*) function. The two interrupt vectors are also restored with the ***InitSetInterruptVector*** (*INITA.ASM*) and the DMA buffers are returned using the ***dmaFreeBuffer*** and ***dmaFreeBufferPong*** (both *INITA.ASM*) functions.

DRV_QUERYCONFIG : Sent to the driver to determine if the driver supports custom configuration. The driver returns the value zero to indicate that software configuration of the waveform device is supported. This feature is only included to allow the device driver's information dialog box, describing its hardware requirements, to be displayed from the *Drivers* applet from the *Control Panel* of the *MAIN* group in *Program Manager*. Otherwise the *SETUP* push button of the *Drivers* applet, which allows software modification of the selected device driver, would be inactive resulting in the inability to display the device driver's information dialog box.

DRV_CONFIGURE : Sent to the driver so that it can display a custom configuration dialog box. The configuration dialog box only displays information about the device driver and the interrupt and DMA requirements of the sound card because the card does not support dynamic software configuration. This function is contained in the *DRIVER.C* file of the device driver.

DRV_INSTALL : Sent to the driver when the driver is being installed. Appendix I describes the installation disk for the device driver. Windows will transfer the driver to the Windows *SYSTEM* subdirectory and modify the *[drivers]* section of SYSTEM.INI to include this driver. The *DRVCNF_RESTART* value is returned which will cause Windows to display a dialog box, asking the user if they wish to restart Windows. If a change in the device driver has occurred, Windows must be restarted to allow this change to take effect on the system.

DRV_REMOVE : Sent to the driver when it is being removed from the system. It must remove its entries in the *SYSTEM.INI* file. The function, ***ConfigRemove*** (*CONFIG.C*) is called which uses the *WritePrivateProfileString* function to remove the device driver from the *[drivers]* section of the *SYSTEM.INI* file. The return value is *DRVCNF_RESTART* which was explained previously in *DRV_INSTALL*.

6.6 The Message Processing Entry Point Functions

Windows requires that the waveform input and output message processing entry point functions be called **widMessage** and **wodMessage** respectively [11]. These functions are similar to the driver installation function in that they were constructed with a *C CASE-SWITCH* structure to identify and process the messages sent to them by the MMSYSTEM module. There is a standard set of messages which must be processed for each waveform device, with a further set of optional messages. The standard set covers opening, closing and querying devices, sending data to them and generally controlling their operation while the optional set covers volume control and preparation of data blocks. These functions are only called once the Windows environment has been initialised and the system's device drivers have been loaded.

6.6.1 The widMessage Function

The **widMessage** function [11] is the entry point function for controlling the waveform input device and is situated in the *WAVEIN.C* file. The format of this function is as follows.

DWORD FAR PASCAL **widMessage** *(WORD id, UINT msg, DWORD dwUser,*
DWORD dwParam1, DWORD dwParam2)

This function first determines if the driver is enabled by examining the driver's enable flag *gfEnable*, this flag is zero if the driver is disabled and non zero if enabled. If the driver is enabled the function checks the *id* parameter which is the identification number for the target device. This identification number value can range from zero to one less than the number of input devices that the waveform device driver supports. Therefore it must be zero as the waveform device driver only supports a single waveform input device.

The *msg* parameter specifies the message being sent to the driver. This message is identified and processed using a *C CASE-SWITCH* statement. The *dwParam1* and *dwParam2* parameters are message dependant. The *dwUser* parameter is used when more than one waveform input device is supported. Separate instance structures containing device information will be created by the device driver for every waveform input device opened.

The standard messages which this function must support are described next, along with the processing done by the device driver for each one. Figure 6.5 illustrates the messages processed by the function.

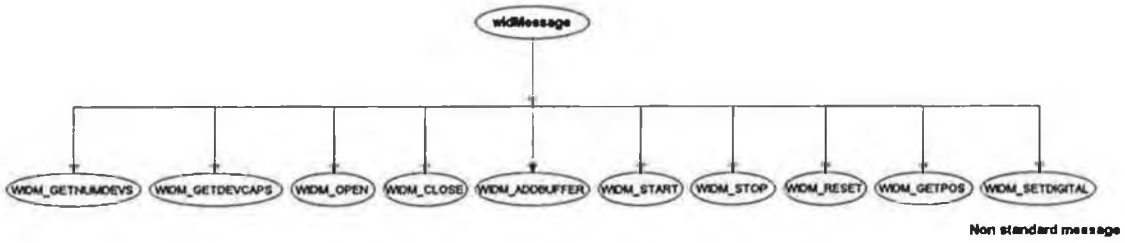


Figure 6.5 Messages processed by the **widMessage** *function.*

WIDM_GETNUMDEVS : Sent to the device to determine the number of devices the hardware supports. The hardware will only support one device, therefore the value one is returned.

WIDM_GETDEVCAPS : Sent to determine the capabilities of the waveform input device. The function ***widGetDevCaps*** is called which will fill the *WAVEINCAPS* structure passed as a pointer with the waveform input device's capabilities and other device driver information. The value zero is returned.

WIDM_OPEN : Sent to acquire and open the waveform input device. The *PCMWAVEFORMAT* structure passed as a pointer will be compared with the device driver structure to determined if the requested format is supported by the waveform input device. If the requested format is not supported then the error code *WAVERR_BADFORMAT* is returned. The *dwParam2* parameter is compared with *WAVE_FORMAT_QUERY* to determined if device information was only requested. The value zero is returned if this is true and otherwise the waveform input device is opened as follows.

The function ***widAquireHardware*** (*WAVEA.ASM*) is called which examines the *gbIntUsed* and *gWaveInFlags* flags to determine if the waveform input device is free. If the device is free then these flags are set to *INT_WAVEIN* and *WIF_ALLOCATED* respectively, to prevent the waveform device from being opened by another application.

The memory required for the driver's *pInClient* structure is allocated from the driver's local heap. This structure is initialised with information regarding the Window's application which has opened the device and the format of the waveform input device. The function ***waveCallback*** (*WAVEFIX.C*) is called to notify the application that the device has been opened. This function in turn calls the MMSYSTEM's *DriverCallback* function which notifies the application in its chosen callback method. The value zero is returned.

WIDM_CLOSE : Sent to close the waveform input device. The waveform input device is only closed if recording is not taking place. The *glpWIQueue* pointer is examined to determine if a recording queue exists. If a recording queue exists

then *WAVERR_STILLPLAYING* is returned, otherwise the function ***widStop*** (*WAVEA.ASM*) is called which masks the DMA channel using the ***dmaMaskChannel*** (*MMTEACH.ASM*) function and returns any remaining data using the ***widSendPartBuffer*** (*WAVEIN.C*) function. The ***waveCallback*** (*WAVEFIX.C*) function then notifies the application that the waveform input device has been closed. The *pInClient* structure's memory is now released from the driver local heap using the Windows function *LocalFree*. The value zero is then returned to indicate that the waveform input device was successfully closed.

WIDM_ADDBUFFER : Sent when an application's data memory buffer is to be filled with recorded data. The flags of the received *WAVEHEADER* structure pointer are examined to determine if the *WAVEHEADER* in already in the data queue and if it has been locked in memory. An error value is returned if the *WAVEHEADER* is not valid, otherwise the function ***widAddBuffer*** (*WAVEIN.C*) is called which adds the *WAVEHEADER* to the queue. This function walks down the queue to the last *WAVEHEADER* and places the pointer to the new *WAVEHEADER* in the last *WAVEHEADER's next waveheader_pointer_*field. If no queue exists then the top of the data queue is set to this *WAVEHEADER*. The value zero is returned if the *WAVEHEADER* was successfully added to the queue otherwise an error code is returned.

WIDM_START : Sent to start recording. The function ***widStart*** (*WAVEA.ASM*) is called to start recording. This function will first set the *gbWaveInFlags* to *WIF_STARTED* to state that recording has started. The function ***dmaInitDMAIn*** is now called to initialise the DMA controller and then the ***dspReset*** function will reset the waveform input device. The *_gbDigital* flag is then examined to determine if the digital input mode has been selected and the card is then configured accordingly. Recording is started when the card's logic is reset, by issuing the reset signal, *MResetbar*. Finally the DMA and Interrupts signals are unmasked by writing to their respective I/O ports. The ***widStart*** function now returns to **widMessage**. The value zero is returned.

WIDM_STOP : Sent to stop recording. The function ***widStop*** (*WAVEA.ASM*) is called which will stop recording if it has been started. This function will then call the ***dmaMaskChannel*** (*MMTEACH.ASM*) function to mask the DMA channel before calling the ***widSendPartBuffer*** (*WAVEIN.C*) function to return the unfilled *WAVEHEADER*s back to the device. The *gbWaveInFlags* flag is then cleared. The value zero is returned.

WIDM_RESET : Sent when recording is to be stopped and all data memory buffers are to be returned. The function ***widStop*** (*WAVEA.ASM*) is first called to stop recording. The function ***widFreeQ*** (*WAVEIN.C*) is then called to return all *WAVEHEADERs* in the recording queue to the Windows application. The value zero is returned.

WIDM_GETPOS : Sent to request the present recording position in bytes from the start of recording. The function ***waveGetPos*** (*WAVEOUT.C*) is called which first examines the passed *MMTIME* structure pointer to determine the requested time format. If the requested format is *TIME_BYTES* (time in bytes), then the *MMTIME* structure is set to the *pInClient* structure's *dwBytesCount* field. This contains the total number of bytes recorded since recording started.

WIDM_SEDTDIGITAL : This message is unique to the MTP device driver and it enables the digital recording capability by setting the *_gbDigital* flag to the *DIGITAL* mode. This flag is examined by the ***WidStart*** function which will configure the card for digital recording if this flag is set.

Optional messages are **WIDM_PREPARE** and **WIDM_UNPREPARE** which if not supported in **widMessage** and are instead processed by the MMSYSTEM module [11]. These messages are sent by the *waveInPrepareHeader* and the *waveInUnprepareHeader* functions respectively which are required to page lock and unlock the data blocks in memory.

The messages that **widMessage** sends to the application are the **MM_WIM_OPEN** and **MM_WIM_CLOSE** messages [11]. They are sent using the *DriverCallback* function in the format chosen by the application when the waveform

input device was opened. The interrupt service routines are responsible for sending the **MM_WIM_DATA** message to the application whenever a data memory buffer has been filled and removed from the data queue.

6.6.2 The wodMessage Function

The **wodMessage** function [11] is is the entry point function for controlling the waveform output device and is situated in the *WAVEOUT.C* file. The function is declared as follows.

DWORD FAR PASCAL **wodMessage** *(WORD id, UINT msg, DWORD dwUser,*
DWORD dwParam1, DWORD dwParam2)

The **wodMessage** function checks the device's identification number, *id*, before allowing any messages to be processed. The waveform output device only supports one instance so the identification number must be zero.

Figure 6.6 describes the messages processed by the device driver's **wodMessage** function. The standard messages which this function must support and the processing done by each are now briefly described.

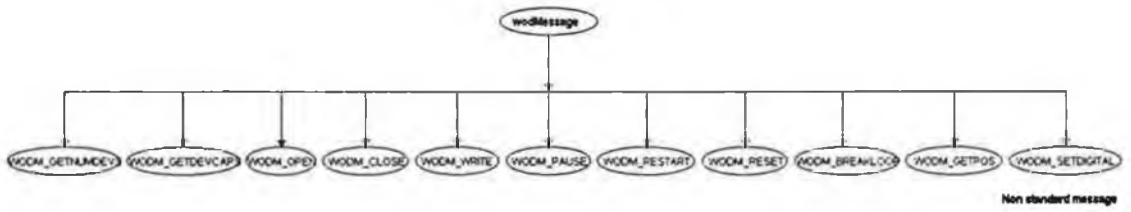


Figure 6.6. *Messages processed by the* **wodMessage** *function.*

WODM_GETNUMDEVS : Sent to the device to determine number of devices the hardware supports. The hardware only supports one device, therefore the value one is returned.

WODM_GETDEVCAPS : Sent to determine the capabilities of the waveform output device. The function ***wodGetDevCaps*** (*WAVEOUT.C*) is called which fills the application's *PCMWAVEFORMAT* structure (passed as a pointer) with the waveform output device's capabilities. The value zero is returned or a relevant error value if an error occurs.

WODM_OPEN : Sent to acquire and open the waveform output device. The *PCMWAVEFORMAT* structure passed as a pointer, parameter *dwParam1*, which defines the desired format that the application wants to playback in, is checked against the waveform output devices supported format, PCM 44.1 kHz 16 bit stereo. If the desired format is not supported then the error value *WAVERR_BADFORMAT* is returned. The *dwParam2* parameter is compared with *WAVE_FORMAT_QUERY* to determine if device information only was requested. The value zero is returned if information only was required. The function ***wodAcquireHardware*** (*WAVEA.ASM*) is now called. This function examines the *gbIntUsed* and *gbWaveOutFlags* to determine if the waveform device is free. If the device is free then these flags are set to *INT_WAVEOUT* and *WOF_ALLOCATED* respectively. The *MMSYSERR_ALLOCATED* value is returned if an error occurred and the device was not allocated. The *pOutClient* structure is now initialised. First, memory is obtained from the device driver's local heap using the Windows function *LocalAlloc* and second, the structure is initialised with application information and the waveform output format. The ***waveCallback*** (*WAVEFIX.C*) function is now called to inform the application that the waveform output device is now opened. The value zero is returned to indicate successful opening of the device.

WODM_CLOSE : Sent to release and close the waveform output device. The *glpWOQueue* pointer is examined to determine if a playback data queue exists. If a queue exists then the waveform output device is still playing and is not closed. The *WAVERR_STILLPLAYING* error value is returned to the application. If the waveform output device is not playing then it is released with the ***widRelease*** *(INITA.ASM)* function and the application is notified that the device is being closed with the ***waveCallback*** (*WAVEFIX.C*) function. The *pOutClient* structure's memory is freed using the Windows *LocalFree* function. The value zero is then returned.

WODM_WRITE : Sent when the application sends a data memory buffer to be played. The passed *WAVEHEADER flags* field is examined to determine if it has been prepared and is not already in the data queue. If it is fails then the relevant error

code, *WAVERR_UNPREPARED* or *WAVERR_STILLPLAYING* is returned. The function ***wodWrite*** (*WAVEA.ASM*) is called if the *WAVEHEADER* is valid. This function places the *WAVEHEADER* in the queue by updating the queue's last *WAVEHEADER* entry's *next_waveheader_pointer* field to point to this *WAVEHEADER*. If no queue exists then the *WAVEHEADER* is placed at the top of the queue and the ***wodKickStartWaveOut*** (*WAVEA.ASM*) function is called to start playback. This function fills the DMA buffers, resets the sound card and places it in the requested playback mode by examining the *_gbDigital* flag. The function then initialises and unmasks the DMA channel with the ***dmaStartDMAOut*** before enabling the DMA / Interrupt mode of the sound card. The value zero is returned.

WODM_PAUSE : Sent when playback is to be suspended. The function ***wodPause*** (*WAVEA.ASM*) is called which sets the *_gfWaveOutPaused* flag to TRUE (the value one). This function then calls the ***wodWaitForDMA*** (*WAVEA.ASM*) function to allow the last DMA session to be completed before the waveform device is paused. The value zero is returned.

WODM_RESTART : Sent when playback is to be restarted. The function ***wodResume*** is called which tests the *_gfWaveOutPaused* flag to determine if the waveform output device was paused. The function exits if the device was not paused, otherwise it clears the *_gfWaveOutPaused* flag and calls the ***wodKickStartWaveOut*** (*WAVEA.ASM*) function to start playback. The value zero is returned.

WODM_RESET : Sent to stop playback and return all data memory buffers to the application. The ***wodHalt*** (*MMTEACH.ASM*) function is called to stop the current DMA session. This function clears the *_gfBusy* flag to stop any further DMA sessions from being started before the DMA channel is masked by the ***dmaMaskChannel*** (*MMTEACH.ASM*) function. The ***wodFreeQ*** (*WAVEOUT.C*) function is then called which returns all unplayed *WAVEHEADERs* back to the application. The *gfWaveOutPaused* and *bBreakLoop* flags are cleared along with the *dwByteCount* element of the *pOutClient* structure.

WODM_BREAKLOOP : Sent to break the looping of data memory buffers. Buffers can be replayed continuously with the loop flags of the buffers' *WAVEHEADER* structures. The *bBreak* flag is set to one which will break the playback loop when the next DMA session is started by the ***wodLoadDMABuffer*** (*WAVEFIX.C*) function. This function is called at every playback interrupt by their interrupt service routines. The value zero is returned.

WODM_GETPOS : Sent to request the present playback position in bytes from the start of playback. The function ***waveGetPos*** (*WAVEOUT.C*) is called which was explained earlier in the *WIDM_GETPOS* case for the **widMessage** function.

WODM_SETDIGITAL. This message is unique to the MTP driver and sets the digital playback mode. The *_gbDigital* flag is set to its *DIGITAL* state by this message.

None of the optional messages for the waveform output device are supported by the waveform device but for reference they are **WODM_GETPITCH, WODM_SETPITCH, WODM_GETVOLUME, WODM_SETVOLUME WODM_GETPLAYBACKRATE, WODM_SETPLAYBACKRATE, WODM_PREPARE** and **WODM_UNPREPARE**. These messages are concerned with the playback sampling rate and volume control which the waveform output device does not support. The **WODM_PREPARE** and **WODM_UNPREPARE** are identical to the waveform input's optional messages **WIDM_PREPARE** and **WIDM_UNPREPARE** and are processed by the MMSYSTEM module [11].

The messages that **wodMessage** will send to the application are the **MM_WOM_OPEN** and **MM_WOM_CLOSE** messages [11]. The *DriverCallback* function is used to send these messages to the application.

6.7 Maintaining Continuous Recording

Once recording has started the device driver's interrupt service routines will maintain continuous recording if the application is capable of sustaining the data queue. The application creates the data queue by sending *WAVEHEADER* structures to the driver. The *WAVEHEADER* structure inturn points to an empty data buffer used to store the incoming audio data. The application must empty the returned *WAVEHEADER's data*

field, (the data memory buffer), to disk and send the *WAVEHEADER* back to the waveform device before the data queue is empty, otherwise data may be lost due to both Swinging Buffers being full.

When a Swinging Buffer has been filled, the Switching interrupt is generated which triggers its interrupt service routine, ***Switch_ISR*** *(MMTEACH.ASM)*. This function will set up the DMA controller to read the first 2 kwords from the full Swinging Buffer, causing the DMA controller to fill the empty DMA ping buffer. Meanwhile ***Switch_ISR*** will call the ***widFillBuffer*** *(WAVEFIX.C)* function to transfer the full DMA pong buffer to the *WAVEHEADER's* data field. The terminal count signal (*TC*) from this DMA session will generate the DMA interrupt, causing its interrupt service routine, ***DMA_ISR*** *(MMTEACH.ASM)* to be called. This ISR will set up the DMA controller to read the next 2 kwords from the Swinging Buffer and fill the empty DMA pong buffer, while the ***widFillBuffer*** *(WAVEFIX.C)* function transfers the driver's full ping buffer to the data field of the *WAVEHEADER* at the top of the data queue.

Another *TC* signal is generated when the previous DMA session is completed, which generates another DMA interrupt which will transfer the next 2 kwords from the Swinging Buffer to the empty DMA ping buffer while the full DMA pong buffer is transferred to the *WAVEHEADER*'s data field. The filling of one DMA ping-pong buffer occurs while the other is transferred to the *WAVEHEADER*'s data field. Seven DMA interrupts are required to empty a *FIFO* after the switching interrupt (7x2k + 2k = 16k). A count variable is used to determine when seven DMA interrupts have occurred. This eighth interrupt will just reset the count variable. The flag *gbDMABuffer* determines which DMA buffer is empty, and which is full and is updated by every ISR.

The initial Switching interrupt does not cause the DMA pong buffer to be transferred to the *WAVEHEADER*'s data field because it has not yet been filled. The following section describes the function called by the ISR, which is responsible for transferring the DMA buffers to the *WAVEHEADER*'s data field and for maintaining the data queue when a *WAVEHEADER* is filled and returned.

6.7.1 The *widFillBuffer* Function

The ***widFillBuffer*** *(WAVEFIX.C)* function is called by the ISRs to empty a full DMA buffer. This function transfers the data to the data field of the *WAVEHEADER* at the top of the data queue. The function's flowchart is shown in Figure 6.7. The parameters passed to this function are, the far pointer to the DMA buffer and the DMA buffer's size.

