Foundations of the Virtual Orchestra

IQP By: **Ian Pushee**

Advisor: **William R. Michalson**

Date: **A Term (October) 2000**

Table of Contents

1 Introduction

The Virtual Orchestra Project is aimed as developing a computerized alternative to a live orchestra for use in theatrical performances. Conventional (live) orchestras are often too expensive for small theatrical troupes to afford. These small performances are either forced to make due with less than a full orchestra, or to use pre-recorded music. The Virtual Orchestra will add a third option, to hire a live performance with the range of a full orchestra that is much more portable, and much less expensive.

This system needs to be simple enough to be operated by only a few people (ideally, only a single 'conductor'), small enough to be extremely portable, and yet powerful enough to produce a viable replication of a full orchestra. From the technological standpoint, the Virtual Orchestra needs to be versatile enough to produce the full range of a concert orchestra and be responsive enough to operate in real time.

2 Background

2.1 Virtual Orchestras

While this project is aimed at producing a specific machine termed the 'Virtual Orchestra', the term has been applied to other systems with a similar purpose and function. The lowest level product termed a *virtual orchestra* is a system that plays back a prerecorded or preproduced musical score. Such a system is not interactive, and must be completely redone for each production. Another level up is a completely modular system, one that can play any score of music using a standard method (like MIDI). The up side of this solution is that it takes little prep time to get the system ready for each production. This system also allows for some run-time alterations to the score, such as changes to the tempo, and other adjustments normally affected by the conductor. The down side is that there is no way to customize the performance for each production, no real way to add much of a human touch. The most complex version of a *virtual orchestra* combines the best features of these two systems. It is modular, allowing for a similar system to be used and reused in many different productions. This system also allows greater customization of the sounds used to create the music, as well as interactive control or tempo and others.

2.1.1 Examples of Virtual Orchestras

Recently, many different companies have designed and begun to use virtual orchestras in performances. Each system has a different design, different purpose, and is used in a different venue. Operaworks is a New York based company founded in 1983 by David Leighton. This small opera troupe does not gross enough to pay for a full orchestra, but this is ok because it brings its own orchestra to all performances. Operaworks' orchestra is located not in the pit, but on the back balcony of the hall. Patrick Casey handles a virtual orchestra from there, as Leighton conducts the singers from the traditional conductor's position up front. Mr. Casey also follows the conductor's signals, but his control of the orchestra is limited to small adjustments such as tempo changes. All of the actual music was programmed into the computer long before the performance, and is taken from CD samples. Each minute of

performed music takes between three and four hours to enter into the computer. The system has a very large setup cost (a $4000 synthesizer, thirty $300-400 CDs, mixer, computer, speakers, etc.), but now Operaworks is free of the high per-performance costs of a real orchestra. Operaworks doesn't make a lot of money, but they put on 8-12 productions per year, and can sell tickets at relatively low prices. A recent production of Strauss's "Ariadne auf Naxos" cost only $18 for adults[1].

Pocket Coach Publications claims that their "... Virtual Orchestra replaces piano-accompanied operatic performances and provides a warm and enveloping orchestral sound." This company rents out their orchestra to small theaters with limited budgets, who do not have the money, nor the space for the full orchestra called for by many operas. In California Michael Fissolo and Dietrich Erbelding have been working on and perfecting their system. This system has been used in many productions so far (portions of Die Walküre, all of Die Fledermaus in Fairfield, California and Hänsel und Gretel with Golden Gate Opera), and its creators have high expectations for the future[2].

Finally, the virtual orchestra system most related to this project is that created by Frederick Bianchi and David Smith. These two professors designed and run a virtual orchestra system that has well over 700 productions under its belt. Bianchi first applied this technology in 1989 at a production of the University of Cincinnati[3], and the system has evolved a very long way from there. Bianchi & Smith's virtual orchestra has gone through many revisions, and is still evolving. The current incarnation of the virtual orchestra is made up of "23 flat-panel NXT loudspeakers laid out roughly in the pattern of an orchestra."[4] Each speaker produces the sound of a single instrument, and the conductor/technician can control the tempo, as well as inserting pauses, on the fly. This project is intended to build on Bianchi & Smith's pioneering work.

---

[1] CLASSICAL MUSIC AND DANCE GUIDE; The New York Times; June 18, 1999, Friday, Late Edition

[2] About the Virtual Orchestra; http://www.pocketcoach.com/vrtlorch.htm

[3] The Money pit; Opera News; March 2, 1996

[4] Musicians beware the Virtual Orchestra; PAUL HORSLEY : THE KANSAS CITY STAR; July 30, 2000, Sunday METROPOLITAN EDITION

The first major opera orchestrated by Bianchi & Smith was the Kentucky Opera's 1995 performance of Hansel and Gretel. The Kentucky Opera was in a bind for this performance, because they were unable to hire a live orchestra due to both monetary constraints and availability. Bianchi & Smith placed their hodge-podge of computers, monitors, samplers, amplifiers, large speakers, and a mixing console all into the pit of the Palace Theatre, and produced the musical accompaniment for the opera for about 1/10th the cost of a live orchestra[3]. The year after, they helped produce 4-5 other performances. Bianchi & Smith's first national tour was with Annie, which would normally have toured with 9-13 musicians. Another use for the virtual orchestra technology is being tested by scientists at the Helsinki University of Technology. Conductors in conductor training classes at the Sibelius Academy are directing computer-generated musicians in the same manor that they would a live orchestra. The 'audience' of these performances can listen to the music as if they were in any of the seats of a concert hall, in order to better judge the conductor's skill. This allows not only additional control over the practice environment, but also saves the school the cost of a rehearsal orchestra, and allows them to work for as long as necessary[5].

2.1.2 Virtual Problems

Of course, no technology is without its demons, and the virtual orchestra has a few rather persistent ones. The first is a purely technical demon, that the music produced by the virtual orchestra just plain doesn't sound all that good. This problem is being overcome as the technology evolves, but it does serve as a base for other problems - first impressions are often the most important. Of Bianchi & Smith's debut in the Kentucky Opera, one reviewer had this to say, "If I had heard this performance over the radio without explanation, I would have had difficulty telling what instruments were being used... generally the sound was alternated between an amorphous haze and a wheezing carousel colliding with a skating-rink mechanical organ."[6] A recent (1996) staging of Salome received this review, "It sounds like an antique CD, perhaps like an oddly appealing cross between an accordion and a calliope. The fake

---

[5] Virtual orchestras are to help train the next generation of conductors; Chris Price : Sunday Times (London); May 23, 1999

strings tend to emerge thin and tinny. The woods (nasal) and brass (brassy) come closer to reality."[7] Is this decrease in quality worth the cost savings of using a virtual orchestra over a live orchestra? It depends on who you are, maybe it is for the new generation -- raised on television and digital music. Of course, when considering the small theatrical troupes that seem more likely to use the virtual orchestra a better comparison may be between a full virtual orchestra, and mere piano accompaniment -- which makes the question much easier to answer.

Another demon haunting the virtual orchestra is that of artistic content. Even beyond the pure question of which type of orchestra sounds better, many will argue that a live orchestra has a dynamic, 'human' quality that is lacking in the synthesized version. In addition, a live orchestra mixes the efforts of the individual musicians. A virtual orchestra will at most show the human touch of a single musician (the one controlling the system), and even that is limited by the amount of on-the-fly control the musician has over the music produced. "Until a thinking computer can replace a violinist's particular love for a piece of music, the good or poor day the brass section has had, or any of a hundred personal nuances that make a performance unique the synthesizer has no key place in the orchestra."[8] says one live orchestra supporter.

The final demon plaguing this technology is that of jobs. What does the virtual orchestra do, but take jobs away from live musicians. One side of the issue says that only small productions who couldn't hire an orchestra anyway are the only one who would rent a virtual orchestra. Opponents would argue that if the technology improves enough to combat the other problems, and proves financially viable -- even larger scale companies may begin to use it instead of live musicians. "The virtual orchestra must seem an unlikely threat to the livelihood of classical musicians. But that's what pop musicians once thought. In the past orchestras were used regularly for everything from soaring string tracks on Motown songs to disco. Nowadays if the sound of an orchestra is desired in pop, it is as likely provided by digital keyboards and

---

[6] The Money Pit; Charles H. Parsons : Opera News; March 2, 1996

[7] Virtual music hits off Off Broadway; Martin Bernheimer : Financial Times (London); June 23, 1999, Wednesday USA EDITION

computers."[9] Paul Horsley of the Kansas City Star (July 30, 2000, Sunday METROPOLITAN EDITION) rebuts this best when he says, "On the other hand, the player piano hasn't really put any pianists out of work."

2.2 MIDI Specification

MIDI seems to be the answer to the versatility problem of a virtual orchestra. MIDI stands for Musical Instrument Digital Interface and is a standard which allows many different types of musical devices to talk with one another. It is a common system that allows even a normal home computer to communicate with even the most advanced music synthesizers. The MIDI specification covers both the hardware and the language used for this communication.

The hardware aspects of MIDI is mainly in how data is transported between devices. MIDI transports data through a serial cable, at a rate of approximately 32 Kbaud/sec (a relatively slow connection). MIDI devices have two interfaces, one specified as 'MIDI OUT' and the other 'MIDI IN'. The MIDI cable connects the output on one device to the input of another device. Generally this means that an output device, like a keyboard or other MIDI-equipped instrument is connected directly to an input device, such as a computer or synthesizer which will actually play the music being produced. The MIDI standard does not specify a chain, a device does not automatically pass on any data it receives, like in SCSI or USB. Some devices however, have a 'MIDI THRU' port, this port acts as another output interface, except that it sends data directly as it comes in on the input interface. The 'MIDI THRU' interface is not required by the standard, but it allows for chaining of MIDI devices.

The language of the MIDI specification is simple, and makes a great effort at using as little data as possible for the maximum result, in deference to the limited bandwidth of the hardware. Each 'word' of the MIDI language is composed of status byte, followed by one to two data bytes. The status byte tells the receiver what type of message will follow, and the data bytes convey the message. There are a few exceptions, non-standard words if you will. Some

---

[8] Letter by Carolyn Ahrens, Union City, NJ; Opera News; May 1996

[9] The Singer Is Real, but the Orchestra Cannot Hear Her; ANTHONY TOMMASINI : The New York Times; June 16, 1999, Wednesday, Late Edition

system messages contain only a status byte, in these cases the type of the message is all the device needs to know in order to take appropriate action. The other exception is known as 'running status'. In order to preserve precious bandwidth, some devices will only send a status byte for the first of a long string of messages with the same status. Status bytes are distinguishable from data bytes in that their Most Significant Bit is always 1. Data bytes have an MSB of 0, which means that they are effectively only 7 bytes long (in terms of how much data they can pass.

The two types of system messages which only require a status byte are real-time and system exclusive messages. Real-time messages are used to convey time critical messages, and may be sent at any time, even in the middle of another message. System Exclusive messages are the jack-of-all-trades of the MIDI world. They consist of a status byte to mark them, followed by any number of data bytes. The message ends when any status byte is sent, but the EOX byte marks the end of a specific message (ie: the exclusive message can be interrupted by normal messages, but the message continues until the EOX byte is sent). Exclusive messages are used to carry any information which does not specifically fit the MIDI standard, and are defined individually by the manufacturers of each MIDI device.

MIDI devices can mimic the workings of the orchestra by use of channels and voices. Voice is the term used for the algorithm which converts a note message into an audible sound. Essentially, different voices represent different instruments in an orchestra. There are a number of voices predefined by the MIDI standard, as well as any number specific to a given output device. Notes can be sent to individual voices through channels, which are separate streams of music. Each channel is assigned a voice through which it plays it's notes.

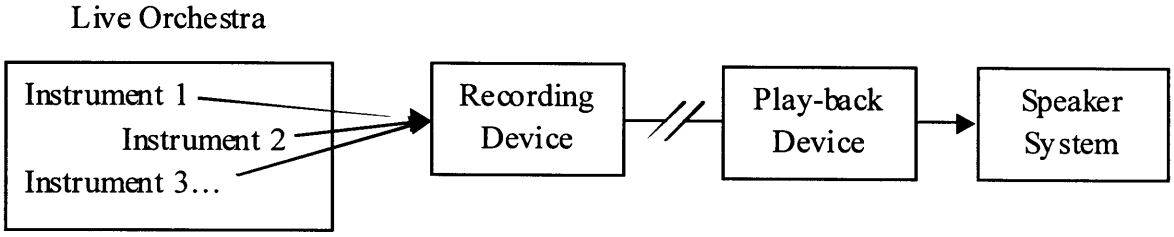
3 The Virtual Orchestra Project

The project in general is to produce a small, powerful, and versatile Virtual Orchestra System. There are many design decisions and question for even the over-all layout of a Virtual Orchestra. Some of the system designs which are called *virtual orchestras* have been discussed earlier, but now we will take a closer look at how they actually work... and why ours will work better. Problems which crop up in some designs are: lack of interaction, level of possible complexity, and depth of music. We will look at these concerns in reference to the two basic systems of *virtual orchestras*: pre-recorded and synthesizer. Finally we have our actual system to consider, a combination of the best features of the first two types.

