2006

An architecture for adaptive real time communication with embedded devices

Amol S. Patwardhan
Louisiana State University and Agricultural and Mechanical College, apatwa1@lsu.edu

AN ARCHITECTURE FOR ADAPTIVE REAL TIME COMMUNICATION WITH EMBEDDED DEVICES

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Systems Science

in

The Interdepartmental Program in Computer Science

by
Amol S Patwardhan
B.E, University of Mumbai, 2004
August 2006

ACKNOWLEDGEMENTS

Sincere thanks to all my friends and family members who provided valuable help and moral support while working on this thesis. Also a special thanks to my Major Professor Dr Baumgartner for his excellent guidance and timely feedback without which this thesis would not have been possible. Finally, I am very grateful to LSYOU, for providing me with the infrastructure and lab equipments to carry out my thesis related experiments.

TABLE OF CONTENTS

ABSTRACT

The virtual testbed is designed to be a cost-effective rapid development environment as well as a teaching tool for embedded systems. Teaching and development of embedded systems otherwise requires dedicated real time operating systems and costly infrastructure for hardware simulation. Writing control software for embedded systems with such a setup takes prolonged development cycles. Moreover, actual hardware may get damaged while writing the control software. On the contrary, in a virtual testbed environment, a simulator running on the host machine is used instead of the actual hardware, which then interacts with an embedded processor through serial communication. This hardware-in-the-loop setup reduces development time drastically but is reliable only if it behaves as close to real time as possible. Use of non-real time architecture like Windows NT on the host machine and the Win32 API causes an overhead in the serial communication that slows down the simulator. The problem is that the simulator is unable to cope with the communication speeds offered by the embedded processor. We propose the development of a kernel mode device driver that overcomes inefficiencies in the Win32 API.

The result is faster communication between the simulator and the embedded processor. Another problem that arises with an increase in the simulator's communication capabilities is whether the operating system can support such a dynamic and high speed interaction. To solve this problem we propose the use of efficient process and thread management and utilization of Windows NT's support for real time execution and utilization of intelligent buffer and interrupt handling to process the high frequency requests coming from the embedded processor to the host machine. Another hurdle is the

diverse nature of hardware that is being simulated: from simple features with low data volume to fairly complex features with high data volume, and with the data rate ranging from very small to very high. Hence, we propose to make the simulator and the kernel mode device driver adaptive. All these strategies culminate into an architecture for adaptive real time communication with the embedded processor, giving the virtual testbed an edge over other design methodologies for embedded systems.

CHAPTER 1. INTRODUCTION

Embedded systems [7, 16, 18, 21] find application in a vast number of devices ranging from wrist watches and dish washers all the way to nuclear plants. As a result, embedded systems have certain characteristic requirements associated with them: they have to be real time, the production cost and time right from design and manufacturing to their testing phase has to be short, they have to be reliable due to their application in critical fields like medicine, military etc, and finally they have to reach the market in time. Thus it is important to design and test embedded systems in a timely and cost effective manner. Most of the embedded systems design [3] involves significant amount of time and effort in software and hardware development. An important design goal is to make the software and hardware design and development more and more concurrent instead of waiting for the control software to be written and tested, after the hardware has been designed and fabricated. Currently, any problems or lack of support for the software results in a significant design change in the hardware and the whole cycle has to repeat again. This calls for the need of a better design and development methodology.

One of the prospective environments for concurrent development and teaching embedded systems instruction is a hardware-in-the-loop (HIL) [11, 20] virtual testbed environment. Hardware in the loop is also a potentially useful technique for developing and testing complex embedded systems. In such a virtual environment, an actual embedded processor is coupled with a simulation of an external mechanical or electrical device instead of the actual hardware itself. Existing HIL environments typically use a dedicated simulation computer, often with a real-time operating system, and specialized hardware for establishing electrical connections to the embedded processor. In our virtual

testbed, the simulation runs on a host machine based on popular operating system platform such as the Windows NT using a low-cost network connection with the embedded processor. The presence of simulation eliminates the waiting time between fabrications of the hardware. Also there is no risk of damaging the hardware while writing the control software of the embedded system. For this virtual testbed to provide valid results, it is very important that the simulation runs as close to real time as possible.

The performance of the simulation is measured through simulation frame and highly depends on the operating system running on the host machine. Using a real-time operating system to run the simulation is the best solution for achieving real-time performance. But an important issue, before the virtual testbed can be actually put to use as an embedded systems development and training environment is that it has to be cost effective and simple to administer. Using off-the-shelf equipment with Windows XP and simple networking hardware makes the HIL environment more cost effective than with dedicated simulation computers and custom hardware. For systems under test with a low I/O rate and a simulation that is not overly complex, Windows XP provides a fairly valid and useful HIL simulation environment. But there are some hurdles in this choice. The simulator is unable to cope with the fast communication capabilities of the embedded processor. The interface between the simulator and the embedded processor is the serial port. In Windows XP, the simulator relies on the Win32 API to handle the data transfer to and from the serially connected embedded processor. With every additional layer in the communication architecture, there are overheads due to copying of data between buffers and the extra time that it takes. This thesis explores adding support to the simulation at the Windows XP kernel level for improving the performance of the simulator. The thesis

proposes the development of a kernel mode device driver to make the communication between the simulator in the user mode and the embedded processor faster.

Chapter 2 provides a brief description of the Windows XP architecture. It is important to understand the Windows XP components and how the device driver fits in. In the same chapter, we also provide an overview of a typical hardware device. Chapter 3 gives details about serial communication and the serial device. Chapter 4 describes the structure of a kernel mode device driver [2, 17]. We discuss the various device and kernel level objects and data structures that are used in the kernel mode device driver development in Chapter 5. We propose an important strategy for implementing an efficient kernel mode device driver. The simulation runs on the Windows XP operating system. Windows XP does not allow applications running in user mode to directly access the serial port. This can be overcome by manipulating the I/O permission map [5, 19], thereby giving unrestricted serial port access to a particular user mode application such as the simulation. The resulting communication is much faster than the one using the Win32 API function calls in conjunction with the standard serial drivers provided by the Windows XP operating system. The simulator is able to perform significantly faster and is able to cope with the communication speeds of the embedded processor, as is explained in Chapter 6. However, a new problem arises due to the drastic improvement in communication performance.

The Windows XP operating system should be able to handle the higher rate of data exchange with the embedded processor. To deal with this problem we propose using API support from Windows XP for real time execution of processes and threads. As a process inside the operating system, the requests from the embedded processor, and the

request from the simulator can be given higher priority to achieve performance within the expected time frame. But the problem is not completely solved here. Due to the wide variety of hardware that can be simulated and the diverse applications of embedded systems [7, 16, 18, 21], it is important that the virtual testbed is robust enough to deal with the different data volume requirements and time-critical nature of the different systems being developed or taught. As a result, we propose providing an architecture for adaptive real-time behavior as a platform for the simulator running on the host machine. The buffer size, interrupt, thread and process priority will be set, on the basis of the data volume during the communication between the simulator and the evaluation board.

The adaptive behavior of the architecture is based on a heuristics that are derived from performance measurements of the virtual testbed [6, 10] under different scenarios such as variable buffer size, interrupt trigger levels, and thread and process priorities. Chapter 7 discusses the implementation of this architecture. Chapter 8 has a description of the performance measurements of the communication between the simulator and the embedded processor, obtained using the kernel mode device driver [2, 17], process priorities, thread priorities, the buffer management and interrupt handling strategies. Chapter 9 provides the heuristics for implementing the adaptive architecture. The Chapter 10 summarizes the results and a conclusion of how the adaptive architecture can be successfully used to improve the interaction between the embedded processor and the simulation running on Windows XP is made. It provides insights and points out the areas to improve current architecture implementation and it also provides a description of future application for the architecture.

CHAPTER 2. ARCHITECTURE OF THE WINDOWS NT KERNEL MODE COMPONENTS

Windows NT consists of three basic kernel mode components as shown in Fig 1. and are described as follows:

Hardware Abstraction Layer

The hardware abstraction layer (HAL) provides a thin software layer abstraction used by the system to interact with hardware that is not part of the *Central Processing Unit* (CPU). HAL provides abstraction of the platform.

Kernel

The kernel provides services for interrupt and exception dispatching, thread scheduling and synchronization, multiprocessor synchronization and time keeping. It provides an object based interface to the higher level components of the operating system.

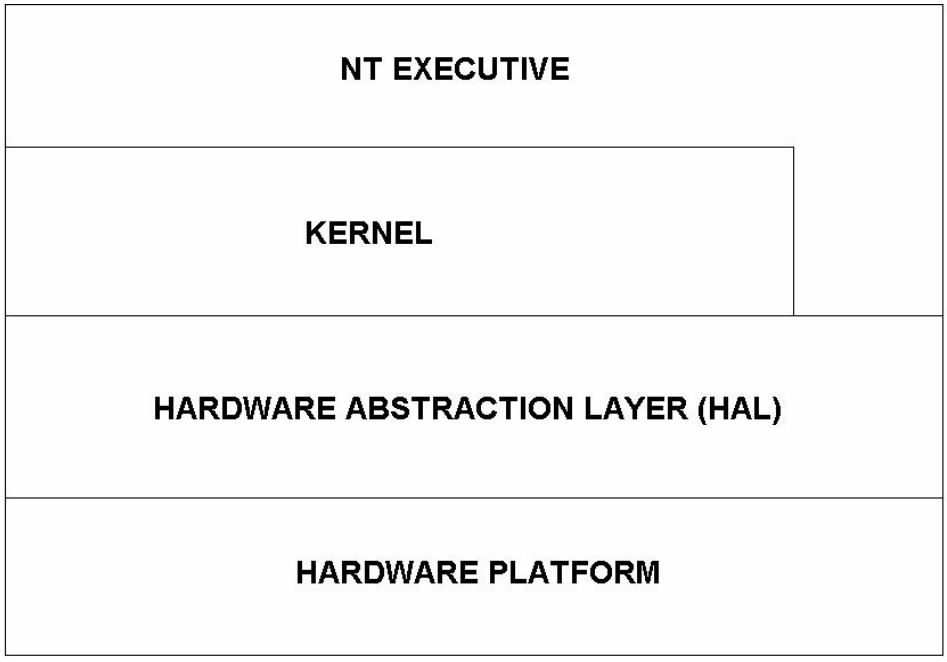

Fig 1. Overview of Windows NT Architecture [2]

The higher level operating system components call the kernel mode services to implement various tasks.

Executive

The executive consists of many independent software components that interact through well defined interfaces.