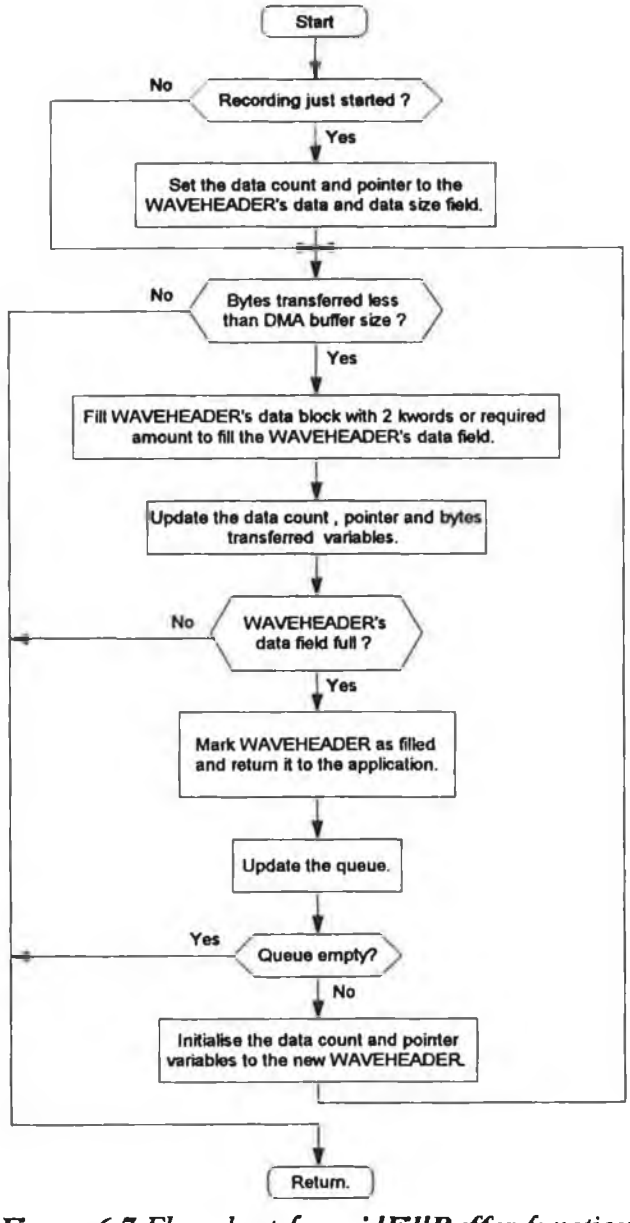


Figure 6.7 *Flowchart for* ***widFillBuffer*** *function.*

Initially the data queue pointer, *glpWIQueue*, is checked to determine if there is a data queue. If there is no queue then the function returns immediately. Otherwise the *wBytesTransfered* variable is set to zero. A *WHILE* loop is entered which will empty

the DMA buffer if the sum total of the data fields in all the *WAVEHEADER*s in the data queue is large enough. This *WHILE* loop compares the *wBytesTransferred* variable with the DMA buffer size and breaks from the loop when they are equal, which is when a DMA buffer has been emptied.

If the data count *(dwCurInCount)* and data pointer *(hpCurInData)* variables are zero then recording has just been started. These variables are then initialised to the *WAVEHEADER's data size* and *data field* address respectively. If the *WAVEHEADER's data field* is large enough, the DMA buffer is transferred to this field and the data count and pointer variables are updated. If the *data field* is not large enough then the *data field* is filled and the *WAVEHEADER* is returned with its flag field set to *WHDR_DONE*. The *WAVEHEADER* is returned using the ***widBlockFinished*** (*WAVEFIX.C*) function which also calls the ***waveCallBack*** (*WAVEFIX.C*) function to send a **MM_WIM_DATA** message [11] to the application.

The top of the data queue is then updated to the next *WAVEHEADER* using the old *WAVEHEADER's next_waveheader_pointer* field. If this field is zero then this is the end of the queue and the function returns the number of bytes transferred. If there is another *WAVEHEADER* then the data count and pointer variables are updated as before. The *wBytesTransferred* variable is updated with the number of bytes transferred from the DMA buffer and this variable ensures the DMA buffer is emptied even if several *WAVEHEADER*s are required. The *WHILE* loop will be repeated until the DMA buffer is empty or the data queue is reduced to zero.

6.8 Maintaining Continuous Playback

Playback is started once a *WAVEHEADER* has been sent to the waveform output device. The ***wodWrite*** (*WAVEA.ASM*) function, which adds *WAVEHEADER*s to the data queue, is responsible for starting playback. Playback is maintained if the application can refill and return played *WAVEHEADER*s to the waveform device before the data queue is empty. As explained previously two interrupts are responsible for accessing the waveform device's Swinging Buffers, the DMA and the Switching interrupt. The process of filling an empty Swinging Buffer from the data queue is described next.

When a Swinging Buffer has been emptied the Switching interrupt will be generated. The resulting ISR, ***Switch_ISR***, will set up a DMA session to transfer the 2 kwords from the full DMA ping buffer to the empty Swinging Buffer. The ISR then

calls the ***wodLoadDMABuffer*** *(WAVEFIX.C)* function which fills the DMA pong buffer from the *WAVEHEADER's data field*. The *TC* signal from this DMA session will generate the DMA interrupt whose ISR, ***DMA_ISR***, will setup another DMA session. This DMA session will transfer the next 2 kwords from the full DMA pong buffer to the Swinging Buffer while the empty DMA ping buffer is filled from the *WAVEHEADER's data field*. The DMA interrupts will be processed until the Swinging Buffer has been filled. Seven DMA interrupts are required to fill the Swinging Buffer along with the Switching interrupt (7x2k + 2k = 16k). A count variable determines when seven DMA interrupts have been processed and on the eighth interrupt this count is reset for the next set of DMA interrupts. The *gbDMABuffer* flag determines which DMA ping-pong buffer is full and which is empty.

The start up process is different to the normal transfer process, and is started by the ***wodKickstartWaveOutput*** (*WAVEA.ASM*) function. First, both DMA ping-pong buffers are filled if there is sufficient data in the data queue. Next, a DMA session is set up to transfer the 2 kwords from the DMA ping buffer to the waveform device. Both of the Swinging Buffers are empty initially, therefore when the first Swinging Buffer has been filled (after the seven DMA interrupts), a Switching and a DMA interrupt will be generated simultaneously. There is no DMA conflict because this eighth DMA interrupt only resets its count variable. The normal playback operations described earlier occur from now on.

6.8.1 The *wodLoadDMABuffer* Function

The ***wodLoadDMABuffer*** (*WAVEFIX.C*) function is called from the Switching and DMA ISRs and is responsible for filling the empty DMA buffer with data from the *data field* of the *WAVEHEADER* which is at the top of the *WAVEHEADER* queue. This function also maintains the queue and returns empty *WAVEHEADERs* to the application along with controlling the looping of *WAVEHEADERs*. The parameters passed are a far pointer to the DMA ping-pong buffer and the size of this buffer. The flowchart is described in Figure 6.8.

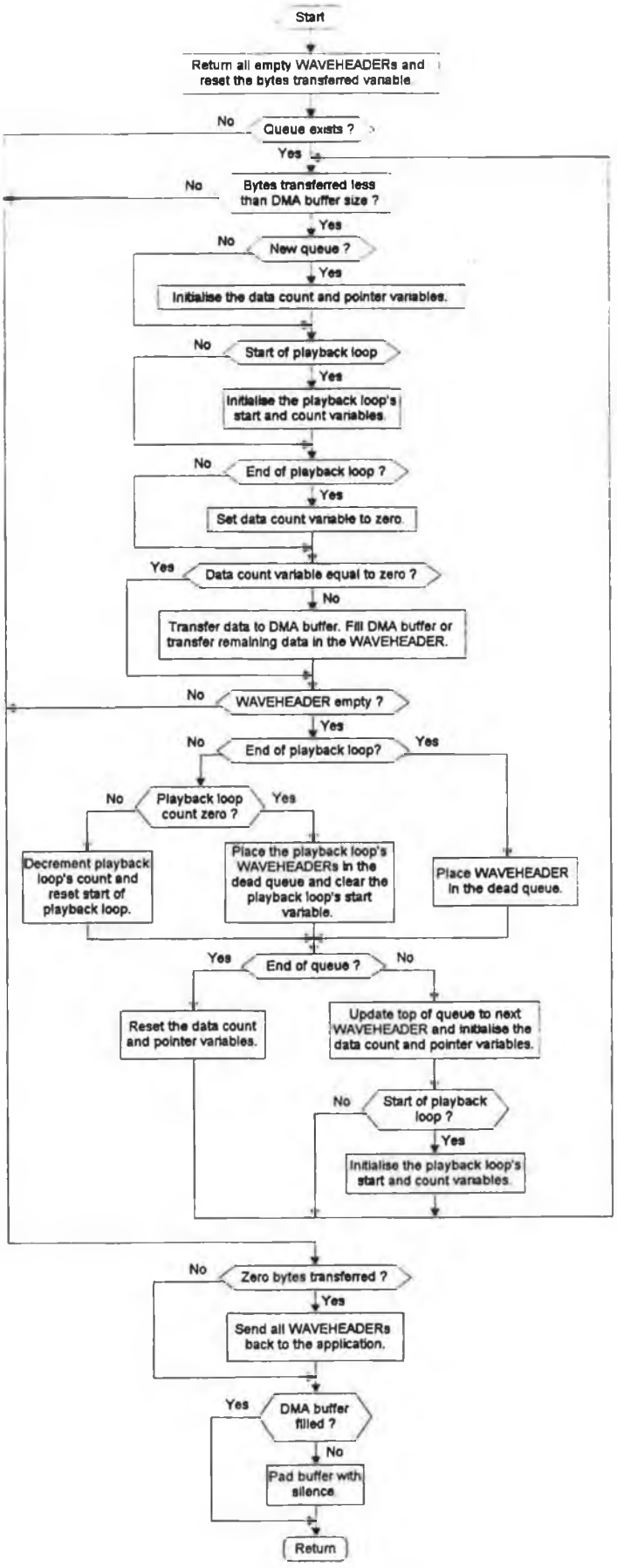


Figure 6.8 Flowchart for the ***wodLoadDMABuffer*** *function.*

This function will first return any empty *WAVEHEADER*s with the ***wodPostAllHeaders*** *(WAVEFIX.C)*. Empty *WAVEHEADER*s are placed in their own dead queue (*lpDeadHead*s) which the ***wodPostAllHeaders*** function walks down, sending each *WAVEHEADER* back to the application. Next, the *wBytesTransferred* variable is set to zero. This variable ensures that the DMA buffer is filled unless there is insufficient data in the queue.

The *WHILE* loop compares the *wBytesTransferred* variable with the DMA buffer size and breaks from the loop when they are equal which is when a DMA buffer has been filled. This loop performs the following operations on the DMA buffer and the data queue. If the queue is new then the data count *(dwCurCount)* and data pointer *(hpCurData)* variables are initialised from the *WAVEHEADER* at the top of the queue. If there is no queue then the DMA buffer is filled with silence before the function returns.

Next the *flags* field of the *WAVEHEADER* is examined to determine if this is the start of a playback loop. The *lpLoopStart* and *dwLoopCount* variables are copied from the *WAVEHEADER* fields if the *WAVEHEADER's flag field* is set to *dwLoops*. Next the *bBreakLoop* flag is examined to determine if the user has requested a break in the playback loop process. The playback loop's start and count variables are examined to determine if this is the end of a playback loop. If it is, then the data count variable is set to zero and the DMA buffer is filled with silence (0000h) before the function returns. The data count variable is compared with zero to determine if data is to be transferred to a DMA ping-pong buffer. The empty DMA ping-pong buffer is filled with the data pointed to by the *WAVEHEADER* (the *WAVEHEADER's data field*).

The data count, data pointer and the *wBytesTransferred* variables are then updated. If the data count variable is zero then the *WAVEHEADER* is empty and the queue must be updated. If a playback loop has not finished and the *WHDR_ENDLOOP* flag has not been set in the *WAVEHEADER's flags* field, then the *dwLoopCount* variable is compared with zero. If the *dwLoopCount* variable is zero then the empty playback loop *WAVEHEADERs* are placed in the dead queue and the top of the queue is updated to the next *WAVEHEADER*, otherwise the *dwLoopCount* variable is decremented and the playback loop restarted. If there is no playback loop active then the *WAVEHEADER* is placed in the dead queue and the top of the queue is updated to the

next *WAVEHEADER*. The above *WHILE* loop is repeated until the DMA buffer has been filled or the queue is empty.

The *WAVEHEADER*s in the dead queue are returned if no data was transferred to the DMA buffer. This may occur if a zero length *WAVEHEADER* was sent to the waveform device. Returning these dead *WAVEHEADER*s may cause the application to restart the waveform device by sending more *WAVEHEADER*s. The DMA buffer is padded with silence if the DMA buffer was not filled. The function ***MemFillSilence*** (*MMTEACH.ASM*) fills the buffer with zeroes. The ***wodLoadDMABuffer*** returns the number of bytes transferred, *wBytesTransferred*.

6.9 Summary

This chapter describes the layout of the MTP's waveform device driver. The functions which the driver must support are explained, namely **DriverProc**, **WidMessage** and **wodMessage** along with the Windows messages processed by each. The procedure for communicating with the application is also described long with the algorithms employed by the driver during recording or playback.

Chapter 7

Debugging the Preset Faults of the MTP

7.1 Introduction

This chapter explains the procedure for debugging the Multimedia Teaching Platform when hardware and software faults are introduced. General hardware and software debugging flow charts for recording and playback are described and are used to detect the source of various deliberate faults. Two example faults, one a hardware recording fault and the other a software playback fault, are debugged using these flowcharts. The range of possible software and hardware faults is also briefly described.

7.2 Hardware and Software Debugging Aids

The two most common hardware debugging aids are a logic analyser and a digital storage oscilloscope while the software aids are the debugging applications from MICROSOFT or BORLAND for debugging Windows programs developed in their respective development environments [2]. These software debugging aids provide source code debugging of the Windows application and allow tracing of the messages received and sent by an application. Another valuable debugging technique is the use of test files and test signals for playback and recording respectively. These signals provide continuity and consistency testing for the hardware and software.

7.2.1 Logic Analyser

The logic analyser provides a digital snap-shot of the signals it is sampling. It can typically view up to sixteen signals over a wide range of time intervals, from nanoseconds (ns) to seconds. The general operating mode of the sound card can be easily tested by viewing the sequences of Swinging Buffer read and write requests, Swinging Buffer switches, interrupts and DMA transfers and comparing them with the expected signal sequences.

7.2.2 Digital Storage Oscilloscope

The digital storage oscilloscope provides an analogue snap-shot of the signals but as with the normal oscilloscope it is limited to displaying only two signals at a time. It displays the signal's actual voltage levels and is therefore suitable for examining the analogue output signal levels when test files are being replayed.

7.2.3 Test Signals and Test Files

Test signals and test files play an important part in detecting and identifying faults and in confirming that the sound card's recording and playback modes are operating correctly. These file and signal tests are described in Chapter 3, Section 3.5.4 and are briefly discussed here. These tests allow comparisons between adjacent samples and allow tracing of the samples through the data paths of the sound card. Test signals cannot be generated for the digital recording mode but this does not pose too much of a problem because its operation is nearly identical to the analogue recording mode.

Test signals are used to test the analogue recording mode of the sound card by examining the recorded file and comparing it with the expected recording. These signals also allow samples to be traced through the sound card thereby verifying its operation. Muting one channel allows the card's channel consistency to be verified. If one sample is lost the signal will switch channels.

Several test files were generated for testing playback continuity which included a ramp waveform, a sine waveform and a one channel ramp waveform. These files are replayed in the analogue and digital modes and their outputs are stored with the digital storage oscilloscope for examination. For the digital mode the signal from the digital amplifier's analogue output is stored. The one channel ramp waveform allows playback consistency to be verified.

These file and signal tests are one of the first steps taken when attempting to identify an unknown hardware or software fault. They provide invaluable information regarding the source of faults because they provide a reference against which the actual signals from the oscilloscope or logic analyser can be compared.

7.2.4 Test Points

The most important signals in each section are labelled as test points in their schematic diagrams. The test points allow the student to quickly determine the operating status of

each particular section and also reduces the time to locate signals on the card. These test points are all readily available on the sound card and none are contained internally in the FPGA. The first two letters of each label refers to the section they belong to, for example in the Address Decoding section, ADTP-2 is the enable signal for the lower data byte's System Bus buffers (U-22). The test points for each section are tabulated in Appendix B, Tables B1 to B7.

7.2.5 Software Debugging Aids

The debugging packages from MICROSOFT and BORLAND provide similar debugging utilities. Both include utilities to trace the messages received by the Windows application and to examine Windows global memory. They also contain a source debugger for stepping through Windows programs. Breakpoints can be set and the application's variables and local memory can be examined. Unfortunately the device driver cannot be debugged with these debugging packages [2].

7.3 Design Faults

The hardware faults can be introduced into the circuit by replacing the source code in the FPGA's PROM while software faults are introduced by providing faulty source code for the device driver or the Windows application source code. There is a huge range of possible software and hardware faults, but the example faults chosen are ones that are easily identifiable without requiring detailed knowledge of the hardware or software.

The software faults are selected to highlight a particular aspect of Windows Multimedia such as maintaining continuous data flow. The software faults are explained in Section 7.5 while the hardware faults are explained next in Section 7.4.

7.4 Hardware Design Faults

Hardware design faults can be easily introduced into the circuit without having to modify any physical connections by replacing the PROM. This PROM is responsible for programming the FPGA at power-up time.

A large number of design faults can be introduced because the FPGA implements the Address Decoding, Clock Generation, Control Logic and the DMA and Interrupt sections. To identify a hardware fault which has been traced to one of the sections

implemented in the FPGA, the student uses the FPGA's schematic design entry and programming software. This allows the student to examine the particular section's logic circuits and allows simulation of the circuit to verify its operation. A hardware fault is described for each section implemented in the FPGA as follows.

7.4.1 Address Decoding Fault

One example fault for the Address Decoding section is to make the I/O port addresses for the digital and analogue recording modes identical. This will result in the serial data received by the Serial Interface in the recording modes, being the logical OR combination of the serial data from the digital receiver and the serial data from the analogue to digital converter. The symptoms for this type of fault would suggest that the fault lies in one of the following sections, Serial Interface, Address Decoding, Clock Generation or in the software configuration of the card as carried out by the device driver. Further investigation by the student would reveal that this fault lies in the Address Decoding section. Observing the test points ADTP-4 (ANALOGUE IN) and ADTP-6 (DIGITAL IN) would identify this fault.

7.4.2 DMA and Interrupt Generation Fault

One example of a fault in the DMA and Interrupt Generation section is if the Switching interrupt's generating circuit is modified to fire only once. This will result in the second Switching interrupt never being generated and the sound card ceasing further operations. The symptoms for this fault occur in every recording or playback mode. This circuit modification involves removing the t-type flip-flop of the generation circuit and replacing it with a d-type flip-flop. The possible causes of the fault's symptoms could be the interrupt service routines, DMA and Interrupt Generation section, Logic Control or incorrect initialisation of the DMA controller by the software. Examination by the student of the sound card's operating modes would confirm that the second Switching interrupt (test point DITP-10) was not being generated and further investigation would pinpoint the error.

7.4.3 Clock Generation Fault

One example of a fault for the Clock Generation section is to invert the serial shift clock of the SiPo converter of the Serial Interface, resulting in the serial data from the I/O Interface being sampled on their transitional edges. This would be achieved by removing the inverter which converts the PiSo serial shift clock into the SiPo serial shift clock. This would cause random errors to occur frequently in the recorded data in both analogue and digital modes. The possible causes of the symptoms of this fault might be in the Serial Interface section, Address Decoding section, the DMA controller initialised to an incorrect memory address, the software writing wrong data to the hard disk or the Control Logic producing simultaneous data buffer enables for the Swinging Buffers resulting in data contention. The source of random errors is usually difficult to detect but the Serial Interface is one of the first places to be examined by the student and this would quickly reveal this fault. Observing the test point SITP-4 (SIPOSCK) in the Serial Interface would force the student to investigate the generation of this clock signal in the Clock Generation section using the FPGA's functional test software.

7.4.4 Control Logic Fault

One example of a fault in the Control Logic section is if the switching of the WENbar signals is not delayed long enough to allow the final Swinging Buffers' read or write requests to be completed. This will cause the channels to be exchanged at every Swinging Buffer switch. This is a subtle fault because losing one sample every 16 k samples is not likely to be detected by the human ear. But if the sound card was being tested for channel consistency with test files it would be very evident if the one channel ramp waveform was played back. Therefore this fault could be used when the card's performance was being thoroughly tested. Identification of this fault would involve examination of the Control Logic section with the FPGA's software.

