More and more frequently we hear that real-time computer music performers would like to be able to do "high-level", real-time programming. Yet, the means; I mean, the music, I do not think we are going to be able to do "high-level", real-time programming. However, we should be able to do the processing. I am doing these things in hardware. I am implementing instructions that do sufficient, that parallelize the world into the "argot" and the "not argot", whereas the latest CPU/GPU combinations of tasks in a real application often cover a much wider spectrum. However, few computer musicians want to drive inter-rupt routines itself.

An important limitation in real-time systems in general has been the lack of effective ways to emulate small systems into larger ones. The problem of parallelization is often more acute than when there are conflicting real-time constraints on the objects regulating the sequences. The need for parallelization and the need for recall, for the reason of making the parallelism of a large system into a real system can sometimes be solved by programmatic means. The need for making the parallelism of a large system into a real system can sometimes be solved by programmatic means. Programmatic means are effective because they enable to operate on the execution of smaller tasks, large real-time systems which require a high level of interaction among tasks, especially through communication or resource-sharing, are hard to develop and verify. One idea is just the sort of system that computer musicians now want to build.

It is time to build a general real-time scheduler which makes these things easy to do. In fact, several research groups are now focusing on this problem of the real-time integration of programs. The complex systems that they design are able to be handled by programmatic means. Programmatic means allow us to forget about circuits, systems, code, memory segments, and the like, and attach message-passing to allow the most complete modularity we know how to put in a software system so far.

Central to current efforts to support more highly developed real-time software environments is the need for an integrated real-time protocol to handle communication and resource location, in a way which allows coherence that software will act as a collection of independent entities in situations where the designer did not have that challenge in mind. It is a collection of intercommunication and scheduling primitives simple enough to make the real-time interaction easy enough for the user, but powerful enough to meet great real-time needs. I think we need a model for doing this.

I would regard anything else, for example an operating system designed for the performance, for another language, as being oriented and inevitable by its nature. Fortunately we cannot implement much of these things that make it easy to do complicated things, but it is essential to make sure that everything in the design is not quite obvious. These three ideas form a kind of a basis for doing something different, and there is no profit in having a built-in notion of hierarchy, it is usually a hindrance. Instead, I would drop the idea of "continuously-running processes that create packets of things, do one packet after another", through the network layer. Third, there should be few devices. Rather, little computing I would keep it stable as an entity in a network of itself.

The MAX real-time system. Research in real-time systems is ongoing at many computer music centers and a new real-time system is no longer a rarity. Yet, musicians I would like to mention another one which has been in development, under various names for about five years and is ready today to be explained. This system is called "MAX" and uses a real-time computer music performer serving many simultaneous digital signal-processing control nodes. It is currently a soft real-time system, making no hard guarantees about meeting its deadlines; there are instructions based on it which might someday allow making it a "hard" real-time system. It currently runs on the MAC at IRCAM and with MIDI equipment at MIT.

In MAX the tasks are carried out by active objects which communicate through message-passing, MAX is also carried out through message-passing, hence real-time computing and interactive systems. A message sent from one object to another has an associated start time and stop time. The program knows its destination object to carry out some calculation; this calculation is engaged to some one no sooner than the start time and no later than the deadline. The calculation is carried out by the receiving object's method for the message. It may turn pass messages to other objects, which have their own start times and deadlines. A message is not required to be delivered immediately when it is sent, but only when the receiving object is ready to receive it. The calculation is carried out by the receiving object's method for the message. It may turn pass messages to other objects, which have their own start times and deadlines. A message is not required to be delivered immediately when it is sent, but only when the receiving object is ready to receive it.
offered through monitors. This simplicity is crucial for the design and verification of feasible real-time schedulers.

The power of the MAX paradigm is in its ability to specify more complex actions from simpler ones. In real-time applications, long streams of actions are required which can not only start and stop on command but adjust their speeds and other properties as a result of other stimuli. This type of scheduling is beyond the scope of straightforward graph-based models of real-time computation, and yet more complex implementations, such as those using MAX, can quickly become very hard to model reliably because of the unknown problem, which is a process which shares resources must be able to lock each other out, creating an often intractable system of constraints on the scheduler. In MAX's demonstration last year, model passing made it easier to deal with the constraint problem in real-time applications.

What a real-time system should be:

MAX in its current state has one major limitation in its ability to carry out tasks according to real-time constraints, which is its lack of context switching. Even if we send a message to an object not in MAX the method for that message is carried out by the system, regardless of whether more urgent tasks appear in the queue. This is a problem if the user has a way to handle urgent tasks. For example, a way described in the next paragraph, I will present the ideas in the form of a lower specification for a proposed real-time system, I call "MAX-Plus". MAX-Plus has a very simple real-time system which consists of a kernel and a collection of objects which carry out specific real-time tasks. The kernel does nothing but handle I/O requests from objects (sending messages to them, as I/O completion) and sends deferred messages passing between them.