3.1 The Systems

There are two distinct systems which are the basis for any *virtual orchestra*. It would be possible to term either of these systems on their own as a *virtual orchestra* in their own right, but most *virtual orchestra* are more of a combination of these two systems. Some systems lean more heavily in one direction, other more heavily towards the other... each system trying to strike the correct balance.

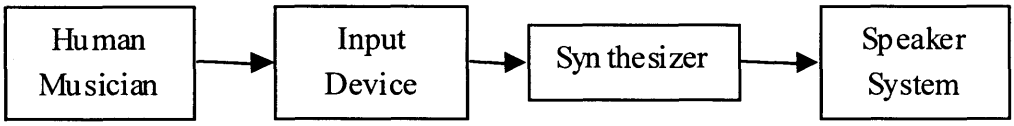
3.1.1 Pre-recordings



This system is based on recording before-hand the music to be played. Once inside the production, the music can then be played back, with little to no input from anyone With a system of this type you can capture all of the nuances of a live performance, and bring them into a virtual production, exactly the same every time. How the music is recorded, and how it

is played back change from system to system, and are really rather trivial to the overall worth of the system idea.

3.1.2 Synthesizers



A synthesizer is a device for synthesizing music. That is, it creates music which sounds like it comes from other instruments. The most commonly seen and recognized synthesizers are electronic keyboards which can 'play' the notes typed as a whole variety of instruments. A synthesizer however, could also be a little black box which takes in MIDI messages, and sends sounds out through a separate speaker. The synthesizer is the part which converts input into specific sounds... and does not need to include the ability to actually produce either input or sound. A synthesizer system, when we are talking about the base systems of a *virtual orchestra*, is a system which interprets real-time inputs from a musician in order to output the preformed music. This system could be as simple as a keyboard, or complex enough to exactly mimic hordes of instruments at once, these details are not important to the nature of the base system. What is important about the synthesizer system is that it uses real-time input to create a real-time performance, and that the arbitrary input is used to produce arbitrary sounds.

3.2 Interaction

Interaction is important to a *virtual orchestra* because it is important to a live orchestra. In a live orchestra, the musicians don't just play a piece straight through, starting at the first note and keeping the same tempo the whole time. They need to be able to adapt to and follow what is going on in the rest of the production. If a singer sings more slowly than is expected, the musicians need to slow their tempo. If there is a snag, and the production needs to stall -- the musicians need to be able to repeat or stretch the piece as necessary... and make it seem as natural as possible. For this reason, live performances have a conductor who can guide the

musicians. The conductor tells them to slow down, or speed up, repeat, or leave out a section... he is the one who makes sure the orchestra keep pace with the rest of the production.

A pre-recorded system has minimal interaction. It can start and stop arbitrarily, but not elegantly. If you stop a recording in the middle, it is generally rather obvious, and slowing down or speeding up a recording gets worse and worse the more you change it. Generally speaking, if you go the pre-recorded route, you loose out on all of the advantages of interaction. Your production will have to follow the music, rather than the other way around. A synthesizer on the other hand, is all about interaction. When you are in control of the input on a synthesizer, you have almost complete control over everything which comes out of the speakers.

3.3 Complexity

The ability to play complex musical pieces is one of the main reasons that a high-end production uses an orchestra, instead of a single pianist or even a one-man-band. There is just something more powerful about a piece of music played one way on some instruments, with base undertones from others, and all being led by yet another section. The complex interactions between the parts of a piece are the very reason that people listen to an orchestra in the first place, and not including this level of complexity into a *virtual orchestra* would be missing the point entirely. You'd be bringing the lone pianist to a Broadway musical, and that just wouldn't fly.

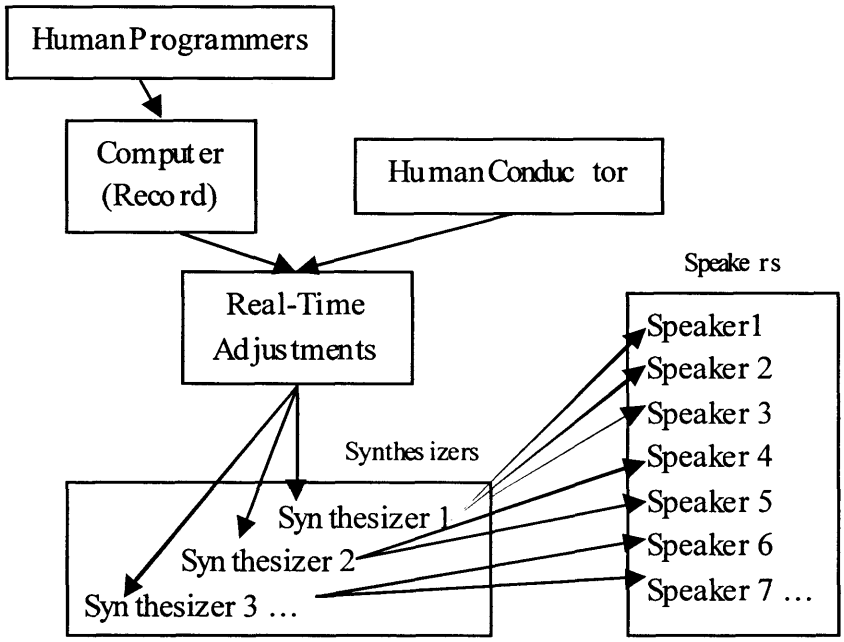
This issue is where pre-recorded music has its largest strength. Because you are recording an actual performance, you don't loose out on any of the complexity brought to the piece by the full, live orchestra. A synthesizer system though, really is brining a one-man-band into a concert hall. No matter how talented, any single performer acting in real-time, can not create the levels of complexity embodied in a full orchestra of 25+ musicians. Here is where the synthesizer shows its limits: no matter how many instruments can be synthesized, you still only have a single musician controlling them, and there is a hard limit to how many things he can do at once.

3.4 Depth

This issue is related to complexity, but it is something which is factored into our system, so it seems that it should be covered. By depth of music, I mean how intense, deep or real it sounds. This is related to complexity in that the more complex a piece, the more depth needs to be provided for it to sound real. A single piano could be replicated on a single (good) speaker, but as you add more instruments (even playing the exact same thing), a single source of audio feels less and less realistic. Also, a single audio source becomes less and less able to support the range of sound needed. A piece very quickly starts to sound flat as the separate sounds are all combined and mixed to come out of a single point source.

Neither system on its own really fills the bill in this area. A recording might have the complexity to make use of a deep system, but generally recordings are flat in and of themselves, and so there is not way to separate more than the very basic sounds from each other (ie: base form everything else). A synthesizer on the other hand, with its massive modular capabilities can easily send separate sounds to separate outputs... but since it lacks the complexity of a pre-recorded piece, you can't really make good use of this capability.

3.5 Our System



The system the project is helping to create is a blend of the two basic systems. This project will use synthesizers to convert inputs from human and computer into the music of the performance. All of the music played by the system could be considered pre-recorded. An orchestra wasn't recorded performing the score, but the notes and nuances of the piece would be programmed and saved on the computer prior to the performance.

This system handles the interaction question using its synthesizer background. Because the music is being synthesized in real-time, we can easily add some levels of interaction for the operator (conductor) to utilize. It is, for example, a very easy matter to create an interface for speeding up or slowing down the tempo of a piece (a simple fly-wheel would serve for that). Similarly, we could allow the conductor, with minimal prior thought, to begin to fade out the music early, or repeat the last strain again... as the situation calls for.

For complexity, we turn to the other end of the *virtual orchestra* scale, and take advantage of the pre-recorded side of the system. Since the musical score is entered into the system prior to the performance, all it takes is extra time to program in levels of complexity that even a live orchestra would have trouble with. You could have each instrument of the orchestra playing different pieces for all the system cares, because each instrument can be told beforehand to play whatever notes you want. The more time and effort are put into the preparation, the more complex the result can be. This compared with a fully real-time synthesizer system, where you are limited to what one person can accomplish in real-time.

Depth is the final criteria we will be looking at. Neither basic system alone could produce the level of depth that a live orchestra is capable of. Our system however, does not have this problem. By crossing the complexity allowed by pre-recording with the instrument separation afforded by synthesizing, we end up with a system that can rival the depth of a live orchestra. The system specifies that the speaker array be set up to mimic the placement of an actual orchestra. This gives each instrument its own separate speaker... just like in a live orchestra.

3.6 My Part

So now we have the layout and concept of the Virtual Orchestra we are designing, next we discuss how it is implemented. The base of the system is a custom-designed computer running Real-Time Linux. This central system will interface with the rest of the system (input

devices, synthesizers, etc.) using a custom MIDI interface card. MIDI is a versatile standard for communication between musical devices, and so was an obvious choice for the communication inside the Virtual Orchestra. The problem is that MIDI development tools are lacking for the Linux operating system. We felt that it would be best to allow development of the applications to be used on the Virtual Orchestra to be done on other operating systems, namely Microsoft Windows.

The problem that arises is in porting these Windows developed applications onto the Linux box used for the system. Maximum MIDI defines a system and libraries for easy development of Windows MIDI applications, so this system has been chosen as the development environment of choice for the Virtual Orchestra. MaxMIDI however, relies on a number of high-level functions in Windows which are not implemented in Linux. This mean that many things which the Windows system does for the programmer, Linux requires programmers to deal with themselves. The main feature relevant to this project which Linux does not define are: multi-media (MIDI in/out, control, timing, etc.), message passing (how Windows communicates among its processes), and memory management. Since these features are needed to easily port MaxMIDI applications, my job was to implement them on the RTLinux platform.

4 Multi-Media System

4.1 Timing

The first thing to deal with in a multi-media system is timing. We need to make sure that sounds are played at the correct time, and that information being recorded are marked as coming in at the right time. Timing is also the reason that we use RT-Linux for this project. RT-Linux is a real-time system, meaning that tasks will be done when they are told to happen, rather than when the system has the free time to handle them. In most cases, we are better off letting an operating system decide what is most important, and thus what should receive which resources. For this new multi-media system, we need the computer to look at keeping good time as its first priority, and that's why we are using RT-Linux.

4.1.1 Modules

Under RT-Linux 2, real-time tasks must be implemented as modules. Under Linux, modules are bits of code which can be loaded into a running system, but act as if they are an integral part of the kernel (ie: they have access to the very low-level functions which are generally reserved for system processes). My timer design in based on a module which starts, stops, and increments timers. Each individual timer can be started and stopped separately, as well as have its own settings (such as how often it increments). The downside of using modules is that they operate in a different memory space than normal programs, so it can be tricky for a normal program to interact with a loaded module. There are two schemes for passing information to and from a module: FIFOs (first-in-first-out pipes) and shared memory. Each has it's advantages and disadvantages, and I used both of them for this part of the project.

4.1.2 FIFOs

A fifo is a data pipe which conveys data one-way (from a single source to a single target) in a first-in-first-out manner. This means that data is received from the pipe in the same order that it was sent in. These pipes are accessed through device files ('/dev/rtf*'). Normal (user

space) programs access these files just as they would normal files (open them for reading or writing, then write into or read from of them). Modules (system space programs) access a fifo using functions defined 'rtl_fifo.h' (part of the rtlinux package):

rtf_create(<fifo number>, <size>)
sets-up a fifo at file '/dev/rtf<#>' which can hold <size> data at once.

rtf_destroy(<fifo number>)
destroys the fifo at the file '/dev/rtf<#>'.

rtf_create_handler(<fifo number>, <handler function>)
when data comes in on <fifo number>, <handler function> is run.

rtf_get(<fifo number>, <buffer>, <count>)
gets up to <count> bytes from <fifo number> and places them in <buffer>.

rtf_put(<fifo number>, <buffer>, <count>)
puts <count> bytes from <buffer> into <fifo number>.

This project uses FIFOs for one way communication into the timer module. This is because very little information will need to be sent info the module (mainly just telling it to create and/or destroy timers). For sending data out of the module (namely to tell the program when a timer ticks), we need to find something else so that we can increment a counter w/out having to waste space by sending send a message each time.

4.1.3 Shared Memory

Shared memory is a block of memory that is set aside when the system starts up. This memory is not allocated to either the system, or user space. Both normal programs and modules can access shared memory space. Shared memory is set aside by adding the line ' append="mem=<size>m"' to '/etc/lilo.conf' (where <size> is the total memory in megabytes, minus the size of the shared memory block). The size of the shared memory block can't be larger than the page size on the given system. For this project, we will use a 1 megabyte block of shared memory. Modules and normal programs have different methods for accessing data

in shared memory. A module can directly access the data by creating a pointer with 'void *ptr = __va(<size>*0X100000)'.