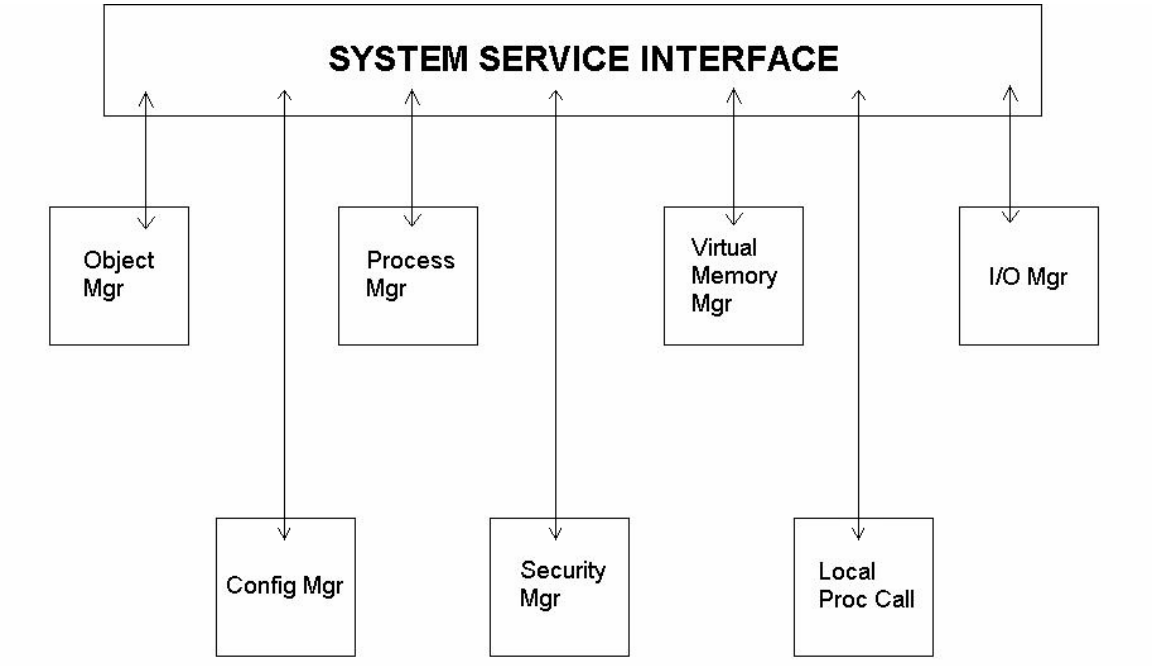

Fig 2. Structure of Executive [2]

The interfaces of the executive are shown in Fig 2. and are described as follows:

1. System service interface:

   The system service interface gives user mode access to executive service, using a technique based on CPU's hardware exception mechanism.

2. Object Manager:

   Object Manager manages the executive objects such as files, processes, threads, and shared memory segments. This includes creating, deleting and keeping track of object references.

3. Configuration Manager:

   Configuration manager maintains a database of hardware and software information called the Registry.

4. Process Manager:

   The process manager is responsible of handling creation, deletion and management of processes and threads. It also provides thread synchronization services.

5. Security Reference Monitor:

   The security reference monitor enforces the system security policies and provides a set of primitives that both kernel and user mode components can call.

6. Local Procedure Call facility:

   The local procedure call facility is a message passing mechanism used for communication between processes on the same machine [2].

7. I/O Manager:

   The I/O manager uses a well defined formal interface to communicate with the device drivers. The I/O manager converts I/O requests from user and kernel mode threads into properly sequenced driver routine calls.

Now that we have a basic understanding of the architecture of various kernel mode components of Windows NT, we discuss the structure of a typical hardware environment. This is needed to understand the concepts behind the serial device which would be used as a means of communication between the simulator and the DSP board that hosts the embedded processor. A typical hardware environment consists of special purpose device registers that are extensively used for handling device operations.

Device Registers

Drivers communicate with peripheral devices using a set of device registers. The device register will perform the following functions:

1. Command: These bits control the operation and behavior of the device.
2. Status: These bits give the information on the current state of the device.
3. Data Buffer: Input and Output data is stored in these buffers.

Accessing the registers:

We need to know the base address of the device registers and the address space they reside in to access them. The address space may be I/O address space or memory mapped address space.

Device Interrupts

The hardware devices generate interrupts in the form of an electrical signal on the interrupt line in the bus whenever they need some form of attention from the CPU.

Interrupt Priorities

When several devices are connected to the same bus, the CPU uses a priority mechanism to service the interrupts generated by the devices. The mechanism depends on the bus and it works by assigning a priority value to each interrupt request line [2].

Interrupt Vector

The interrupt vector is a unique, bus-relative number that allows the CPU to identify the source of an interrupt and call the appropriate service routines [2].

CHAPTER 3. SERIAL COMMUNICATION

A serial port [1, 8, 15] is an input/output port available in most computers. It follows the RS232 or EIA232 specification and is usually a 9 pin or a 25 pin connector.

Transmission

The serial port has one transmit buffer and a transmit shift register. The transmit buffer is used to transmit one byte of data to the transmit shift register. The data from the shift register is placed on the serial line one bit at a time. As soon as the transmit buffer gets empty, the serial port sends an interrupt to the CPU over a dedicated interrupt line. The CPU runs the serial port device driver to check the serial port registers. It finds that the transmit buffer is empty and sends a byte to the I/O address, if there is a byte to send. The CPU sends one byte of data to the I/O address, over a 32 bit bus, which is a very inefficient way of using the bus band width. There is also a lot of overhead in interrupt handling. When the CPU receives an interrupt, the device driver has to check the registers before it knows what happened at the serial port. The serial port buffer size ranges from 1 byte to 16 byte. Recently serial ports with buffer size of 64 bytes are also available. This buffer is a FIFO buffer.

Reception

The serial port receives data from the external cable. It stores this data in a receive shift register. The receive shift register puts this data into the receive buffer. When the receive buffer is full, it generates an interrupt which causes the CPU to stop the current process execution and pick up the byte from the receive buffer. The CPU interacts with the serial port buffer, through a larger buffer that resides in memory.

The serial port is usually called RS232C, EIA232 or TIA232. RS stands for Recommended Standard, EIA for Electronics Industries Association, and TIA for Telecommunication Industries Association, respectively. The various port addresses for a serial device are shown in Table 1.

Table 1. Standard Port Addresses for Serial Port

Name	Address	IRQ
COM1	0x3F8	4
COM2	0x2F8	3
COM3	0x3E8	4
COM4	0x2E8	3

The serial port uses the following registers:

1. DLAB: Divisor Latch Access Bit
2. IER: Interrupt Enable Register
3. IIR: Interrupt Identification Register
4. FCR: FIFO Control Register
5. LCR: Line Control Register
6. LSR: Line Status Register
7. MCR: Modem Control Register
8. Scratch Register

Table 2. FIFO Control Register Bits 6 and 7

Bit 6	Bit 7	Interrupt Trigger Level
0	0	1 Byte
0	1	4 Byte
1	0	8 Byte
1	1	14 Byte

The register of interest for us is the FIFO Control, a write-only register, in which bits 6 and 7 are used to control the interrupt trigger level. The received data available

interrupt will occur at the trigger level set using bits 6 and 7 in the FCR. Table 2. shows the various values that can be set for bit 6 and 7 of the FCR. Thus when we set the bit 6 and 7 to 0 and 0 respectively, it means that the buffer size id 1 byte. Whenever the buffer receives 1 byte of data an interrupt will be generated.

CHAPTER 4. KERNEL MODE DEVICE DRIVER

The difference between an application program and a kernel mode device driver [2, 17] is that the application program runs under the control of a main or a WinMain function, whereas a kernel mode device driver has no main. Instead a kernel mode device driver has a collection of subroutines that are called as needed by the I/O Manager. The major categories of routines that make up the driver are:

Driver Initialization and Cleanup Routines

1. DriverEntry routine:

   The I/O manager calls this routine when it loads the driver [2]. The routine performs various initialization functions, hardware resource allocation and making the driver name available.

2. Unload routine:

   The I/O manager calls this routine when a driver is unloaded. The routine destroys any kernel objects and de-allocates any hardware resources.

3. Shutdown routine:

   The routine puts the device in a known state if the system happens to shut down.

I/O System Service Dispatch Routines

When the I/O Manager gets a request it uses the function code to call one of the several dispatch routines in the driver [2].

Open and close operations:

The drivers provide a routine to handle CreateFile Requests and the CloseHandle function calls.

Device Operations:

The driver may have one or more Dispatch routines for handling actual data transfer in response to the Win32 ReadFile, WriteFile and DeviceIoControl requests [2].

1. Data transfer routines:

   Depending on nature and complexity of the device a number of different driver routines are involved.

2. Start I/O routine:

   The I/O manager calls this routine in order to begin a device operation.

3. Interrupt service routine (ISR):

   The ISR routine handles the interrupts generated by the device and executes the *deferred procedure call* (DPC) routine. The ISR routine should be as small as possible because we do not want the system to spend too much time executing this routine.

4. DPC routines:

   One or more DPC routines perform clean up, release of system resources, error reporting and starting the next device operation.

Other Driver Routines

1. Timer routine:

   Drivers can keep track of time that has passed during a device operation using the timer routine.

2. I/O Completion routine:

   The lower level drivers notify completion of request processing using a call to the higher level driver I/O completion routine.

3. Cancel I/O routine:

   If the request is canceled, the I/O manager calls the Cancel I/O routine to perform cleanup operations [2].

We further discuss various data structures associated with a kernel mode device driver in the next chapter.

CHAPTER 5. DRIVERS AND KERNEL MODE OBJECTS

I/O is packet driven under Windows NT. Each I/O transaction is described in the form of a data structure called an *I/O Request Packet* (IRP) which is shown in Fig 3.

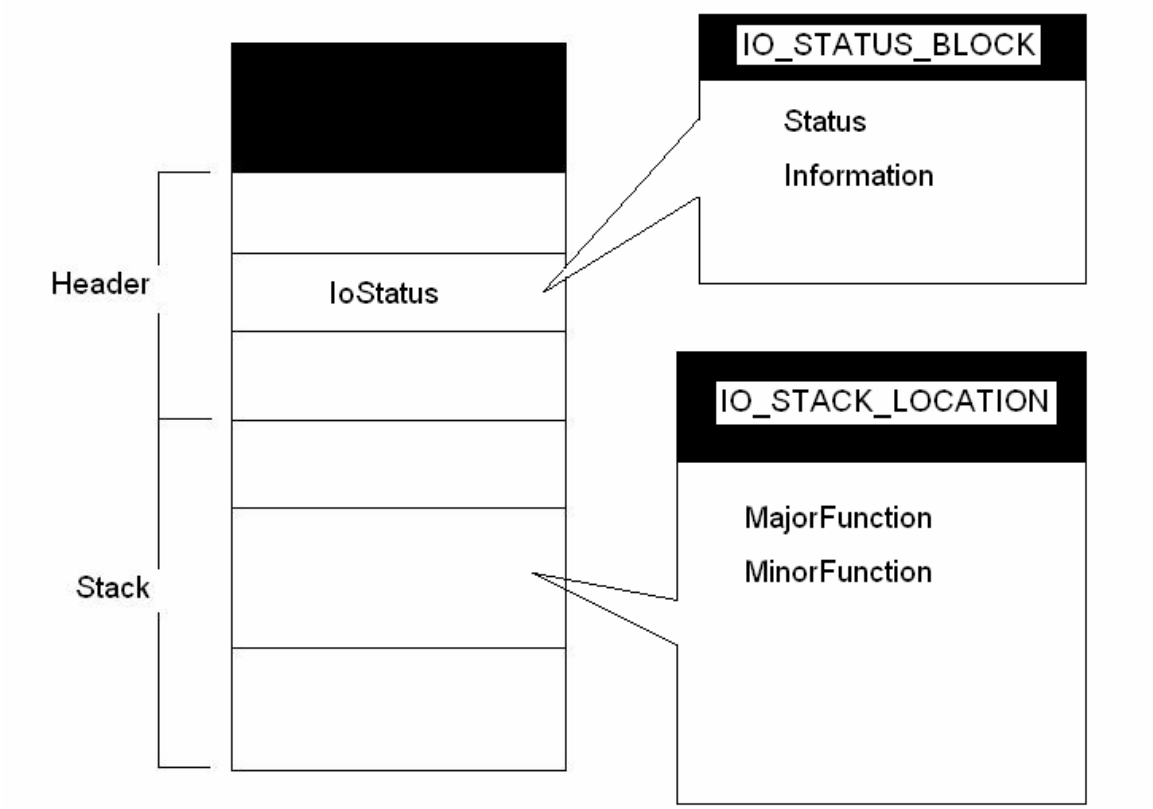

Fig 3. Structure of IRP [2]

The IRP header holds information about the IRP request. The IoStatus member holds the final state of the operation. The Status field holds the status value for the current IRP processing. The information field holds a 0 value or a function-code-specific value. The I/O stack location holds the function code and parameters for an I/O request. For requests sent directly to the lowest-level driver the corresponding IRP will have only one I/O stack location in the stack.