The user starts up 3 by allocating all the objects he wants (which is fairly straightforward and simple). At startup, the objects do as much memory allocation and file I/O as can be done. Whatever I/O requests an object makes are handled by the kernel. The objects are known to the kernel in advance, since they will probably require memory allocation.

The deferred messages in X will have a start-time and a deadline as in MAX but only the start-time would be specified by the sending process. The deadline would be specified by adding the latency of the receiver process. This is in order to ensure that all messages sent to an object have the same latency, why we want this to be true is explained in the next paragraph.

Suppose a method is running when associated deadline is a second away, when I/O completes and generates a message whose deadline is 1 second from now. We need to suspend the slow method while the urgent one runs. This urgent method might yield other deferred messages in turn, causing a whole tree of method calls to fill off of higher urgency than that of the original method. We can do all these context-switched without fear as long as we never pass a message to the original object; its data may be in its immediate state since it is already being handled by another method. But if we yield in the context - switching while a message is sent to that object, will have a deadline later than the deadline already has, we will never want to carry out the method before finishing the suspended processing.

It would be desirable to pass some messages directly without pausing and scheduling them. This would not only reduce overhead but also allow the
The "schedule" object would replace the "play" CV in MAX. It takes a stream of "signal" messages to convert "message" to "readiness" to decide when to send the desired messages. It is implemented in C++.

The "message" object would replace the "note" CV in MAX. It takes a stream of "message" messages to convert "message" to "readiness" to decide when to send the desired messages. It is implemented in C++.

The "message" object would replace the "note" CV in MAX. It takes a stream of "message" messages to convert "message" to "readiness" to decide when to send the desired messages. It is implemented in C++.

The "message" object would replace the "note" CV in MAX. It takes a stream of "message" messages to convert "message" to "readiness" to decide when to send the desired messages. It is implemented in C++.
It is important to limit the computation overhead associated with the real-time scheduler. This overhead generally consists of the computation devoted to making scheduling decisions, context-switching between processes, and communication between processes. In this section, we describe a proposed implementation of an interrupt-driven scheduler for a time-sharing real-time system: it will be seen that the major cost associated with the scheduler comes from interprocess communication, a situation which suggests that algorithms which work well under this scheduler will also work well in distributed systems.

Deferred messages will appear when another object creates one (while it itself is handling a message) or as a result of 1/0 completion. They may or may not be immediately runnable, depending on their start-times. The scheduler that stores the runnable ones and the nonrunnable ones in separate pools at clock ticks messages in the nonrunnable-message pool may become runnable and are moved to the runnable-message pool either at interrupt time or as a scheduled task itself.

Suppose the intensities which arrive in the system are \( d_1 \) and \( d_2 \). A runnable task at latency \( d_1 \) appears as a message which is to be passed to an object of that intensity hence the scheduler keeps a pool of deliverable messages, which we organize according to latency as shown in the figure. A message is delivered to the object by creating a stack frame for the object and then executing the object as a procedure.

The scheduler keeps the runnable-message pool in the form of a separate queue for each intensity. The scheduler always sends the first message in the lowest intensity queue. A message can only be delivered by a method which returns the scheduler another message or no message. One of the only situations in which we need to deliver a method before its next is when 1/0 (including the clock) causes a low-latency message to appear. The figure shows the situation in which a message of intensity \( d_1 \) has appeared while a method of intensity \( d_2 \) was running. In this case the scheduler causes a software interrupt to occur by pushing a new stack frame onto the stack and executing the lower-latency method. When this method returns and another event message results from it where intensities are less than \( d_2 \) have been serviced) we pop the stack back to the prior frame at latency \( d_2 \) and resume the associated method.

Note that we do not have the buffer and express of maintaining several stacks. This will keep context switching over low and greatly ease debugging. The major cost associated with the implementation will be storing space for messages and copying them to the runnable-message pool. The scheduling queues themselves are quite inexpensive, especially if the number of different intensities required is small.

At the lowest level possible 1/0 should be handled inside the kernel, but we would like to work concurrently with "true" 1/0 and with the result of some other processing. For instance, if the object "exists" to parsing MIDI from a keyboards and we want switch number 5 being pressed to result in a message being sent to object "harp" containing the data "bang", one could say (next command will result in a "click" when the switch was just pressed):

- The reason for this bang is on

[ICMC 86 Proceedings] 46