7.5 Software Faults

The scope of possible software faults is vast but the faults selected are those which highlight a particular aspect of Windows Multimedia such as data queue management. The following faults highlight the operation of the hardware interrupt service routines, data queue management, file formats and the correct termination of the playback operation.

7.5.1 Interrupt Service Routine Fault

One example of a fault in the interrupt service routine is if the DMA interrupt counter is not reset to zero after the eighth DMA interrupt. This will corrupt recording and playback after the first switch of the Swinging Buffers. The DMA interrupts will not set up DMA sessions or transfer data from the application's data buffer resulting in the Swinging Buffers never being emptied or filled fully. This fault is easily located, once the general operation of the sound card is investigated using the logic analyser. The second DMA interrupt does not generate another DMA interrupt when filling or emptying the second Swinging Buffer after the first Swinging Buffer switch.

7.5.2 File Access Fault

One example of a fault in the area of file access is if the format chunk in the recorded WAVE file is incorrectly set to the PCM 44.1 kHz mono 16 bit format which none of the playback modes support. The application will display an error informing the user that the requested playback file's format is not supported by the sound card. This is a software error which does not require any hardware debugging. The recorded file will be examined to determine its format before investigating the file creation function for correct file creation, particularly creating the format subchunk.

7.5.3 Device Driver Fault

One example of a fault in the device driver is if the silence padding function of the device driver's playback mode writes the incorrect value into the partially filled DMA buffer. An audible click will be heard at the end of playback. The value 8000h is written instead of 0000h which results in the output signal going to its maximum negative level causing the audible click. The symptoms of this fault would point to a software problem rather than a hardware fault. This fault may not be heard or observed with a test file if the test file's length is a multiple of the DMA buffers size, 4 kbytes. If it is a multiple, then this silence padding function is never used and no fault is observed on the digital storage oscilloscope. This discrepancy would point to a fault in the silence padding function.

7.5.4 Data Queue Maintenance Fault

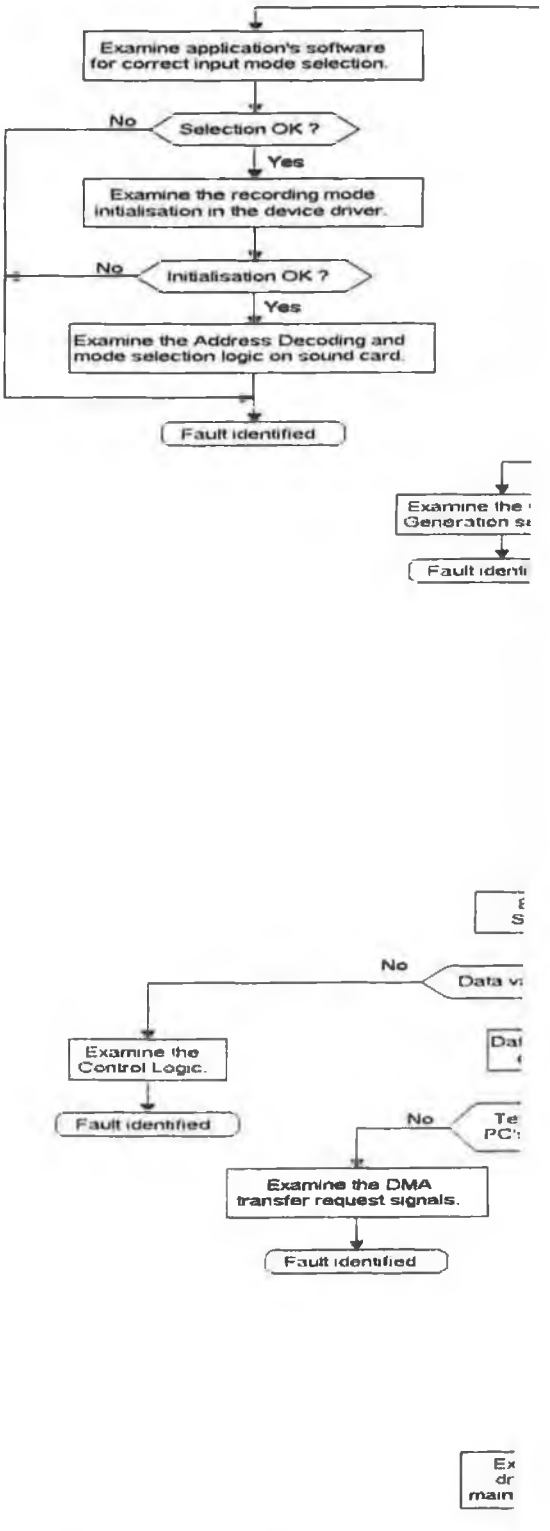
One example of a fault in the area of data queue maintenance is if the size of the data buffers used by the application are set too large, continuous recording or playback cannot be maintained. Accessing the disk takes up too much of the systems time resulting in the interrupts being serviced too late to maintain the data flow between the Swinging Buffers and the I/O Interface. Observing the sound card's operation for small recording or playback lengths would prove that the hardware was performing correctly. This would cause the student to concentrate on the management of the data queue by the software.

7.6 Debugging a Recording Fault

The flowchart for identifying a general recording fault is shown in Figure 7.1 and it describes the process of locating the area of the Multimedia Teaching Platform which contains the fault. There are different types of recording faults such as corrupted recordings faults or terminal faults where the recording process is never completed. The debugging steps for the different recording faults are grouped together to form a general debugging flowchart. Once the area of the fault has been identified, the system is examined in detail and compared against its normal operating process to identify the cause of the fault. Non terminal recording faults are usually noticed when the recorded file is replayed and an audible difference is detected by the listener when compared with the original audio signal. An important consideration when beginning to debug a non terminal recording fault is to first confirm it is not a playback fault.

7.6.1 Description of an Example Recording Error

Consider the following example of a terminal recording fault. The recording process is started by the user selecting the *Record* push button in the analogue recording dialog box (see Appendix D Figure D.2). The user knows recording has been started because the text of the *Cancel* push button in the recording digital box has been changed to *Stop*. The change only occurs when recording has been started by the application and its text returns to *Cancel* when recording has finished. This push button's text is changed to indicate its different function during recording, which is to stop recording, whereas normally this push button closes the dialog box.



SBuffer = Swinging Buffer

Figure 7.1 Flowchar

The recording dialog box's scroll bar shows the percentage of recording which has occurred but this is not updated, indicating that recording is not taking place. Also the push button's text is never changed back to *Cancel* from *Stop*. After the user has manually stopped recording with this *Stop* push button and attempts to replay the recorded file, a message box is displayed stating that the selected file is corrupt.

This terminal fault could be in the hardware or software. For example, the hardware may be operating perfectly while the software is not maintaining the data queue or the hardware may not be generating the interrupt service routines. The steps involved in locating this recording fault are now described.

7.6.2 Debugging Process

The first step when debugging a recording fault is to determine if the fault is common to both recording modes, analogue and digital. The second step is to determine whether a signal is present on the analogue inputs, otherwise further investigation will be pointless. If it is not common then this immediately reduces the possible sources of the fault. The example fault is common to both recording modes, therefore tests signals are placed on the analogue inputs. The oscilloscope is used to verify that these signals are present on the inputs of the analogue to digital converter (tests points AITP-6 and AITP-7).

The recording process is repeated in the analogue mode and the recorded file is examined. If the file is not corrupt then recording has taken place which points to a playback error rather than a recording error. The example fault in this case does not produce a valid recording. The file is viewed using an editor which can display the file in hexadecimal. This will reveal if the file is in the correct RIFF WAVE format and if the file creation function of the application is correct. For the example fault this test reveals that one Swinging Buffer's data (32 kbytes) was recorded to disk, but the file was not properly closed because the data size field of the *RIFF* chunk does not equal the size of the combined *format* and *data* subchunks of the file. The size of the recorded data suggests a hardware or an interrupt fault resulting in only the first Swinging Buffer being emptied and written to disk.

The logic analyser is now used to examine the recording operation of the sound card. This will determine if the hardware and software are maintaining recording for the specified length of time and will provide valuable information on the possible sources

of the fault. Further investigation is always required to pinpoint the fault. The following signals, Switching and DMA interrupts (DITP-10 and DITP-9), DACK7bar (DITP-5), DMA's TC (DITP-11), MResetbar(ADTP-8), Swinging Buffers full and empty flags (DITP-1, 2, 3 and 4), WEN1bar (DITP-12) and the Swinging Buffers' read and write requests (SBTP-6, 7, 8 and 9) are sampled by the logic analyser. The logic analyser is configured to trigger when the MResetbar signal is activated at the start of recording, and begins sampling for approximately one second. These signals will determine if the card is sustaining recording. If recording is being sustained then these signals should be identical to those shown in Figure 7.2.

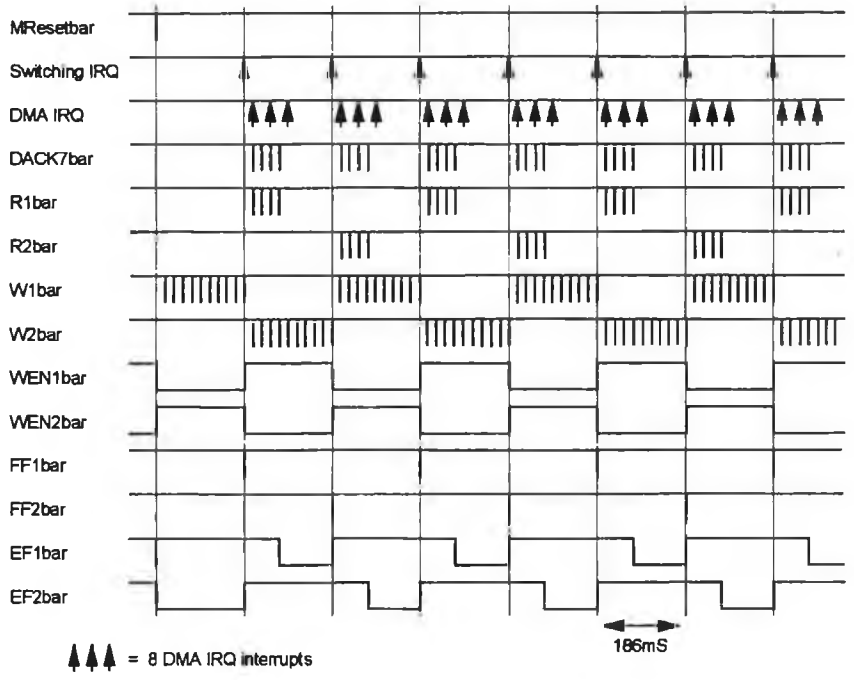


***Figure 7.2** Operation of the Sound Card during Recording.*

In the example fault, recording is not being sustained as shown in Figure 7.3. Examination of these signals reveal that the first Swinging Buffer is filled and subsequently emptied by the Switching and DMA interrupts but the second Swinging Buffer is not emptied. The WENbar signals do not toggle after the first switch, resulting in the Switching interrupt not being generated which in turn does not generate the DMA interrupts required to empty the second Swinging Buffer. Therefore the fault is identified as being situated in the switching circuits of the Control Logic section.

This fault requires detailed examination of the switching circuits to determine why a switch occurs only after the first Swinging Buffer has been filled. The playback modes are unaffected which indicates that the fault lies in the recording mode's triggering circuits for switching the Swinging Buffers. The schematic diagrams for the Control Logic section used when generating the PROM program for the FPGA are examined with the FPGA's schematic design entry and programming software package. Examination of these switching circuits reveals that switching is only triggered by the first Swinging Buffer's empty flag whereas it should be the logical AND combination of the two Swinging Buffers' empty flags.

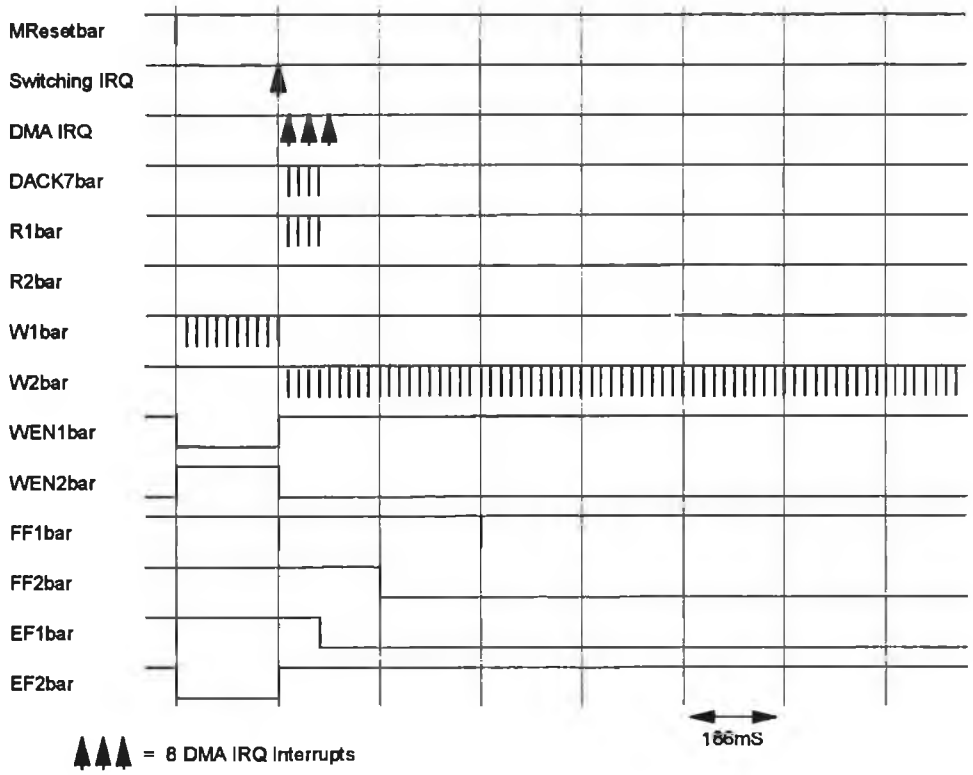
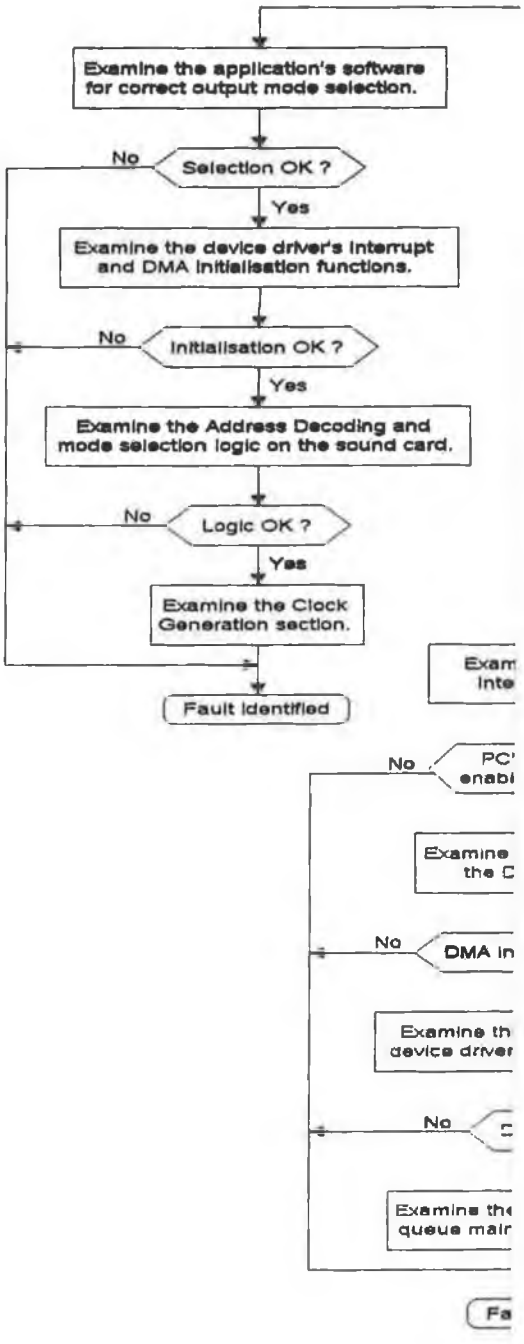


Figure 7.3 Fault in both Recording Modes of the Sound Card.

7.7 Debugging a Playback Fault

The general flowchart for identifying a playback fault is shown in Figure 7.4. This is a general debugging flowchart which describes the steps required to identify the two types of playback faults, terminal or corrupted (non terminal) playback. The flowchart will pinpoint the area where the fault lies and the student is then required to investigate this area further.



SBuffer = Swinging Buffer

Figure 7.4 Flowchar

7.7.1 Description of the Playback Fault

Consider the following non terminal playback fault. A file is played and the application starts and terminates playback correctly but the audio playback is corrupt. The corruption is repetitive and consists of skips in the audio playback. The playback process can be paused and restarted correctly so it would not appear to be a hardware fault.

7.7.2 Debugging Process

The first step is to determine if the fault is present in both playback modes. In this case the fault is present in both modes, therefore the analogue mode is used for convenience to identify this fault. Otherwise the digital mode signal must be converted to analogue by the digital amplifier before being stored with the digital storage scope. The analogue removes the need for the digital amplifier. The one channel ramp test file is replayed in the analogue mode and is examined with the digital storage oscilloscope (test points AITP-8 and AITP-9). The output waveforms are corrupted as shown in Figure 7.5. The channels are being inadvertently switched and a small section of the ramp is being lost at every channel switch. The frequency of this fault is 340 ms, which suggests a software rather than a hardware problem. A channel switch could be caused by a sample being lost during every Swinging Buffer switch but the frequency of this fault does not match the Swinging Buffer switching frequency (186 ms).

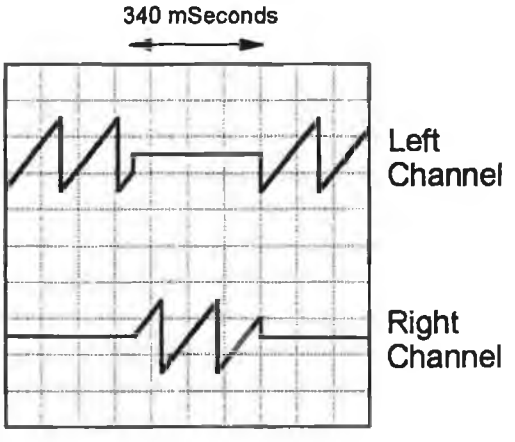


Figure 7.5 Output Waveforms observed on the Storage Oscilloscope.

The missing parts of the waveform also point to a periodic fault and it is also noted that the continuous parts of the waveform are not corrupt. The general playback operation, as described in Section 7.6.2, is nevertheless investigated with the logic analyser, to confirm that continuous playback is occurring. The waveforms in Figure 7.6 were observed indicating correct playback operation of the sound card.

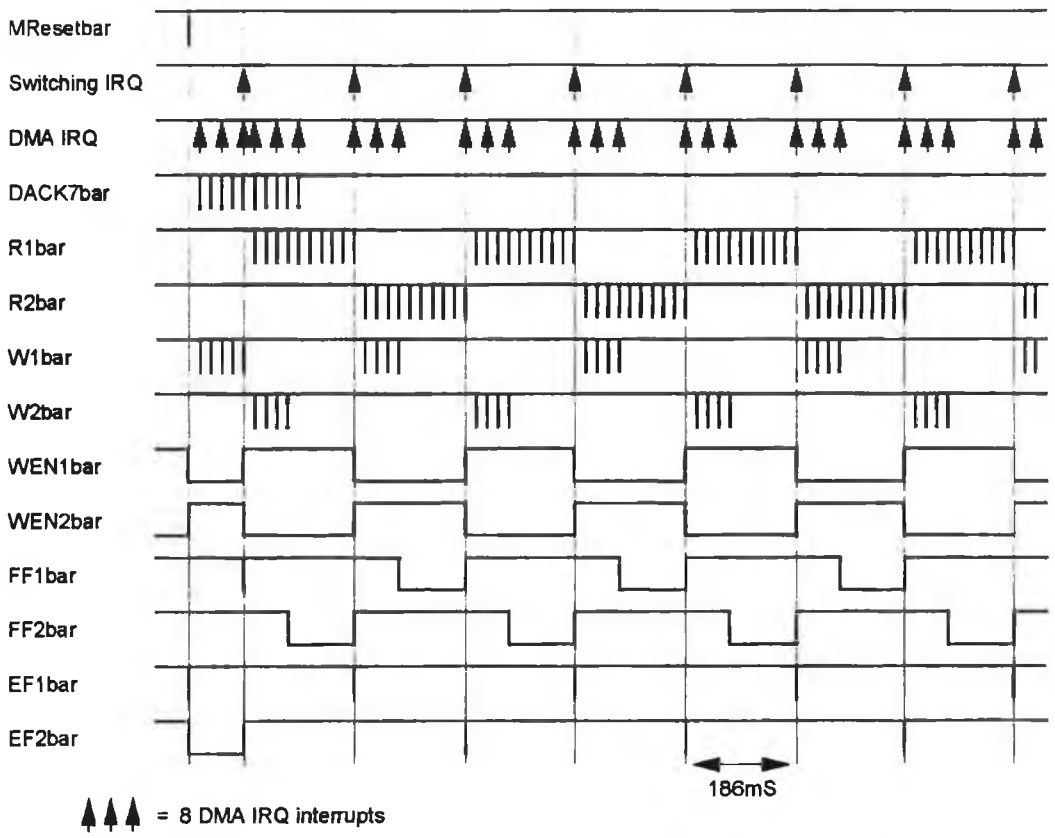


Figure 7.6 *Operation of the Sound Card during Playback.*

The oscilloscope waveforms in Figure 7.5 indicate a software fault because the fault is repetitive at a stable frequency and does not stop continuous playback. The interrupt service routines' in the device driver which are responsible for the initialisation of the DMA controller cannot be the source of the fault, because of the frequency of the fault. Therefore the application software is examined, particularly the data pointers used when reading from the file and the data pointers sent to the device driver.