Normal (user space) programs need to map shared memory into user space memory, which they can then access. This is done by opening '/dev/mem', then mapping the address using mmap(). Memory is unmapped once we are done with it by calling munmap().

```
void * ptr;
int fd = open("/dev/mem", O_RDONLY);
ptr = (char * ) mmap( 0, <shared size>, PROT_READ,
                        MAP_FILE |MAP_PRIVATE, fd, <size>*0X100000);

munmap( ptr, <shared size> );
```

4.1.4 User Space

The user space (normal program) portion of the timing system is contained in the 'mmsystem.c' file. This system communicates with the module to create, read and destroy timers. To keep track of the timers which have been created, break the shared memory block up into a structure. This structure contains a lock (only one copy of the mmsystem should be running at once), and three arrays. The first array holds the thread number for the real-time thread corresponding to this timer. The second array holds the thread number for the user-space thread that monitors the timer. The third array holds the 'value's of the timers, basically keeping track of the ticks since the timer was started. When a timer is created, the program is passed the index to the specific timer in these three arrays.

The first thing to do when dealing with timers is to initiate the mmsystem if it hasn't already been started. Initiating the mmsystem maps the shared memory block into a global pointer. It also creates an array of midiTimer's, that will contain the index, who created the timer, and the function to call each time the timer ticks. When you create a new timer, it finds a free space in the array (a free timer index) and marks that it will soon be in use. Then a message is sent to the module to create and start a new timer with the given parameters. Finally, a new thread is created which polls the timer, and executes the specified function whenever the timer ticks. Removing a timer works by sending a message to the module, then ending the polling thread for the timer. Finally, the timer is marked as being free again.

The polling method for dealing with the timer was chosen because it is flexible, and continues to perform well even on a highly loaded system. When using a signal/callback

approach, I found that signals would get lost under heavy system load. With the polling system there are no signals to be lost or interrupted, and the system keeps track of its own state. If the system falls behind during very heavy usage (ie: infrequent heavy lag, thrashing, etc.), it will automatically correct itself when there is more idle time available. Since we are dealing with mSecond long ticks, the lags will not be noticeable, unlike the cumulative effect of missed signals.

4.2 MIDI Handling

The multi-media layer of Windows has a specific way of dealing with MIDI, and the MaxMIDI system expects things to be handled in that way. The multi-media layer handles everything for locating MIDI devices, to opening a connection to the device, to send data to the device, to receiving and presenting the data in a specific format. Linux does none of these things for you, so we have to define them (and make them look like Windows functions).

4.2.1 MIDI Devices

The first step to handling MIDI is dealing with the MIDI connections to the computer. These connections are referred to as MIDI devices. Each MIDI device has an input and an output, and we have separate structures to deal with each. When a MIDI device is opened, we open the '/dev/midi**' file which corresponds to the device, for both reading and writing. When we open a device for input/output, we create a new structure which will handle the data on this particular device-direction. The overall device structure stores a pointer to this I/O device, and the I/O device remembers which actual device it came from. The I/O device deals with callbacks, running status, locks, and buffers (for input devices). Also related to devices are functions that get the number of MIDI devices, and fetch the device's capabilities.

4.2.2 MIDI Output

The multi-media system handles MIDI output by defining functions to open, close, and send messages to a MIDI output device. We open a MIDI output device by creating an output structure, and linking it with the actual MIDI device that will be used for the output. A midiOut structure contains a pointer to the actual MIDI device, a lock (so that we don't try to

send two messages at the same time), a status byte (for running status), and a pointer to the callback function which will be called when a long message has been sent out. That brings us to the idea of having two different types of messages -- short and long. A short message is a single normal (non-sysex) MIDI message. This is fine for sending one message at a time, independent of anything else. Another feature of short message is running status. If multiple messages in a row use the have the same status byte, the MIDI standard allows us to omit the status byte from subsequent message with the same status. This can save a lot of bandwidth, and so is important to implement. The function to send a short message (midiOutShortMessage) saves the status when it sends a message, and omits the status byte from the next message if it is the same.

Sometimes you need to send either a long single message (ie: sysex), or an ordered string of messages (say to control a bunch of channels at once), for this the mmsystem also defines a way to send long messages. These long messages are defined in a 'midiHdr' header structure which contains the full data of the message, and attributes needed to send the whole message. Long messages do not keep track of running status (in fact, they reset it), for two reasons. First of all, the main intent of long messages is for sending sysex data. A sysex message ends with the first non-data byte sent (ie: when a status byte is sent), if we were using running status, a normal message might be sent w/out its status byte, and would be interpreted as still being part of the sysex message. The other reason for abandoning running status while sending a long message is that if the message is used to send multiple smaller messages, we have no way of knowing what the last status byte of the long message was. This means there'd be no way to figure out what the running status is once we come out of the long message.

4.2.3 MIDI Input

MIDI input is handled by the multi-media system using a polling thread. Running on this thread is a function which continual checks for input, reads the input, and passes it along to the program. Input devices are can be started, stopped and reset, as well as opened and closed. Opening a MIDI input device consists of creating the device structure and specifying the callback function which will be responsible for handling the data. Closing a device mainly just frees the memory reserved by the input device, and allows it to be opened by someone

else. Starting the MIDI input device starts a polling thread for the device, and stopping it ends the polling thread. Resetting an input device clears its sysex buffers (see below).

Like MIDI output, input is handled differently for short and long (sysex) messages. Short messages are sent to the callback function as soon as they are received. Running status is also kept when a short message is received. If the next message received does not have a status byte, this status is used. Long messages are stored into a buffer as they are being received. Once the buffer is full, or the sysex message has been completely received, the buffer is sent to the callback and the next buffer moves up on the queue. If there are no free buffers, sysex messages are lost. If a sysex message is started while there are no buffers, the whole message will be ignored, even if buffers become available before the message ends. This is because sending only the a part of a sysex message would not make any sense.

5 Windows Functions

Not only did I have to mimic the windows multi-media system for this project, I also had to mimic much of the general windows functionality. The main 'windows' functionality I needed to implement was, well... windows. Windows uses its window objects to assign responsibility to different parts of a program. Each 'window' runs concurrently, and is responsible for handling its own elements and functions. Also notable are the windows-like message passing and memory management systems. These systems are defined and implemented in windows.h and WinRT.c. Also of note is the WinRT.h file, which defines direct conversions from windows types to RTLinux types (ie: BYTE->char, UINT->unsigned int, LPSTR->char *, and so on).

5.1 Window Objects

Window objects start as window classes. The window class defines most of the parameters for a window, and can be reused for every window that shares these parameters. My system really only implements two of the attributes of the window class, because it does not actually control any physical (GUI) windows, class name and windowProc. WindowProc is the function which will handle any messages sent to a window of this class. Window classes must be registered with the system, and are stored in a global queue to be used in opening new windows. That brings us to the main aspect of the window handling system, the windows themselves. Windows can be opened and closed by a program, and are mainly used to send and receive messages from different parts of the system. The MaxMIDI input system for example sends a message to the main window when a MIDI message is received. Once we are done with a window, we close it to free up memory space.

5.2 Message Passing

Messages are passed between windows to share information, and since the MaxMIDI system uses this functionality, we needed to implement it. Windows messages are posted, retrieved, and processed. The message system is centered around the message queue structure. This structure stores the posted messages (in the order received), and allows them to be retrieved. The message queue is defined as scalable (ie: it gets bigger as more messages are

added) -- we do this because we do not expect there to ever be more than a few messages on the queue, but need to be prepared for a large number.

Messages are sent to a specific window, but they can be received by any window if that window calls 'GetMessage' without specifying a window. Getting messages is a blocking process, this means that it will wait until a message is found before returning. This makes the 'get message loop' the center of most Windows programs. In such a loop, the window waits until it receives a message, processes the message, then waits for the next message to come in. When a message is 'gotten', it is removed from the message queue. Once received, messages are dispatched to the WindowProc that is supposed to handle them. The user defines this handling function to deal with the messages as need be.

5.3 Memory Allocation

One of the unexpected difficulties that I ran into is that Windows and Linux have very different schemes for memory management. When you request a block of memory from Windows, the system remembers what size the memory block is. This makes it very easy to say, double the size of a memory block. Linux does not remember such things, so a management abstraction system was needed to record the size of allocated memory blocks. This abstraction system deals with 'global handles', which are pointers to structures that store a pointer to the real memory block, and the size of the memory block. The system offers conversions from handle to pointer, and from pointer to handle (by searching the handle list for a matching pointer value). It also handles allocation, reallocation, and deallocation of memory blocks. The handle list is a double linked list, this means that each element of the list has a pointer both to the next element, and the previous element. This allows an element to be removed from the list using information within the element, with out having to search the whole list to find the elements exact placement. I implemented this list as a linked-list because I do not expect that very many blocks will be allocated in this way (really, only the mmsystem should use it), and so a linked-list provides a low over-head. Finding an element's handle based on a pointer to the actual memory location is slow, because it requires a search of the unordered list. I feel that this is an acceptable loss, because I expect elements to be added and deleted

more often than they are searched for (and adding/deleting elements from a linked-list is very fast, especially with a double linked-list).

6 General Techniques

There are a few general techniques used throughout the systems programmed for this project. Linked lists, recycling, locks and polling threads are just a few of the techniques used, but I will concentrate on these and attempt to explain their advantages, and to show why I chose them over other techniques.

6.1 Linked Lists

I have used linked-lists to hold arrays of structures throughout this project's system. I chose linked-lists as my primary array/list type mainly because they are scalable and very easy and fast to add to and delete from. The trade-off is that searches on these lists must be done linearly (looking at each element in turn, from first to last). Another problem to linked-lists is that they require creation/deletion of structs with each element added or removed. These are CPU intensive tasks, and should be minimized, not encouraged. This disadvantage will be addressed below in the 'recycling' section. Most of the lists in this project are queue's, which need to perform many additions/deletions, and very few searches. For this reason, unordered linked-lists seem the best choice. The other reason to use linked lists s that they have very little overhead. I do not know how many elements there will be in a given list, so it is good to use a system which grows linearly in size as I add more elements, rather than taking up the maximum amount of space no matter how much it gets used. Since I expect the lists to stay relatively small most of the time, this seems a good strategy.

6.2 Recycling

Along with the concept of linked-lists of structures comes the idea of structure recycling. It is assumed that allocating and dealocating memory are two of the most CPU consuming tasks of a program, and so they are generally minimized as much as possible. This is a major problem with the linked-list paradigm, because it requires so many allocations/deallocations. As in other cases where resource conservation is required, recycling is the answer. Recycling of structures requires the creation of a new type of queue, a 'free_struct' queue which contains any structures of a given type which are not in use. When a new structure is needed, the system first

checks this queue, and only creates a new struct if the queue is empty. When a struct is no longer needed (ie: its element has been deleted), it is added to the queue and awaits reuse. In this way, you never have to spend the resources to delete structures, and you only need to create as many structs as the maximum number of elements used in your list. One thing to watch for though, is that you must be very careful to clear any unwanted data from the 'free' structures before using them again, they are not empty like freshly allocated blocks.

6.3 Locks

The system implemented for this project uses threads to a great degree. Whenever you use threads, and access the same data from multiple threads, you run the risk of data corruption. For this reason, this project uses locks on all of the global queues. When a function is accessing a queue, it sets the lock on that queue and other functions are forced to wait until it is done before they get a chance to access the queue. Functions wait for a lock to open by use of a while loop. Each iteration of the loop forces the thread to 'usleep(1)'. This voluntarily gives resource control back to the CPU, so waiting for a lock to free does not cause any resource drain.

6.4 Polling Threads

A polling thread is a thread whose only purpose is to check for changes to some element of data. I use polling threads for both timers and watching MIDI input. Polling threads can be a resource drain, because they are generally implemented as continuos loops. This drain is minimized however by giving resource control back to the CPU by use of the 'usleep(1)' function. Some of the functionality of a polling thread could be handled using BSD signals, but I found that on heavily loaded systems, signals can be lost. In both places which I use polling threads, lost signals can have a cumulative effect which is very noticeable, and therefore to be avoided.

A Appendix: Glossary

Byte	A unit of computer data -- 8 bits (on/off). Each typed character is usual represented by a single byte.
C	A programming language. The code for this project is in C.
Callback	A function which is run in response to an action, such as a midi even coming in, or a timer ticking.
FIFO	First In First Out, a scheme for dealing with queues. Also refers to a type of pipe used by RTLinux to communicate between processes.
Linux	An open source operating system. Linux runs on many different computer platforms, and is developed by a wide community effort.
MIDI	Musical Instrument Digital Interface, specifies a 'language' for communicating with digital instruments.
Real-Time Linux	A Linux operating system which incorperates real-time scheduling to allow programs to specify *exactly* when something should be done.
Serial	A type of computer cable/connection. Only allows data flow in one direction at a time.
Signal	A way to communicate between and within processes, you send a signal and if a program has been set to catch it, a function is run. Also referred to as an interrupt (though they aren't quite the same thing).
System Space	The memory space which holds the operating system, and any modules

	loaded into the OS.
User Space	The memory space occupied by user-run programs.
Virtual Orchestra	A computer music system designed to take the place of conventional live orchestras in theatrical productions, mainly when a live orchestra would be cost prohibitive.

