Toward Modular, Portable, Real-Time Software[1]

Roger B. Dannenberg and **Dean Rubine**

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
dannenberg@cs.cmu.edu

Spark L.P.
150 E. 58th St., 35th Floor
New York, NY 10155 USA
rubine@cs.cmu.edu

ABSTRACT: W is a systematic approach toward the construction of event-driven, interactive, real-time software. W is driven by practical concerns, including the desire to reuse existing code wherever possible, the limited real-time software support in popular operating systems, and system cost. W provides a simple, efficient, software interconnection system, giving objects a uniform external interface based on setting attributes to values via asynchronous messages. An example shows how W is used to implement real-time computer music programs combining graphics and MIDI.

1. Introduction

Real-time interactive music systems are among the most demanding computer applications. In addition to the usual algorithm, data structure, and implementation problems, interactive music systems must respond to input in milliseconds and deal with a variety of input and output media. Special structures are required to satisfy these real-time requirements, leading to programming styles not supported by ordinary software development tools or languages. As a consequence, interactive music applications tend to be hard to develop and hard to port to new environments.

We have developed a new software system called "W" to address these problems. W has roots in constraint-based systems (Myers, *et al.* 1990), object-oriented systems (Pope 1991), the CMU MIDI Toolkit (Dannenberg 1986), Formula (Anderson and Kuivila 1990), MAX (Puckette and Zicarelli 1991), and MIDI (Rothstein 1992), taking good ideas from each of these. The goal of W is to simplify software development by incorporating existing software and by encouraging a modular approach to new software development.

[1]Originally published as: Dannenberg and Rubine, "Toward Modular, Portable, Real-Time Software," in *Proceedings of the 1995 International Computer Music Conference*, International Computer Music Association (September, 1995), pp. 65-72.

What is W? W's primary function is to interconnect objects within programs. W is a software layer added to existing objects, modules, and libraries, resulting in "building blocks" that send and receive messages. When one object is connected to another, messages from the first are delivered to the second. Generally, these messages convey state changes, so changes to some field in the sending object are tracked by some field in the receiving object. A good mental model of this idea is a MIDI connection between a controller (the sending object) and a synthesizer (the receiving object).

2. An Example Programming Task

To describe W in more detail, we will show how W implements an example program specification. Our example is trivial, but it has many of the elements that make real implementation difficult. The specification is: *start and stop a MIDI sequence using graphical controls*. Let us examine the issues surrounding the implementation of this program.

2.1 Real-Time Issues

In this program, the timing of MIDI output is critical, so real-time performance is important. The interface involves graphics operations which can be slow. The real-time issue is to prevent graphical updates from interfering with MIDI timing. In this specific case, one could assume that no graphical updates take place while the MIDI sequence is playing, but this is not a general solution.

Another approach is to precompute timestamped MIDI data and send it to a special device driver that outputs messages according to their timestamps. This would simplify the programming task, but it is not always possible to precompute data. This approach is not a general solution either.

A general solution must preempt long-running, non-time-critical tasks to service the time-critical ones. A typical approach is to place time-critical computations within interrupt handlers. For example, the Apple MIDI Manager (Apple 1990) calls application functions from within system interrupts, and the Microsoft Windows API allows MIDI handling functions to be called from within system interrupts. The advantage of this structure is that interrupts preempt running applications, allowing MIDI processing to proceed with low latency. The disadvantage is that many system-dependent steps must be taken to prevent nested interrupts, coordinate interrupts from multiple sources (e.g. timers and MIDI interfaces), and communicate with the main application. These issues make the code system-specific and therefore non-portable.

This "application + interrupt routines" architecture is normally limited to two levels: time-critical and non-time-critical. It is unclear how to implement multiple tasks, each with different latency requirements (*e.g*. audio processing, MIDI processing, graphical animation processing, musical analyses, and user-interface processing). Finally, debuggers and print statements cannot be used to debug interrupt routines.

Another approach uses multiple tasks provided by a real-time operating system. At the least, this results in system-dependent communication between the graphical interface process and the sequencer process. Furthermore, these operating systems have relatively small markets; often fewer, more expensive choices exist for development systems and peripherals, especially MIDI and audio.

2.2 Modularity Issues

A great deal of useful software already exists. *Our goal is to reuse software written by others rather than write everything from scratch.* For the example discussed earlier, we want to use an existing library to provide the start and stop buttons and existing sequencer software (*e.g.,* the CMU MIDI Toolkit) to process MIDI data.