The data pointers used by the application when reading data from the RIFF WAVE file were compared with the data pointers stored in the *WAVEHEADER* structures using the source code debugging applications. These pointers were found to be identical. For the example fault, the frequency of the channel switching is similar to the size of the data field *(dwBufferLength)* of the *WAVEHEADERs* sent to the device driver. A difference in the data buffer sizes used by the application's file functions and the size of the buffers used by the device driver could produce this fault. The *dwBufferLength* field of the *WAVEHEADER* dictates the size of the buffers that the device driver uses (60,000 bytes) and they were found not to be equal to the size of the data buffers used by the function (70,002 bytes) which reads from the WAVE file. This

was due to the existence of two constants, both used for the size of the data buffers. There was a local constant in the C file (MMAPP.C) that processes returned *WAVEHEADERs*, which was used instead of the global constant to set the size of the *WAVEHEADER's dwBufferLength* field. Therefore, there was a size difference between the data read from the WAVE file and the data sent to the sound card which resulted in the periodic channel switching and loss of data.

7.8 Summary

This chapter introduces the debugging techniques for identifying the MTP's preset faults. A selection of hardware and software MTP faults are described and two of these are debugged using the MTP's general playback and recording flowcharts. These flowcharts illustrates the steps taken in identifying the section of the MTP that possessed one of these preset faults.

Chapter 8

Summary and Future Development of the MTP

8.1 Introduction

This chapter discusses and summarises the Multimedia Teaching Platform and examines areas where further development work can be carried out.

8.2 The Multimedia Teaching Platform

The objective of this thesis was to develop a teaching platform for the design of a Windows Multimedia device. This involved the development of a Windows application and device driver and the design, construction and testing of a digital and analogue sound card. The MTP as presented successfully achieved the objective.

The sound card design involved the use of advanced programmable logic in the form of a FPGA to provide the ability to introduce preset hardware faults. The sound card supported both analogue audio signals and the consumer digital format (Appendix A) found in commercial digital audio equipment, such as CD players and DCC players.

The software was composed of a Windows application and a Standard mode device driver. The driver was written in a mixture of C and 80286 assembler and conformed fully to the Windows Multimedia standard for waveform device drivers. The application used the low level multimedia audio functions defined in the MMSYSTEM module of Windows to provide the user with greater control over playback and recording. Algorithms were also developed in the application to maintain the data queue between the application and the device driver, thus ensuring continuous recording or playback. The application also introduces the user to the files which make up a Windows application and the role each one performs.

The MTP also introduces a number of important Windows software features which are often only briefly covered, such as installation software and help files. These features of the application while not necessary, are highly desirable from the user's perspective. Their importance to a Windows application is only noticed by the user when

they are absent. Appendix H explains the process of generating a *.HLP* format help file while Appendix J describes the steps involved in creating the installation disk for the MTP application.

The unique programmable logic of the sound card and the availability of the source code for both the application and device driver allows for the introduction of preset faults in the MTP. These faults were developed to highlight the different aspects of Windows Multimedia, such as data queue management. The ease of introducing new faults provides the MTP with a powerful ability to develop new faults tailored to the student's knowledge and experience. The MTP also provides general recording and playback debugging flowcharts to assist the user in identifying the areas in which the preset faults lie. These flowcharts still require initiative from the student to pinpoint the cause of the preset fault. Solving the faults teaches the student valuable hardware and software debugging techniques

8.3 Future Development of the MTP

The next obvious development of the present MTP would be a laboratory test with undergraduate students to evaluate its performance and the suitability of the preset faults. The feedback from the students would be carefully examined to highlighting any areas of the MTP which could be improved to increase its teaching potential.

Other developments areas might include audio compression and decompression which are not addressed at present by Windows Multimedia. Compression and decompression is vital for certain multimedia file types such as video, where the size of uncompressed data is too large for present hard disks to hold more than a few minutes of uncompressed information. With the demand for better quality video and high definition digital TV *(HDTV)*, compression will become essential.

There are compression standards available but these have not been universally accepted by manufacturers which have already implemented their own compression schemes. For instance Intel use their own Indeo video compression scheme. MPEG is a prime example of a universal compression scheme which is only supported by a limited number of video capture cards. If compression and decompression are implemented independently by manufacturers this leads to incompatibility when dealing with compressed files and a compressed sound file will not be automatically playable on another manufacturer's card.

Compression and decompression capabilities could be achieved using discrete compression and decompression chips but would be inappropriate from a teaching viewpoint. Implementing the compression and decompression algorithms explicitly using a digital signal processor *(DSP)* would allow faults to be introduced into these algorithms. This would greatly increase the teaching capabilities of the MTP and provide the student with experience of compression and decompression algorithms in a real time situation. This DSP facility could be implemented using a second plug-in card or a piggy-back board. The second plug-in card would be more suitable because of the problem of accessing the chips beneath the piggy-back card when debugging the card.

When recording, the DSP would compress a full Swinging Buffer to a partially filled Swinging Buffer, and the DMA controller would transfer this compressed data to the PC's memory. The DMA transfer from the Swinging Buffers would be faster due to the reduction in the data to be transferred. A faster hard disk would also be required to allow sufficient time for the data to be compressed in real time without any data loss from the I/O Interface. When playing back compressed data, the Swinging Buffer would be partially filled by the DMA transfers which the DSP would then fill by uncompressing this compressed data.

References

[1] ***Resource Workshop: User's Guide***. Borland International Inc, California USA, 1991.

[2] ***Turbo Debugger 3.0 for Windows: User's Guide***. Borland International Inc, California USA, 1991.

[3] ***TurboC++ 3.0 for Windows: Programmer's Guide***. Borland International Inc, California USA, 1991.

[4] ***TurboC++ 3.0 for Windows: User's Guide***. Borland International Inc, California USA, 1991.

[5] Byers, T.J. ***Inside the IBM PC AT***. MicroTest Productions Inc, McGraw-Hill, New York USA, 1985.

[6] ***Volume 1 Data Book A/D Conversion ICs***. Crystal SemiConductor Corp, Houston USA, 1994.

[7] Hall, Douglas V. ***Microprocessors and Interfacing: Programming and Hardware***, 2nd Edition. McGraw-Hill International Editions. New York, USA. 1992

[8] Hogan, Thom. ***The Programmers PC Source Book***, 2nd Edition. Microsoft Press, USA, 1991

[9] Hubbard, H. ***Inside the RIFF Specification***, Dr Dobb's Journal, September 1994. United Newspapers Publication, California USA. Pages 38 to 45.

[10] Jeffries, Thomas. ***Multimedia Infrastructures***: Byte, August 1993, McGraw-Hill Publications, New York. Pages 193 to 198

[11] ***Microsoft Development Network and Library (CD-ROM)***, Microsoft Corporation, One Microsoft Way, WA 98052-6399, USA.

[12] Norton, Daniel A. ***Writing Windows Device Drivers***. Addison Wesley Publishing Company, USA, 1992.

[13] Oney, Walter. ***Examining the Windows Setup ToolKit***, Dr. Dobb's Journal, February 1994. United Newspapers Publication, California, USA. Pages 68 to 72.

[14] Petzold, Charles. ***Programming in Windows: The Microsoft Guide to Writing Applications for Windows 3***, 2nd Edition. Microsoft Press, Washington USA, 1990.

[15] Stevens, Al. ***Help for Windows Help Authors***, Dr Dobb's Journal, April 1994. United Newspapers Publication, California USA. Pages 86 to 91.

[16] ***The Programmable Logic Data Book***, Xilinx 1994

[17] LIU, Yu-Cheng and GIBSON, Glenn A. ***Microcomputer Systems: The 8086/8088 Family. Architecture, Programming and Design***, 2nd Edition. Prentice-Hall International, USA, 1986.

Appendix A

Serial Digital Audio Interface

There are two formats in use, Consumer and Professional. The serial digital audio data interface implemented in the sound card, conforms to the following Consumer format standards, IEC-958 Consumer, S/PDIF and CP-340 Type 2. The consumer format is designed for commercial consumer products, CD players for example. The professional format on the other hand is designed for professional or recording studio use. Table A.1 lists the major differences between the two formats.

Consumer	Professional
Hardware	
0.5 Vpp	5 Vpp
75 Ω Terminated Coaxial Cable	Balanced Twisted Wire Pair
Software	
Control Bits	Control Bits

***Table A.1** Differences in the Digital Audio formats.*

The digital audio format is composed of frames. There is a left and right channel subframe per frame while 192 consecutive left or right subframes form a channel status block. The subframe format is shown in Figure A.1. The subframe can have a total of

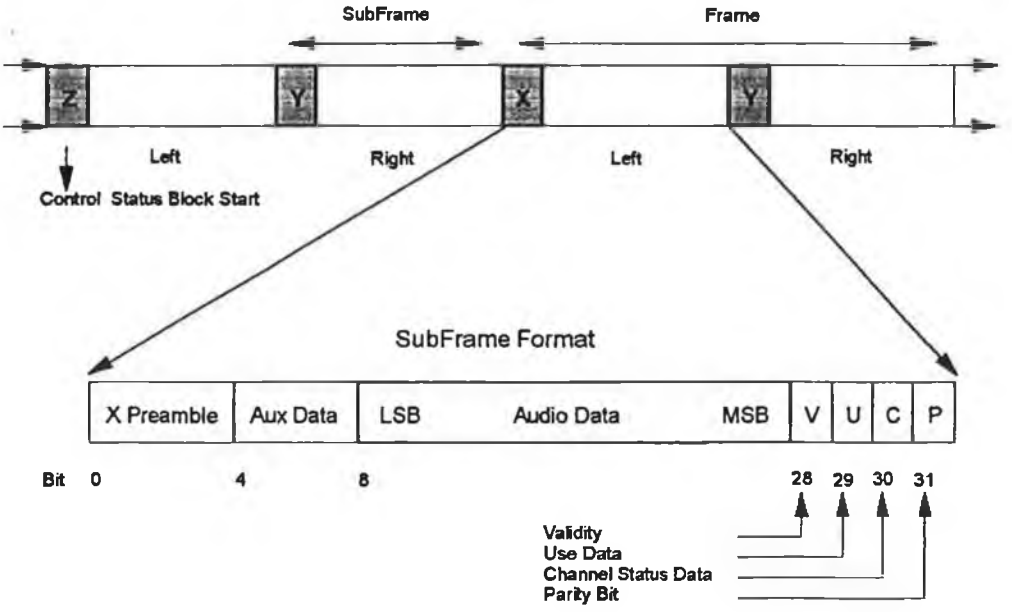


***Figure A.1** Subframe Format.*

24 audio bits but only 16 are used in the consumer format. The parity bit ensures even parity over the subframe. The validity bit indicates if the audio data is suitable for conversion to analogue audio. The user bit is designed for transmitting user information. The control information bits builds up from 192 consecutive subframes to form a channel status control block for each channel.

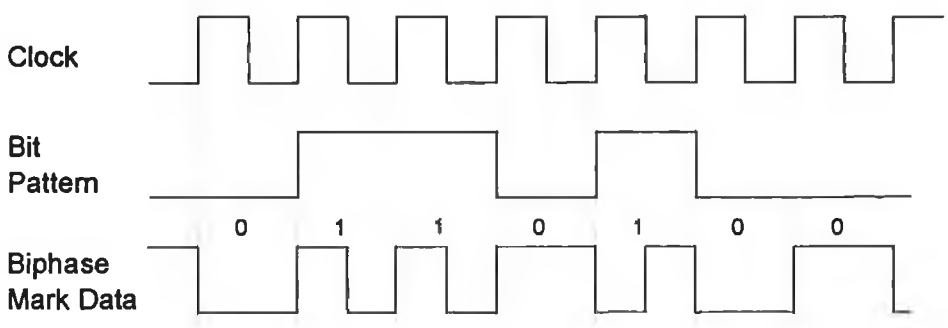


***Figure A.2** Biphase Mark Coding.*

The line coding used is Biphase Mark. Figure A.2 shows the generation of the code from the clock and bit sequence. It is this clock that can be extracted from the coded data by the receiver. This coding scheme has low D.C. content along with good clock extraction capabilities and polarity independence. The low D.C. content is due to a transition occurring at ever bit boundary as shown in Figure A.2. A one is represented by a transition within the bit boundaries while a zero has none.

Synchronisation for block and subframes are achieved through coding violations. These violations signal the start of a block or subframe and are known as preambles. Figure A.3 shows the preambles used for each subframe and status block. The arrows illustrate where coding violations have occurred due to boundary transitions not taking place.

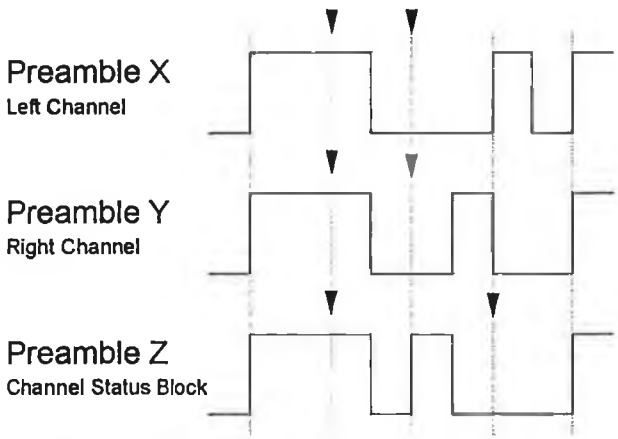


***Figure A.3** Frame and Subframe Synchronisation Preambles.*

Appendix B

Schematic Diagrams and Test Points for the Sound Card

The X labels in the scematic diagrams refer to devices contained in the FPGA while the U labels refer to external devices on the sound card. Figures B.1 to B.8 are the schematic diagrams for the card and Figure B.9 shows the physical layout of the card. The designated test points for each section are tabulated after the schematic diagrams in Tables B1 to B7.