B Appendix: Headers

B.1 mmsystem.h

```
/***
 * Header file for the multi-media system
 ***/
#include <pthread.h>
#include <errno.h>

#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>             /* sizeof() */
#include <sys/mman.h>           /* mmap(), PROT_READ, MAP_FILE, MAP_FAILED */

#include "WinRT.h"

#ifndef _MMSYS
#define _MMSYS

// Define a few data types
#define LPTIMECALLBACK          void *
#define MMRESULT                WORD
#define MMVERSION               DWORD

// Maximum name length
#define MAXPNAMELEN             256

// Midi header messages
#define MHDR_DONE                       1
#define MHDR_INQUEUE            2
#define MHDR_PREPARED           4

// MMSYSTEM error codes
#define MMSYSERR_NOERROR        0
#define MMSYSERR_BADDEVICEID    1
#define MMSYSERR_NOMEM          2
#define MMSYSERR_ALLOCATED      3
#define MMSYSERR_INVALPARAM     4
#define MMSYSERR_INVALHANDLE    5

// MIDI error codes
#define MIDIERR_STILLPLAYING    10

// MIDI OUT messages
#define MOM_DONE                15
#define MOM_CLOSE               16

// MIDI IN messages
#define MIM_DATA                20
#define MIM_LONGERROR           21
#define MIM_LONGDATA            22
#define MIM_CLOSE               23

#define MIDI_MAPPER                     16

#define MMSYSERR_MAXERR                 32

// Status byte stuff
#define STAT_MIN                0x80
```

```
#define STAT_SYSEX                      0xF0
#define STAT_EOX                    0xF7
#define STAT(a)                     ((a) >= 0x80)
#define STAT_CHANNEL(a)             ((a) >= 0x80 && (a) < 0xF0)
#define STAT_REALTIME(a)            ((a) >= 0xF8)

// Arguments
#define TIME_PERIODIC               1
#define CALLBACK_FUNCTION               1

// Base address of the shared memory space
#define BASE_ADDRESS                (47 * 0x100000)

// Maximum timers allowed
#define MAX_TIMERS                      256

// Size of the messages sent through the fifo
#define MMEVENT_SIZE                4

// Messages sent to the module
#define CREATE_TIMER                1
#define KILL_TIMER                  2

#define MODULE_INSTALLED            1


// Status for timers in shared memory
#define BLOCK_MESSAGE_MIN               -5

#define BLOCK_BUSY                      -4
#define BLOCK_TERM                      -3
#define BLOCK_USED                      -2
#define BLOCK_UNUSED                -1

// Define the midi callback type
typedef void  MIDICALLBACK (void * handle, UINT wMsg,
                            DWORD dwInstance, DWORD dwParam1, DWORD dwParam2);
#define LPTIMECALLBACK void *

// A MIDI timer
struct midiTimer
{
    UINT            ID;                 // The timer's id (index)
    DWORD           instance;           // The 'window' which started this timer
    MIDICALLBACK  *callback;            // The function that gets called each tick
};

// MIDI device capability structure
typedef struct
{
    WORD    wMid;                   // Manufacturer identifier of the device driver
    WORD    wPid;                   // Product identifier of the MIDI output device

    MMVERSION vDriverVersion;       // Version number of the device driver for the MIDI output device
                                    // The high-order byte is the major version number,
                                    // and the low-order byte is the minor version number

    CHAR    szPname[MAXPNAMELEN];       // Product name in a null-terminated string
    WORD    wTechnology;            // Flags describing the type of the MIDI output device
    WORD    wVoices;                // Number of voices supported by an internal synthesizer device
    WORD    wNotes;                 // Maximum number of simultaneous notes that can be played
```

```
    WORD    wChannelMask;       // Channels that an internal synthesizer device responds to
                                // The least significant bit refers to channel 0 and
                                // the most significant bit to channel 15

    DWORD   dwSupport;          // Optional functionality supported by the device
} MIDICAPS, MIDIOUTCAPS, MIDIINCAPS;
#define LPMIDICAPS          MIDICAPS *
#define LPMIDIOUTCAPS       MIDICAPS *
#define LPMIDIINCAPS                MIDICAPS *

// MIDI header structure
struct midiHdr
{
     LPSTR  lpData;                         // Pointer to MIDI data
    DWORD  dwBufferLength;                  // Size of the buffer
    DWORD  dwBytesRecorded;                 // Actual amount of data in the buffer
    DWORD  dwUser;                          // Custom user data
    DWORD  dwFlags;                         // Flags giving information about the buffer:
                                            // MHDR_DONE, MHDR_INQUEUE, MHDR_PREPARED
     struct midiHdr *lpNext;                // Reserved
    DWORD  reserved;                        // Reserved
    DWORD  dwOffset;                        // Offset into the buffer when a callback is performed
    DWORD  dwReserved[4];                   // Reserved
};
#define MIDIHDR     struct midiHdr
#define LPMIDIHDR   MIDIHDR *

// MIDI output device structure
struct midiOut
{
    struct midiDev    *device;      // Pointer to the actual device

    MIDICALLBACK  *callback;       // Callback to execute
    DWORD         instance;        // Window which opened the device

    BYTE          status;          // Running status
    BOOL          playing;         // Lock
};
#define HMIDIOUT    struct midiOut *
#define LPHMIDIOUT HMIDIOUT *

// MIDI input device structure
struct midiIn
{
    struct midiDev    *device;      // Pointer to the actual device

    MIDICALLBACK  *callback;       // Callback to execute when we receive data
    DWORD         instance;        // Window which started this device
    pthread_t     task;            // Polling thread for this device

    BYTE          status;          // Running status
    BOOL          running;         // Device running?
    BOOL          lock;            // Lock
    LPMIDIHDR     buffer;          // Sysex buffers
};
#define HMIDIIN     struct midiIn *
#define LPHMIDIIN   HMIDIIN *

// MIDI decivice structure
struct midiDev
{
    HMIDIIN       midiIn;                   // Pointer to the input 'device'
```

```
    HMIDIOUT        midiOut;                    // Pointer to the output 'device'
    int             fd;                         // File descriptor for this device
};

#endif                          // _MMSYS


BOOL init_mmsys( void );
UINT mmsystem_status( UINT ID );

void timeKillEvent( WORD ID );
BOOL timeSendEvent( WORD event[4] );
WORD timeSetEvent(
            UINT uDelay, UINT uResolution, LPTIMECALLBACK lpTimeProc,
            DWORD dwUser, UINT fuEvent );
void *timerThread( void *timer );

BOOL init_midi();
MMRESULT openMidiDev( UINT dev );
MMRESULT midiGetDevCaps(
            UINT uDeviceID, LPMIDICAPS lpMidiCaps, UINT cbMidiCaps );
UINT midiGetNumDevs( void );

MMRESULT midiOutGetDevCaps(
            UINT uDeviceID, LPMIDIOUTCAPS lpMidiOutCaps, UINT cbMidiOutCaps );
UINT midiOutGetNumDevs( void );
UINT midiOutOpen(
            LPHMIDIOUT lphmo, UINT uDeviceID, DWORD dwCallback,
            DWORD dwCallbackInstance, DWORD dwFlags );
MMRESULT midiOutClose( HMIDIOUT hmo );
MMRESULT midiOutReset( HMIDIOUT hmo );

MMRESULT midiOutPrepareHeader( HMIDIOUT hmo, LPMIDIHDR lpMidiOutHdr, UINT cbMidiOutHdr );
MMRESULT midiOutUnprepareHeader( HMIDIOUT hmo, LPMIDIHDR lpMidiOutHdr, UINT cbMidiOutHdr );

MMRESULT midiOutShortMsg( HMIDIOUT out, DWORD msg );
MMRESULT midiOutLongMsg( HMIDIOUT hmo, LPMIDIHDR lpMidiOutHdr, UINT cbMidiOutHdr );

void *midiInThread( void *hmi );
BOOL midiInWait( HMIDIIN hmi );

MMRESULT midiInGetDevCaps(
            UINT uDeviceID, LPMIDIINCAPS lpMidiInCaps, UINT cbMidiInCaps );
UINT midiInGetNumDevs( void );
MMRESULT midiInOpen(
            LPHMIDIIN lphMidiIn, UINT uDeviceID, DWORD dwCallback,
            DWORD dwCallbackInstance, DWORD dwFlags );

MMRESULT midiInPrepareHeader( HMIDIIN hmi, LPMIDIHDR lpMidiInHdr, UINT cbMidiInHdr );
MMRESULT midiInUnprepareHeader( HMIDIIN hmi, LPMIDIHDR lpMidiInHdr, UINT cbMidiInHdr );
MMRESULT midiInAddBuffer( HMIDIIN hMidiIn, LPMIDIHDR lpMidiInHdr, UINT cbMidiInHdr );

MMRESULT midiInStart( HMIDIIN hMidiIn );
MMRESULT midiInStop( HMIDIIN hMidiIn );
MMRESULT midiInClose( HMIDIIN hMidiIn );
MMRESULT midiInReset( HMIDIIN hMidiIn );
```

B.2 WinRT.h

```
/***
 * Define data types for conversion from windows to rtlinux
```

```
***/

#ifndef __WINRT__
    // Define a bunch of empty arguments
    #define __WINRT__
    #define __export
    #define WINAPI
    #define FAR
    #define CALLBACK

    // A simple 'min' program
    #define min(a, b)  ((a) < (b) ? (a) : (b))

    // Define a few windows messages
    #define WM_QUIT         1
    #define WM_USER         255

    // True/False
    #define TRUE    1
    #define FALSE   0

    // Windows data-types
    typedef unsigned char               BYTE;
    typedef unsigned int                DWORD;
    typedef unsigned short int          WORD;
    typedef unsigned int                UINT;
    typedef char                        BOOL;
    typedef char                        CHAR;

    // Common windows pointers
    #define LPSTR           char *
    #define HMENU           void *
#endif
```

B.3 windows.h

```
/***
/ This header file is designed to help port windows programs into rtlinux
/ (specificaly those windows programs based on the MaxMidi DLLs).
***/
#include <stdio.h>
#include <time.h>
#include <ctype.h>
#include "WinRT.h"

#ifndef __RTLINUX__
  #define __RTLINUX__

  #define STRLEN    256

  // Macros for dealing with different data sizes
  #define HIBYTE(a)         ((a / 0x100) & 0xFF)
  #define HIWORD(a)                 ((a / 0x10000) & 0xFFFF)
  #define LOBYTE(a)         (a & 0xFF)
  #define LOWORD(a)                 (a & 0xFFFF)
  #define MAKELONG(l,h)     ((h * 0x10000) + l)

  #define SELECTOROF(a)     (a)

  // Define a few more pointer types
  #define LPCTSTR           char *
```

```
#define LPDWORD         DWORD *
#define HANDLE          FILE *
#define LRESULT         DWORD

#define WPARAM          DWORD
#define LPARAM          DWORD

// Define the constants for global memory allocation (not actualy used)
#define GMEM_MOVEABLE       1
#define GMEM_SHARE          2
#define GMEM_ZEROINIT       3
#define GHND                4
#define GPTR                5

// Global memory structure
struct globalMem
{
  // Navigation pointers
struct globalMem        *next;
  struct globalMem      *prev;

  void *                ptr;            // Pointer to the memory block
  size_t                size;           // Size of the memory block
};
#define HGLOBAL         struct globalMem *

// Window structure
struct hwnd;
#define HWND    struct hwnd *
typedef LRESULT WNDPROC (HWND hWnd, UINT iMessage, WPARAM wParam, LPARAM lParam);
struct hwnd
{
  LPCTSTR       class;                  // Class name
  LPCTSTR       name;                   // Window name
  BOOL          assigned;
  WNDPROC       *proc;                  // WindowProc (handler for messages)
};

// Message queue structure
struct msgque
{
  struct smsg   *end;           // Pointer to the last message in the queue
  struct smsg   *buffer;        // Pointer to the beging of the queue
  BOOL          lock;
};

// Actual message structure
struct smsg
{
  struct smsg   *next;                  // Next message in the queue

  HWND          hWnd;                   // Target window
  UINT          message;                // The message
  WPARAM        wParam;                 // Word paramater
  LPARAM        lParam;                 // Long paramater
  time_t        time;                   // When the message was posted

};
#define MSG             struct smsg *

// A few more empty defenitions
#define HINSTANCE                       DWORD
```

```
#define HICON                   void *
#define HCURSOR                         void *
#define HBRUSH                  void *
#define CS_GLOBALCLASS          0
#define WS_DISABLED             0
#define CW_USEDEFAULT           0
typedef struct _WNDCLASS
{
  struct _WNDCLASS *next;

  UINT          style;
  WNDPROC       *lpfnWndProc;
  int           cbClsExtra;
  int           cbWndExtra;
  HINSTANCE     hInstance;
  HICON         hIcon;
  HCURSOR       hCursor;
  HBRUSH        hbrBackground;
  LPCTSTR       lpszMenuName;
  LPCTSTR       lpszClassName;
} WNDCLASS;
#define PWNDCLASS struct _WNDCLASS *

HGLOBAL GlobalAlloc( int nadda, size_t size );
void *GlobalLock( HGLOBAL ptr );
HGLOBAL GlobalHandle( void *ptr );
HGLOBAL GlobalReAlloc( HGLOBAL hMem, size_t dwBytes, UINT uFlags );
DWORD GlobalSize( HGLOBAL hMem );
void FreeGlobalMem16( void *ptr );

DWORD timeGetTime( void );

PWNDCLASS RegisterClass( const WNDCLASS *lpWndClass );
HWND CreateWindow( char *lpClassName, char *lpWindowName, DWORD dwStlye, int x,
                                        int y, int nWidth, int nHeight, HWND hWndParent, HMENU hMenu,
                                        DWORD hInstance, void *lpParam );
BOOL DestroyWindow( HWND hWnd);

void SetMessageQueue( UINT queueSize );
BOOL PostMessage( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam );
int GetMessage( MSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax );
LRESULT SendMessage( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam );
LRESULT DispatchMessage( const MSG lpmsg );

LRESULT DefWindowProc(HWND hWnd, UINT iMessage, WPARAM wParam, LPARAM lParam);

LPSTR lstrcpy( LPSTR str1, LPSTR str2);
#endif
```