There is much to be learned by looking at a hardware example: Audio/Video systems, which include component stereo systems, professional audio equipment, and component video systems. These have few "data types," *i.e.* microphone signals, line-level signals, video signals, and MIDI signals. Consequently, there are few types of connectors, and components easily plug together to build systems. Components are designed by specialists, but they can be assembled into systems by non-experts. Components make few assumptions about what they are connected to.

Compare this to Object Oriented Programming which, according to conventional wisdom, supports reusability. There are many data types. Every object presents "jacks" in the form of methods, but the data types of these connection points are entire argument lists. Little can be plugged together without hand-crafting special purpose interface code, analogous to soldering special-purpose patch cables. Objects often work together according to specific, inflexible designs.

We advocate an alternative, based on the A/V systems analogy, where objects have an external interface consisting of (mostly) simple data types rather than complex methods. Almost every interface in our system consists of operations of the following form:

SET *Attribute* TO *Value*,

where *Attribute* is a Lisp-like symbol such as X, TEMPO, or FILENAME, and *Value* is a value of type integer, float, string, or boolean. With only these few types, it is much more likely that objects will be simple to interconnect.

As with A/V systems, our components have outputs as well as inputs. For example, a graphical button object has an "output port" from which changing values are sent. External to the button object, a connection can be made to the input of some other object, such as a sequencer's start/stop status. In contrast, object-oriented systems do not have "output ports" or externally defined connections between objects. Rather, they must hold references to message targets in internal variables.

The "boxes and wires" approach is only the first step toward modularity. Previous systems, such as MAX (Puckette and Zicarelli 1991), PCL (Teitelbaum 1985), and Bars and Pipes (Fey and Grey 1989), require programmers to work within tightly constrained runtime environments. This makes it difficult to reuse pre-existing software. In contrast, W places minimal restrictions on pre-existing software. In effect, W is "software glue" that joins modules together. Alternatively, W is a "universal wrapper" that adds a veneer to existing software so that it can communicate with other W software.

2.3 Portability Issues

One of the most perplexing real-time problems is portability. Operating systems and tools change rapidly; applications that are not portable often become obsolete just as they mature into something interesting. Portability is harder with real-time applications because they make heavy use of system-dependent features to achieve real-time performance. Graphical interfaces also lead applications to become system-dependent because nearly every operating system offers a different graphics environment.

W promises to quickly leverage existing software and device drivers for graphics, MIDI, audio, and other components that could otherwise take months to implement in a new system. Of course, any well-designed application could offer reusability, but it is not clear that anyone understands how to do this. This may be because the interesting interfaces involve more than simple subroutine calls. Music applications are organized around graphical user interface toolkits, software interrupts, multiple processes, shared address spaces, callbacks, and other operating-system-dependent features. W abstracts away these system differences.

W also encourages developing objects with simple, well-defined interfaces. If all MIDI output is achieved via a connection to a MIDI output object, programmers will be less inclined to make system-dependent calls to lower-level MIDI interface functions. At the same time, the designer of the W MIDI output module will naturally think about how to define the module in a portable, system-independent fashion. W guides the developer toward more portable systems.

3. Implementing the Example Program

Our example assumes that a library of W objects is available, including buttons and sequencers. This assumption makes the implementation very simple. Later we will look at how to add new objects to W.

3.1 Creating and Connecting Objects

Assume W contains graphical button objects, a sequencer, and MIDI output. The following (slightly simplified) code creates a window and button, and provide labels for two button states. The "Set" functions (WSetW, WSetString, etc.) set attributes, e.g. wa_onlabel, to values, e.g. "Stop":

```
/* first create some W objects: */
window = WNew("aWindow");
wbutton = WNew("aButton");
/* put button in window: */
WSetW(To(wbutton), wa_parent, window);
WSetString(To(wbutton), wa_onlabel, "Stop");
WSetString(To(wbutton), wa_offlabel, "Start");
```

The first lines create instances of class aWindow and class aButton. Then attributes within these instances are initialized, *e.g.* the wa_parent attribute of button is set to the window. The reader may be puzzled by the To(wbutton) notation. Ordinarily, messages are sent from the output port of an object over pre-established connections. Thus, the first parameter to WSetW would normally be self, referencing the current object. However, no connections have been established, yet we still have to deliver messages. The notation To(wbutton) establishes a temporary connection from a "dummy" object to the desired target and returns the dummy object so that the message will be properly delivered.