There is a unique *driver object* for each driver currently loaded in the system. A driver object contains a pointer to a linked list of devices serviced by the driver as shown

in Fig 4. There are no access functions to modify the driver object. The DriverEntry routine sets various fields directly.

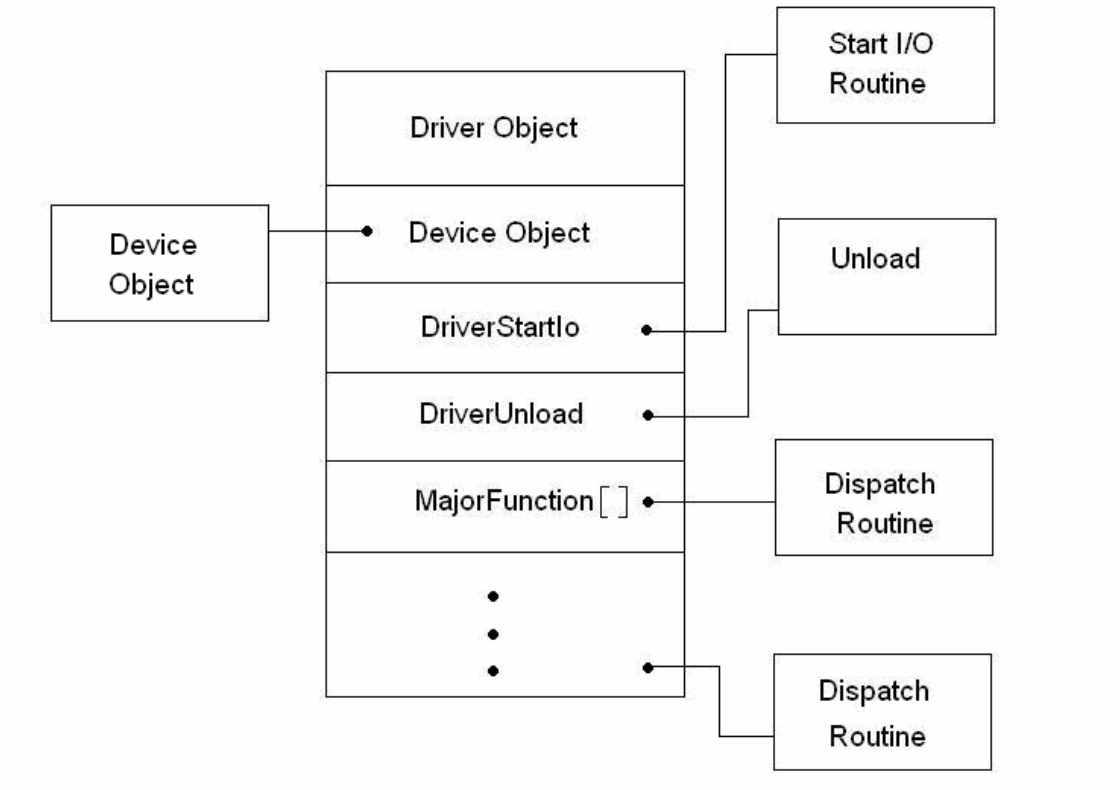

Fig 4. Driver Object Structure [2]

Most of the data in the *device object* is the exclusive property of the I/O manager. Fig 5. shows the structure of a device object. The I/O Manager passes a device object pointer as an argument to most of the routines in the driver. Some of the access functions for a device object are:

1. IoCreateDevice
2. IoCreateSymbolicLink
3. IoDeleteDevice

The device object is connected to another important data structure called *device extension*. It's a block of nonpaged memory automatically attached by the I/O manager.

A device extension structure holds information associated with a particular device. The device extension stores device state, a pointer to an *interrupt object* and a back pointer to the Device object.

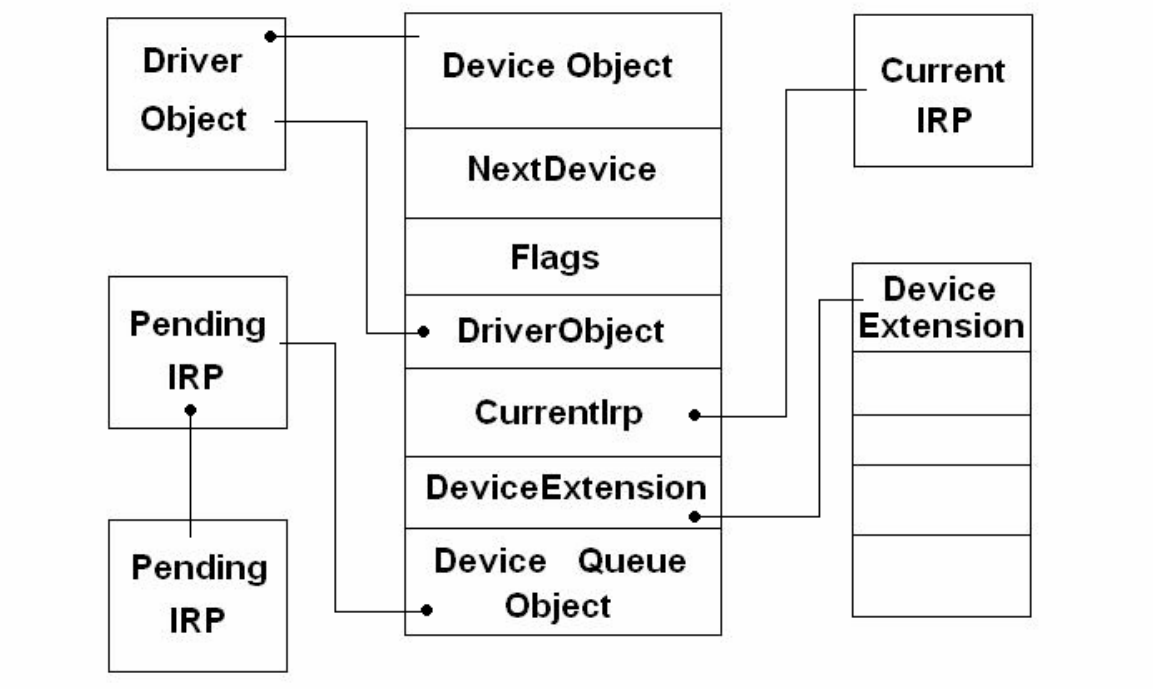

Fig 5. Device Object [2]

The interrupt object is created by the DriverEntry routine for each interrupt vector supported by the device. The interrupt object is used to locate the Interrupt Service routine whenever an interrupt occurs.

Fig 6. shows how the interrupt object is linked to the interrupt service routine. The interrupt object is deleted by the Unload routine. Some of the access functions for an interrupt object are:

1. HalGetInterruptVector
2. IoConnectInterrupt
3. IoDisconnectInterrupt

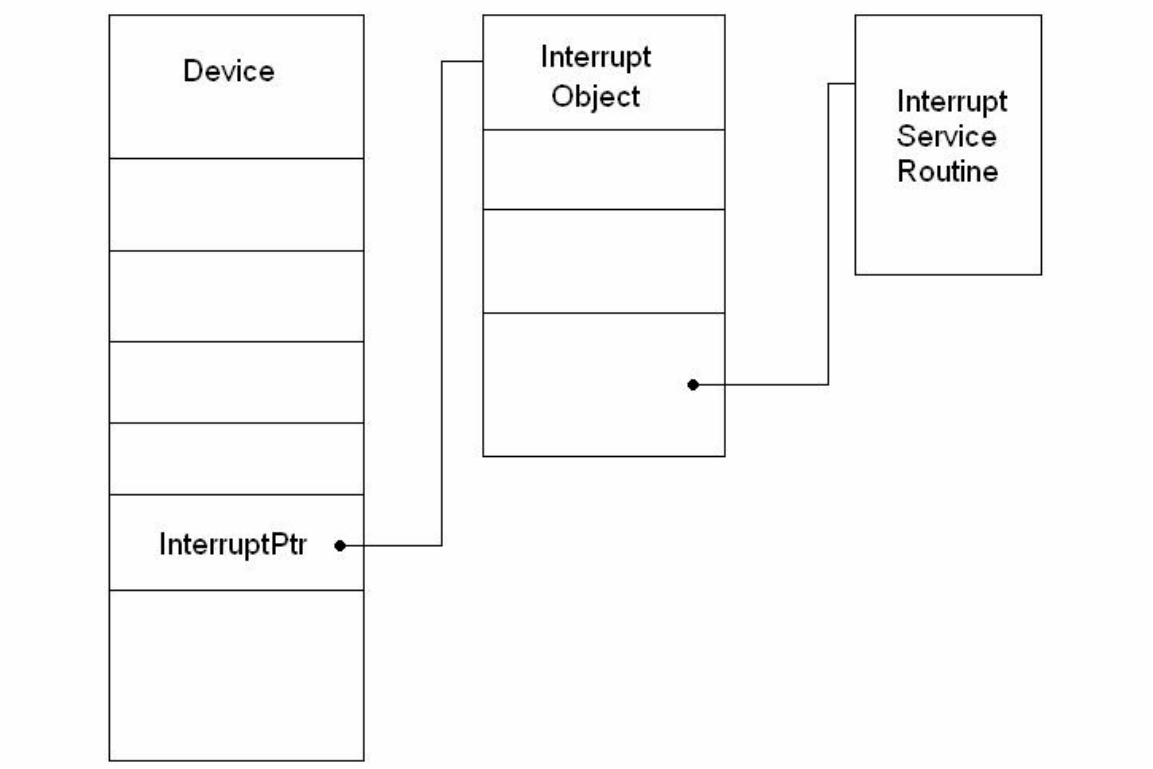

Fig 6. Interrupt Object [2]

CHAPTER 6. DRIVER IMPLEMENTATION

The simulator is an application running in the user mode level which is also called as the ring 3 of the Windows NT architecture. As a result it has less privileges than the kernel level components of the operating system. Any interaction between the simulator and the embedded processor takes place through an intermediate kernel mode device driver. The driver is named RTSerial because of our aim at making the communication over the serial channel as close to real time as possible.

```
void OpenRTSerial(void)
{
     hRTSerial = CreateFile("\\\\.\\RTSerial",
                                GENERIC_READ,
                                0,
                                NULL,
                                OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL,
                                NULL);
}

void CloseRTSerial(void)
{
    CloseHandle(hRTSerial);
}
```

Fig 7. Open and Close Methods

We obtain the reference to our driver handle in the user mode simulator program, using a simple CreateFile call. To close the driver we use the CloseHandle function call. This code implementation is shown in Fig 7.

We use a technique where the simulator communicates with the kernel mode device driver that uses an optimized mechanism for communicating with the serial port

under it. Before we discuss this any further let us find out what IOPM (Input/Output permission map) [5, 19] is. IOPM is an area of memory that stores an access bit map of 8192 bytes (2000 Hex Bytes). Within the IOPM each byte location represents 8 I/O addresses. Whenever the processor comes across an I/O request the I/O permission map is checked for access privileges on the address for which the access has been requested.