C Appendix: Program Files

C.1 Makefiles

C.1.1 rtl.mk

```
# rtl.mk -- contains definitions for the Makefile
#Automatically generated by rtl Makefile
RTL_DIR = /usr/src/rtlinux-2.0/rtl
RTLINUX_DIR = /usr/src/rtlinux-2.0/linux
INCLUDE= -I/usr/src/rtlinux-2.0/linux/include -I/usr/src/rtlinux-2.0/rtl/include -I/usr/src/rtlinux-2.0/rtl
CFLAGS = -I/usr/src/rtlinux-2.0/linux/include -I/usr/src/rtlinux-2.0/rtl/include -I/usr/src/rtlinux-2.0/rtl -
I/usr/src/rtlinux-2.0/rtl/include/posix -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -D__SMP__ -D__RTL__ -
D__KERNEL__ -DMODULE -pipe -fno-strength-reduce -m486 -malign-loops=2 -malign-jumps=2 -malign-
functions=2 -DCPU=686
ARCH = i386
CC = gcc
RTL_MODULES=/lib/modules/2.2.13-rtl2.0/misc
```

C.1.1 Makefile

```
all: mm_module.o sync_test

include rtl.mk

sync_test     : main.c Midiin.c Midiout.c Sync.c mmsystem.c WinRT.c windows.h WinRT.h MxDLL.h mmsystem.h
      $(CC) ${INCLUDE} -O2 -Wall -lpthread main.c Midiin.c Midiout.c Sync.c mmsystem.c WinRT.c -o sync_test

mm_module.o: mmsystem_module.c mmsystem.h WinRT.h MxDLL.h
      $(CC) ${INCLUDE} ${CFLAGS} -c mmsystem_module.c -o mm_module.o
```

C.2 mmsystem_module.c

```
/***
 * Real-time part of the mmsystem. This module controls the mmsystem
 * timers. Compile using:
 *    INCLUDE= -I/usr/src/rtlinux-2.0/linux/include -I/usr/src/rtlinux-2.0/rtl/include
 *     -I/usr/src/rtlinux-2.0/rtl
 *    CFLAGS = -I/usr/src/rtlinux-2.0/linux/include -I/usr/src/rtlinux-2.0/rtl/include
 *     -I/usr/src/rtlinux-2.0/rtl -I/usr/src/rtlinux-2.0/rtl/include/posix -Wall
 *     -Wstrict-prototypes -O2 -fomit-frame-pointer -D__SMP__ -D__RTL__ -D__KERNEL__
 *     -DMODULE -pipe -fno-strength-reduce -m486 -malign-loops=2 -malign-jumps=2
 *     -malign-functions=2 -DCPU=686
 *
 *    mm_module.o     : mmsystem_module.c mmsystem.h WinRT.h MxDLL.h
 *              $(CC) ${INCLUDE} ${CFLAGS} -c mmsystem_module.c -o mm_module.o
 *
 * It expects a shared memory space at BASE_ADDRESS (defined in 'mmsystem.h'), and
 * will use rt-fifo #1 for communication ('/dev/rtf1').
 ***/

#include <linux/errno.h>
#include <rtl.h>
#include <time.h>
```

```
#include <rtl_sched.h>
#include <rtl_fifo.h>

#include "mmsystem.h"
#include "MxDLL.h"

// This structure makes up the first part of the shared memory block
struct shared
{
    BOOL        lock;                   // Lock the timer
    pthread_t   rt_task[MAX_TIMERS];    // Keep track of the rt thread for this timer
    pthread_t   pr_task[MAX_TIMERS];    // Keep track of the program thread for this timer
    DWORD       timer[MAX_TIMERS];      // This stores the current 'time'
};

// This thread is the timer, it runs periodicly to iterate the timer
void *timer_thread( void *t )
{
    DWORD *timer = (DWORD *) t;             // pointer to the timer device

    while( 1 )
    {
        pthread_wait_np();                  // Wait untill the period is up
        *timer += 1;                        // Increment the timer
    }

    return NULL;
}

// When a message comes in on the fifo, this handler is run
int task_handler( unsigned int fifo )
{
    struct shared *ptr= (struct shared *) __va(BASE_ADDRESS);   // Shared memory
    pthread_attr_t attr;
    // Empty thread attributes
    struct sched_param sched_param;                             // Thread scheduling paramaters
    int i;
    WORD com[MMEVENT_SIZE];

    // Get the message from the fifo
    rtf_get( fifo, com, MMEVENT_SIZE * sizeof(int) );
    // Make sure the message makes sense
    if( (i = com[0]) >= MAX_TIMERS )
        return 1;

    // Switch on the message type
    switch( com[1] )
    {
    // Create a new timer
     case CREATE_TIMER:
        pthread_attr_init (&attr);
        sched_param.sched_priority = SCHED_RR;          // Round-robin scheduling
        pthread_attr_setschedparam (&attr, &sched_param);

        // Create a new thread for the this timer, store the task number
        //  The thread calls timer_thread, and passes a pointer to the timer to increment
        pthread_create( &(ptr->rt_task[com[0]]), &attr, timer_thread,
                                        (void *) &(ptr->timer[com[0]]) );

        // Make the thread periodic (run the loop over and over)
        // First message argument (com[2]) is the start_time (in mseconds)
```

```
                // Second message argument (com[3]) is the resolution (msec between ticks)
                pthread_make_periodic_np( ptr->rt_task[com[0]],
gethrtime()+(10000000*com[2]), 10000000*com[3] );
                break;

   // Destroy a timer
      case KILL_TIMER:
                // If the given timer isn't set, don't try to kill it
                if( BLOCK_MESSAGE(ptr->rt_task[i]) )
                        return 1;

                // Stop the thread
                pthread_delete_np( ptr->rt_task[i] );

                // Mark this timer as dead
                ptr->rt_task[i] = BLOCK_TERM;
                break;
       default:
                return 1;
      }

      return 0;
}

// This function is called when the module is loaded
int init_module(void)
{
      struct shared *ptr = (struct shared *) __va(BASE_ADDRESS);
      int i;

      // Clear the fifo we will use (rtf1)
      rtf_destroy(1);

      // Create the fifo again
      if( rtf_create(1, 65536) )
                return 1;

      // Mark all timers as free
      for( i = 0; i < MAX_TIMERS; i++ )
                ptr->rt_task[i] = BLOCK_UNUSED;

      // Set the handler for the fifo
      rtf_create_handler(1, &task_handler);

      // Mark that the module is installed (in shared memory)
      ptr->lock = MODULE_INSTALLED;
      return 0;
}

// This is called when the module is removed
void cleanup_module(void)
{
      struct shared *ptr= (struct shared *) __va(BASE_ADDRESS);
      int i;

      // Delete any threads that are still active
      for( i = 0; i < 16; i++ )
      {
                if( ptr->rt_task[i] < 0 && ptr->rt_task[i] > BLOCK_MESSAGE_MIN )
                {
                        pthread_delete_np(ptr->rt_task[i]);
                        ptr->rt_task[i] = BLOCK_TERM;
```

```
                }
        }

        // Unlock the system
        ptr->lock = 0;
}
```