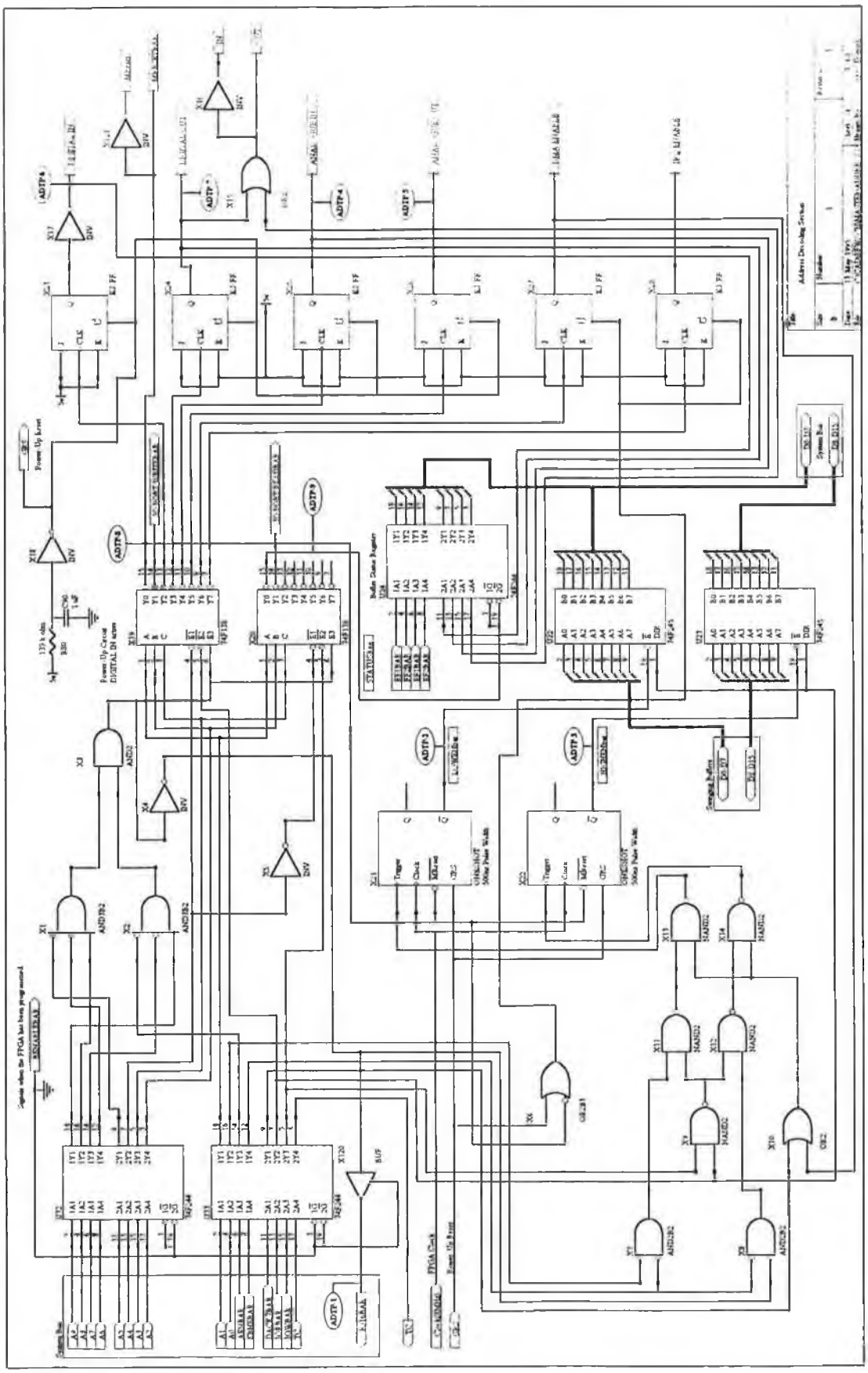


Figure B.1 *Address Decoding Section.*

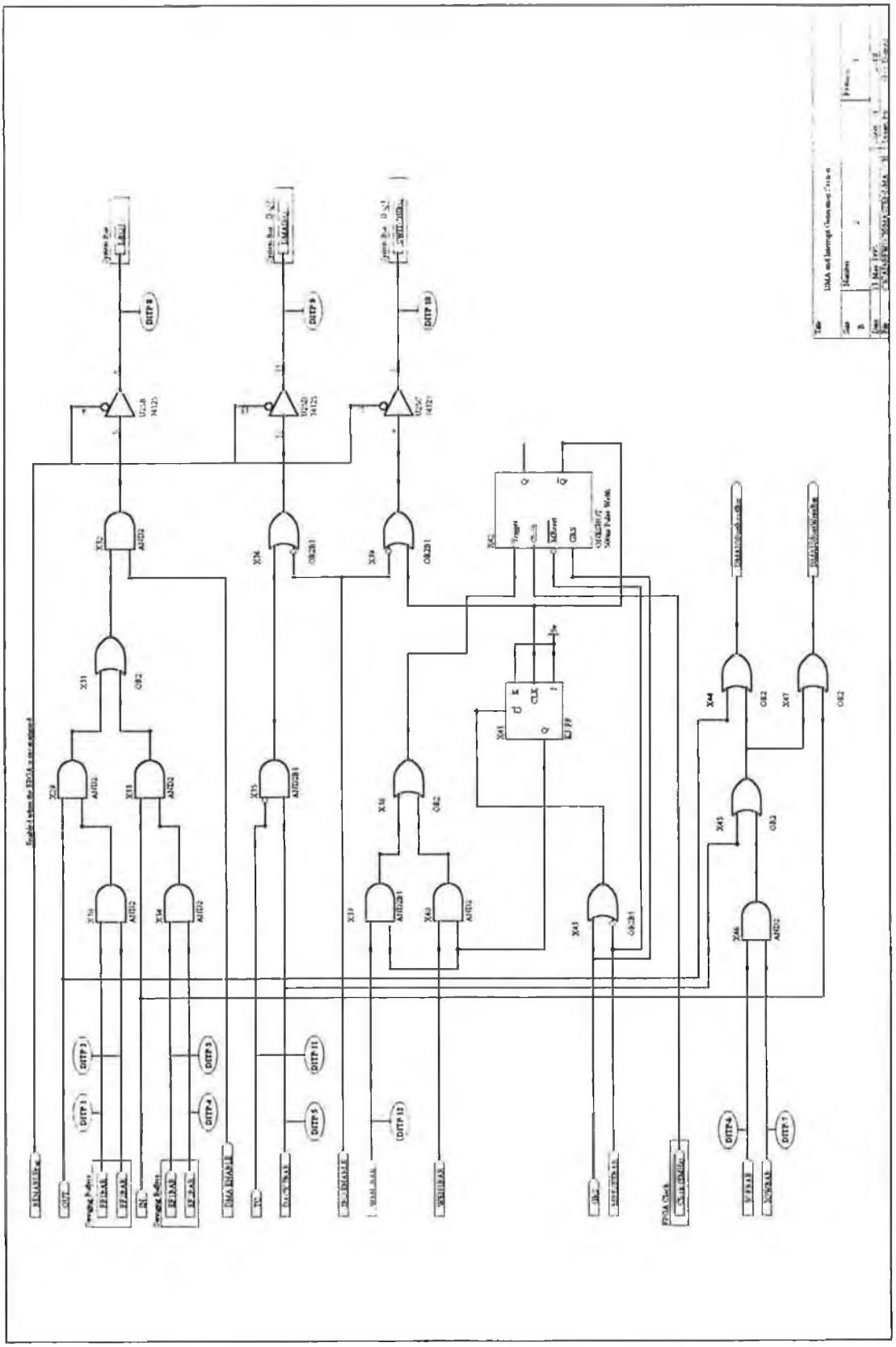


Figure B.2 *DMA and Interrupt Generation Section.*

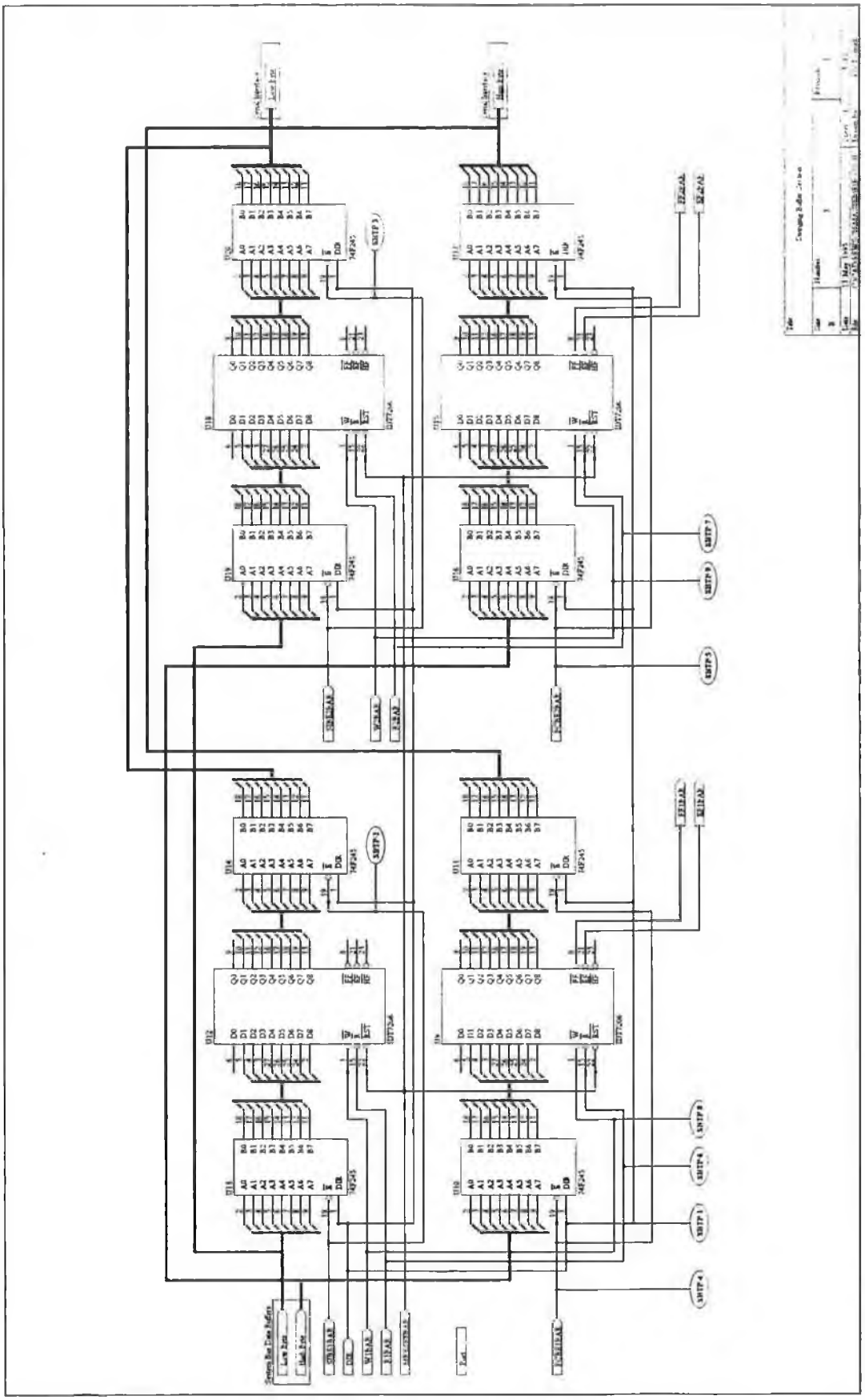
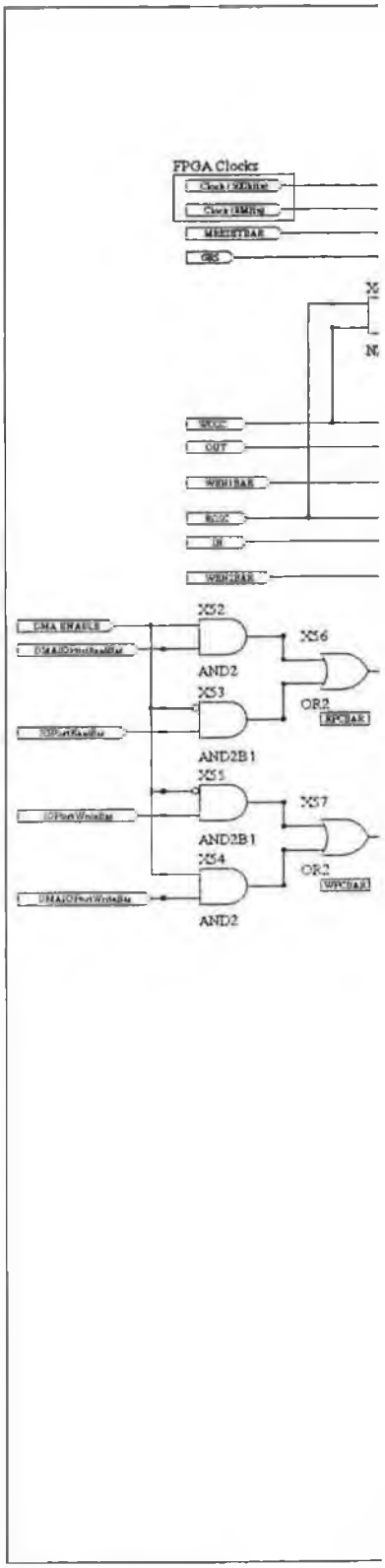


Figure B.3 *Swinging Buffers Section.*


FPGA Clocks
OUT
DMA ENABLE
XS2
AND2
XS6
XS3
OR2
AND2B1
XS5
XS7
AND2B1
XS4
OR2
AND2

Figure

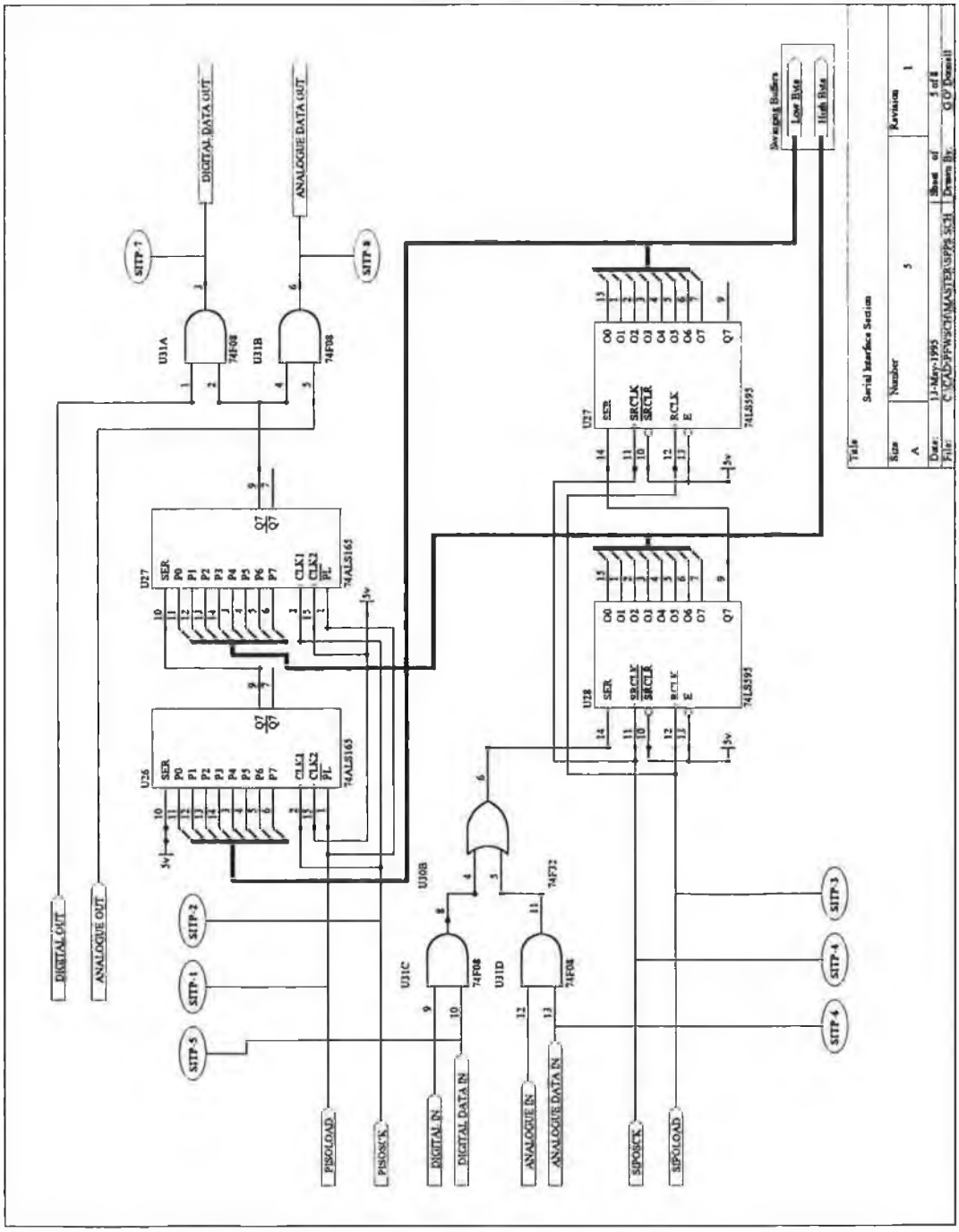


Figure B.5 Serial Interface Section.

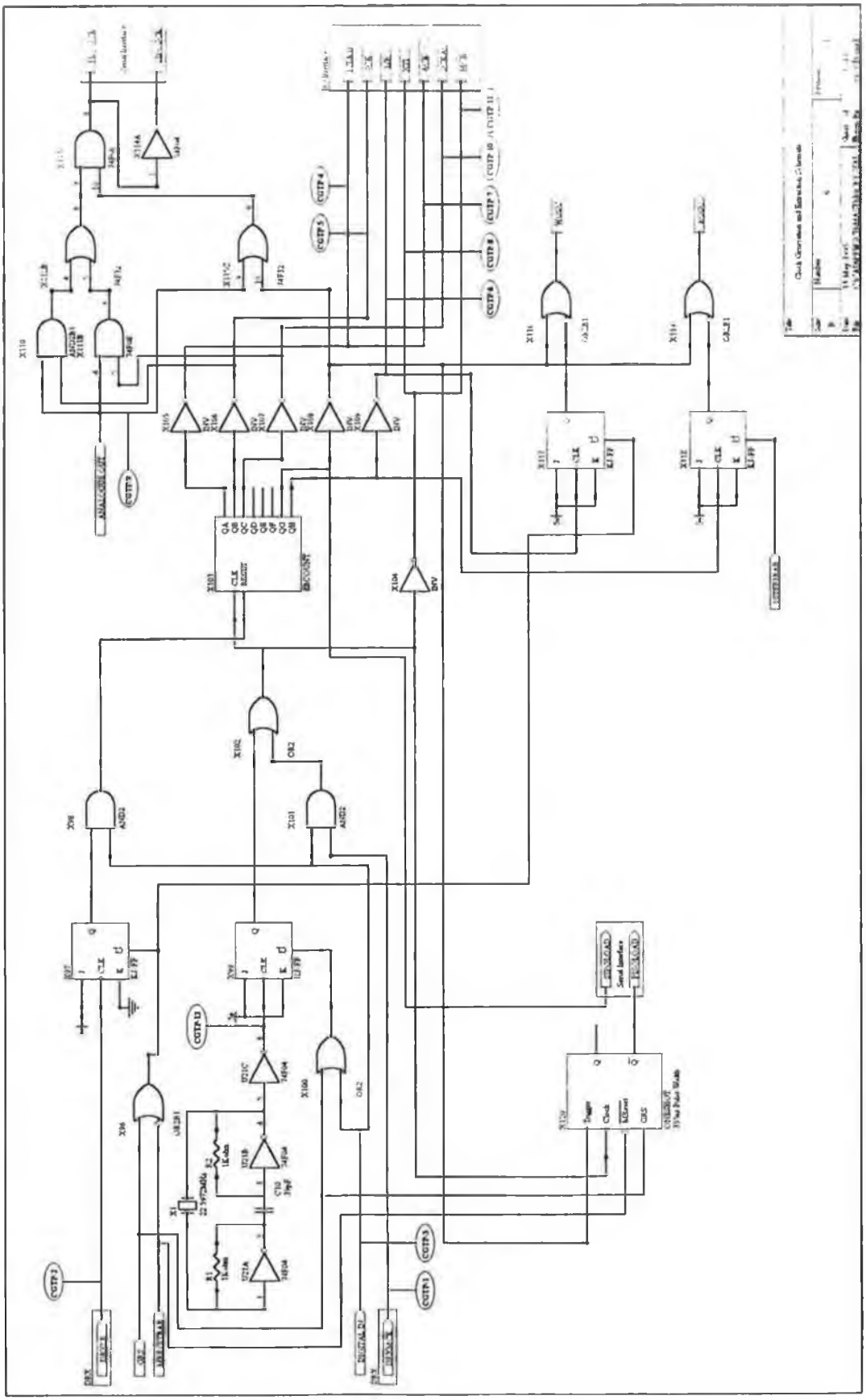


Figure B.6 *Clock Generation Section.*

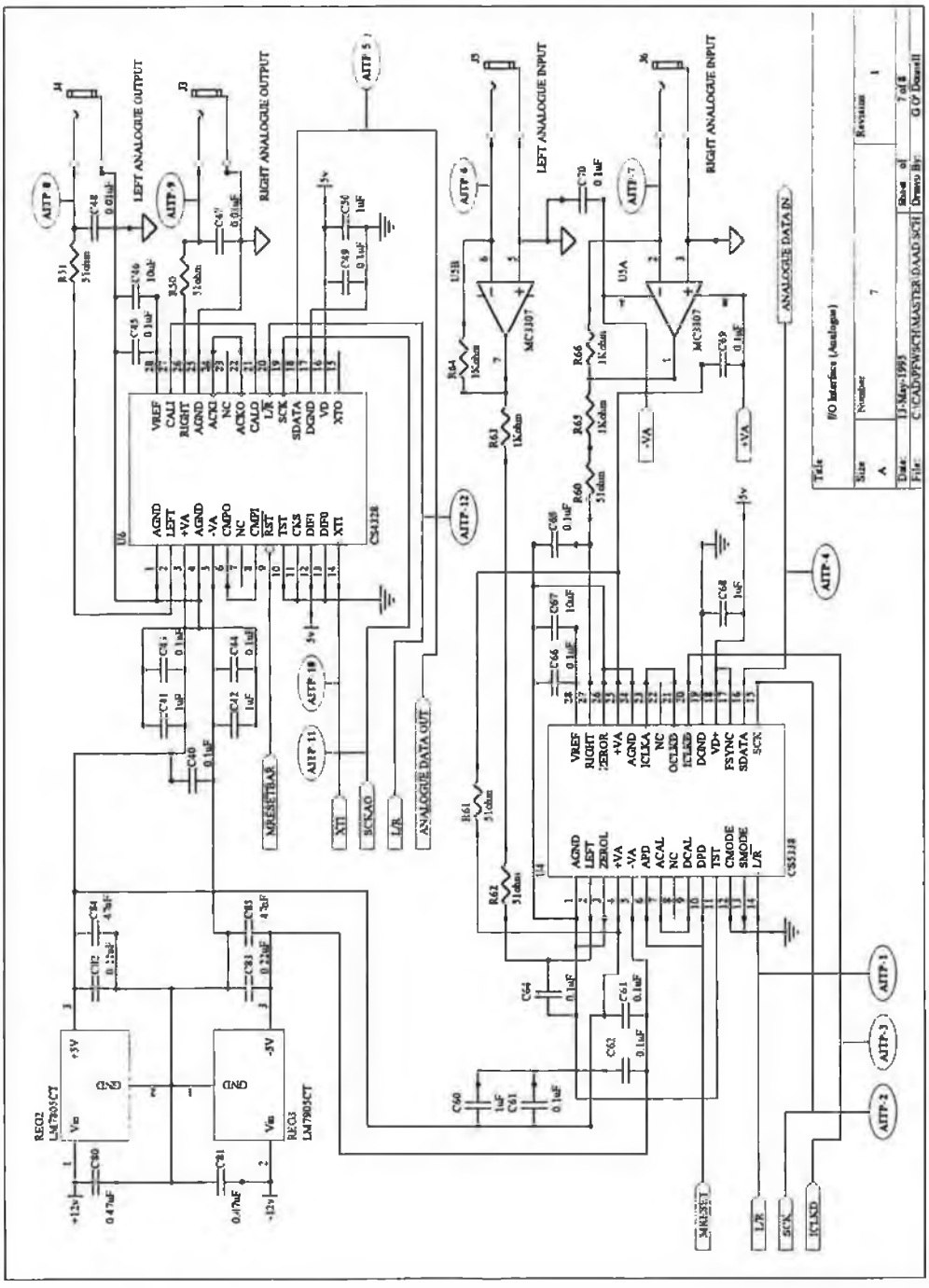


***Figure B.7** Analogue I/O Interface Section.*

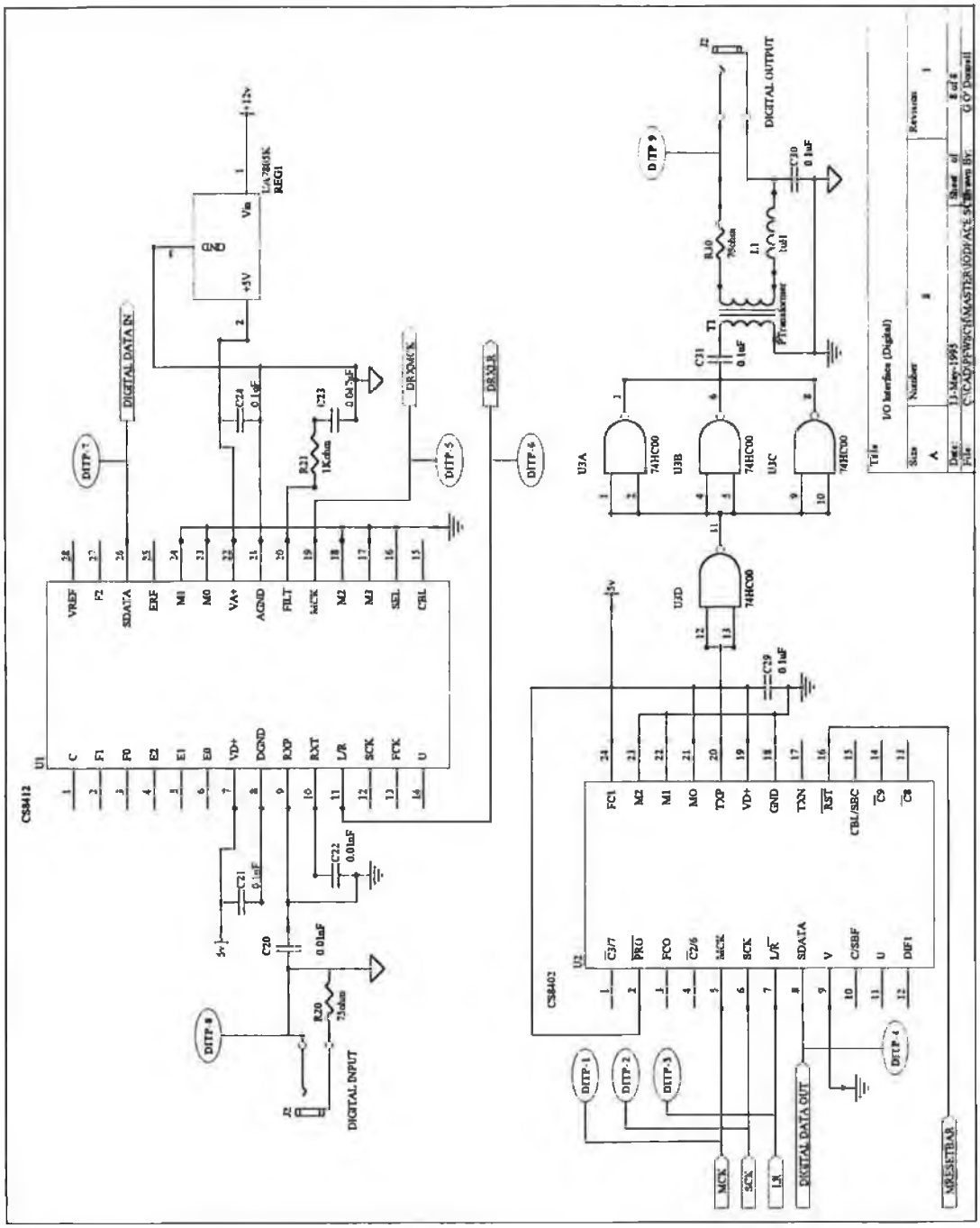


Figure B.8 Digital I/O Interface Section.