We also need to create a sequencer and load a sequence (the sequence is loaded as a side effect of setting the filename attribute):

```
wseq = WNew("aSeq");
WSetString(To(wseq), wa_filename,
            "testseq.mid");
```

We want to connect the button to the slider, but there is a minor problem. The button sends messages of the form "SET ISON *Boolean*", but the sequencer uses the attribute RUNFLAG to start playing. For this purpose, W provides a special object called a Transformer to change the attribute name in a message and then forward the message to another object. Below, we create a transformer and make connections from the button through the transformer to the sequencer:

```
t = WNew(“aTransformer”);
WSetW(To(t), wa_to, wseq);
WSetW(To(t), wa_toattribute, wa_runflag);
WSetW(To(t), wa_from, wbutton);
WSetW(To(t), wa_fromattribute, wa_ison);
```

Connections are made as a side-effect of setting the TO and FROM attributes (coded as wa_to and wa_from) of the Transformer. An alternative way to make connections is to use WConnect(wbutton, t) and WConnect(t, wseq). The example is finished by connecting the sequencer to a MIDI output object. We will omit this step; it is similar to the steps already shown.

3.2 Objects and Connections

We now describe in detail how the code from the previous section works. W objects are referenced by 32-bit identifiers. The identifier has 3 fields, designating address space, zone, and index. Zones are described later. For simplicity, let us assume everything is implemented in one zone and address space. The index field is an index into a table of W objects. (See Figure 1.) If this were a typical object-oriented system, W would be a base class and every object in the system would inherit from it. In contrast, W is a “glue” layer added to existing objects, so there is no option of making every object a subclass of W objects. Instead, each W object contains a pointer to the “real” object such as a button, MIDI I/O object, or sequencer. W objects also contain the receive function address and list pointers that maintain inter-object connections. In Figure 1, W object 1 (a Button) is connected to objects 2 and 3 (sequence players). This figure differs from the code example in that the intermediate transformer object is omitted and there are two sequence objects.

When a message is sent, the send function iterates over the Senders list of connections and sends a copy of the message to each receiver. In this example, the receiver’s receive function is called immediately. In this way, messages can be passed on the

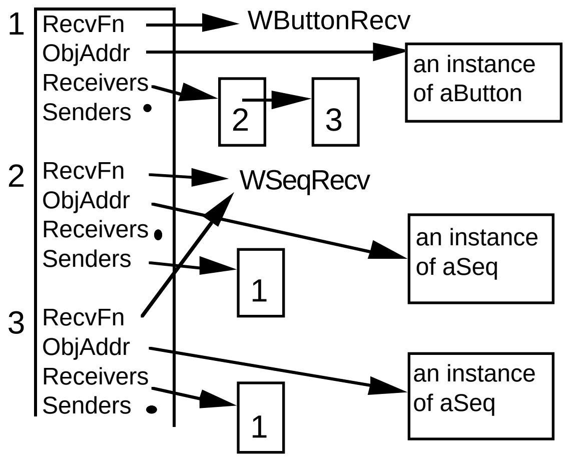

Figure 1. W runtime structure. Object 1 is connected to Objects 2 and 3, which are of the same type and hence share the receive function WSeqRecv. Note that no extra fields are added to pre-existing objects.

stack with no overhead for message buffering, copying, and scheduling the receiving task.

Where do messages originate? Any button implementation will have some provision for calling a function or method when the user clicks on the button. This function should call WSetBool(self, wa_ison, value), where self is the W reference for the button, wa_ison is the attribute to set, and value is the current state of the button.

3.3 Integrating Objects Into W

We now examine in detail how pre-defined objects can be used within a W system. In other words, how do we endow an existing object (*e.g.* a sequencer) with a W interface? We will use a simplified version of the sequencer object from the CMU MIDI Toolkit as an example. W uses a preprocessor to build interfaces to objects. The first step is to annotate the existing C structure definition (usually found in an “include” file) with some special comments. These comments (delimited by */*W* and *W*/*) are read by the preprocessor but ignored by the compiler:

```
struct seq_struct {
    boolean runflag;
    /*W in out whenset RunflagChanged W*/
    event_type event_list;
    long transpose; /*W in out W*/
```

```
        char filename[MAXTXT];
            /*W in out whenset SeqLoad W*/
        ... many more fields omitted ...
    }
```