In a typical Windows NT architecture the Input/ Output permission map has bits set to 1 and that means user mode applications are not allowed to directly access or communicate with the devices. In our implementation we manipulate the address that corresponds to the port address of the serial port and set the corresponding bits to zero. The kernel mode device driver [2, 17] manipulates the IOPM such that it can directly write to the serial port thereby bypassing the strict restrictions of Windows. As a result, the driver completely eliminates the need for a kernel level buffer for temporary data storage while the data is being sent to or received from the serial channel. This is achieved using a few undocumented calls such as:

1. void Ke386SetIoAccessMap(int, IOPM *);
2. void Ke386QueryIoAccessMap(int, IOPM *);
3. void Ke386IoSetAccessProcess(PEPROCESS, int);

Now we take a look at the DeviceIoControl routine responsible for handling the actual read and write requests coming from the user mode application program such as the simulator. As one can see in Fig 8., it is a simple switch statement that reads or writes to the serial channel using built macros provided by the device driver API, depending on the type of request it gets. But how does the simulator communicate with the driver? The answer lies in the asyncRTRead and asyncRTWrite functions. We name the function

prefixed with ‘async’ because we make use of the asynchronous communication while performing our data transfer.

```
switch ( irpSp->Parameters.DeviceIoControl.IoControlCode )
 {
  case IOCTL_READ:

        if ((inBufLength >= 2) && (outBufLength >= 1)) {
            (UCHAR)Value = READ_PORT_UCHAR((PUCHAR)ShortBuffer[0]);
            CharBuffer[0] = Value;
        } else ntStatus = STATUS_BUFFER_TOO_SMALL;
        pIrp->IoStatus.Information = 1;
        ntStatus = STATUS_SUCCESS;
        break;

  case IOCTL_WRITE:
        if (inBufLength >= 3) {
            WRITE_PORT_UCHAR((PUCHAR)ShortBuffer[0], CharBuffer[2]);
        } else ntStatus = STATUS_BUFFER_TOO_SMALL;
        pIrp->IoStatus.Information = 0;
        ntStatus = STATUS_SUCCESS;
        break;
    .
    .
    .
    .
}
```

Fig 8. DeviceIoControl

The ‘RT’ stands for Real Time. These functions are the interface for the communication between the simulator at the user mode level and the kernel mode device driver. These functions will be used to replace the slower counterparts such as the ReadFile and WriteFile functions available in the Win32 API as discussed later.

One can easily see in Fig 9. how the asyncRTRead or asyncRTWrite functions encapsulate the DeviceIoControl function call which is responsible to pass the control code such as read or write and the address of the device to communicate down to the driver. Note that the access to the device would not have been possible without manipulating the IOPM.

```
BOOL asyncRTRead(unsigned short pAddress, unsigned char *ptrByte,
                 OVERLAPPED *ptrOverlap, DWORD *ptrDwBytesRead)
{
    BOOL error;
    unsigned char Buffer[3];
    unsigned short * pBuffer;
    pBuffer = (unsigned short *)&Buffer;
    *pBuffer = pAddress;

    error = DeviceIoControl(hRTSerial,
                            IOCTL_READ,
                            &Buffer,
                            2,
                            &Buffer,
                            1,
                            ptrDwBytesRead,
                            ptrOverlap);
    ptrByte = Buffer;
    ucharTmp = ptrByte[0];
    return error;
}
```

Fig 9. asyncRTRead

Now where do we call this function from our simulator code? The answer is simple. We use a thread that runs in the background for handling any kind of interrupt generated by the serial device. This thread is invoked as soon as the simulator is up and running. The thread implements a mechanism called overlapped I/O [14] which allows simultaneous execution of reads and writes to the device and at the same time the graphical user interface (GUI) of the simulator does not get stalled. The way this works is the thread waits on read and write events to occur instead of continuously polling for these events. As a result any read and write call returns with some kind of status code such as ‘success’ or ‘pending’ thereby letting the normal flow of program execution to continue uninterrupted. Fig 10. shows how to use the overlapped I/O technique [14]. The WaitCommEvent waits on the events specified for the particular device asynchronously.

```
unsigned __stdcall CSerialCommHelper::ThreadFn(void*pvParam)
{
    OVERLAPPED ov;
    memset(&ov,0,sizeof(ov));
    ov.hEvent = CreateEvent( 0,true,0,0);
    HANDLE arHandles[2];
    arHandles[0] = apThis->m_hThreadTerm;
    DWORD dwWait;
    SetEvent(apThis->m_hThreadStarted);
    while (  abContinue )
    {
        BOOL abRet = ::WaitCommEvent(apThis->m_hCommPort,
                                     &dwEventMask,
                                     &ov) ;
        if ( !abRet )
        {
            ASSERT( GetLastError () == ERROR_IO_PENDING);
        }

        arHandles[1] = ov.hEvent ;
        dwWait = WaitForMultipleObjects (2,
                                     arHandles,
                                     FALSE,
                                     INFINITE);
        switch ( dwWait )
        {
        case WAIT_OBJECT_0:
            {
                _endthreadex(1);
            }
            break;
        case WAIT_OBJECT_0 + 1:
            {
                DWORD dwMask;
                if (GetCommMask(apThis->m_hCommPort,&dwMask) )
                {
                    if ( dwMask == EV_TXEMPTY )
                    {
                        AfxMessageBox("Data sent");
                        ResetEvent ( ov.hEvent );
                        continue;
                    }
                }
                //read data here...
            break;
         }
    }
    return 0;
}
```

Fig 10. Structure of Interrupt Handler

The GetCommMask function gets the information about the event raised. As soon as data is available for the device to read we issue an asyncRTRead call and whenever we need to send data we make an asyncRTWrite call.

```
try
{
    BOOL abRet = false;
    DWORD dwBytesRead = 0;
    OVERLAPPED ovRead;
    memset(&ovRead,0,sizeof(ovRead));
    ovRead.hEvent = CreateEvent( 0,true,0,0);
    std::string szData;
    do
    {
        ResetEvent( ovRead.hEvent  );
        unsigned char szTmp[1];
        int iSize  = sizeof ( szTmp );
        memset(szTmp,0,iSize);
        OpenRTSerial();
        abRet = asyncRTRead(0x3F8, &szTmp[0], &ovRead, &dwBytesRead);
        CloseRTSerial();

        if (!abRet )
        {
            abContinue = FALSE;
            break;
        }
        if ( dwBytesRead > 0 )
        {
            myStr += szTmp[0];
            myStr.append("\r\n");
            apThis->my_DataRx->SetWindowText(myStr.c_str());
            iAccum += dwBytesRead;
            apThis->m_theSerialBuffer.Flush();
        }
    }while (0);
    CloseHandle(ovRead.hEvent );
}
catch(...)
{
```

Fig 11. Call to asyncRTRead

Depending on the event we either issue an asynchronous Read or Write call as shown in Fig 10. As can be seen we first open a handle to our driver which in turn gives

us direct access to the serial device and then we make an asyncRTRead call. The same is true with an asyncRTWrite call.

We also make sure that we close the handle. Note that this eliminates the role of standard serial device driver that comes into play if we would have made the regular Win32 Read and Write function calls. The standard serial device driver supports much other functionalities which are not a necessity in our scheme of things.

The same call using the Win32 API would have looked as shown in Fig 12. The elimination of ReadFile and WriteFile calls reduces the execution time of I/O operations drastically. Since we are directly communication with the I/O port we are skipping the overheads caused due to passage of data through buffers at various levels such as at the application level, kernel level and the device level.

The interface used for communication in the virtual testbed [6, 10] environment is the serial channel. The serial device has a built in FIFO whose size can be set using the FCR (FIFO Control Register). An advantage of the kernel mode device driver implementation is that the simulator is in a position to control the FIFO size depending on the data volume and communications speed requirements by performing simple writes to a specific offset from the serial ports base address, which would represent the FIFO Control Register. The call is a simple asyncRTWrite call. We pass the base address of the serial port + 2, in other words the address of FCR (FIFO control register). Thus to set the FIFO size to 4 we write the 0x2 value to this address and set the 6^{th} and the 7^{th} bit of the FIFO control register accordingly. The FIFO size determines how quickly the receive buffer will get filled with incoming data and how often the Receive data ready interrupt will be generated. The FIFO size can be controlled at the user mode level, through the

simulator too. Thus we have a flexibility to control the FIFO size through both layers of the architecture, i.e. through User mode and also through Kernel mode.

```
try
{
    BOOL abRet = false;
    DWORD dwBytesRead = 0;
    OVERLAPPED ovRead;
    memset(&ovRead,0,sizeof(ovRead));
    ovRead.hEvent = CreateEvent( 0,true,0,0);
    std::string szData;
    do
    {
        ResetEvent( ovRead.hEvent  );
        unsigned char szTmp[1];
        int iSize  = sizeof ( szTmp );
        memset(szTmp,0,iSize);
        abRet = ::ReadFile(apThis->m_hCommPort,
                           szTmp,sizeof(szTmp),&dwBytesRead,&ovRead);
        if (!abRet )
        {
            abContinue = FALSE;
            break;
        }
        if ( dwBytesRead > 0 )
        {
            myStr += szTmp[0];
            myStr.append("\r\n");
            apThis->my_DataRx->SetWindowText(myStr.c_str());
            iAccum += dwBytesRead;
            apThis->m_theSerialBuffer.Flush();
        }
    }while (0);
    CloseHandle(ovRead.hEvent );
}
catch(...)
{
```

Fig 12. ReadFile Call

This gives us better control over the FIFO buffer size and also supports adaptive characteristics of the architectural better. This implies that we have control over the interrupt trigger level. User mode manipulation of the buffers is achieved using registry

manipulation techniques. The registry stores device specific information and can be used to change the device register values as shown in Fig 13. and Fig 14.

```
void SetRxFifoSize(int size)
{
    HKEY hKey;
    char FIFOstr[10];
    if(size == 1 || size == 4 || size == 8 || size == 14)
    {
        strcpy(FIFOstr,"RxFIFO");
        RegOpenKeyEx(HKEY_LOCAL_MACHINE,
                     "SYSTEM\\CurrentControlSet\\Services\\Serial",
                     0,
                     KEY_ALL_ACCESS,&hKey);
        RegSetValueEx(hKey, // handle of key to set value for
                      FIFOstr,  // address of value to set
                      0,    // reserved
                      REG_DWORD,    // flag for value type
                      (unsigned char*)&size,    // address of value data
                        sizeof(size)    // size of value data
                        );
        RegCloseKey(hKey);
    }
}
```

Fig 13. Setting the Receive Buffer Size

```
void SetTxFifoSize(int size)
{
    HKEY hKey;
    char FIFOstr[10];
    if(size <= 16 && size >= 1)
    {
        strcpy(FIFOstr,"TxFIFO");
        RegOpenKeyEx(HKEY_LOCAL_MACHINE,
                     "SYSTEM\\CurrentControlSet\\Services\\Serial",
                     0,
                     KEY_ALL_ACCESS,&hKey);
        RegSetValueEx(hKey, // handle of key to set value for
                      FIFOstr,  // address of value to set
                      0,    // reserved
                      REG_DWORD,    // flag for value type
                      (unsigned char*)&size,    // address of value data
                        sizeof(size)    // size of value data
                        );
        RegCloseKey(hKey);
    }
}
```

Fig 14. Setting the Transmit Buffer Size

We first open the appropriate key in the registry for the serial device and then set the buffer size using the function RegSetValueEx. Some of the parameters the RegSetValueEx accepts are a handle to the registry key and the size to be set for the buffer. The acceptable values for the receive buffer size are 1, 4, 8 or 14. The acceptable values for transmit buffer size are 1 to 16. We use the kernel mode device driver to improve the communication, which results in a drastic improvement of the time needed for data transmission as illustrated in Fig. 15.