1 = Digital Receiver	9 = FIFO1A	24 = 74F244
2 = Digital Transmitter	12 = FIFO2A	25 = 74F125
3 = 74HC00	15 = FIFO1B	27 & 26 = PiSo
4 = ADC	18 = FIFO2B	28 & 29 = SiPo
5 = 5532 Op-Amp	10, 13, 16 & 19 = 74F245	30 = 74F32
6 = DAC	11, 14, 17 & 20 = 74F245	31 = 74F08
7 = FPGA (XC4003)	21 = 74F04 (Osc)	32 = 74F244
8 = PROM	22 & 23 = 74F245	33 = 74F244

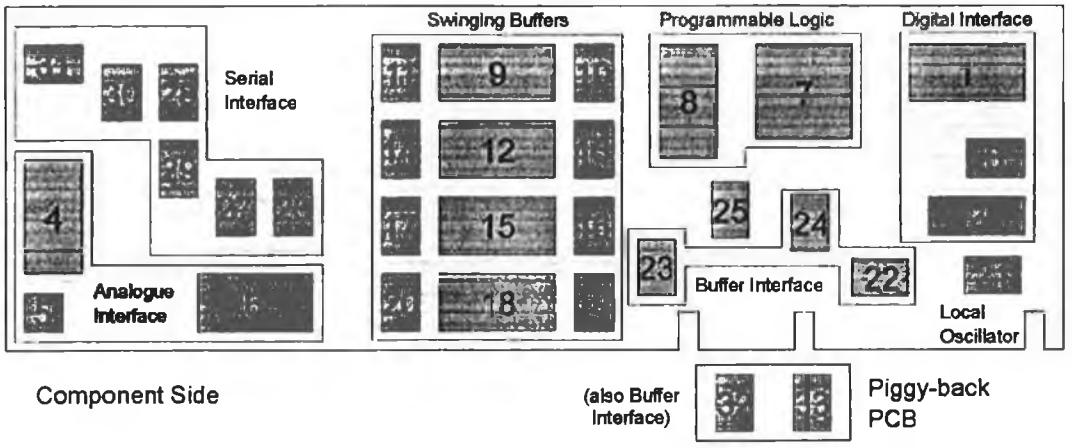


Figure B.9 *Physical Layout of the Sound Card.*

Address Decoding	
ADTP Number	Label
1	IO16BAR
2	LOWENBAR
3	HIGHENBAR
4	ANALOGUE IN
5	ANALOGUE OUT
6	DIGITAL IN
7	DIGITAL OUT
8	MRESETBAR
9	BUFFER STATUS REG.

Table B.1 *Test Points for the Address Decoding Section.*

DMA / Interrupt Generation	
DIGTP Number	Label
1	FF1BAR
2	FF2BAR
3	EF1BAR
4	EF2BAR
5	DACK7BAR
6	IORBAR
7	IOWBAR
8	DRQ7
9	DMAIRQ
10	SWITCHIRQ
11	TC
12	WEN1BAR

Table B.2 *Test Points for DMA / Interrupt Generation Section.*

Swinging Buffer and Control Logic	
SBTP Number	Label
1	DIR
2	SIBE1BAR
3	SIBE2BAR
4	PCBE1BAR
5	PCBE2BAR
6	R1BAR
7	R2BAR
8	W1BAR
9	W2BAR

***Table B.3** Test Points for the Swinging Buffer and Control Logic Sections.*

Serial Interface	
SITP Number	Label
1	PISOLOAD
2	PISOSCK
3	SIPOLOAD
4	SIPOSCK
5	DIGITAL DATA IN
6	ANALOGUE DATA IN
7	DIGITAL DATA OUT
8	ANALOGUE DATA OUT

***Table B.4** Test Points for the Serial Interface Section.*

Clock Generation	
CGTP Number	Label
1	DRXMCK
2	DRXLR
3	DIGITAL IN
4	ICKLD
5	SCK
6	L R
7	ACKO
8	XTI
9	ANALOGUE OUT
10	SCKAO
11	MCK

***Table B.5** Test Points for the Clock Generation Section.*

Analogue I/O Interface	
AITP Number	Label
1	L R
2	SCK
3	ICKLD
4	ANALOGUE DATA IN
5	ANALOGUE DATA OUT
6	LEFT ANALOGUE I/P
7	RIGHT ANALOGUE I/P
8	LEFT ANALOGUE O/P
9	RIGHT ANALOGUE O/P
10	XTI
11	SCKAO
12	L R

***Table B.6** Test Points for the Analogue I/O Interface.*

Digital I/O Interface	
DITP Number	Label
1	MCK
2	SCK
3	L R
4	DIGITAL DATA IN
5	DRXMCK
6	DRXLR
7	DIGITAL DATA IN
8	DIGITAL I P
9	DIGITAL O P

***Table B.7** Test Points for the Digital I/O Interface.*

Appendix C

Direct Memory Access

The generic Direct Memory Access *(DMA)* controller chip used in the AT Personal Computer *(PC)* is the Intel 8237. This device is used to transfer data between memory and peripheral devices or from one memory location to another without microprocessor control. In PCs the current DMA controllers are integrated into a VLSI device which also contains the Priority Interrupt Controller *(PIC)* along with other peripheral devices. The AT computer possesses two DMA controllers, the first is cascaded through the second so that there is only one source of DMA requests to the microprocessor. Table C.1 describes the PC's DMA channels [8]. The first DMA controller only permits 8 bit transfers while the second only supports 16 bit transfers. The second DMA controller must therefore be used by the MTP's device driver to allow 16 bit I/O transfers between the PC's memory and the sound card. Each DMA controller supports four DMA channels which can be configured separately by writing to several internal registers located in the PC's low I/O port address map.

Channel	0	1	2	3	4	5	6	7
Function	Spare	SDLC	Disc	Spare	Cascade	Spare	Spare	Spare
Bit Transfer	8	8	8	8		16	16	16

***Table C.1** DMA Channels in the AT PC.*

DMA Transfer Cycle

The DMA controller has a request line (*DRQ#*) for each of its channels, which are active high. If a DMA request is received on one of these lines then the DMA controller will signal the microprocessor by activating the Hold Request *(HOLD)* line. The microprocessor when its has finished its present operation, will release control of the bus to the DMA controller, notifying it with its Hold Acknowledgement *(HLDA)* signal. The DMA controller will now perform the DMA operation that the requesting DMA channel

was initialised for. The DMA acknowledgement *(DACK-bar)* signal notifies the peripheral device of a DMA I/O cycle on the particular DMA channel. The DACK#bar signal is active low. At the end of a DMA session, the Terminal Count *(TC)* signal is issued on the last transfer cycle. This signal is common to all the DMA controller's channels.

DMA Registers and Initialisation

The DMA controller is initialised by writing to several internal 8 and 16 bit registers that control the size of transfer, type of transfer and the starting memory address for the transfers. Table C.2 describes the registers used in initialising DMA channel seven, the fourth channel of the second DMA controller. The registers used for transfers are the Page Address register, Base and Current Address registers, Base and Current Count registers, Mode register and the Mask register.

Register	I/O Port Address (hex)
Channel 7	
Page Address Register	8A
Base and Current Address Registers	CC
Base and Current Count Registers	CE
Second DMA Controller	
Mode Register	D6
Mask Register	D4
Clear Byte Pointer	D8

***Table C.2** DMA Controller Registers accessed by the Device Driver when programming the DMA Controller.*

The two Base registers are write only registers which reside at the same I/O port addresses as their corresponding read only Current registers as shown in Table C.2. The Current registers are initialised to the Base registers whenever the Base registers are written to. These Base registers are only written to when the controller is being initialised. The Current registers indicate the present address location and the remaining number of transfers left in the DMA session.

The four Address and Count registers are 16 bit internal registers that are accessed through 8 bit I/O address ports. Writing to these 16 bit registers is achieved by an internal flip-flop which toggles between the high and low bytes upon every I/O port access to these registers. This flip-flop can be reset by writing to its I/O port, Clear Byte Pointer. Upon reset the 8 bit I/O port address accesses the lower byte of the internal 16 bit register.

The Base and Current Address registers are different for each DMA controller. For the first controller, which only supports 8 bit transfers, they hold the address range A0 to A15 while for the second one, which only supports 16 bit transfers, they hold A1 to A16. Figure C.1 illustrates the difference between the Address registers for the 8 and 16 bit DMA controllers. The Page registers holds the same address bits A16 to A23 for both controllers. The Current and Page Address registers combine to yield the total address, which can be in the range 0 to 16 Mbytes. The Page register is a static register which limits the size of DMA sessions to the size of the Base Address and Count registers. These 16 bit registers therefore, impose a maximum transfer size of 64 kbytes (2^{16}) for the 8 bit controller and 128 kbytes for the 16 bit controller.

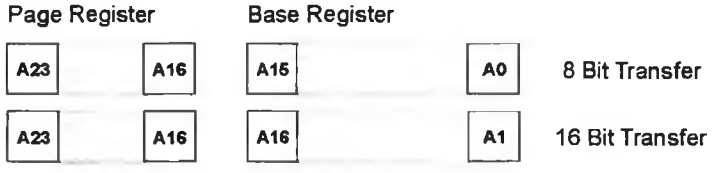


***Figure C.1** Bit Configurations for the Page and Address Registers.*

The bit configurations for the Mode register are defined in Figure C.2. Bits 0 and 1 define which channel is to be initialised, while bits 2 and 3 define the type of data transfer, I/O read or write or memory to memory. Bit 4 is the auto-initialisation mode, which defines whether upon completion of the specified number of data transfers, another is automatically started. Bit 5 determines if the memory address in the Current Address register is incremented or decremented after each transfer. Bits 6 and 7 define the basic type of DMA transfer in relation to bus management. In *Single Mode*, the DMA controller will release control of the bus after every DMA transfer, irrespective of the current state of the DMA request line, thus ensuring the processor is never locked out. For the *Block Mode*, the DMA controller will transfer the complete block of data without yielding control of the bus. In *Demand Mode*, the controller will transfer data

while the channel's request line is active and a terminal count has not been reached. The terminal count is an active high signal issued by the controller whenever a channel's count register has reached zero. The final mode is the *Cascade Mode* which programs all the DMA channels for *Single Mode* transfer.

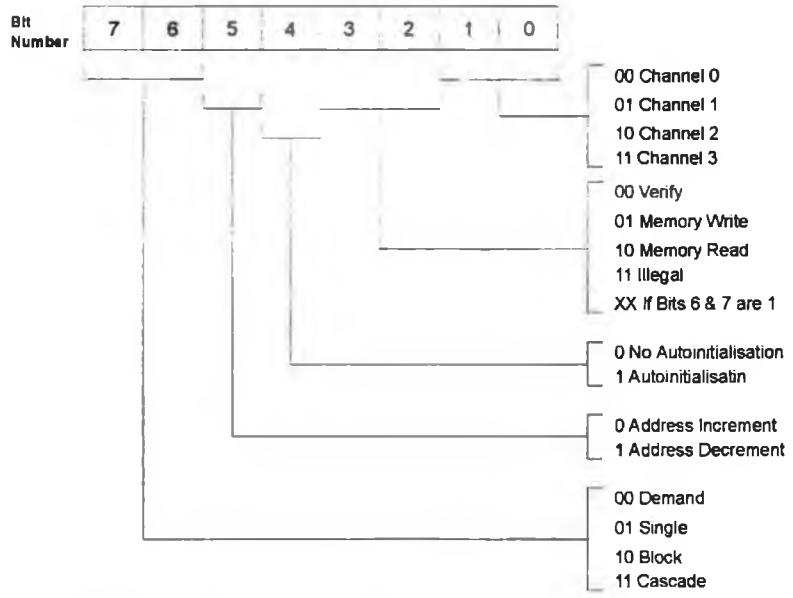


***Figure C.2** Bit Definitions of the Mode Register.*

There are three more registers, the Command Register, the Status Register and the Write All Mask register. The command register is initialised by the PC's ROM BIOS at boot-up and establishes the DMA Request *(DRQ#)* and DMA Acknowledge *(DACK#bar)* active levels and the priority scheme of the DMA channels. This Command register is a write only register which shares the same I/O port address with the read only Status register. The Status register defines the state of each DMA channel and is described in Figure C.3. The Write All Mask register is used to mask or unmask each channel collectively rather than individually as in the Mask register.

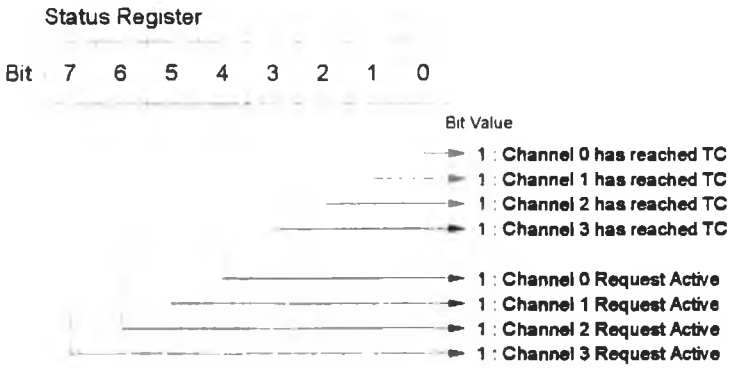


***Figure C.3** Bit Definitions of the Status Register.*

Programming the DMA Controller

The steps involved in programming DMA channel seven using assembler language are illustrated in Listing C.1. These step are applicable to all channels. First, the channel must be masked to stop any DMA activity on the channel and to ensure that DMA cycles are only started at the required time, which is after the channel has been fully initialisation. The channel is masked by writing 07h to the DMA controller's Mask

```
mov   dx, MaskReg       ; Masking channel 7.
mov   al, 07h
out   dx, al

mov   dx, PageReg       ; Set up the Page register.
mov   al, PageAddress
out   dx, al

mov   dx, ClearByte     ; Clear byte pointer.
out   dx, al

mov   dx, AddressReg    ; Set up the Base Address
mov   ax, Address       ; register.
out   dx, al            ; Low address byte.
mov   al, ah
out   dx, al            ; High address byte.

mov   dx, CountReg      ; Set up Base Count register.
mov   ax, Count-1
out   dx, al            ; Low count byte.
mov   al, ah
out   dx, al            ; High count byte.

mov   dx, ModeReg       ; Set up Mode register.
mov   al, TransferType  ; Type of DMA transfer.
out   dx, al

mov   dx, MaskReg       ; Unmask channel 7.
mov   al, 03h
out   dx, al
```

***Listing C.1** Programming DMA Channel 7.*

register, I/O port D4h. The value seven corresponds to masking channel 4 of the DMA controller but this is the PC's second DMA controller whose channels are designated 4, 5, 6 and 7.

Once the channel has been masked, the DMA controller can be safely initialised. The memory buffer's Page and Base Address register values are now written to their respective registers at I/O port addresses 8Ah and CCh. These values are calculated from the memory buffer's starting address as illustrated previously in Figure C.1. Before writing to the Base Address register, the DMA's internal flip-flop must be cleared to guarantee access to the lower byte of the 16 bit Base registers first. This is accomplished by writing to the Clear Byte Pointer at I/O port D8h.

The value written to the Base Count register is one less than the required number of DMA cycles because the DMA's cycles are stopped when the count register reaches FFFFh and not 0000h. The type of DMA session determines the value written to the Mode register. For the memory read and I/O port write transfer, the value 47h is written while for memory write and I/O read the value 4Bh is written. The value 47h correspond to a memory read from a I/O port on channel seven in single mode DMA transfers with address increment and no auto-initialisation. The value 4Bh differs in that the DMA transfers are memory read and I/O port write cycles. The DMA controller has now been initialised and is now unmasked by writing 03h to its Mask register, I/O port D4h. DMA transfers can now take place on Channel 7.

Interrupts

Interrupts inform the microprocessor of events which it must respond to, for example when a key has been pressed on the keyboard. These events can be software or hardware generated and each interrupt possesses a routine or program, which the microprocessor calls to service the interrupt. This program is called the Interrupt Service Routine *(ISR)*. In the first 1024 bytes of the PC's memory resides the Interrupt Vector Table *(IVT)* which has an entry for each one of the PC's 256 interrupts [17]. Each entry is four bytes long and contains the segment and offset for each interrupt's ISR. When the Intel microprocessors are operating in their protected modes, the IVT is replaced by the Interrupt Descriptor Table *(IDT)* which is similar to the segment descriptor table used for code and data segments. The DOS interrupt 21h functions 35h and 25h can be used to set and read the interrupts vectors in both modes.

The PC possesses 15 hardware interrupts which are available on the PC's system bus. These interrupts are implemented using two PICs, consisting of a master and slave. The slave PIC is cascaded through the third interrupt line of the master PIC resulting in only 15 interrupts being available rather than 16. The master PIC's interrupts are described in Table C.3.

Name	Timer	Keyboard	Cascade	Comm 2	Comm 1	LPT 1	FDisc	LPT 2
IRQ No.	0	1	2	3	4	5	6	7
Vector No.	8	9	A	B	C	D	E	F

***Table C.3** Lower Hardware Interrupt Lines.*

Priority Interrupt Controller

The Priority Interrupt Controller *(PIC)* provides the interface between the microprocessor and the hardware interrupt lines *(IRQs)*. It provides priority and masking of multiple interrupt lines.

The PICs are initialised by the PC's ROM BIOS at boot-up. The master PIC is initialised by writing Initialisation Control Words *(ICW)* to I/O port addresses 20h and 21h. There is a certain bit combination written to I/O port 20h to start the initialisation process and a set sequence of I/O port writes to 21h and 20h are required to configure the PIC. The default configuration for the PC is fixed priority and positive edge triggered interrupts, therefore IRQ lines cannot be simultaneously shared between several devices. The interrupt lines must be held high when inactive and pulsed low to latch them into the PIC.