These annotations inform the preprocessor what attributes should be included in the W interface. The preprocessor also defines the attribute names we have been using such as wa_onlabel and wa_from. The “in out” part specifies the object both receives and sends SET messages corresponding to that field. For each field, a “callback” function may be specified to be called each time the field is set. Thus, when runflag is set, RunflagChanged will be called. It might be defined as follows:

```
void RunflagChanged(seq_type seq)
{
    if (seq->runflag) seq_play(seq);
    else seq_stop(seq);
}
```

3.4 Message Types

To complete the programming example, we must connect the sequencer output to a MIDI object, and W messages must carry the outgoing MIDI stream. To pass MIDI data between objects, we could use a SET message for each parameter (SET STATUS, SET PITCH, SET VELOCITY, etc.), but this approach is awkward. Instead, W supports messages of type MIDI in addition to type SET. W’s MIDI messages use the same mechanisms as SET for object interconnection and message delivery. W objects that exchange MIDI messages are similar to interconnected MIDI applications communicating via the Apple Midi Manager. W has built-in objects to construct MIDI messages from SETable attributes and to deconstruct MIDI messages into SET messages.

Another message type is GET. In some cases, it is necessary to retrieve values from W objects, and this is difficult with the asynchronous message-passing style described so far. For a more synchronous programming style, GET messages request the value corresponding to an attribute and return the value. The programmer sends a GET message using a function call which blocks until a reply is received. GET is not generally safe for real-time communication across zones, so it is used only for special purposes such as debugging and initialization.

4. Real-Time Support

In the example program, it is not good to use the same process to handle the graphical user interface and the MIDI sequencer because slow graphics operations will interfere with the production of accurately timed MIDI. Elsewhere (Dannenberg 1993), we have advocated the architecture illustrated in Figure 2. This architecture is intended to support interactive, event-driven real-time systems. W adheres to this architecture, where the system is divided into *zones*, each with an associated latency. Tasks are allocated to zones according to their computational latency requirements. For example all tasks with very low latency requirements are placed in a low-latency zone, and all high-latency tasks in a high-latency zone. In our model, a *task* is a computation that is invoked to handle an external event such as W message arrival, internal events such as a condition becoming true, or timer events such as the passage of a 10ms interval. Although “task” conventionally means “process,” we are using it to mean something as simple as a subroutine call.

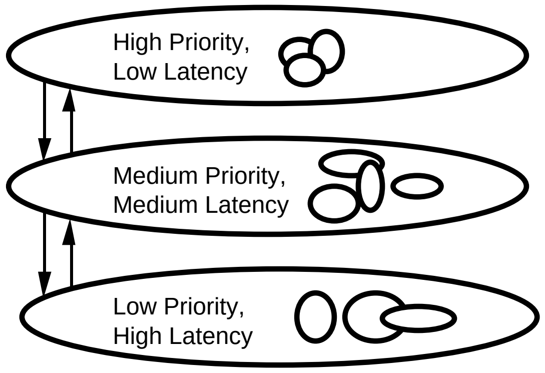

Figure 2. A real-time system composed of three zones, each of which contains a number of tasks. Communication between zones is by messages (with no shared data).

Tasks are serviced by a single thread of control per zone; thread priority increases with decreasing latency requirements. In other words, tasks which must be computed with the smallest amount of delay will preempt tasks with less critical timing requirements. Our model assumes fixed-priority, preemptive scheduling, but other schedulers could be made to work. Furthermore, a zone’s “thread of control” can be a software interrupt as well as a process, provided the preemption and priority assumptions are met.

Communication between zones is exclusively through messages. There are no shared data structures visible to the programmer. An important implication is that W objects, as with other data structures, must each reside in only one zone. Consequently, all operations on a W object must be compatible with the latency associated with the object's zone. Each operation must have a worst-case computation time which is small compared to the latency of the zone, and each operation must tolerate the latency of the zone. We have not found this restriction to be a problem, although sometimes, objects must be designed with this restriction in mind.[1]

Zones provide two programming simplifications. First, tasks within a zone run to completion without preemption (each zone has one thread of control). Thus, there is never concurrent access to data structures and there is no need for locks, semaphores, and other safeguards. Each zone is effectively a monitor (Hoare 1974). (At some implementation level, zones may communicate through shared memory buffers and allocate from shared memory pools, so the *implementation* must use synchronization operations.)