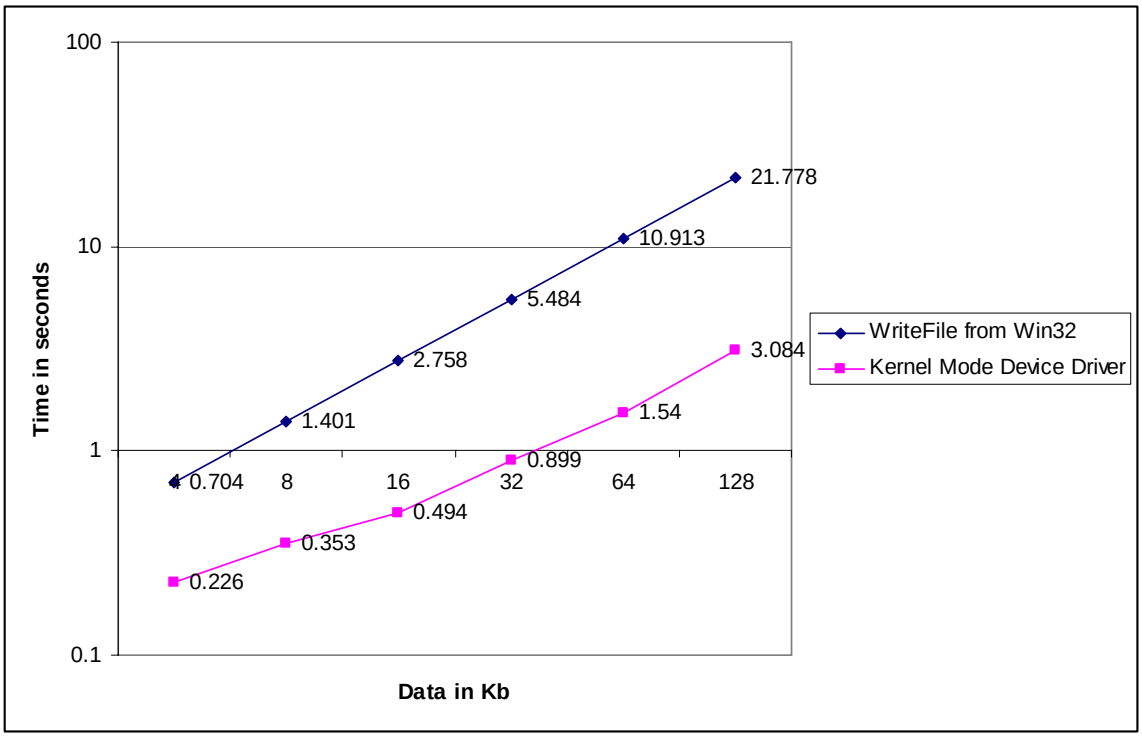

Fig 15. Improvement in Data Transmission Time

The graph shows that the performance has improved by a factor of 7, which is an important achievement in bringing the simulators capabilities to cope with the high speed communication requirements close to expected. But as a direct result of this we see an important performance issue which is explained in the following matrix in Fig 16. There are a number of embedded systems applications that need to transmit large amounts of

data at a very high rate and where the simulator is required to be equally responsive to the requests coming from the embedded processor.

	Small Buffer size	Large Buffer Size
Small Data Volume	Real Time	Latency
Large Data Volume	Performance Issue	Close to Real Time

Fig 16. Behavior Matrix

So improving the communication speed is not enough. Hence we propose designing an architecture that would support and overcome the problem. We discuss the implementation of the architecture in our next chapter.

CHAPTER 7. IMPLEMENTATION OF ARCHITECTURE

We saw in the previous chapter how a kernel mode device driver [5, 19] can make the communication between simulator and the embedded processor faster. But this has its implications on the overall performance of the virtual testbed especially on the host machine end, where the underlying operating system architecture should be robust and scalable enough to support such a high degree of interrupt generation. To handle this we propose using thread and process management support provided by Windows NT and the Win32 API.

We therefore introduce a characteristic to the architecture and provide the simulator with an ability to configure its process and thread priorities on the basis of data volume, communication speed and the nature of hardware it is representing. The simulator can set the process priority to Real Time, which is the highest priority level at which a process can run. Similarly the simulator can set the priority of the thread that listens and writes to the serial channel, thus making these time critical activities get scheduled quicker. This will be discussed in detail in the subsequent sections.

Process Priority

Under Windows NT, it is possible to programmatically set the priority of a process using a function called SetPriorityClass [4, 9, 12]. Depending on the type of hardware, the volume of data transfer and the communication speed requirement of the simulator can set the process priority level at which it needs to run.

We have a number of priority levels available for use but we are specifically interested in the REALTIME_PRIORITY_CLASS. Table 3. illustrates the various priority levels that can be set for a process.

Table 3. Priority Class Values [12]

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS 0x00008000	Process that has priority above NORMAL_PRIORITY_CLASS but below HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS 0x00004000	Process that has priority above IDLE_PRIORITY_CLASS but below NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS 0x00000080	Process that performs time-critical tasks that must be executed immediately. The threads of the process preempt the threads of normal or idle priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class application can use nearly all available CPU time.
IDLE_PRIORITY_CLASS 0x00000040	Process whose threads run only when the system is idle. The threads of the process are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle-priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS 0x00000020	Process with no special scheduling needs.
REALTIME_PRIORITY_CLASS 0x00000100	Process that has the highest possible priority. The threads of the process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

We obtain the handle to the current process that represents the simulator using the function GetCurrentProcess. Then we pass value ranging from 0 to 4 depending on the system performance requirement to the switch case to set the appropriate priority level.

Thus we make use of these facts and available API to set the priority as shown in Fig 17.

```
hProcess = GetCurrentProcess();
switch (procPrt)
{
case 0: stat = SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS);
        break;
case 1: stat = SetPriorityClass(hProcess, ABOVE_NORMAL_PRIORITY_CLASS);
        break;
case 2: stat = SetPriorityClass(hProcess, HIGH_PRIORITY_CLASS);
        break;
case 3: stat = SetPriorityClass(hProcess, REALTIME_PRIORITY_CLASS);
        break;
default: break;
}
```

Fig 17. Setting the Process Priority

Thread Priority

Under Windows NT, it is possible to programmatically set the priority of a thread using a function called SetThreadPriority [9, 13]. Depending on the type of hardware, the volume of data transfer and the communication speed requirement of the simulator can set the thread priority at which it needs to run. This value, together with the priority class of the thread's process, determines the thread's base priority level. Table 4. illustrates the various Priority levels that can be set for a thread.

Table 4. Thread Priorities

Thread Priority Level
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL

Fig 18. shows how to set the priority of the thread that would be performing read and write operations. We first obtain the handle to the thread for which we wish to set the

priority. Then we pass this handle along with the thread priority as arguments to the SetThreadPriority function. In Fig. 18, the thread has been assigned a normal priority, which is the default priority. Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The priority class and the priority level are combined to form the base priority of each thread.

```
m_hThread = (HANDLE)_beginthreadex(0,
                                   0,
                                   CSerialCommHelper::ThreadFn,
                                   (void*)this,
                                   0,
                                   0 );

SetThreadPriority(m_hThread, THREAD_PRIORITY_NORMAL);
```

Fig 18. Setting the Thread Priority

The system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level does scheduling of threads at a lower level take place. The following table shows the base priority levels for different combinations of priority class values and thread priority level values that can be used to determine the base priority:

Table 5. Base Priority Values [9, 13]

Base Priority	Process Priority Class	Thread Priority Level
7	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
8	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
9	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
15	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
24	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
25	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
26	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
31	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL

The architecture uses the priority class of a process to differentiate between applications that are time critical and those that have normal or below normal scheduling requirements. The architecture uses thread priority values to differentiate the relative priorities of the tasks of a process. For example, a thread that handles input from the serial channel could have a higher priority level than threads that perform CPU intensive calculations or threads belonging to other application processes.

A thread with a base priority level above 11 interferes with the normal operation of the operating system. Using REALTIME_PRIORITY_CLASS may cause disk caches to not flush, hang the mouse, and so on, but this behavior is still acceptable considering the most important goal that the simulator must behave as close to real time as possible for successful interaction with the embedded processor and representation of the real world hardware. With this approach we provide an architecture for the virtual testbed environment where the simulator is much more responsive to the high degree of interrupt generation and receives more time slices of the CPU to attend to the high frequency of requests coming from the embedded processor. We perform a number of tests on the architecture and observe the behavior of the simulator and test its communication capabilities and its behavior in case of different process priority, thread priority and interrupt trigger levels. The results are described in the next chapter. We also discuss the test environment setup used for performance measurement and analysis.

CHAPTER 8. PERFORMANCE MEASUREMENT AND ANALYSIS

Testing Tools and the Environment

We make use of the following tools in order to analyze and measure the performance of a user mode application such as the simulator, while communicating through the serial channel:

1. A GUI based, user mode application program, running on the host machine. This program is capable of writing to the serial port [1, 8, 15] and reading from it. The application program represents the simulator. The host machine is running on Windows NT platform.

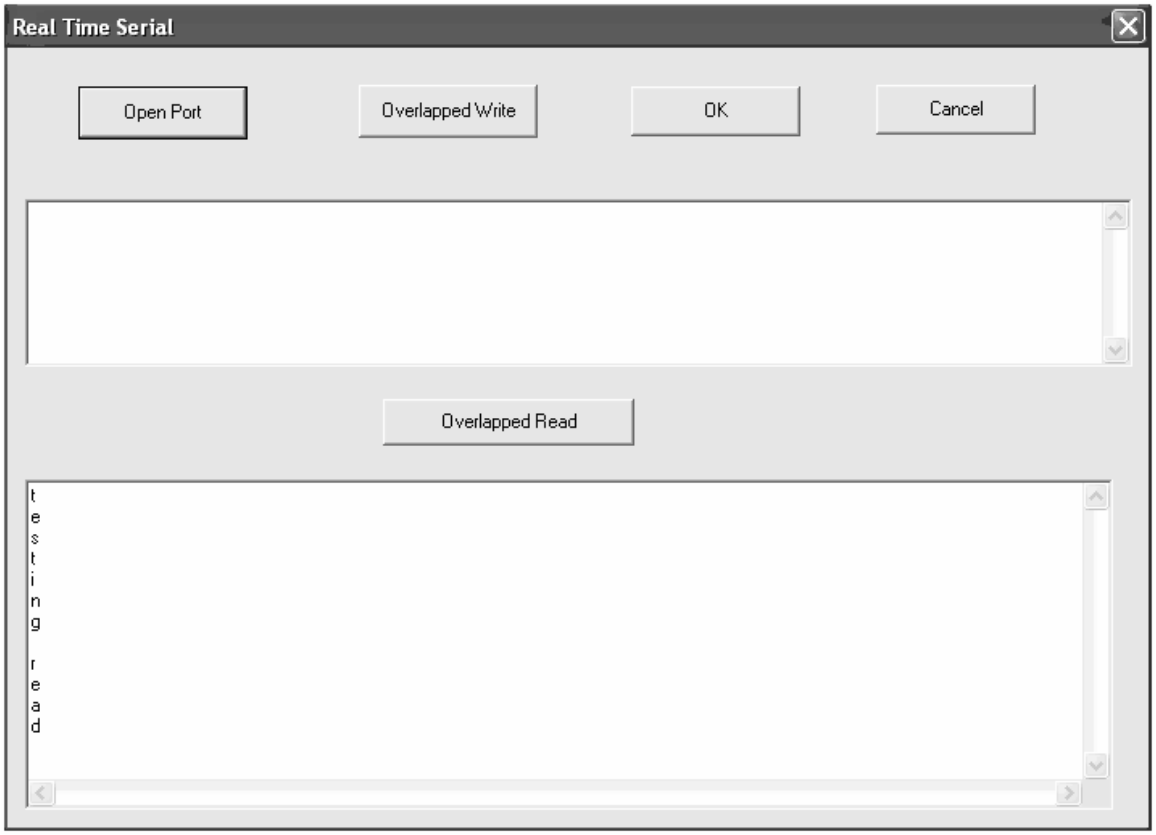

Fig 19. Snapshot of Testing Tool

2. A GUI-based user mode application program, running on a machine connected to the host machine through a null modem cable. This program is also capable of writing to the serial port and reading from it. The application program is a replacement to the embedded processor, purely for analysis purposes. A snapshot of this program user interface is given in Fig 19.

These components help us create an analysis environment where we use each of the two strategies proposed for improving the communication between the simulator and the embedded processor and then measure the performance of the system.