C.3 mmsystem.c

```
/***
 * This file contains the user-space multi-media system code.
 * It implements: timers, and MIDI I/O
 ***/

#include <sys/time.h>
#include "mmsystem.h"
#include "MxDLL.h"

BOOL get_line( FILE *fd, LPSTR outstr );

// Structure used in shared memory
struct shared
{
        BOOL            lock;
        pthread_t       rt_task[MAX_TIMERS];
        pthread_t       pr_task[MAX_TIMERS];
        DWORD           timer[MAX_TIMERS];
};

/* Global pointers */
// Shared memory
struct shared           *MMSYS = NULL;

// MIDI devices
struct midiDev          *MIDIDEV = NULL;
// MIDI timers
struct midiTimer *MIDITIMER = NULL;
// Keep track of when the system starts up
time_t          start_time = 0;

// Initiate the multi-media system
BOOL init_mmsys( )
{
        int fd;

        // Set the start time
        if( !start_time )
                start_time = time(NULL);

        // Open memory device
        if ((fd = open("/dev/mem", O_RDWR)) < 0)
        {
printf( "Couldn't open /dev/mem.\n" );
return FALSE;
        }

      // Map the shared memory
      MMSYS = (struct shared *) mmap(0, (sizeof(struct shared)),
                                     PROT_READ | PROT_WRITE,
                                     MAP_FILE | MAP_SHARED,
```

```
                                    fd, BASE_ADDRESS );

    if( MMSYS == MAP_FAILED )
    {
      printf( "Couldn't map memory (%d).\n", errno );
      return FALSE;
    }

    // Close the memory device
    close(fd);

    // Create the midiTimer array
    MIDITIMER = (struct midiTimer *) calloc( MAX_TIMERS, sizeof(struct midiTimer) );
    return TRUE;
}

// Free-up a timer
void mmsys_free( UINT ID )
{
      // This isn't a valid timer
      if( !MMSYS || ID >= MAX_TIMERS )
                return;

      // Kill the timer if it is actualy alive
      if( MMSYS->rt_task[ID] >= 0 )
                timeKillEvent( ID );

      // Mark this timer as free
      MMSYS->rt_task[ID] = BLOCK_UNUSED;

      return;
}

// Get the status of a timer
UINT mmsystem_status( UINT ID )
{
      if( !MMSYS || ID >= MAX_TIMERS )
                return MMSYSERR_BADDEVICEID;

      return MMSYS->rt_task[ID];
}

// Sends a message (event) to the mmsystem module
BOOL timeSendEvent( WORD event[MMEVENT_SIZE] )
{
      int fd;

      // Open the fifo for writing
      if( (fd = open( "/dev/rtf1", O_WRONLY )) < 0 )
                return FALSE;

      // Write the message
      write( fd, event, MMEVENT_SIZE*sizeof(WORD) );
      close( fd );

      return TRUE;
}

// Kill the timer
void timeKillEvent( WORD ID )
{
      WORD com[4];
```

```
    struct midiTimer *timer = &MIDITIMER[ID-1];

    if( !MMSYS || !timer )
            return;

    // Set-up the message we will send
    com[0] = timer->ID; com[1] = KILL_TIMER;
    timeSendEvent( com );

    // Stop the timer thread
    pthread_join( MMSYS->pr_task[timer->ID], NULL );
    // Show that the timer is now free
    MMSYS->rt_task[timer->ID] = BLOCK_UNUSED;

    return;
}

// Create a new timer
// Delay            -- Time we wait untill starting the timer
// uResolution      -- Time in mSec between each tick
// lpTimeProc-- The function which will handle the timer tick
WORD timeSetEvent
                    ( UINT uDelay, UINT uResolution, LPTIMECALLBACK lpTimeProc, DWORD dwUser,
UINT fuEvent )
{
    WORD com[4];
    UINT ID, i;

    // Initiate the mmsystem if it is off
    if( !MMSYS )
            init_mmsys( );

    // Find the first unused timer
    for( i = 0; i < MAX_TIMERS; i++ )
    {
            if( MMSYS->rt_task[i] == BLOCK_UNUSED )
                    break;
    }

    // Fail if there are no unused timers
    if( i >= MAX_TIMERS )
            return 0;

    ID = i;
    // Mark the timer as in use
    MMSYS->rt_task[i] = BLOCK_USED;

    MIDITIMER[ID].ID = ID;
    MIDITIMER[ID].callback = lpTimeProc;
    MIDITIMER[ID].instance = dwUser;

    // Send the event which will create the rt_timer
    com[0] = ID; com[1] = CREATE_TIMER;
    com[2] = uDelay; com[3] = uResolution;
    timeSendEvent( com );

    // Create a thread to handle poll the timer
    pthread_create( &(MMSYS->pr_task[ID]), NULL, timerThread, (void *) &MIDITIMER[ID]);

    return ID+1;
}
```

```
// This thread polls the timer
void *timerThread( void *t )
{
	struct midiTimer *timer = (struct midiTimer *) t;
	DWORD position = 0;

	// End the function when the timer dies
	while( mmsystem_status(timer->ID) != BLOCK_TERM )
	{
		// Call the callback on each tick of the timer
		while(  position != MMSYS->timer[timer->ID] )
		{
			if( position > MMSYS->timer[timer->ID] )
				position = MMSYS->timer[timer->ID];
			else
				position++;

			timer->callback( timer, 0, timer->instance, 0, 0 );
		}

		// Give control back to the system (let others in the queue go)
		usleep(1);
	}

	return NULL;
}

// Initiate the midi device array
BOOL init_midi()
{
	int devs = midiGetNumDevs();

	if( !start_time )
		start_time = time(NULL);

	// Create an array of midi devices
	MIDIDEV = (struct midiDev *) calloc( devs, sizeof(struct midiDev) );

	return TRUE;
}

// Open a specific midi device
MMRESULT openMidiDev( UINT dev )
{
	int fd;
	char buf[256];

	// Make sure the device is legal
	if( dev >= midiGetNumDevs() )
		return MMSYSERR_BADDEVICEID;

	// Make sure the device hasn't alreayd been opened
	if( MIDIDEV[dev].fd )
		return MMSYSERR_ALLOCATED;

	// Open the midi device file
	sprintf( buf, "/dev/midi%d%d", dev/10, dev%10 );
	if( (fd = open( buf, O_RDWR, O_NDELAY )) == -1 )
		return MMSYSERR_NOMEM;

	// Set the array with the file descriptor
	MIDIDEV[dev].fd = fd;
```

```
	return MMSYSERR_NOERROR;
}

// Close a specific midi device
void closeMidiDev( struct midiDev *dev )
{
	if( !dev->fd || dev->midiIn || dev->midiOut )
		return;

	dev->fd = 0;

	return;
}

// Get the capabilities of the given device
MMRESULT midiGetDevCaps( UINT uDeviceID, LPMIDICAPS lpMidiCaps, UINT cbMidiCaps )
{
	FILE *fd;
	char buf[256], *ptr = NULL, i = -1;

	// Open the sndstats file
	if( (fd = fopen( "/dev/sndstat", "r" )) < 0 )
		return MMSYSERR_NOMEM;

	// Look for the MIDI devices section, and get info
	for( buf[0] = '\0'; strcmp(buf, "Midi devices:"); get_line(fd, buf) );
	do
	{
		get_line(fd, buf);
		for( ptr = buf; isdigit(*ptr); ptr++ );
		*ptr = '\0'; i = atoi(buf);
	} while( isdigit(buf[0]) && i != uDeviceID );

	fclose(fd);

	// Fill in the device capabilities (since we don't have a database of devices,
	// most things will be NULL)
	if( !isdigit(buf[0]) )
		return MMSYSERR_BADDEVICEID;

	lpMidiCaps->wMid			= 0;
	lpMidiCaps->wPid			= 0;
	lpMidiCaps->vDriverVersion		= 0;

	while( isspace(*(++ptr)) ); ptr++;
	lstrcpy( lpMidiCaps->szPname, ptr );
	lpMidiCaps->wTechnology		= 0;
	lpMidiCaps->wVoices			= 0;
	lpMidiCaps->wNotes			= 0;
	lpMidiCaps->wChannelMask		= 0;
	lpMidiCaps->dwSupport		= 0;

	return 0;
}


// Get the number of installed MIDI devices
UINT midiGetNumDevs()
{
	FILE *fd;
	char buf[256], i = 0;
```

```
    BOOL err = TRUE;

    // Open the sndstat file
    if( (fd = fopen( "/dev/sndstat", "r" )) == NULL )
        return MMSYSERR_NOMEM;

    // Find the section on MIDI devices
    for( buf[0] = '\0'; err && strcmp(buf, "Midi devices:"); err = get_line(fd, buf) );
    if( !err )
        return 0;

    // Count the devices
    get_line(fd, buf);
    while( err && isdigit(buf[0]) )
    {
        i++;
        get_line(fd, buf);
    }

    fclose(fd);

    return i;
}


// Get the size of a MIDI messages data section, given its status byte
int getMidiSize( BYTE status )
{
    if( status >= 0x80 && status < 0xC0 )
        // Normal messages, using both data bytes
        return 2;
    else if( status < 0xE0 )
        // Messages using just one data byte (Program Change, Channel Aftertouch)
        return 1;
    else if( status < 0xF0 )
        // Pitch Blend messages, using two data bytes
        return 2;
    else if( status == 0xF1 || status == 0xF4 || status == 0xF5 )
        // Undefined system common messages
        return -1;
    else if( status == 0xF2 )
        // Song Position pointer -- 2 data bytes
        return 2;
    else if( status == 0xF3 )
        // Song select mesage, only one data byte
        return 1;
    else if( status == 0xF6 )
        // Tune end message, no data
        return 0;
    else if( status == 0xF7 )
        // End of Sysex data
        return 0;
    else if( status >= 0xF8 )
        // Sysyem real-time message, no data bytes
        return 0;

    // Invalid status, or SYSEX byte
    return -1;
}

// Open an output 'device', and specify its attributes (like callback)
UINT midiOutOpen(
```

```
                        LPHMIDIOUT lphmo, UINT uDeviceID, DWORD dwCallback,
DWORD dwCallbackInstance, DWORD dwFlags )
{
    // Make sure the MIDI system is started
    if( !MIDIDEV )
            init_midi();

    // Make sure the device is legal
    if( uDeviceID >= midiOutGetNumDevs() )
            return MMSYSERR_BADDEVICEID;

    // Make sure we haven't already opened this device for output
    if( MIDIDEV[uDeviceID].midiOut )
            return MMSYSERR_ALLOCATED;

    // The only flag we can handle is CALLBACK_FUNCTION
    if( dwFlags != CALLBACK_FUNCTION )
            return MMSYSERR_INVALPARAM;

    // Allocate memory for the device
    if( (*lphmo = (HMIDIOUT) malloc(sizeof(struct midiOut))) == NULL )
            return MMSYSERR_NOMEM;

    // Open the device if needs be
    if( !MIDIDEV[uDeviceID].fd )
            if( openMidiDev(uDeviceID) )
                    return MMSYSERR_NOMEM;

    // Point to the new output 'device', and fill its attributes
    MIDIDEV[uDeviceID].midiOut = *lphmo;
    (*lphmo)->device      = &(MIDIDEV[uDeviceID]);

    (*lphmo)->playing           = FALSE;
    (*lphmo)->callback          = (void *) dwCallback;
    (*lphmo)->instance          = dwCallbackInstance;

    return MMSYSERR_NOERROR;
}

// Close a give MIDI output 'device'
MMRESULT midiOutClose( HMIDIOUT hmo )
{
    // Make sure the pointer is valid
    if( !hmo )
            return MMSYSERR_INVALHANDLE;

    // Report that we are still sending output
    if( hmo->playing )
            return MIDIERR_STILLPLAYING;

    // Send a close message to the callback
    (*hmo->callback)( hmo, MOM_CLOSE, hmo->instance, 0, 0 );
    // Remove the pointer from the acctual device to the output 'device'
    hmo->device->midiOut = NULL;

    // Close the actual device if it is not being used anymore
    if( !hmo->device->midiIn )
            closeMidiDev( hmo->device );
    // Free this 'device'
    free( hmo );

    return MMSYSERR_NOERROR;
```

```
}

// Send a reset message to the output 'device'
MMRESULT midiOutReset( HMIDIOUT hmo )
{
	BYTE n;
	DWORD msg;

	// Make sure the device handle isvalid
	if( !hmo )
		return MMSYSERR_INVALHANDLE;

	// Send a reset message to each channel
	for(n = 0; n < 16; n++)
	{
		msg = (0x00007BB0 | (DWORD)n);
		write( hmo->device->fd, &msg, sizeof(DWORD) );
	}

	return MMSYSERR_NOERROR;
}

// Prepare an output header for use
// Store the start of data in 'reserved'
MMRESULT midiOutPrepareHeader( HMIDIOUT hmo, LPMIDIHDR lpMidiOutHdr, UINT cbMidiOutHdr )
{
	lpMidiOutHdr->reserved = (DWORD) lpMidiOutHdr->lpData;
	return MMSYSERR_NOERROR;
}

// Reset an output header
// Set the data pointer back to the begining of the data
MMRESULT midiOutUnprepareHeader( HMIDIOUT hmo, LPMIDIHDR lpMidiOutHdr, UINT cbMidiOutHdr )
{
	lpMidiOutHdr->lpData = (LPSTR) lpMidiOutHdr->reserved;
	return MMSYSERR_NOERROR;
}

// Get the capabilities of an output 'device'
MMRESULT midiOutGetDevCaps(
			UINT uDeviceID, LPMIDIOUTCAPS lpMidiOutCaps, UINT cbMidiOutCaps )
{
	return midiGetDevCaps( uDeviceID, lpMidiOutCaps, cbMidiOutCaps );
}

// Get the number of output 'devices' (same as input)
UINT midiOutGetNumDevs( void )
{
	return midiGetNumDevs();
}

// Write a MIDI message to the device
void writeMidi( int fd, BYTE *message, int size )
{
	int i = 0;

	// Keep sending data until everything is sent
	while( i < size )
		i += write( fd, message+i, (size-i) * sizeof(BYTE) );

	return;
}
```

```
// Send a short (non-sysex) message out
MMRESULT midiOutShortMsg( HMIDIOUT hmo, DWORD msg )
{
	int size;
	BYTE *status	= (BYTE *) &msg;
	BYTE *data	= (BYTE *) (&msg)+1;

	// Make sure the handle is valid
	if( !hmo )
		return MMSYSERR_INVALHANDLE;

	// Get the length of the data segment
	size = getMidiSize( *status );
	if( size < 0 )
		return MMSYSERR_NOERROR;

	// Wait until previous note has played, then lock 'device'
	while( hmo->playing ) usleep(1);
	hmo->playing = TRUE;

	// Running status message, only send the data bytes
	if( *status && *status == hmo->status && STAT_CHANNEL(*status) )
		writeMidi( hmo->device->fd, data, size );
	else
	{
		writeMidi( hmo->device->fd, status, size+1 );
		// Set running status
		if( STAT_CHANNEL(*status) )
			hmo->status = *status;
	}

	// Unlock device because we are done with it
	hmo->playing = FALSE;

	return MMSYSERR_NOERROR;
}

// Write a long (sysex) message to the midi output 'device'
// hmo		-- Handle to the MIDI output device.
// lpMidiOutHdr		-- Pointer to a MIDIHDR structure that identifies the buffer to be prepared.
// cbMidiOutHdr		-- Size, in bytes, of the MIDIHDR structure.
MMRESULT midiOutLongMsg( HMIDIOUT hmo, LPMIDIHDR lpMidiOutHdr, UINT cbMidiOutHdr )
{
	// Make sure the handle is valid
	if( !hmo )
		return MMSYSERR_INVALHANDLE;

	// Wait until the device is free
	while( hmo->playing ) usleep(1);
	hmo->playing = TRUE;

	// Reset the running status... since we don't know the state after the 'long message'
	// (which could contain any amount of any type of data)
	hmo->status = 0;
	writeMidi( hmo->device->fd, (BYTE *) lpMidiOutHdr->reserved, lpMidiOutHdr->dwBytesRecorded );
	// Execute the callback, telling it that the message is done
	(*hmo->callback)( hmo, MOM_DONE, hmo->instance, (DWORD) lpMidiOutHdr, 0 );

	// Unlock the device
	hmo->playing = FALSE;
```

```
    return MMSYSERR_NOERROR;
}


// Open a MIDI input 'device'
MMRESULT midiInOpen(
      LPHMIDIIN lphmi, UINT uDeviceID, DWORD dwCallback, DWORD dwCallbackInstance, DWORD
dwFlags )
{
    // Initiate the device array if it isn't already
    if( !MIDIDEV )
            init_midi();

    // Make sure we are opening an ok device
    if( uDeviceID >= midiInGetNumDevs() )
            return MMSYSERR_BADDEVICEID;

    // Make sure this device isn't alreayd open for input
    if( MIDIDEV[uDeviceID].midiIn )
            return MMSYSERR_ALLOCATED;

    // CALLBACK_FUNCTION is the only valid flag
    if( dwFlags != CALLBACK_FUNCTION )
            return MMSYSERR_INVALPARAM;

    // Allocate memory for this device
    if( (*lphmi = (HMIDIIN) malloc(sizeof(struct midiIn))) == NULL )
            return MMSYSERR_NOMEM;

    // Open the actual device if it isn't already
    if( !MIDIDEV[uDeviceID].fd )
            if( openMidiDev(uDeviceID) )
                    return MMSYSERR_NOMEM;

    // Link the input device to the actual device
    MIDIDEV[uDeviceID].midiIn = *lphmi;
    (*lphmi)->device  = &(MIDIDEV[uDeviceID]);

    // Mark the device as running, and store the attributes
    (*lphmi)->running = TRUE;
    (*lphmi)->callback          = (void *) dwCallback;
    (*lphmi)->instance          = dwCallbackInstance;
    (*lphmi)->task              = 0;

    return MMSYSERR_NOERROR;
}

// Prepare a MIDI input header (don't need to do anything)
MMRESULT midiInPrepareHeader( HMIDIIN hmi, LPMIDIHDR lpMidiInHdr, UINT cbMidiInHdr )
{
    return MMSYSERR_NOERROR;
}

// Unprepare a header (again, nothing to do)
MMRESULT midiInUnprepareHeader( HMIDIIN hmi, LPMIDIHDR lpMidiInHdr, UINT cbMidiInHdr )
{
    return MMSYSERR_NOERROR;
}

// Get the input capabilities of a MIDI device (same as normal caps)
MMRESULT midiInGetDevCaps( UINT uDeviceID, LPMIDIINCAPS lpMidiInCaps, UINT cbMidiInCaps )
{
```

```
    return midiGetDevCaps( uDeviceID, lpMidiInCaps, cbMidiInCaps );
}

// Get the number of input devices
UINT midiInGetNumDevs( void )
{
    return midiGetNumDevs();
}

// Adds a new input buffer for use by the input device
MMRESULT midiInAddBuffer( HMIDIIN hMidiIn, LPMIDIHDR lpMidiInHdr, UINT cbMidiInHdr )
{
    // Make sure we are using a valid handle
    if( !hMidiIn )
            return MMSYSERR_INVALHANDLE;

    // Mke sure the new buffer we are trying to add is realy free
    if( lpMidiInHdr->dwFlags & MHDR_INQUEUE )
            return MIDIERR_STILLPLAYING;

    // Wait until the MIDI input 'device' is not busy
    while( hMidiIn->lock ) usleep(1);
    hMidiIn->lock = TRUE;

    // Add the new buffer after the first (only the first buffer will ever be in use)
    if( hMidiIn->buffer )
    {
            lpMidiInHdr->lpNext = hMidiIn->buffer->lpNext;
            hMidiIn->buffer->lpNext = lpMidiInHdr;
    }
    // If this is the first buffer, just add it
    else
    {
            lpMidiInHdr->lpNext = hMidiIn->buffer;
            hMidiIn->buffer = lpMidiInHdr;
    }

    // Unlock the device
    hMidiIn->lock = FALSE;

    return MMSYSERR_NOERROR;
}

// Start the MIDI input device
MMRESULT midiInStart( HMIDIIN hMidiIn )
{
    DWORD buf[MAXPNAMELEN], r = MAXPNAMELEN;

    // Make sure the handle is valid
    if( !hMidiIn )
            return MMSYSERR_INVALHANDLE;

    // The device is already started if it has a thread assigned to it
    if( hMidiIn->task )
            return 0;

    // Clear the input device -- keep reading from input until there is nothign left
    while( r >= MAXPNAMELEN )
    {
            fd_set rfds;
            struct timeval timeout;
```

```
                timeout.tv_sec = 0;
                timeout.tv_usec = 1;

        FD_ZERO( &rfds );
                FD_SET( hMidiIn->device->fd, &rfds );
        r = select( (hMidiIn->device->fd)+1, &rfds, NULL, NULL, &timeout );

        if( r )
                r = read( hMidiIn->device->fd, buf, MAXPNAMELEN );
    }

    // Mark the device as running, and start the thread
    hMidiIn->running  = TRUE;
    hMidiIn->status             = 0;
    pthread_create( &(hMidiIn->task), NULL, midiInThread, (void *) hMidiIn );

    return MMSYSERR_NOERROR;
}

// Stop a MIDI input device
MMRESULT midiInStop( HMIDIIN hMidiIn )
{
    // Valid handle?
    if( !hMidiIn )
        return MMSYSERR_INVALHANDLE;

    // If the device isn't running... our work is done
    if( !hMidiIn->running )
        return MMSYSERR_NOERROR;

    // Set the device to stopped, and stop the polling thread
    hMidiIn->running = FALSE;
    pthread_join( hMidiIn->task, NULL );
    hMidiIn->task = 0;

    return MMSYSERR_NOERROR;
}

// Reset the input device (remove it's buffers)
MMRESULT midiInReset( HMIDIIN hMidiIn )
{
    struct midiHdr *cur, *next;

    // Valid handle?
    if( !hMidiIn )
        return MMSYSERR_INVALHANDLE;

    // If there are no buffers, our work is already done
    if( !hMdidIn->buffer )
        return MMSYSERR_NOERROR;

    // Wait until the device is not locked
    while( hMidiIn->lock ) usleep(1);
    hMidiIn->lock = TRUE;

    // Clear all of the queued buffers
    cur = hMidiIn->buffer;
    while( cur )
    {
        next = cur->lpNext;
        hMidiIn->callback( hMidiIn, MIM_LONGERROR, hMidiIn->instance, (DWORD) cur, 0 );
        cur = next;
```

```
    }

    hMidiIn->buffer = NULL;

    // Unlock the device
    hMidiIn->lock = FALSE;

    return MMSYSERR_NOERROR;
}

// Close a MIDI input device
MMRESULT midiInClose( HMIDIIN hMidiIn )
{
    // Valid handle?
    if( !hMidiIn )
        return MMSYSERR_INVALHANDLE;

    // Stop the MIDI input device
    if( midiInStop(hMidiIn) != MMSYSERR_NOERROR )
        return MIDIERR_STILLPLAYING;

    // There shouldn't be any buffers in the queue (reset first)
    if( hMidiIn->buffer )
        return MIDIERR_STILLPLAYING;

    // Tell the program that we've closed the input
    (*hMidiIn->callback)( hMidiIn, MIM_CLOSE, hMidiIn->instance, 0, 0 );
    hMidiIn->device->midiIn = NULL;

    // Close the actual device if we are done with it
    if( !hMidiIn->device->midiOut )
        closeMidiDev( hMidiIn->device );

    // Free the memory
    free( hMidiIn );

    return MMSYSERR_NOERROR;
}

// Read <size> bytes (or until end of message) in from the MIDI input
BYTE readMidi( HMIDIIN hmi, BYTE *data, int size)
{
    int i = 0;

    // Keep reading until we have everything
    while( i < size )
    {
        read( hmi->device->fd, data+i, sizeof(BYTE) );

        // If it is a real-time message, send it to the program, and just keep running
        if( STAT_REALTIME(*(data+i)) )
        {
            DWORD message = (DWORD) *(data+i);
            hmi->callback( hmi, MIM_DATA, hmi->instance, message,
                    1000 * (time(NULL) - start_time) );
        }
        // If we hit another status byte, this message is done
        else if( STAT(*(data+i)) )
            // Pass this new status byte
            return *(data+i);
        // Otherwise just move on to the next byte
        else
```

```
                        i++;
    }

    return MMSYSERR_NOERROR;
}

// Poll the MIDI input device and deal with input
void *midiInThread( void *t )
{
    DWORD message;
    BYTE temp;
    int size;

    // Status byte is the first byte of a message
    BYTE *status = (BYTE *) &message;
    BYTE *data = status + 1;
    HMIDIIN hmi = (HMIDIIN) t;
    LPMIDIHDR hdr;
    BOOL long_data = FALSE, no_buffer = FALSE;

    // Wait for input... loop until the device is stopped
    while( midiInWait( hmi ) )
    {
        message = 0;
        // Read in the status byte
        read( hmi->device->fd, status, sizeof(BYTE) );

        // We are still recieving long data (sysex)
        if( long_data && !STAT_REALTIME(*status) )
        {
            // Mark that there is no buffer to save in
            if( !hmi->buffer )
                no_buffer = TRUE;

            // If there is no buffer, we have nothing to save
            if( no_buffer )
            {
                // Keep looping, because this is a data byte
                if( !STAT(*status) )
                    continue;

                // The byte was a status byte, so this is no longer a sysex message
                // We don't stop here, because we have to handle the status byte
                long_data = no_buffer = FALSE;

                // If the status byte was just ending the sysex
                if( *status == STAT_EOX )
                    continue;
            }
            else
            {
                // Record any data byte, or EOX
                if( !STAT(*status) || *status == EOX )
                    *(hmi->buffer->lpData + hmi->buffer->dwBytesRecorded++) = *status;

                // If the buffer is full, or the sysex ends (with a status byte)
                if( hmi->buffer->dwBytesRecorded >= hmi->buffer->dwBufferLength
                                                                    || STAT(*status) )
                {
                    // Take the full buffer out of the queue
                    hdr = hmi->buffer;
                    hmi->buffer = hdr->lpNext;
```

```
                        // Send the buffer to the callback
                        hmi->callback(hmi, MIM_LONGDATA, hmi->instance,(DWORD) hdr, 0);

                        // We are finished with this sysex if it was a status byte
                        if( STAT(*status) )
                                long_data = FALSE;

                        // Go on if it was a status byte
                        if( !STAT(*status) || *status == EOX )
                                continue;
                }
                else
                        continue;
        }
}

// We need to watch for the case of one sysex message being ended by another
if( *status == STAT_SYSEX )
{
        // Start a new sysex message
        long_data = TRUE;

        // We can't save anything if there is no buffer
        // We will not save any part of the sysex, even if a buffer gets added
        // (because it wouldn't make any sense to keep only part fo the sysex message)
        if( !hmi->buffer )
                no_buffer = TRUE;
        else
        {
                // Add this bit to the buffer
                *(hmi->buffer->lpData) = *status;
                hmi->buffer->dwBytesRecorded++;

                // If this fills the buffer, send it to the callback
                if( hmi->buffer->dwBytesRecorded >= hmi->buffer->dwBufferLength )
                {
                        hdr = hmi->buffer;
                        hmi->buffer = hdr->lpNext;
                        hmi->callback(hmi, MIM_LONGDATA, hmi->instance,(DWORD) hdr, 0);
                }
        }

        continue;
}

// Running status -- there is no status byte on the message
if( !STAT(*status) )
{
        // We are not currently in running status, so ignore the byte
        if( !STAT_CHANNEL(hmi->status) )
                continue;

        // Set the status to the running status
        *data = *status;

        // Ge the size of this type of message
        size = getMidiSize(hmi->status) - 1;

        // Fill in the rest of the data
        *status = readMidi( hmi, data+1, size );
```

```
                        // Unless we didn't get all of the data (ran into a status byte in the middle
                        // of it) we send this message on the the callback.
                        if( !(*status) )
                        {
                                // Set the status to the running status
                                *status = hmi->status;
                                hmi->callback( hmi, MIM_DATA, hmi->instance, message,
                                                1000 * (time(NULL) - start_time) );
                                continue;
                        }
                }

                // Keep looping until we are able to grab a good (full) message
                do
                {
                        if( (size = getMidiSize(*status)) < 0 )
                                break;

                        temp = readMidi( hmi, data, size );
                        if( temp )
                                *status = temp;
                } while( temp );

                // Set running status
                if( !STAT_REALTIME(*status) )
                                hmi->status = *status;

                // Only admit to having recived good messages
                if( size >= 0 )
                        hmi->callback( hmi, MIM_DATA, hmi->instance, message,
                                        1000 * (time(NULL) - start_time) );
        }

        return 0;
}

// This function blocks for input on the given MIDI device
BOOL midiInWait( HMIDIIN hmi )
{
        fd_set rfds;
        int timedout;
        struct timeval timeout;

        // We'll check to make sure we aren't supposed to be quiting, at most
        // once a millisecond.
        timeout.tv_sec = 0;
        timeout.tv_usec = 1000;

        do
        {
                // Set the file file to check
                FD_ZERO( &rfds );
                FD_SET( hmi->device->fd, &rfds );

                // Time out after 1 millisecond
                timedout = !select( (hmi->device->fd)+1, &rfds, NULL, NULL, &timeout );

        // Loop if the select timed out
        } while( hmi->running && timedout );

        return hmi->running;
}
```