Operational Control Words *(OCW)* also reside at the same I/O port locations and are used to individually mask and unmask interrupts and to issue End of Interrupt Commands *(EOI)*. When a hardware generated interrupt's ISR is finished it must issue an EOI command to the first OCW. This command resets the PIC IRQ's input, allowing further interrupts to be latched by the PIC. There is also a general EOI command which can be issued by writing 20h to the first OCW (I/O port address 20h). The second OCW is known as the Mask register. This is situated at I/O port address 21h. Each IRQ line can be masked by setting its corresponding Mask register bit to one or unmasked by

setting it to zero, bit zero controls IRQ0. The sound card uses two interrupts and these are the Comm2 and LPT2 interrupts, which are known as the Switching and DMA interrupts respectively.

Programming the PIC

The procedure for programming the hardware interrupt IRQ3 (Comm2) using assembler language is shown in Listing C.2. The hardware interrupt IRQ3 corresponds to the interrupt vector Bh. First, the interrupt being changed is masked by setting its bit in the PIC's Mask register to one. Next the DOS interrupt 21h function 35h is used to retrieve the present ISR from the interrupt vector and store it the *gdwOldSwitchISR* variable so it can be returned it to its original state when the program is finished. This interrupt vector is then set to the new ISR using the DOS interrupt 21h function 25h. The IRQ line is now unmasked by setting its mask bit in the Mask register to zero.

```
mov   dx, MaskReg
in    al, dx
or    al, 08h           ; Mask IRQ3.
out   dx, al

mov   al, 0Bh           ; Retrieve ISR for vector Bh.
mov   ah, 35h
int   21h               ; Old ISR in ES:BX.
mov   ax, bx            ; Save old ISR.
mov   dx, es
mov   [gdwOldSwitchISR].sel, dx
mov   [gdwOldSwitchISR].off, ax

push  ds                ; Save DS.
mov   al, 0Bh
mov   ah, 25h           ; New ISR in DS:BX.
lds   bx, lpNewISR
int   21h               ; Set the new vector for Bh.
pop   ds                ; Restore DS.

mov   dx, MaskReg
in    al, dx
and   al, 0F7h          ; Unmask IRQ3.
out   dx, al
```

***Listing C.2** Programming Interrupt Vector Bh, Hardware Interrupt IRQ3.*

Appendix D

Using the MTP Application

This appendix explains how to record and play WAVE files with the MTP application. The application's menu bar and the options it presents to the user are also described.

Playing and Recording WAVE Files

The WAVE files can be recorded and played back in the analogue or digital modes using dialog boxes called by the *Play* and *Record* menu bar items. These menu bar items provide an analogue or a digital option which will display the appropriate dialog box when selected.

The Play Dialog Box

The dialog boxes for the analogue and digital playback options are identical except for their titles or *Caption Bars*. The analogue playback dialog box is displayed in Figure D.1. In the case, the user has selected the WAVE file *FLUTE.WAV* to be played for one second, beginning one second from the start of the file. The length of the file is displayed in the *File Length* section. The start position and playback duration can be selected though edit windows as illustrated. The scroll bar at the bottom of the dialog box can also set the start position and when playback is active, it moves to show the present playback position.

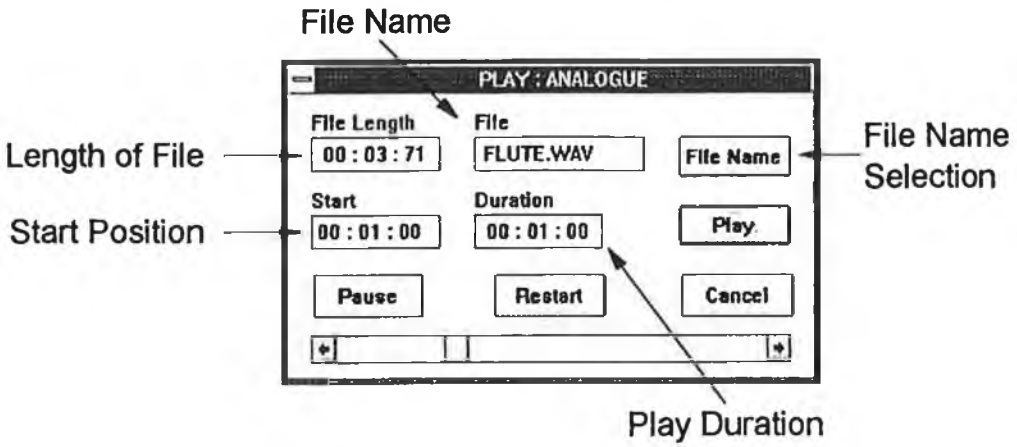


Figure D.1 The Analogue Playback Dialog Box.

The *File Name* push button allows the user to select the filename for playback using the standard Windows *Open* dialog box. The push buttons *Play*, *Pause* and *Restart* control playback while the *Cancel* push button closes the dialog box. When playback is taking place the *Cancel* push button's text is changed to *"Stop"*. This push button will then only stop playback and set its text back to *"Cancel"*. The push button's text is automatically returned to *"Cancel"* when playback has finished. The *Play* push button in the analogue or digital play dialog box starts playback of the chosen file from the requested start position and for the requested duration. If the duration is zero then the file is played from the start position to the end of the file.

The Record Dialog Box

The record dialog box is identical for the analogue and digital options except for the *Caption Bars*. Figure D.2 shows the record digital dialog box. This dialog box allows the file name and recording duration to be chosen. The maximum recording duration is displayed above the selected recording duration. The durations are in the form, minutes, seconds and hundreds of seconds. There are push buttons for starting and stopping recording along with a push button *File Name*, for changing the recording file name. The file name is changed using the standard Windows *Save As* dialog box. The *Cancel* push button will close the dialog box if recording is not taking place. When recording is taking place, the *Cancel* push button's text is changed to *"Stop"* and is returned when recording is stopped or has finished. Pressing the *Stop (Cancel)* push button during recording only stops recording. The scroll bar can also be used to choose the recording

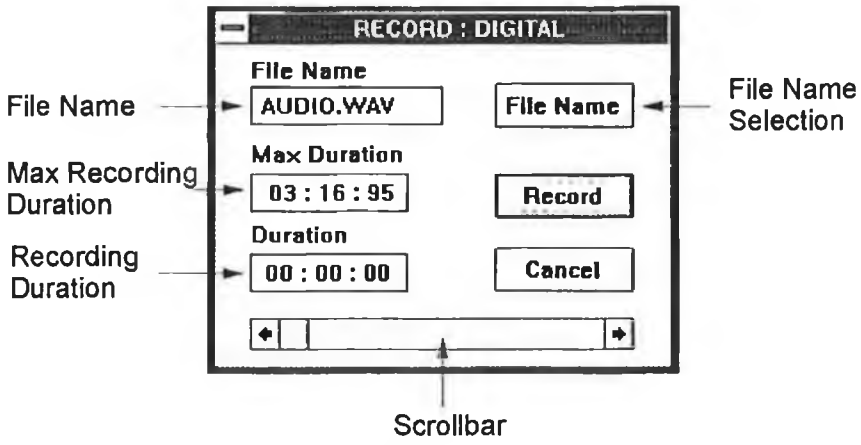


Figure D.2. *The Digital Recording Dialog Box.*

duration in relation to the maximum duration possible. When recording is taking place, the scroll bar moves to represent the present recording position in relation to the chosen recording length.

Default File Selection

The *FILE* menu bar item allows the user to select the default files for recording and playback. The *FILE* pull-down menu consists of the common Windows' *Open* and *Save As* options. These pull-down items will call the standard Windows dialog boxes for opening and saving files. The file name selected in the *Open* dialog box will be the default file name for the playback dialog boxes. Similarly the *Save As* dialog box sets the default recording file name. File names can also be selected in the dialog boxes but these do not update the default file names.

Waveform Device Information

The *Info* menu item displays two pull-down items, *Wave Out* and *Wave In*. These elements select dialog boxes which displays the waveform output and waveform input capabilities of the waveform device.

Help and Application Information

The *Help* menu bar item contains the *Help* and *Application* pull-down elements. The *Application* element displays a dialog box which provides general information about the application. The *Help* element calls the Windows help application which in turn loads the application's help file, *MTPHELP.HLP* and displays its contents page.

Exiting from the Application

The *Exit* menu bar item displays two pull-down options, *Yes* or *No*. The *Yes* pull-down item when selected displays a message box prompting the user to confirm exit from the application. The *No* pull-down element will cancel the *Exit* menu selection.

Appendix E

Windows Operating Modes

There are three operating modes for Windows, Real, Standard and Enhanced. A brief description of the major features of each follows.

Real Mode

This was the first operating mode for Windows when it was introduced in 1985 and was designed to run on the Intel 8086 microprocessor. Real mode used the Intel 8086 microprocessor's *segment* and *offset* technique to generate the 20 bit physical memory address. The total data memory for the system and the active applications was restricted to the lower 640 kbytes of the memory space which proved a major handicap as the size of Windows applications increased. This mode was discarded from Windows 3.1 onwards [12].

Standard Mode

Standard mode was designed for the Intel 80286 microprocessor and required at least one megabyte of extended memory to run. It was introduced in Windows 2.0 [12]. This mode used the microprocessor's *protected mode* to increase Windows' performance. The size of system memory was increased by the microprocessor's *segment selector* and *offset* method. This allowed up to 16 Mbytes of physical memory [12]. The *selector* points to a *selector table* in memory which contains the base address. The offset is added to this base address to generate the physical memory.

The *selector table* also contains information regarding the selector type and its priority levels. They provide protection against unauthorised access. Each program operating under protected mode has a priority level ranging from zero to three. Level zero is reserved for the Windows operating system and is known as the supervisor mode while normal Windows applications run at level three and is therefore known as the user mode.

The priority level of the accessing application must be greater or equal to the *selector's* priority level otherwise access will be refused. This protects the system from

accidental corruption by Windows applications. This protection does not prevent Window applications from overwriting each other as they run at the same priority level. Higher priority segments can be accessed by lower priority applications using *call gates* but these are strictly controlled.

The Interrupt Vector Table *(IVT)* is replaced by the Interrupt Descriptor Table (*IDT*). This IDT is similar to the normal segment selector table. The interrupt *selector* is similar to the normal code *selector*. The I/O port addresses can also be prioritised, and an I/O privilege map in Windows can restrict the I/O port address space available to applications [12].

DOS applications can be run in Windows Standard mode, but they require the microprocessor to switch from *protected mode* to *real mode* [12]. They cannot run when minimised and all Windows applications are suspended while the DOS application is active.

80386 Enhanced Mode.

The 80386 Enhanced mode was introduced in Windows 3.0 and can only be run on an Intel 80386 microprocessor or higher. Enhanced mode retains all the benefits of Standard mode but includes increased memory and Virtual 8086 mode. The maximum size of segments is increased to 4 Gbytes and each segment can be divided into 4 kbytes pages. These small pages can be swapped between the hard disk and memory when physical memory is over committed, thus increasing system performance. This paging is implemented in hardware and introduces virtual memory to Windows [12].

The Virtual 8086 mode allows DOS applications to run in the microprocessor's *protected mode*, eliminating the need to switch to the microprocessor's *real mode*. Multiple DOS applications can therefore run simultaneously along-side normal Windows applications. The DOS applications are now no longer suspended when minimised.

Appendix F

RIFF File Format

The format for Windows Multimedia files is the RIFF format. RIFF stands for *Resource Interchange File Format* and was developed jointly by Microsoft and IBM [9]. The format was designed to supported different types of Multimedia files such as the waveform audio *(WAVE)* files and audio video interleave *(AVI)* files, and to allow for the introduction of new formats as Windows Multimedia develops.

RIFF files are composed of blocks of data known as chunks. The primary or parent chunk is the *RIFF* chunk. This chunk contains all the other chunks of the file, known as subchunks. The types of subchunks contained in the *RIFF* chunk depend on the RIFF file type. A chunk is composed of three elements or fields. The first field is a four-character code specifying the chunk ID, a double word specifying the size of the data contained in the chunk's data field, and the chunk's data itself. *RIFF* and *LIST* chunks contain an extra field known as a form field which defines the format of the data in the file. The *LIST* chunk is the only other chunk which can contain subchunks. The maximum size of the data is limited to 4.29 Giga bytes which is more than adequate at present but may become a problem for future formats such as Digital High Definition Television *(HDTV)*.

RIFF WAVE File Format

The first chunk encountered in a WAVE file is a *RIFF* chunk which possesses a form field of type *WAVE*. The data size field of this chunk is the size of all its subchunks. The WAVE format can contain several different subchunks but only the *FORMAT* and *DATA* chunks are used here. Figure F.1. illustrates the WAVE file format used by the application.

The *FORMAT* subchunk describes the format of the data contained in the *DATA* chunk and its identification *(ID)* field is *fmt*. The data of this chunk is a *PCMWAVEFORMAT* structure. This structure contains the sampling frequency, bits per sample, bytes per second, stereo or mono, minimum number of bytes per complete samples (left and right if stereo) and the data type which is always PCM. The next subchunk is the *DATA* chunk which contains the actual waveform data. This chunk has

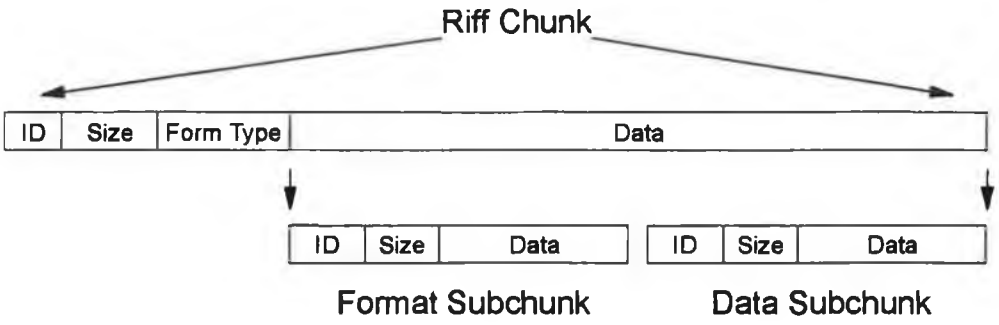


Figure F.1. *WAVE File Structure.*

an ID field of *data*. The maximum size of the data the file can hold is approximately 6.7 hours of uncompressed 44.1 kHz 16 bit stereo data. The WAVE format can also contain additional subchunks, a *fact* subchunk, associated data *LIST adtl* and a cue-points subchunk for identifying different locations within the file for synchronisation with other multimedia devices.

PCM Audio Format

The most common RIFF WAVE format is the Pulse Code Modulation *(PCM)* format. For the PCM 16 bit stereo format the data is stored left channel first and least significant byte first [11]. The format chunk contains all the data required to set up the waveform device for this format. Because the digital format only involves a different transmission medium and the coding of the signal is also PCM, the input or output medium is irrelevant when dealing with WAVE files.

Multimedia File I/O Functions and Structures

The application uses the Multimedia File I/O functions for accessing the WAVE files on the hard disk rather than the normal Windows or C functions for several reasons. First, these functions are part of Windows so they do not increase the size of the application. Second, they are specifically written for RIFF files and provide easier and faster navigation through these files [11].

There are several new data structures associated with these functions. *HMMIO* is a file handle similar to the normal file handles but is not compatible with C or DOS file functions. The *MMCKINFO* structure contains information on a chunk. The most important of these being the type of chunk, size of data field and a pointer to the chunks

data field from the beginning of the chunk. The *MMIOINFO* structure contains information on the current state of a file. For a complete description of the different elements of each structure refer to the Microsoft Multimedia Programmer's Reference, listed in the Bibliography.

The Multimedia File I/O Functions can be identified by their *mmio* prefixes and they contain the standard set of file functions, opening, closing, reading and writing. They also contain unique functions such as *mmioCreateChunk*, *mmioDescend* and *mmioAscend*. The *mmioAscend* and *mmioDescend* functions are used to navigate through the various chunks of the RIFF file without calculating the various pointers for the data fields manually. The *mmioDescend* function will move the file pointer to the data field of the present chunk while *mmioAscend* moves the pointer to the beginning of the next chunk. The *mmioDescend* function will fill the *MMCKINFO* structure with the chunk details which the *mmioAscend* function utilises when moving to the next chunk.

Appendix G

Hungarian Notation

The *Hungarian notation* used in the Windows application's C programs and the device driver's C and assembly language programs is shown in Table G.1. The variable types which may not be familiar are the *DWORD*, *BYTE* and *BOOL* which are simply modifications of previous types *LONG* and *CHAR*, which do not require the *UNSIGNED* keyword.

Prefix	Data Type
c	char (8 bits)
by	BYTE (unsigned char)
n	short or int (8 bit integer)
i	int (16 bit signed integer)
x, y	short (used as x, y coordinates)
b	BOOL (16 bit integer)
ui or w	UINT or WORD (unsigned 16 bit integer)
l	LONG (32 bit signed integer)
dw	DWORD (unsigned 32 bit integer)
fn	function
s	string
lpsz	long pointer to a string terminated by zero

***Table G.1** Examples of Hungarian Notation.*

Appendix H

Help File for the MTP Application

The help implemented in the Windows MTP application provides explanations for the menu bar items and the process of recording or playing files. This is not a comprehensive help document but does demonstrate the common help principles. The Windows program *WINHELP.EXE*, situated in the Windows directory, provides the interface for accessing Windows *HELP* format files [11, 15].

The application's help file consists of a *CONTENTS PAGE* with *HYPERTEXT* jumps to other screens and *GLOSSARY* items which display pop-up boxes when selected. The *CONTENTS PAGE* of the *HELP* file is shown in Figure H.1. The text shaded (Green in colour) and underlined by a solid line in Figure H.1, are *HYPERTEXT* jumps which allow the user to call up a new text screen related to the highlighted text. They provide links between the different sections of the help file. The *GLOSSARY* hot spot text areas are coloured and underlined with a dashed line. The help file also contains bitmaps with *GLOSSARY* pop-up boxes.

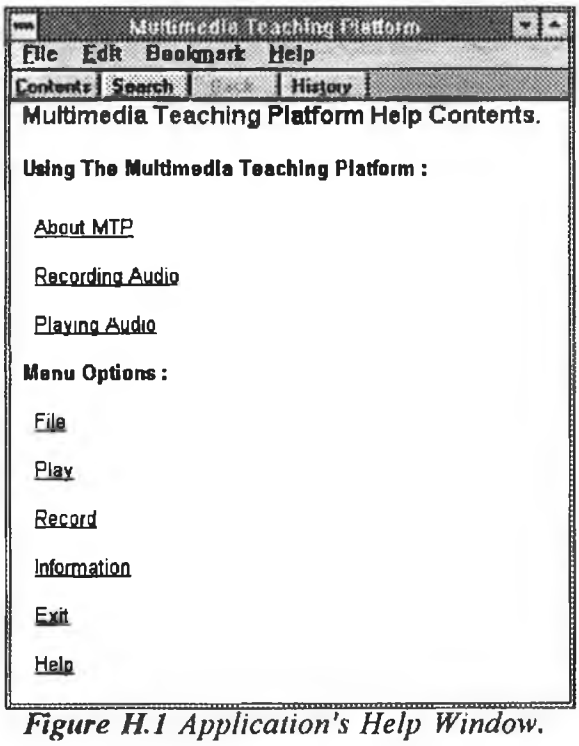


Figure H.1 Application's Help Window.

Generating the Help File

The help file is generated by the HC31.EXE compiler from the help project file MTPHELP.HPJ and the separately compiled help source file MTPHELP.RTF. The help file MTPHELP.HLP, is generated from the following two commands issued at the system prompt;

HC31 MTPHELP.RTF

HC31 MTPHELP.HPJ

The help project file is shown in Listing H.1 and has the same format as a *.INI* Windows file. The *[OPTIONS]* section contains optional information regarding the help file. The *TITLE* keyword sets the Windows *WINHELP* program's title bar while *COMPRESS* determines whether the file is compressed. The *[FILES]* section specifies the source files which make up the help file. The *[MAP]* section is optional and defines the help file's context sensitive elements. One of these elements can be accessed immediately when the help file is called, if the *WinHelp* function's context flag is set and its context value equals one of these numbers.

```
[OPTIONS]
TITLE=Multimedia Teaching Platform
COPYRIGHT=Kevin Street, DIT.
COMPRESS=TRUE

[FILES]
mtphelp.rtf

[MAP]
#define H_About_MTP      10
#define H_File           20
#define H_Recording      40
#define H_Playback       30
#define H_Information    50
#define H_Help           60
#define H_Exit           70
```

***Listing H.1** Project File for the Help File.*

The Help File's Source File

This source file is in the Rich Text Format *(.RTF)* and is shown in Listing H.2. Displayed in this listing are the *CONTENTS PAGE*, the *HYPERTEXT* jump *Play* and the *GLOSSARY* item *Analogue Audio*. The double underlined text shown in Listing H.2, states *HYPERTEXT* jump points while single underline text states *GLOSSARY* items. The text in brackets immediately after the underlined text is the reference name of the *GLOSSARY* item or the *HYPERTEXT* jump section. This text is hidden in the RTF format and this is illustrated by brackets enclosing the text.