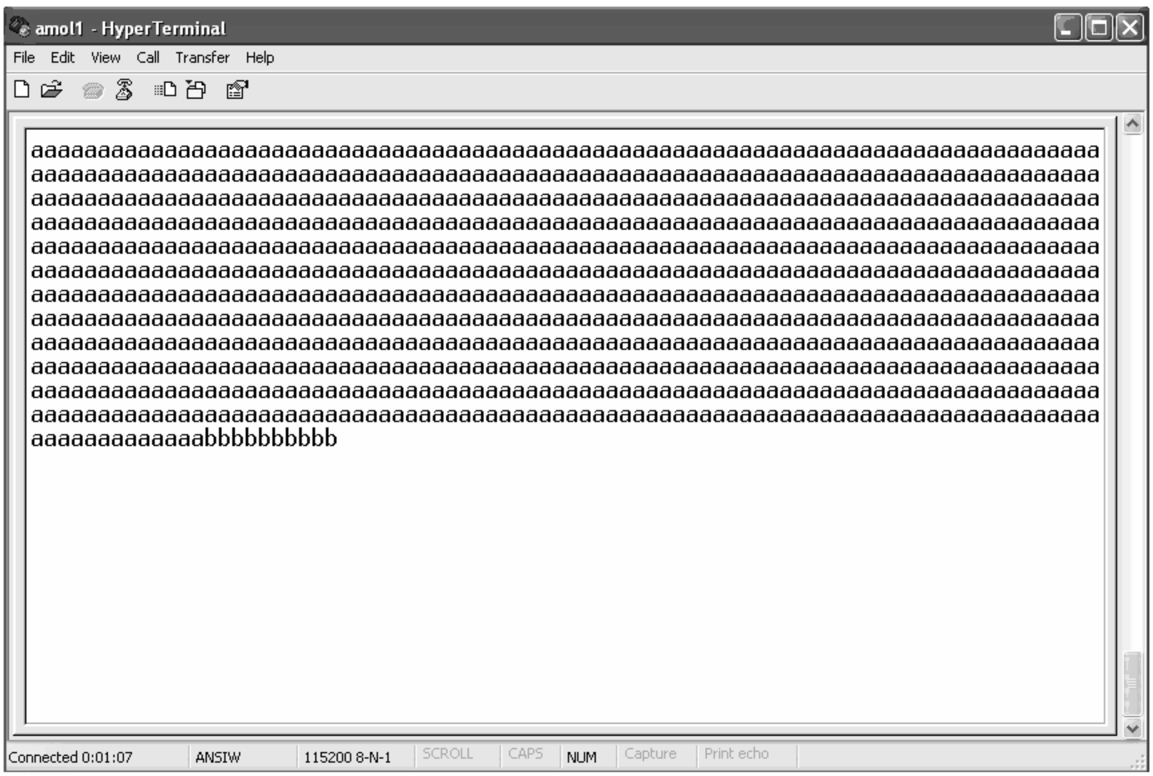

Fig 20. Verification of Data using HyperTerminal

An interesting point to note is that the plain and simple HyperTerminal proves to be a great verification tool for analyzing if there is any loss of data as shown in Fig 20. We capture the transmitted data in a text file using the HyperTerminal and measure the

size of the file to see whether the entire data was transmitted or not. For example, when 4kB of data is transmitted over the serial channel the size of the text file should be 4kB.

Test Cases

We first measure the performance of a process that writes data to the serial channel using an asynchronous WriteFile Function Call. This measurement is taken for different values of data volume ranging from 4kB to 128kB. Other important metrics used are the process and thread priorities.

```
for( i = 0; i<msgCnt; i++ )
for( j = 0; j<1024; j++ )
{
    if (i == msgCnt-1 && j == 990)
                szBuffer = "b";
    status = WriteFile(
                hDevice,
                szBuffer,
                strlen( szBuffer ),
                &dwBytesWritten,
                &overlap );
    if( !status &&
        GetLastError() != ERROR_IO_PENDING )
    {
        dwErrorCode = GetLastError();
        ErrorMessage( "WriteFile", dwErrorCode );
        ExitThread( dwErrorCode );
    }
    status = GetOverlappedResult(
                hDevice,
                &overlap,
                &dwBytesWritten,
                TRUE );
    if( !status )
    {
        dwErrorCode = GetLastError();
        ErrorMessage( "WriteFile", dwErrorCode );
        CloseHandle( overlap.hEvent );
        ExitThread( dwErrorCode );
    }
}
```

Fig 21. Test Code for WriteFile

Fig 21. shows the code of the test for the WriteFile function call used for testing the simulator performance. The measurements are shown in Fig 22-29. and the readings are specified in Table 6-13. Fig 30. compares the measurements for different combinations of priority and buffer size. A character represents 8 bits of data. Hence 1024 iterations of a character written to the serial channel represent 1kB of data transmission. At the other end an application program listens on the serial channel and captures the incoming data. The listener program displays the time when data reception began and generates a beep at the end of data reception and also displays the end time.

The first column in Fig 31. indicates the process and thread priority used for the I/O operation. ‘0’ indicates low priority and ‘3’ indicates high priority. Thus a combination of 00 means low process and low thread priority, whereas a combination of 33 means a high process and high thread priority. The second column represents a FIFO buffer of size 1 byte. The third column represents a FIFO buffer of size 16 bytes. We observe that the larger the buffer gets, the more responsive, efficient and faster the communication gets. For example, the communication speed for a buffer size of 1 byte but with high process and thread priorities is more than that with low process and thread priorities. This is because the CPU is able to attend to the interrupts generated by the serial device faster as a result of high priorities given to simulator process and to the thread listening and writing to the serial device. Similarly, there is an improvement in communication speed when a larger buffer size and high process and high thread priority are used. This is due to the fact that the FIFO buffer is well fed with data due to the larger buffer size. In addition to this, any interrupts generated due to emptying of the buffer are quickly attended by the CPU to refill the buffer with data if available.

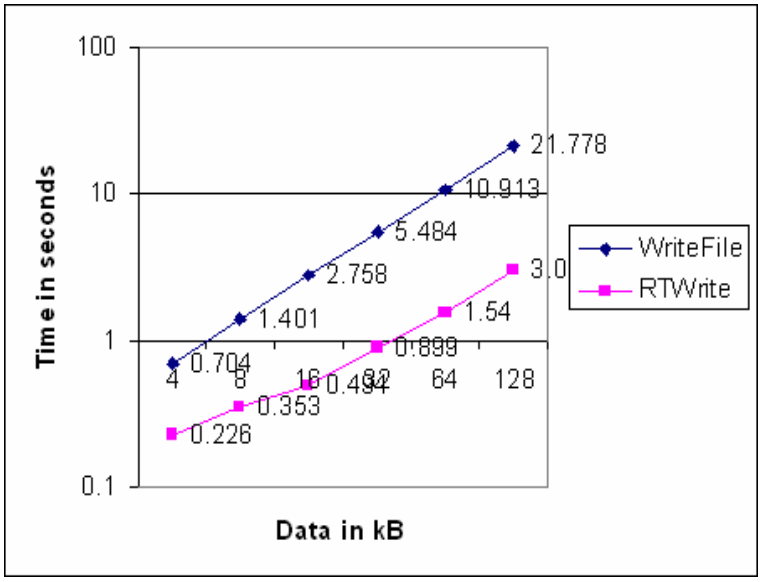

Fig 22. Write using Low Priority and Small Buffer Size

Table 6. Readings for Write using Low Priority and Small Buffer Size

Low Process Priority	Low Thread Priority	Buffer size = 1 byte
Data in KB	WriteFile	RTWrite
4	0.704	0.226
8	1.401	0.353
16	2.758	0.494
32	5.484	0.899
64	10.913	1.54
128	21.778	3.084

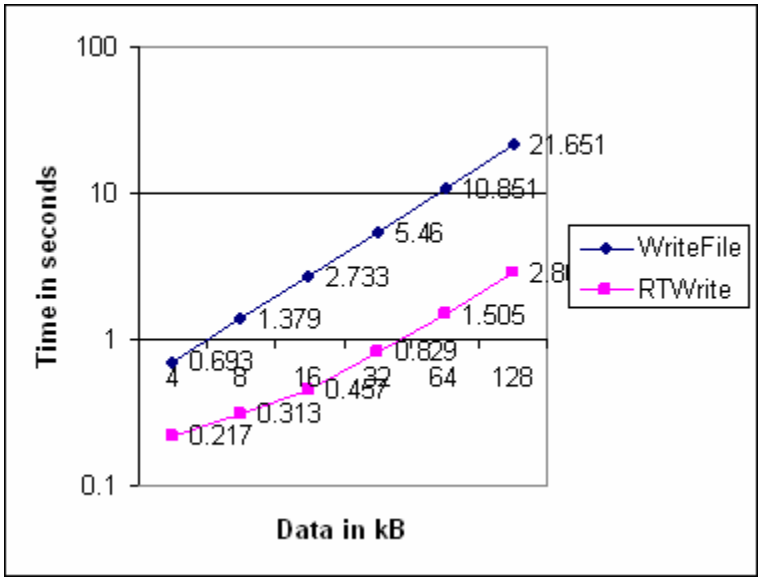

Fig 23. Write using Low Priority and Large Buffer Size

Table 7. Readings for Write using Low Priority and Large Buffer Size

Low Process Priority	Low Thread Priority	Buffer size = 16 byte
Data in KB	WriteFile	RTWrite
4	0.693	0.217
8	1.379	0.313
16	2.733	0.457
32	5.46	0.829
64	10.851	1.505
128	21.651	2.861

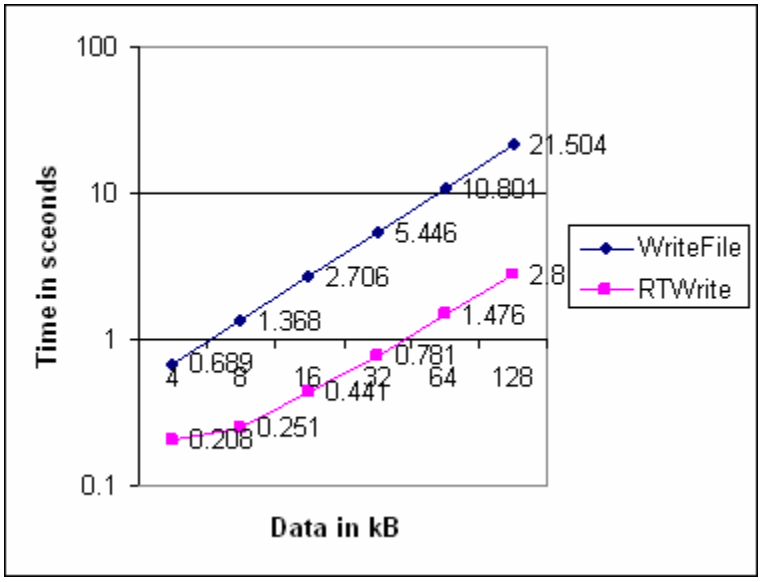

Fig 24. Write using High Priority and Small Buffer Size

Table 8. Readings for Write using High Priority and Small Buffer Size

High Process Priority	High Thread Priority	Buffer size = 1 byte
Data in KB	WriteFile	RTWrite
4	0.689	0.208
8	1.368	0.251
16	2.706	0.441
32	5.446	0.781
64	10.801	1.476
128	21.504	2.819