C.4 WinRT.c

```
/***
 * The source for functions needed to port from Windows to RTL.
 * Implements: global (handle) memory, windows, and message passing
 ***/

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "windows.h"
#include "WinRT.h"

struct globalMemQueue
{
	BOOL lock;
	struct globalMem *mem;
};

extern time_t start_time;

struct globalMemQueue GLOBALMEM;
struct globalMem *FREEMEM = NULL;

struct msgque MSGQUEUE;
MSG free_msg = NULL;

WNDCLASS *CLASSQUEUE = NULL;

// Allocate some memory
HGLOBAL GlobalAlloc( int nadda, size_t size )
{
	struct globalMem *mem;
	void *ptr;

	ptr = (void *) malloc( size );
	if( ptr == NULL )
		return NULL;

	// Wait for the memory queue to unlock
	while( GLOBALMEM.lock ) usleep(1);
	GLOBALMEM.lock = TRUE;

	// Get a memory struct (recycled if possible)
	if( FREEMEM )
	{
		mem = FREEMEM;
		FREEMEM = mem->next;
	}
	else
		mem = (struct globalMem *) malloc( sizeof(struct globalMem) );

	// Remember where the memory is, and how big
	mem->ptr = ptr; mem->size = size;

	// Place the struct in the queue
	mem->next = GLOBALMEM.mem;
	mem->prev = NULL;
	if( GLOBALMEM.mem )
```

```
                GLOBALMEM.mem->prev = mem;
        GLOBALMEM.mem = mem;

        // Unlock the queue
        GLOBALMEM.lock = FALSE;

        // Return a pointer to this struct
        return mem;
}

// Give the actual location in memory
void *GlobalLock( HGLOBAL ptr )
{
        if( ptr )
        return ptr->ptr;

    return NULL;
}

// Get the handle for this block of memory
HGLOBAL GlobalHandle( void *ptr )
{
        HGLOBAL hGlobal;

        // Lock the queue
        while( GLOBALMEM.lock ) usleep(1);
        GLOBALMEM.lock = TRUE;

        // Search through the handles for this pointer
        for( hGlobal = GLOBALMEM.mem; hGlobal && hGlobal->ptr != ptr; hGlobal = hGlobal->next );

        // Unlock the queue
        GLOBALMEM.lock = FALSE;

        return hGlobal;
}

// Reallocate a new block of memory
HGLOBAL GlobalReAlloc( HGLOBAL hMem, size_t dwBytes, UINT uFlags )
{
        void *new_ptr;

        if( !hMem )
                return NULL;

        // Free the handle if we are reallocating it to 0 blocks
        if( !dwBytes  )
        {
                FreeGlobalMem16(hMem->ptr);
                return NULL;
        }

        // Lock the queue
        while( GLOBALMEM.lock ) usleep(1);
        GLOBALMEM.lock = TRUE;

        // Can't do anything if the handle isn't valid
        if( hMem->ptr == NULL || hMem->size == 0 )
                return NULL;

        // Actualy reallocate
        new_ptr = realloc( hMem->ptr, dwBytes );
```

```
    if( new_ptr )
    {
            hMem->ptr = new_ptr;
            hMem->size = dwBytes;
    }

    // Unlock the queue
    GLOBALMEM.lock = FALSE;

    // Return the handle if we were successful
    return new_ptr ? hMem : NULL;
}

// Get the size of the handled block
DWORD GlobalSize( HGLOBAL hMem )
{
    if( !hMem || hMem->ptr == NULL )
            return 0;

    return hMem->size;
}

// Free a block of memory
void FreeGlobalMem16( void *ptr )
{
    HGLOBAL hGlobal;

    if( !ptr )
            return;

    // Find the handle (if there is one)
    hGlobal = GlobalHandle(ptr);

    // Lock the queue
    while( GLOBALMEM.lock ) usleep(1);
    GLOBALMEM.lock = TRUE;

    // If there is a handle for this pointer, deal with the handle
    if( hGlobal )
    {
            // Route the queue around the handle
            if( hGlobal->prev )
                    hGlobal->prev->next = hGlobal->next;
            else
                    GLOBALMEM.mem = hGlobal->next;

            if( hGlobal->next )
                    hGlobal->next->prev = hGlobal->prev;

            // Set this handle to NULL, and put it in the recycle bin
            hGlobal->ptr = NULL; hGlobal->size = 0;
            hGlobal->next = FREEMEM;
            FREEMEM = hGlobal;
    }

    // Unlock the queue
    GLOBALMEM.lock = FALSE;

    // Free the block of memory
    free(ptr);
    return;
}
```

```
// Time since the program was started -- in mSeconds
// We are only looking at the time since the program started because otherwise
// the number would be too large
DWORD timeGetTime( void )
{
    return 1000 * (time(NULL) - start_time);
}

// Register a window class -- each window created has to use a created class
PWNDCLASS RegisterClass( const WNDCLASS *lpWndClass )
{
    PWNDCLASS wndClass;

    // Make sure we were given a class name
    if( !lpWndClass->lpszClassName || lpWndClass->lpszClassName[0] == '\0' )
            return 0;

    // Allocate memory
    wndClass = (WNDCLASS *) malloc( sizeof(WNDCLASS) );

    // Set the paramaters (most don't do anything)
    wndClass->style             = lpWndClass->style;
    // Pointer to the function which will be run when a message is recieved
    wndClass->lpfnWndProc   = lpWndClass->lpfnWndProc;
    wndClass->cbClsExtra    = lpWndClass->cbClsExtra;
    wndClass->cbWndExtra    = lpWndClass->cbWndExtra;
    wndClass->hInstance     = lpWndClass->hInstance;
    wndClass->hIcon             = lpWndClass->hIcon;
    wndClass->hCursor       = lpWndClass->hCursor;
    wndClass->hbrBackground = lpWndClass->hbrBackground;
    if( lpWndClass->lpszMenuName )
            wndClass->lpszMenuName = strdup(lpWndClass->lpszMenuName);
    else
            wndClass->lpszMenuName = NULL;
    wndClass->lpszClassName         = strdup(lpWndClass->lpszClassName);

    // Add this class to the queue
    wndClass->next                  = CLASSQUEUE;
    CLASSQUEUE                      = wndClass;

    return wndClass;
}

// Create a new window
HWND CreateWindow( char *lpClassName, char *lpWindowName, DWORD dwStlye, int x,
                                    int y, int nWidth, int nHeight, HWND hWndParent, HMENU hMenu,
                                    DWORD hInstance, void *lpParam )
{
   HWND newWnd;
   PWNDCLASS wndClass;

   // Find the class in the queue
   for( wndClass = CLASSQUEUE; wndClass
            && strcmp(wndClass->lpszClassName, lpClassName); wndClass = wndClass->next );

   // Is the class legal?
   if( !wndClass )
     return NULL;

   // Allocate the memory for this window
   newWnd = (HWND) malloc( sizeof(struct hwnd) );
```

```
    // Assign the parameters to the window
    newWnd->class = strdup( lpClassName );
    newWnd->name = strdup( lpWindowName );

    // When a message is sent to the window, this function will be run
    newWnd->proc = wndClass->lpfnWndProc;
    newWnd->assigned = TRUE;

    return newWnd;
}

// Destroy the given window
BOOL DestroyWindow( HWND hWnd)
{
    // Make sure the window handle is valid
    if( hWnd->assigned != 1 )
            return FALSE;

    hWnd->assigned = FALSE;

    // Free the memory used by this window
    free( hWnd );

    return TRUE;
}


// Posts a message to the message queue
BOOL PostMessage( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    MSG     newMsg;

    // If there are any free message structs, use one
    if( free_msg )
    {
            // Lock the queue
            while( MSGQUEUE.lock ) usleep(1);
            MSGQUEUE.lock = TRUE;

            // Grab a free message
            newMsg = free_msg;
            free_msg = newMsg->next;

            // Unlock the queue
            MSGQUEUE.lock = FALSE;
    }
    // Otherwise just allocate memory for a new struct
    else
            newMsg = (MSG) malloc( sizeof(struct smsg) );

    // Fill in the attribites the message (target window, message, paramaters, and time)
    newMsg->hWnd  = hWnd;
    newMsg->message = msg;
    newMsg->wParam          = wParam;
    newMsg->lParam          = lParam;
    newMsg->time    = time(NULL);
    newMsg->next    = NULL;

    // Lock the message queue
    while( MSGQUEUE.lock ) usleep(1);
            MSGQUEUE.lock = TRUE;
```

```
	// Place the message at the end of the queue
	if( MSGQUEUE.end )
		MSGQUEUE.end->next = newMsg;
	else
		MSGQUEUE.buffer = newMsg;

	// Mark the end of the queue
	MSGQUEUE.end = newMsg;

	// Unlock the queue
	MSGQUEUE.lock = FALSE;

	return TRUE;
}

// Get the next message for the given window
// This function will wait until a message is received
// Filters are not implemented
int GetMessage( MSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax )
{
	MSG	curMsg;
	struct smsg **preMsg;

	// Make sure we were given an ok message to place the received message into
	if( lpMsg == NULL )
	return -1;

	// Just grab the next message
	if( !hWnd )
	{
		// Loop to wait for a message to read -- usleep to volintarily give up CPU control
		while( !MSGQUEUE.buffer || MSGQUEUE.lock ) usleep(1);

		MSGQUEUE.lock = TRUE;

		// Fill the passed message structure with the first message, and remove the
		// message from the queue
		curMsg			= MSGQUEUE.buffer;
		lpMsg->hWnd		= MSGQUEUE.buffer->hWnd;
		lpMsg->message		= MSGQUEUE.buffer->message;
		lpMsg->wParam		= MSGQUEUE.buffer->wParam;
		lpMsg->lParam		= MSGQUEUE.buffer->lParam;
		lpMsg->time		= MSGQUEUE.buffer->time;
		MSGQUEUE.buffer		= MSGQUEUE.buffer->next;
	}
	else
	{
		// Loop while waiting for a new message
		while( TRUE )
		{
			// Give up cpu control while waiting
			while( !MSGQUEUE.buffer || MSGQUEUE.lock ) usleep(1);
			MSGQUEUE.lock = TRUE;

			for( preMsg = &(MSGQUEUE.buffer), curMsg = MSGQUEUE.buffer;
				curMsg && curMsg->hWnd != hWnd;
preMsg = &(curMsg->next), curMsg = curMsg->next );

			if( curMsg )
				break;
```

```
                MSGQUEUE.lock = FALSE;

                // Give up control of the CPU before we loop
                usleep(1);
            }

            // Fill the supplied message struct
            lpMsg->hWnd     = curMsg->hWnd;
            lpMsg->message  = curMsg->message;
            lpMsg->wParam   = curMsg->wParam;
            lpMsg->lParam   = curMsg->lParam;
            lpMsg->time     = curMsg->time;
            *preMsg         = curMsg->next;
    }

    // Update the queue
    if( !MSGQUEUE.buffer )
            MSGQUEUE.end = NULL;

    // Add the removed message to the free queue
    curMsg->next = free_msg;
    free_msg = curMsg;

    // Unlock the queue
    MSGQUEUE.lock = FALSE;

    if( lpMsg->message == WM_QUIT )
            return 0;
    else
            return TRUE;
}

// Sends a message directly to the window's processing function
// Specify the arguments of the message directly
LRESULT SendMessage( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    if( !hWnd || hWnd->assigned != 1 )
            return 0;

    return (*hWnd->proc)( hWnd, msg, wParam, lParam );
}

// Send a message struct to the window's processing function
LRESULT DispatchMessage( const MSG lpmsg )
{
    if( !lpmsg || !lpmsg->hWnd || lpmsg->hWnd->assigned != 1 )
            return 0;

    return (*lpmsg->hWnd->proc)( lpmsg->hWnd, lpmsg->message, lpmsg->wParam, lpmsg->lParam );
}

// Handles messages not handled by the user-defined WindowProc
LRESULT DefWindowProc(HWND hWnd, UINT iMessage, WPARAM wParam, LPARAM lParam)
{
    return TRUE;
}

// Tool function: get a line from a file descriptor
BOOL get_line( FILE *fd, LPSTR outstr )
{
    char ch;
```

```
    // Eliminate leading white-space
    while( isspace(ch = getc(fd)) );

    while( ch != -1 && ch != '\n' )
    {
            *outstr = ch;
            outstr++; ch = getc(fd);
    }

    *outstr = '\0';

    // Return TRUE if this is a normal line, FALSE if it's the last line of the file
    return ch == '\n' ? TRUE : FALSE;
}

// No effect, because we define an infinite message queue
void SetMessageQueue( UINT queueSize )
{
    return;
}

// Copy the string (just redefining the windows function)
LPSTR lstrcpy( LPSTR str1, LPSTR str2)
{
    return strcpy( str1, str2 );
}
```