The *GLOSSARY* and *HYPERTEXT* sections are declared using # *footnote*. The *HYPERTEXT* element can contain a number of properties which are also declared with footnotes. For example the *$ footnote* determines the string which appears in the *Go To* list box in the *Search* dialog box of the *WINHELP* program when this element is selected in the *Search* dialog box. Table H.1 shows the footnotes used for the

```
{bml icon.bmp}
Multimedia Teaching Platform Help Contents.

Using The Multimedia Teaching Platform :
  About MTP{H_About_MTP}
  Recording Audio{H_Recording}
  Playing Audio{H_Playback}

Menu Options :
  File{H_File}
  Play{H_Playback}
  Record{H_Recording}
  Information{H_Information}
  Exit{H_Exit}
  Help{H_Help}
------------------------------ Page Break ------------------------------
 # $ K + Play
The process of playing analogue or digital files is identical.
Playback is accomplished with one of the dialog boxes from the Play
menu bar item. The dialog box shown below allows the user to select
the WAVE file, the start position and the duration of playback.

{bmc playanlg.shg}
------------------------------ Page Break ------------------------------
 # Analogue Audio
The analogue audio is the 44.1 kHz 16 bit stereo PCM WAVE format.
```

***Listing H.2** Sections of the Help File's Source File.*

HYPERTEXT jumps in the help file and their functions. The + *footnote* assigns a context number to the *HYPERTEXT* element which is used in context sensitive help for jumping immediately to the subject, bypassing the *CONTENTS PAGE*.

The bitmap ICON.BMP is inserted into the start of the *CONTENTS PAGE* with the hidden text *bml icon.bmp*. The *bml* statement inserts the bitmap on the left-hand side of the *CONTENTS PAGE*. The *bmr* statement places the bitmap on the right while the *bmc* statement places the bitmap in the current line of text. This bitmap can itself be defined as a *HYPERTEXT* jump or a *GLOSSARY* item by underlining the bitmap statement and declaring the jump or pop-up reference after it. The *Play HYPERTEXT* text also contains a bitmap image, *bmc playanlg.shg*. This bitmap is in the *segmented-graphics bitmap* format *(.SHG)*. This format contains hot spot areas which can in turn

Footnote	Function
#	Reference Name.
S	String displayed in the ***Search*** dialog box.
K	String displayed in the ***Go To*** list box of the Search dialog box.
+	Definition of context number.

Table H.1 *FOOTNOTES used for HYPERTEXT jumps.*

call *GLOSSARY* and *HYPERTEXT* items. The bitmap is created in the *SHED.EXE* application from a normal Windows bitmap file. The *SHED.EXE* application allows the user to define graphical hot spot areas on the bitmap and link these to *GLOSSARY* or *HYPERTEXT* elements in the help file's source file. This application is one of the visual programming aids which are included with most Windows programming packages. The *PLAY ANALOGUE* and *RECORD DIGITAL* dialog boxes displayed in the *Recording* and *Play HYPERTEXT* screens will display *GLOSSARY* pop-up boxes when one of their controls is selected. These pop-up boxes explain the function of each control. The *PLAY HYPERTEXT* screen is shown in Figure H.2.

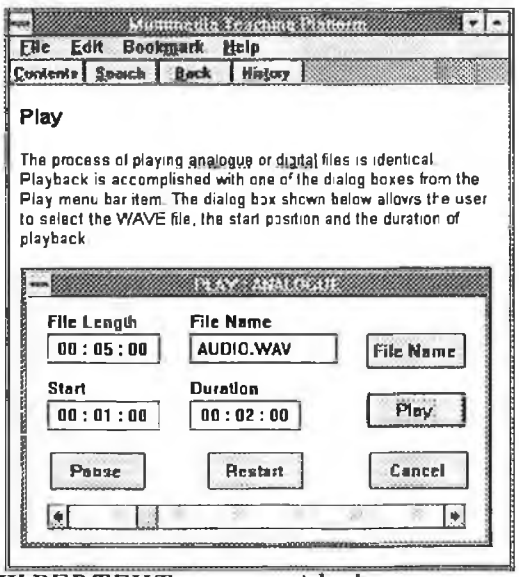


Figure H.2 *Play HYPERTEXT screen with the segmented-graphics bitmap, PLAYHELP.SHG.*

Appendix I

Device Driver Installation

The device driver can be installed in Windows manually or automatically. Manual installation involves placing the line *"WAVE=MMTEACH.DRV"* in the *[drivers]* section of the *SYSTEM.INI* file, and copying the device driver *MTEACH.DRV*, to the *SYSTEM* subdirectory in the main Windows directory. Automatic device driver installation is accomplished by the *Drivers* applet in *Control Panel*. *Control Panel* is in the *Main* group of *Program Manager*. This installation requires an installation disk containing the device driver *MMTEACH.DRV*, and the standard setup file *OEMSETUP.INF*, for third party device drivers [11]. This file is listed in Listing I.1. and contains the number of disks required for installation along with the type of device driver *"WAVE"*, and the name of the device driver *MMTEACH.DRV*. The *DRIVERS* applet will perform the same process as described for manual installation.

```
[disks]
    1 =. ,"Multimedia Teaching Platform",disk1

[Installable.Drivers]
MMTEACH = 1:mmteach.drv,  "Wave", "Multimedia Teaching Platform",
```

Listing I.1. *Device Driver's Setup file, OEMSETUP.INF.*

Appendix J

Installing the MTP application

This Appendix explains the process involved in installing a Windows application from its installation disk using the Microsoft Windows Setup Utility [11, 13]. The steps involved in building the installation disk for the MTP application are also described.

Windows Setup Utility

Windows applications are installed using the Windows Setup Utility. This utility can be broken into two sections. The first section is composed of SETUP.EXE and its list file SETUP.LST. The SETUP.EXE program is responsible for transferring the files, listed in its list file, to a temporary directory on the hard disk. These files are required by the next part of the installation process, _MTEST.EXE the installation program. This program is a *BASIC* interpreter which reads the installation setup script, MMAPP.MST. This script controls the installation process, it is responsible for displaying dialog boxes and processing the options which the user selects.

The *Microsoft Windows Software Development Kit Ver.3 (SDK)* [11] contains a setup development kit in its MSSETUP directory situated in its main directory. This kit contains applications and source code for building installation disks for Windows applications.

Installing the application

This Windows application is installed on to the PC's hard disk from its installation disk, labelled *"Installation Disk for the Multimedia Teaching Platform's Windows Application"*. The application is installed by running the executable file SETUP.EXE on this disk. This program requires Windows to be active and will start Windows if Windows is not already running.

The process of installing the Windows application from the user's perspective is explained. First, a dialog box is displayed which informs the user that the installation process is being initialised. This initialisation process involves transferring files required by the installation program to a temporary directory. This temporary directory is deleted when the installation process finishes. When this process is completed, the installation

program is started which displays a dialog box requesting the user to *CONTINUE* or *EXIT* the installation process as shown in Figure J.1. There is also a *HELP* option. This option displays a dialog box which provides information on the installation process. If the *EXIT* option is selected then the installation process is stopped and none of the application's files are transferred.

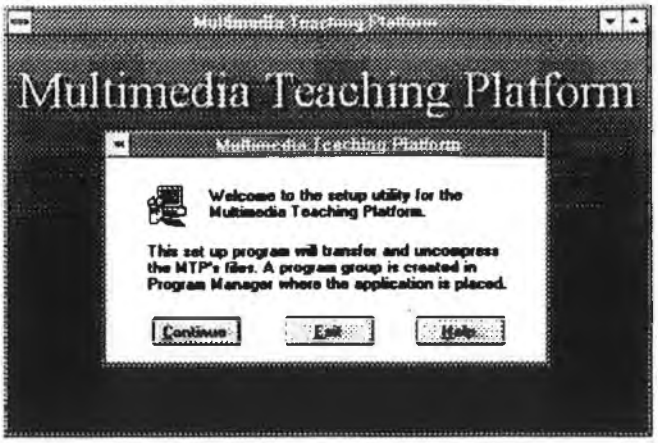


***Figure J.1** The Installation Program displaying the Welcome Dialog Box.*

If the *CONTINUE* option is taken, then the application's files can now be decompressed and transferred from the installation disk to the default directory. A dialog box is displayed as shown in Figure J.2, which provides information on the present application file being transferred. The dialog box also contains a bar indicator that reports the percentage of the installation process which has been completed.

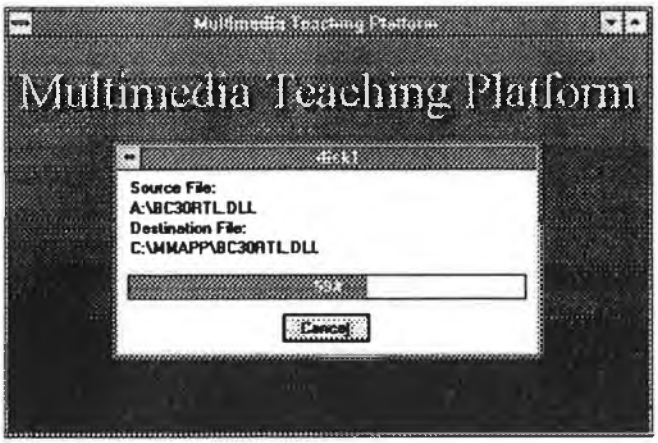


***Figure J.2** Installing the Application's Files.*

Building the Installation Disk

The process of building the installation disk for the application can be summarised as follows:-

Step 1 : Identify the files which the application needs to run and store these in a new working directory along with the files required by the installation process. The files needed are shown in Table J.1.

Step 2 : Create the setup script MMAPP.MST, in the working directory and modify the dialog box templates.

Step 3 : Create the layout file MMAPP.LYT with the Windows program DSKLAYT.EXE.

Step 4 : Create the disk image and the MMAPP.INF file using the DOS program DSKLATY2.EXE. The MMAPP.INF file determines the files needed by the application to run and on which disk they are stored on.

Step 5 : Modify the MMAPP.INF file to only include the application's files.

Step 6 : Compress the MMAPP.INF file.

Step 7 : Modify the SETUP.LST file.

Step 8 : Transfer the disk image files, the compressed image file and setup list file to the installation disk.

Setup Utility's Files	Application Files
Setup.exe	mmapp.exe
Setup.lst	mmapp.bmp
_mstest.exe	mmapp.ico
mscomstf.dll	speaker.ico
mscuistf.dll	comdisc.ico
msdestf.dll	compact.ico
msinsstf.dll	recorder.ico
msshlstf.dll	mtphelp.hlp
msuilstf.dll	bc30rtl.dll
setupapi.inc	bwcc.dll
mmapp.mst	

***Table J.1** Files required by the DSKLAYT program.*

The first step involves creating a working directory for developing the installation disk. This directory is filled with the application's files and the files required by the setup utility as described in Table J.1. The next step is to create the setup script MMAPP.MST. This is in the form of a BASIC language source file as shown in Listing J.1. The first section of the script involves customising the user interface. References to

the library MSCUISTF.DLL are declared. This library is a DLL which contains the dialog boxes and their processing functions along with the bitmap which is displayed in the installation program's main window as shown in Figure J.1

```
'$INCLUDE 'setupapi.inc'
GLOBAL DEST$
CONST WELCOME       = 100
CONST APPHELP       = 900
DECLARE SUB Install
DECLARE FUNCTION MakePath (szDir$, szFile$) AS STRING
INIT:
    CUIDLL$ = "mscuistf.dll"            ''Custom user interface dll
    HELPPROC$ = "FHelpDlgProc"          ''Help dialog procedure
    SetBitmap CUIDLL$, 2
    SetTitle "Multimedia Teaching Platform"
    szInf$ = GetSymbolValue("STF_SRCINFPATH")
    IF szInf$ = "" THEN
        szInf$ = GetSymbolValue("STF_CWDDIR") + "MMAPP.INF"
    END IF
    ReadInfFile szInf$
    DEST$ = "C:\MMAPP"
WELCOME:
    sz$ = UIStartDlg(CUIDLL$, WELCOME, "FInfoDlgProc", APPHELP,
                                                        HELPPROC$)
    IF sz$ = "CONTINUE" THEN
        UIPop 1
    ELSE
        GOTO QUIT
    END IF
    Install
QUIT:
    END

SUB Install STATIC
    SrcDir$ = GetSymbolValue("STF_SRCDIR")
    CreateDir DEST$, cmoNone
    AddSectionFilesToCopyList "Files", SrcDir$, DEST$
    CopyFilesInCopyList
    CreateProgmanGroup "Multimedia Teaching Platform", "", cmoNone
    ShowProgmanGroup  "Multimedia Teaching Platform", 1, cmoNone
    CreateProgmanItem  "Multimedia Teaching Platform",  "MMTEACH",
MakePath(DEST$,"mmapp.exe"), "", cmoOverwrite
END SUB

FUNCTION MakePath (szDir$, szFile$) STATIC AS STRING
    IF szDir$ = "" THEN
        MakePath = szFile$
    ELSEIF szFile$ = "" THEN
        MakePath = szDir$
    ELSEIF MID$(szDir$, LEN(szDir$), 1) = "\" THEN
        MakePath = szDir$ + szFile$
    ELSE
        MakePath = szDir$ + "\" + szFile$
    END IF
END FUNCTION
```

***Listing J.1** The Setup Script file, MMAPP.MST.*

The *Welcome* dialog box as shown in Figure J.1 is displayed with the *UIStartDlg* function. For more information on the installation program's functions refer to the setup documentation in the SDK. This dialog box's processing function will return control to the installation program when the user has selected the *Quit* or *Continue* option. If the *Continue* option has been selected then the application is installed with the *Install* procedure *(SUB Install Static)*. The dialog boxes *Welcome* and *Help*, are modified from the sample source code for the MCUISTF.DLL library. The "Multimedia Teaching Platform" bitmap is created and displayed using the statement *SetBitMap CUIDLLS, 2*, where 2 is DLL reference number for this bitmap. The sample code is in the directory MSSETUP\BLDCUI, situated in the main SDK directory.

The third step is to create the layout file, MMAPP.LYT using the DSKLAYT.EXE Windows program. This program sets the attributes for all the installation files and is in the directory MSSETUP\DISKLAY, situated in the main SDK directory. The attributes which can be set include which type of installation disk the file is to be stored on (read only), whether to compress and decompress the file and whether to overwrite a file if it already exists. The files shown in Table J.1 possess the following attributes; Any Diskette, Overwrite Always, Compress, Decompress, Vital File and File Date is the same as the source file's date. The exceptions are the SETUP.EXE and SETUP.LST which must not possess the Compress and Decompress attributes because they initialise the installation process.

The fourth step uses the layout file to build the installation disk and the image file MMAPP.INF. This file is used by the installation program to determine where the files are stored and what their attributes are. The DOS program DSKLAYT2.EXE creates this image file and the installation disk in the specified directory. The files with the Compress attribute are compressed by this program using the COMPRESS.EXE program which resides in the same directory as the DSKLAYT programs. DSKLAYT2.EXE is run from its own directory with the following switches

DSKLAYT2 MMAPP.LYT (full path to working directory)\MMAPP.INF /d \setup /f /k 144

The first file is the layout file, followed by new image file with the path to the working directory. The */d \setup* switch determines the directory where the setup disks are to be created. The installation disk image will be stored in the directory SETUP\DISK1. The */f* switch overwrites the MMAPP.INF file if one already exists when the installation

disks are being built. The *k 144* switch instructs DSKLAYT2 to build installation disks for 1.44 Mbyte floppy disks.

The MMAPP.INF file created by DSKLAYT2.EXE unfortunately includes the files required by the initialisation and installation programs. These will be transferred along with the application's files to the application's directory. In this installation process where the user does not have a selection of installation options these files are not required. For a more sophisticated application where the user has a choice of installation options then these files may be transferred. The MMAPP.INF file is edited and the installation files are removed. The MMAPP.INF file is shown in Listing J.2.

```
[Source Media Descriptions]
      "1", "disk1", "bc30rtl.dl_", "..\disk1"

[Default File Settings]
"STF_BACKUP"     = ""
"STF_COPY"       = "YES"
"STF_DECOMPRESS" = "YES"
"STF_OVERWRITE"  = "ALWAYS"
"STF_READONLY"   = "YES"
"STF_ROOT"       = ""
"STF_SETTIME"    = "YES"
"STF_TIME"       = "0"
"STF_VITAL"      = "YES"

[Files]
      1, comdisc.ico,,,, 1993-08-04,,,, !READONLY,,,,,, 766,,,,,
      1, compact.ico,,,, 1993-07-28,,,, !READONLY,,,,,, 766,,,,,
      1, mmapp.ico,,,, 1994-10-20,,,, !READONLY,,,,,, 766,,,,,
      1, recorder.ico,,,, 1994-10-20,,,, !READONLY,,,,,, 766,,,,,
      1, speaker.ico,,,, 1994-10-20,,,, !READONLY,,,,,, 766,,,,,
      1, mtphelp.hlp,,,, 1994-11-11,,,, !READONLY,,,,,, 11166,,,,,
      1, mmapp.bmp,,,, 1994-10-01,,,, !READONLY,,,,,, 44118,,,,,
      1, mmapp.exe,,,, 1994-12-01,,,, !READONLY,,,,,, 156498,,,,,
      1, bc30rtl.dll,,,, 1992-06-10,,,, !READONLY,,,,,, 143802,,,,,
```

***Listing J.2** Installation Disk's Image file, MMAPP.INF.*

The fifth step involves manually compressing the image file into the MMAPP.IF_ file as follows,

*COMPRESS (path to working directory)\MMAPP.INF (path to working directory)\MMAPP.IF_*

The underscore character *(_)* at the end of the file name denotes a compressed file.

The penultimate step is to modify the sample list file SETUP.LST. This file determines which files are transferred to the temporary directory and is shown in Listing J.3. The setup script MMAPP.MST, is added to the list as shown in Table J.3. The *[Files]* section determines the transferred files along with their names when transferred

to the hard disk. The *[Options]* section determines the properties of the dialog box which is displayed when the files are being transferred, such as the title "Multimedia Teaching Platform Setup" and the message "Initialising Setup..." in the center of the dialog box. The name of the temporary directory and the disk space required for installation are also declare in this section. The statement *CmdLine = _mstest mmapp.mst /C "/S %s %s"* is responsible for running the installation program with the setup script, MMAPP.MST.

```
[Params]
WndTitle    = Multimedia Teaching Platform Setup
WndMess     = Initialising Setup...
TmpDirSize = 900
TmpDirName = ~msstfqf.t
CmdLine     = _mstest mmapp.mst /C "/S %s %s"
DrvModName = DSHELL

[Files]
mmapp.if_     = mmapp.inf
mmapp.in_     = mmapp.ing
mmapp.ms_     = mmapp.mst
setupapi.in_  = setupapi.inc
mscomstf.dl_  = mscomstf.dll
msinsstf.dl_  = msinsstf.dll
msuilstf.dl_  = msuilstf.dll
msshlstf.dl_  = msshlstf.dll
mscuistf.dl_  = mscuistf.dll
msdetstf.dl_  = msdetstf.dll
_mstest.ex_   = _mstest.exe
setup.exe     = setup.exe
setup.lst     = setup.lst
```

Listing J.3 *The Setup List file, SETUP.LST.*

The last step is to transfer the installation disk image files to the installation disk. The setup list and compressed image file are also transferred to this disk. The files contained on the installation disk are shown in Table J.2. The application can now be installed from this disk.

File Name	Transferred Name
mtest.ex	_mtest.exe
bc30rtl.dl_	bc30rtl.dll
comdisc.ic_	comdisc.ico
compact.ic_	compact.ico
mmapp.bm_	mmapp.bmp
mmapp.ex_	mmapp.exe
mmapp.ic_	mmapp.ico
mmapp.if_	mmapp.inf
mmapp.in_	mmapp.ing
mmapp.ms_	mmapp.mst
mscomstf.dl_	mscomstf.dll
mscuistf.dl_	mscuistf.dll
msdestf.dl_	msdestf.dll
msinsstf.dl_	msinsstf.dll
msshlstf.dl_	msshlstf.dll
msuilstf.dl_	msuilstf.dll
mtphelp.hl_	mtphelp.hlp
recorder.ic_	recorder.ico
setup.exe	setup.exe
setup.lst	setup.lst
setupapi.in_	setupapi.inc
speaker.ic_	speaker.ico

Table J.2 Installation Disk Files.