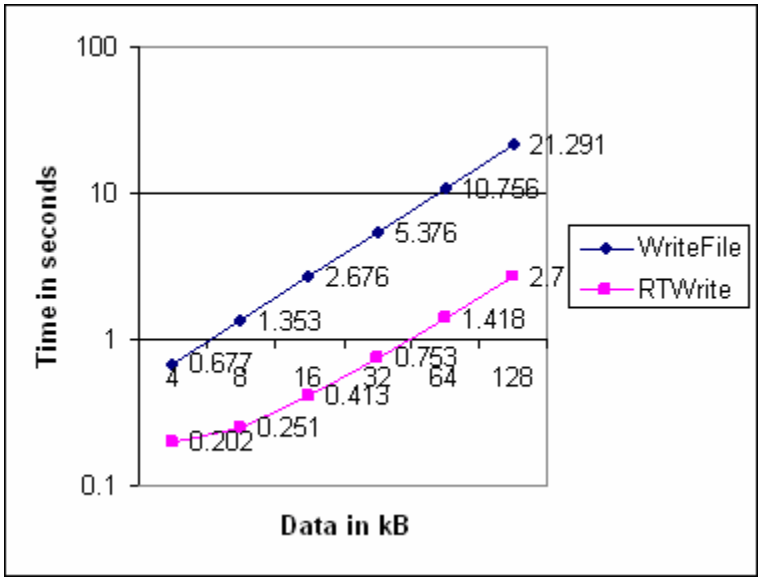

Fig 25. Write using High Priority and Large Buffer Size

Table 9. Readings for Write using High Priority and Large Buffer Size

High Process Priority	High Thread Priority	Buffer size = 16 byte
Data in KB	WriteFile	RTWrite
4	0.677	0.202
8	1.353	0.251
16	2.676	0.413
32	5.376	0.753
64	10.756	1.418
128	21.291	2.715

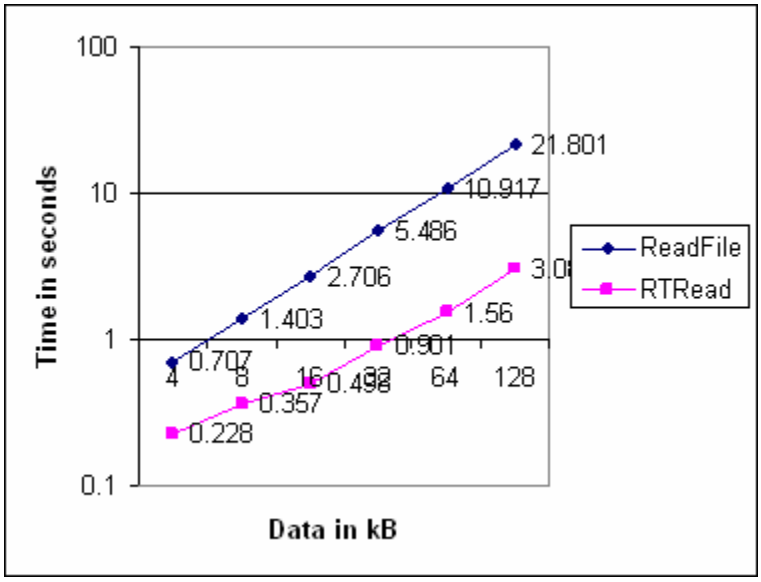

Fig 26. Read using Low Priority and Small Buffer Size

Table 10. Readings for Read Using Low Priority and Small Buffer Size

Low Process Priority	Low Thread Priority	Buffer size = 1 byte
Data in KB	ReadFile	RTRead
4	0.707	0.228
8	1.403	0.357
16	2.706	0.498
32	5.486	0.901
64	10.917	1.56
128	21.801	3.087

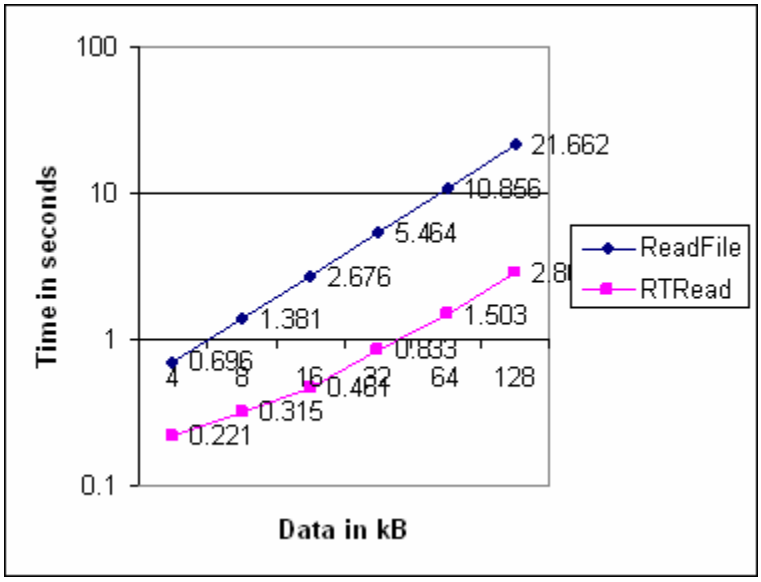

Fig 27. Read using Low Priority and Large Buffer Size

Table 11. Readings for Read using Low Priority and Large Buffer Size

Low Process Priority	Low Thread Priority	Buffer size = 16 byte
Data in KB	ReadFile	RTRead
4	0.696	0.221
8	1.381	0.315
16	2.676	0.461
32	5.464	0.833
64	10.856	1.503
128	21.662	2.863

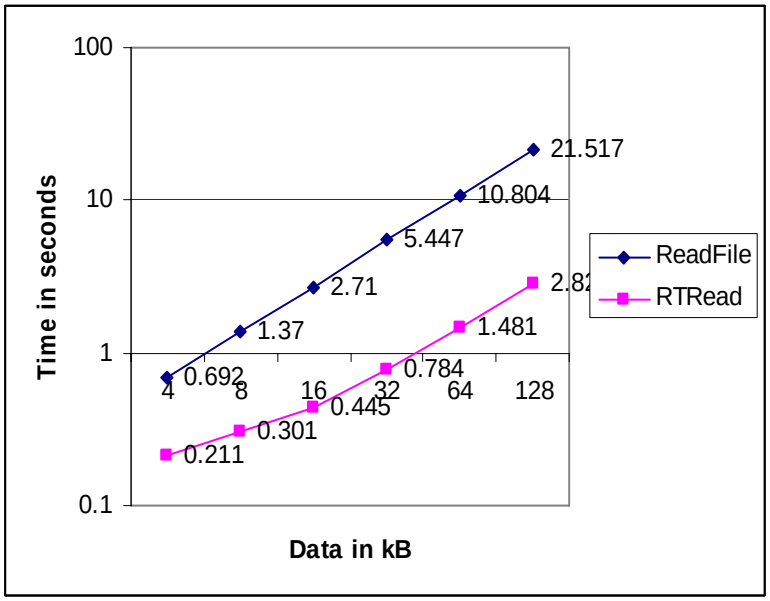

Fig 28. Read using High Priority and Small Buffer Size

Table 12. Readings for Read using High Priority and Small Buffer Size

High Process Priority	High Thread Priority	Buffer size = 1 byte
Data in KB	ReadFile	RTRead
4	0.692	0.211
8	1.37	0.301
16	2.71	0.445
32	5.447	0.784
64	10.804	1.481
128	21.517	2.825

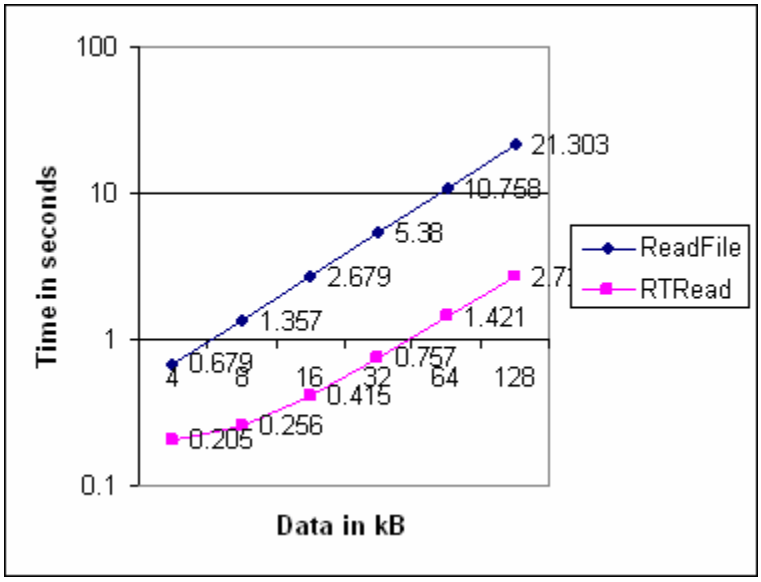

Fig 29. Read using High Priority and Large Buffer size

Table 13. Readings for Read using High Priority and Large Buffer

High Process Priority	High Thread Priority	Buffer size = 16 byte
Data in KB	ReadFile	RTRead
4	0.679	0.205
8	1.357	0.256
16	2.679	0.415
32	5.38	0.757
64	10.758	1.421
128	21.303	2.721

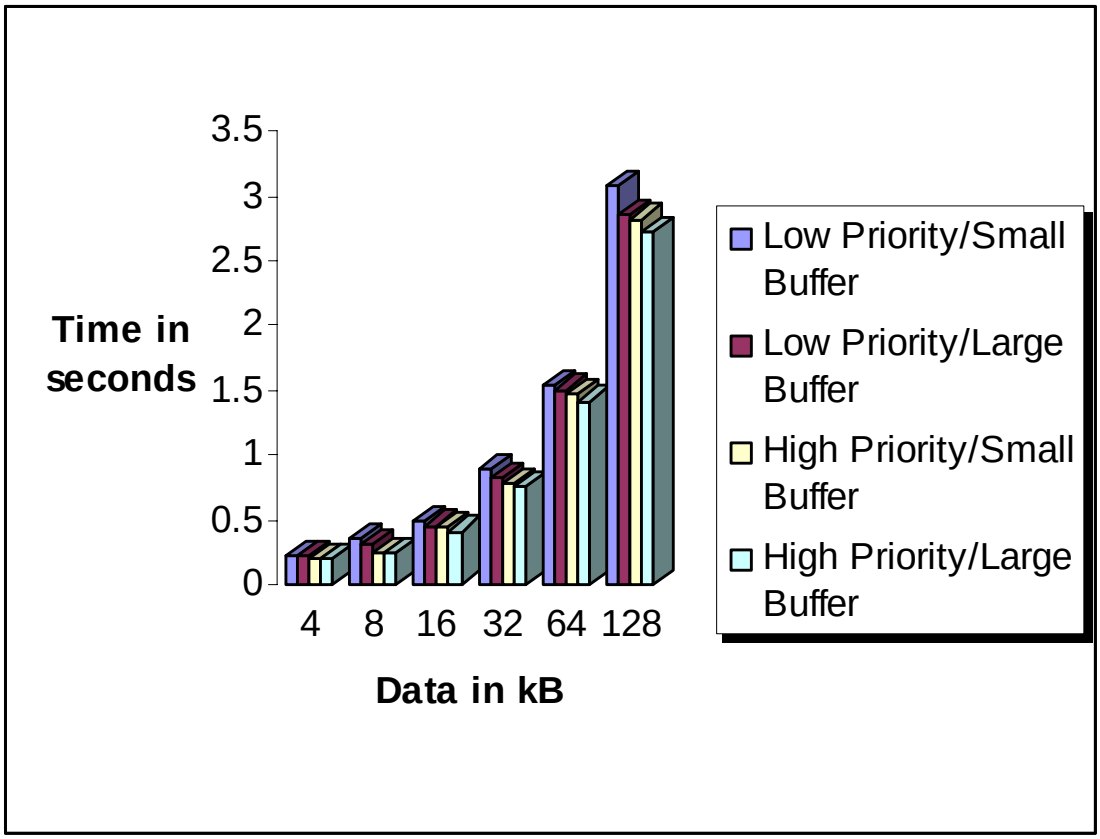

Fig 30. Performance Analysis

Process Thread Priority \ Buffer Size	1	16
00	Normal	Fast
33	Faster	Fastest

Fig 31. Performance Matrix

CHAPTER 9. HEURISTICS FOR ADAPTIVE REAL TIME COMMUNICATION

With the increase in communication capabilities of the simulator arises the problem of operating system support for the dynamic nature of data exchange and faster interrupt generation. We have proposed the use of process and thread priority management, coupled with FIFO buffer size adjustments to control the interrupt trigger levels as a technique to make the operating system more responsive to the high speed communication between the simulator and the embedded processor. With this the problem looks almost solved but for the inability of the virtual testbed to support the diverse nature of embedded systems that would be developed or taught using the testbed environment.