Zones may preempt one another, but concurrent access to data structures is avoided even in this case. All communication between zones is restricted to message passing. Messages are processed only after the current task runs to completion because there is only one thread of control per zone. This is similar to the stack-based scheme called X (Puckett 1986), but less restricted because messages can pass from high-priority objects to low-priority objects.

Another nice property of a zone is that tasks are completed sequentially in the order of arrival. This means, for example, that the processing of a MIDI note-on will complete before the processing of the corresponding note-off. Overall, this programming approach has the flavor of sequential programming, which is much simpler than concurrent programming.

1. For example, loading a sequence may be very slow compared to the desired latency of a sequence player. It makes sense to separate sequence readers from sequence player objects and place them in different zones. Sequence data, once loaded, can be delivered by a W message from the reader object to the player object. To avoid copying large sequence structures between zones, the message can carry a pointer so long as the reader and player never attempt to access the structure simultaneously.

4.1 Zones in W

Zones play a key role in the design of W. Zones demand loosely-coupled tasks communicating asynchronously through buffered messages. Only asynchronous communication allows an object to communicate without blocking, which would have adverse effects on real-time performance. We also want one programming paradigm that handles both intra-zone and inter-zone communication. The message-passing style of W is contrary to the synchronous style of object-oriented programming, but it fits our requirements.

When communication is within a zone, messages are passed efficiently on the stack. The measured time to fully process a simple SET message on an 80Mhz IBM Power PC is under 7µs. Buffering and copying is necessary only when messages travel to another zone.

Recall that references to W objects have three fields: the address space, the zone, and the object index. Within a zone, the receive function is located directly in the object table and called. To reach another zone in the same address space, every address space has a table of "zone objects" which transfer messages from one zone to another through message buffers. To reach another address space, there is a similar table of "address space objects" which transfer messages across address spaces. Message transfer may use any mechanism, including shared memory, sockets, networks, or even MIDI.

4.2 Time and Scheduling

All messages carry a timestamp representing the time at which the message should be acted upon. Timestamps may be interpreted in two ways. First, the timestamp can be taken to mean real time. In this case, a message with a future timestamp is delayed and delivered at the indicated time. Second, the timestamp can be used to implement precomputation. In this case, messages are delivered slightly ahead of time so that computation can take place before the indicated time rather than after. The interpretation of timestamps is a property of a zone. By combining a precomputation zone with a real-time zone, event buffering (Anderson and Kuivila 1986) can be implemented (see Figure 3). The precomputation zone (A) runs ahead of real-time, producing output messages with future timestamps. These messages (e.g. MIDI events) are buffered by the real-time zone (B) until the indicated timestamp. Message delivery according to timestamps is the responsibility of the W message system, so every zone must provide time-based scheduling and dispatch of messages (Dannenberg 1989).

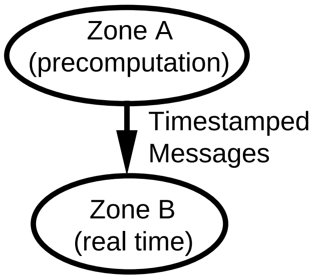

Figure 3. W implements event buffering by combining a precomputation (time advance) zone with a real-time zone.

5. Implementation Details

A receive function is required for each type of W object, and receive functions are error-prone to implement. The receive function must decode the message type (GET, SET, MIDI, etc.) and field names (ISON, RUNFLAG, etc.). Our preprocessor compiles a table of field descriptors for each object type. At initialization time, this table is used to build a hash table for quick lookup.

W provides an interesting mechanism for object creation. At initialization time, "class" objects are created, representing the types of objects in the system. To create an instance of a class, a GET NEW message is sent to the desired class object. The class object creates the instance and returns its W reference.

Entire systems of interconnected W objects can be saved to disk. This is facilitated by the SETALL message (yet another message type), which requests that an object send SET messages corresponding to all of its internal state. These messages are captured by an archiving object and written to a file. Later, the file can be interpreted to recreate a network of W objects. SET messages are used to restore the internal state of every object that was archived.