Bibliography

Maximum MIDI : advanced music applications in C++; Messick, Paul;
©1998 Greenwich : Manning.

The Lexis-Nexis Archives: http://www.lexis-nexis.com;
©2000 Lexis-Nexis, a division of Reed Elsevier Inc. All rights reserved.

Financial Times (London), ©1999 The Financial Times Limited.
May 23, 1999, Sunday; June 23, 1999, Wednesday USA EDITION

The New York Times, ©1999 The New York Times Company.
June 16, 1999, Wednesday, Late Edition; June 18, 1999, Friday, Late Edition;
August 12, 1999, Thursday, Late Edition

Los Angeles Times, ©1999 Los Angeles Times.
October 5, 2000, Thursday, Ventura County Edition

The Kansas City Star, ©2000 The Kansas City Star Co.
July 30, 2000, Sunday METROPOLITAN EDITION

The Houston Chronicle, ©1999 The Houston Chronicle Publishing Company.
November 26, 1999, Friday 2 STAR EDITION

The Journal of the Acoustical Society of America, ©2000 Acoustical Society of America.
May 1998 - Volume 103, Issue 5; October 1999 - Volume 106, Issue 4

Opera News, ©1996 Metropolitan Opera Guild, Inc.
March 2, 1996 - Volume 60, No. 12; May 1996 - Volume 60, No. 16

http://www.pocketcoach.com/, Pocket Coach Publications.

InfoTrac: http://www.infotrac.galegroup.com/, ©2000 Gale Group. All rights reserved.

Mix Online (Mix Magazine): http://www.mixonline.com/, ©1999 Intertec Publishing.

WPI Journal, Spring 1997, ©1997 Worcester Polytechnic Institute.

Frederick W. Bianchi: Associate Professor, Worcester Polytechnic Institute (WPI).