We find applications of embedded systems [7, 16, 18, 21] in a wide range of hardware. Each of these has its own data volume requirement, data exchange rate, communication speed and timing constraints as far as interacting with the embedded processor goes. Since we are simulating the hardware, it is important that the architecture has support for the simulator to be intelligent enough to handle the situation. This is where we propose making the architecture adaptive. Thus all our strategies, methodologies and techniques culminate into what we term an adaptive architecture for real-time communication with embedded devices. We base our heuristics on the readings we obtain from the previous chapter. Using some of the techniques that we discussed for controlling the process and thread priorities, controlling the buffer management and also the underlying kernel mode device driver [2, 17] we try to make the simulator intelligent, adaptive and aware of the changing data volume and data exchange rates.

Thus our heuristic is dependent on the performance measurement and analysis we conducted and gives the architecture an adaptive nature. As our measurements showed, higher process and thread priorities and larger buffer size result in higher throughput (see Fig 30). On the other hand, a smaller buffer size results in less delay until the data is processed and, therefore, in better real-time behavior. This suggests the heuristic to use as small a buffer size as possible and to increase it only if necessary for achieving higher throughput. High process and thread priorities result both in better real-time behavior and in higher throughput, but they might result in the simulator interfering with other applications running on the computer, such as a debugger. We propose for the driver to get information from the user of the simulator on whether other applications need to be considered, and then to set the priorities as high as possible such that they do not interfere with other applications. Fig 32. shows the pseudo-code that represents the heuristic we propose for implementing the adaptive behavior.

1. Repeat steps 2 to 6 indefinitely
2. Store data in user mode buffer
3. Measure data size
4. Measure time taken for data exchange
5. Calculate the data rate
6. Compare the data rate with experimental measurements

   and set buffer size, thread priority and process priority for best results

Fig 32. Pseudo Code for Adaptive Behavior

We develop an interface for handling the data storage and measurement of data sizes which is described in Fig 33. We use the AddData method to store data in the user

mode buffer and GetSize method to measure the data volume. Fig 34. is a code snippet that shows the proposed implementation of the pseudo code for adaptive behavior of the architecture.

```
//---- public interface --
void AddData( char ch ) ;
void AddData( unsigned char ch ) ;
void AddData( std::string& szData ) ;
void AddData( std::string& szData,int iLen ) ;
void AddData( char *strData,int iLen ) ;
std::string GetData() {return m_szInternalBuffer;}

void        Flush();
long        Read_N( std::string &szData,long alCount,HANDLE& hEventToReset);
bool        Read_Upto( std::string &szData,
                       char chTerm,
                       long  &alBytesRead,
                       HANDLE& hEventToReset);
bool        Read_Available( std::string &szData,HANDLE & hEventToReset);
inline long GetSize() {return m_szInternalBuffer.size();}
inline bool IsEmpty() {return m_szInternalBuffer.size() == 0; }
```

Fig 33. Interface to handle User Mode Buffer

The AdaptToEnvironment Function is the one which actually implements the logic for adaptive behavior as shown in Fig 35. With this approach we propose to give the architecture an adaptive behavior in an attempt to overcome the problem of supporting a wide range of embedded systems applications and make the virtual testbed a potential environment for concurrent development of embedded systems and also as a cost effective teaching tool. We propose to implement this architecture to host a simulator for two different kinds of embedded systems applications. The first is a traffic light control system [10] and the other is an answering machine environment [10]. Both of these systems have very different data volume and data exchange rate requirements. The traffic light control system transmits small amounts of data but requires a high data exchange rate for reading the road sensors.

```
int varSize = 0;
int varTime = 0;
count = 1;
SYSTEMTIME st,ed;
GetSystemTime(&st);
do
{
    ResetEvent( ovRead.hEvent  );
    unsigned char szTmp[1];
    int iSize  = sizeof ( szTmp );
    memset(szTmp,0,iSize);

    OpenRTSerial();
    abRet = asyncRTRead(0x3F8, &szTmp[0], &ovRead, &dwBytesRead);
    CloseRTSerial();

    if (!abRet )
    {
        abContinue = FALSE;
        break;
    }

    if ( dwBytesRead > 0 )
    {

        myStr += szTmp[0];
        usrBuff.AddData(szTmp[0]);
        iAccum += dwBytesRead;
        apThis->m_theSerialBuffer.Flush();
    }

count++;
}while (count <= 5)

GetSystemTime(&ed);

AdaptToEnvironment(sd,ed, usrBuff.GetSize());
```

Fig 34. Call to the AdaptToEnvironment Function

The answering machine system requires a very large amount of data volume to be transmitted to the evaluation board that hosts the embedded processor. In both of these systems the hardware is simulated and would be implemented using the adaptive architecture as the driving concept. The virtual testbed [6, 10] is designed to support the

diverse nature of the two embedded systems and we propose that the architecture is robust and intelligent enough to adapt to the different system requirements.

```
if (varTime <= 0.00003 && varSize == 1) {
    SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS);
    SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);
    SetRxBufferSize(14);
}
else
    if (varTime <= 0.00003 && varSize == 4 ) {
          •
          •
          •
```

Fig 35. AdaptToEnvironment

For measuring the data rate, there are two possible approaches, either measuring the data volume in a fixed time interval or averaging the time between successive bytes of data. In the former approach, we set a timer with a fixed time period. After we measure the data that was transferred or received in the given time period we calculate the data rate. Now we find the maximum data rate and the minimum data rate registered so far and determine the buffer size, thread and process priority that would be the most suitable for the best results. Another approach is to keep the amount of data being measured constant and measure the amount of time it takes to send or receive it. The measured time is used to calculate the data rate each time and then all the data rates registered so far are analyzed to determine the buffer, thread and process priority settings, for best results by choosing the optimum data rate. This is particularly useful for those type of simulator implementations where the data rate varies. A typical example would be a simulation of video surveillance system where the data is streamed at different data rates depending on the motion of the object under surveillance.

CHAPTER 10. CONCLUSION

We have successfully designed the adaptive architecture for real time communication with embedded devices. The virtual testbed can be based on this adaptive architecture to make it more efficient, robust, cost effective and fast and a reliable environment for the design, development, testing and even for the purpose of teaching embedded system programming. This architecture shall prove very useful in many embedded systems environments for the purpose of design and development as well as testing.

We have made a significant leap in making the communication between the simulator and the evaluation board close to real time. Making the architecture adaptive furthers our intention to make the virtual testbed environment more reliable and a popular design and development platform in conjunction with the commonly available Windows XP operating systems. We hope use of this architecture would make teaching embedded systems an enjoyable experience with no concerns about damage to the hardware or credibility of the system performance.

We have proposed various approaches for the adaptive algorithm in determining the optimum values for buffer size, thread priority and process priority that would help the virtual testbed give the best results in as many different scenarios as possible. The accuracy of the threshold that the algorithm would set for the testbed depends on more and more precise measurements of data rates and the corresponding buffer size, thread priority and process priority values. This in turn depends on how stable and efficient the device drivers on the DSP board are and also how robust the simulator implementation is. Also with availability of more and more simulators we shall be able to experiment with

different test case scenarios and derive more accurate values for the thresholds of buffer size, thread priority and process priority. For the purpose of future development we discuss the following two areas in which the architecture can be broadened.

1) The use of universal serial bus devices is increasing by the day. USB is finding wide applications in interfacing the hardware device. The capability of USB in communicating serially is many times more than the standard serial communication ports. Embedded processors hosted on evaluation boards are capable of very high data transmission rates. Providing a support for USB communication in the proposed architecture would be a prospective area for research.
2) Another possibility for research is to make an improvement in the heuristics behind the adaptive architecture. There is also scope for fine tuning the heuristics so that the device driver parameters are set precisely depending on the system requirement. The benchmark used against which the optimum data rate is compared should be as precise as possible so that the corresponding buffer size, thread priority and the process priority values are as accurate as possible.

REFERENCES

[1] Axelson, J., *Serial Port Complete,* Madison: Lakeview Research, 1998.

[2] Baker, A., *The Windows Nt Device Driver Book: A Guide For Programmers,* New Jersey: Prentice Hall, 1997.

[3] Berger, A., *Embedded Systems Design: An Introduction to Processes, Tools and Techniques,* California: CMP Books, 2001.

[4] Cottet, F., Delacroix, J., Kaiser, C., Mammeri, Z., *Scheduling in Real-Time Systems* New Jersey: John Wiley & Sons, 2002.

[5] Dabak, P., Phadke, S., Borate, M., *Undocumented Windows NT,* New Jersey: John Wiley & Sons, 1999.

[6] Keyhani, A., Marwali, M. N., Higuera, L. E. , Athalye, G., Baumgartner, G., "An Integrated Virtual Learning System for the Development of Motor Drive Systems," *IEEE Transactions on Power Systems*, Vol. 17, No. 1, February 2002, pp. 1–6.

[7] Li, Q., Yao, C., *Real-Time Concepts for Embedded Systems,* California: CMP Books, 2003.

[8] Linux Documentation Project, "Serial HOWTO," http://www.tldp.org/HOWTO/Serial-HOWTO-1.html

[9] Liu, C. L., Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, Vol. 20 No. 1, January 1973.

[10] Liu, J., *A Virtual Testbed for Embedded Systems Development and Instruction*, Dept. of Computer and Information Science, The Ohio State University, 2004.

[11] Mobley, S. B., Gareri, J. P., "Hardware-in-the-loop simulation (hwil) facility for development, test, and evaluation of multispectral missile systems: update," *Technologies for Synthetic Environments: Hardware-in-the-Loop Testing V*, 4027:11-21, 2000.

[12] MSDN, "Process and Thread Functions," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/setpriorityclass.asp

[13] MSDN, "Scheduling," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/scheduling_priorities.asp

[14] MSDN, “Serial Communication in Win32,” http://msdn.microsoft.com/library/default.asp?url=/library/en-s/dnfiles/html/msdn_serial.asp

[15] Nelson, M., *Serial Communications Developer's Guide*, New Jersey: John Wiley & Sons, 2000.

[16] Noergaard, T., *Embedded Systems Architecture : A Comprehensive Guide for Engineers and Programmers*, Oxford, England: Newnes, 2005.

[17] Oney, W., *Programming the Microsoft Windows Driver Mode*, Redmond: Microsoft Press, 2002.

[18] Randall, J., *Specification And Design Methodology For Real-Time Embedded Systems*, Boston : Kluwer Academic Publishers, 2002.

[19] Soloman, D., *Inside Windows NT*, Redmond: Microsoft Press, 1998.

[20] Song, F., Folleco, A., An, E., “High fidelity hardware-in-the-loop simulation development for an autonomous underwater vehicle,” In *Oceans 2001. An Ocean Odyssey. Conference Proceedings*, pages 444-9 Vol.1. Ocean Eng. Dept., Florida Atlantic Univ., Dania, FL, USA, 2001.

[21] Wilson, G., *Embedded Systems and Computer Architecture*, Oxford, England: Newnes, 2001.

VITA

Amol S Patwardhan has done his bachelor's degree at University of Mumbai, India, in the field of information technology. Currently he is pursuing his master's degree in the field of system science, under the Computer Science Department, at Louisiana State University, Baton Rouge, United States of America.