Zones can be implemented in two ways. In a multitasking operating system like Unix or NT, each zone can be implemented with a conventional thread or process. In a simpler operating system like the Macintosh or Microsoft Windows, the high-priority preemptive zone must be implemented as a software interrupt. W will allow software to be ported between these two rather different configurations because the details and differences are irrelevant to and hidden from W programs. In any operating system, graphical toolkits are likely to provide their own "event loop," so the W zone concerned with graphical interfaces will likely be driven by "callbacks" from user interface objects. Again, W provides great flexibility. The source of flexibility is primarily the fact that computations in W are relatively short execution sequences represented by message receive functions. These always execute to completion and return, so a zone's thread of control can come equally well from callbacks, interrupts, or processes. Communication between zones is strictly by messages, so any number of implementation schemes can be devised, depending upon the host system.

6. Debugging Support

Debugging real-time systems is difficult. Debuggers often provide little or no support for software interrupts and multiple processes. When they do, traditional debugging is not always applicable to real-time programs. With W, objects can be partially tested and debugged in a single zone where debuggers are most effective. Once objects function correctly, the programmer can switch to multiple zones to get reliable real-time performance. Little or no programming is required to make this switch. Sometimes, however, bugs appear only when code is running as an interrupt handler. With W's GET and SET messages, a programmer has some built-in debugging facilities to examine objects in interrupt handlers and objects running on remote machines. Furthermore, W connections can be added to monitor state changes and data streams, a practice familiar to MAX programmers. Because messages are asynchronous, connections that monitor data do not block processing. We expect little impact on real-time performance.

7. Conclusions

At present, W exists as a single-zone system with graphical user interface objects and MIDI I/O. The W kernel is about 5000 lines of C code, including the preprocessor. By the time of publication, we expect to have W running in a true real-time system with multiple zones.

W is a new approach to structuring interactive real-time systems. It draws inspiration from a number of earlier systems. The main goal of W is to use existing software wherever possible. A primary contribution is what W does *not* do. W does not provide threads, processes, user interfaces, device drivers, or interprocess communication. What W *does* provide is a mechanism to "glue" all these components together into a coherent system. W provides an abstract computation and communication environment, which maximizes portability. W is small and efficient. We believe it is a good solution to making effective use of existing operating systems, multimedia devices, and graphical user interface toolkits.

8. Acknowledgments

The authors would like to thank the School of Computer Science at Carnegie Mellon and the IBM Corporation for making this work possible. Belinda Thom provided valuable feedback on an earlier draft of this paper.

References

Anderson, D. P. and R. Kuivila. 1986. "Accurately Timed Generation of Discrete Musical Events." *Computer Music Journal* 10(3):48-56.

Anderson, D. P. and R. Kuivila. 1990. "A System for Computer Music Performance." *ACM Transactions on Computer Systems* 8(1):56-82, February.

Apple Computer, Inc. 1990. *MIDI Management Tools, Version 2.0*. Apple Programmers and Developers Association.

Dannenberg, R. B. 1986. "The CMU MIDI Toolkit." In *Proceedings of the 1986 International Computer Music Conference*, pages 53-56. International Computer Music Association, San Francisco.

Dannenberg, Roger B. 1989. "Real-Time Scheduling and Computer Accompaniment." In System Development Foundation Benchmark Series. *Current Directions in Computer Music Research*. Mathews, M. V. and J. R. Pierce, editors, MIT Press, pages 225-262.

Dannenberg, R. B. 1993. "Software Support for Interactive Multimedia Performance." *Interface Journal of New Music Research* 22(3):213-228, August.

Fey, T. and M. J. Grey. 1989. *Using Bars and Pipes*. Decatur, Georgia, 1989.

Hoare, C. A. R. 1974. "Monitors: An Operating System Structuring Concept." *Communications of the ACM* 17(10):549-557, October. Erratum in *Communications of the ACM*, 18(2) (Feb. 1975), page 95.

Myers, B. A., *et al.* 1990. "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces." *IEEE Computer* 23(11):71-85, November.

Pope, S. T. (editor). 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* Boston: MIT Press.

Puckette, M. 1986. "Interprocess Communication and Timing in Real-time Computer Music Performance." In P. Berg (editor), *Proceedings of the International Computer Music Conference 1986*, pages 43-46. International Computer Music Association.

Puckette, M. and D. Zicarelli. 1991. *MAX Development Package*. Palo Alto, CA.: Opcode Systems, Inc.

Rothstein, J. 1992. *MIDI: A Comprehensive Introduction*. Madison, WI: A-R Editions.

Teitelbaum, R. 1985. "The Digital Piano and the Patch Control Language System." In W. Buxton (editor), *Proceedings of the International Computer Music Conference 1984*, pages 213-216. International Computer Music Association.