FTS: A Real-time Monitor for Multiprocessor Music Synthesis

Miller Puckette
l'Institut de Recherche et Coordination Musique/Acoustique
31 Rue St. Merri, 75004 Paris, France
miller.puckette@ircam.fr

The IRCAM Music Workstation (IMW) is the first real-time computer music synthesis system based on a truly general-purpose processor, the Intel i860. The IMW's hardware consists of one or more NeXT hosts together with between two and 24 i860 coprocessors (CPs) running at 40 MHz, nominally capable of 80 MFLOPS apiece. An operating system, CPOS ("CoProcessor Operating System"), has been written specifically to fill the requirements this hardware poses for real-time musical synthesis and control. A distributed computer program, FTS ("faster than sound"), which runs under CPOS, manages the real-time calculations required both for control and for synthesis. This paper describes FTS and how it interacts with application software running on the host.

The i860 is the first inexpensive general-purpose processor powerful enough that we could consider basing a real-time computer music system on it. Before now, one had to resort to special DSP architectures, as was done for the 4X, IRCAM's previous adventure in real-time music synthesis (Favreau 1986). The 4X combines a general-purpose "control processor" with special synthesis hardware. A 4X application thus consists of two programs which must intercommunicate in real time: the "patch," which defines the heavy numerical calculations involved in computing sounds, and the "control program," which provides parameters for the patch, usually as a function of real-time control inputs. This separation was made out of necessity, not by choice. Forcing the user to maintain two programs, in different languages, whose state must nonetheless be kept coherent, greatly increases the effort required to develop a new application or to merge two existing ones.

Because of the IMW's homogeneous hardware design, a single distributed program running on the CPs, such as FTS, can do almost all of the real-time processing required (the exception is that a host computer is needed as a server to provide certain I/O). A great part of the difficulty of making music on the

4X and similar machines drops away instantly in using the the IMW. The problems of synchronization between a "smart" control processor and a "dumb" sound processor disappear entirely, leaving only the easier and more interesting problem of coordinating several, equal, high-level processors.

It is in the merging of pre-existing applications that this unification between control and synthesis makes the biggest difference. Users of the 4X have traditionally spent more time putting together new configurations of known techniques than in developing those techniques originally. Making a new 4X "patch" (the sound-making part of the application) which merely rearranges existing elements typically requires heavy reworking of source code. The control programs, in C, cannot simply be concatenated either; and their edition must agree with the edition of the patch. One of the most fundamental requirements that we have placed on the IMW is a much greater facility to juxtapose working pieces into working wholes. In FTS , the low-level real-time software base for the IMW, we have tried to lay a foundation which permits this kind of building-block functionality.

The greatest single difficulty in programming the IMW is that it is still, after all, a multiprocessor. FTS provides an explicit remote message-passing feature and an explicit mechanism for sending a continuous stream of samples from one processor to another. But FTS leaves it up to application software to try to hide the existence of the machine boundary – or simply to leave it explicit. FTS does, however, confront the problem of synchronizing many processors in real time in a deterministic way.

The Design Goals of FTS

A Music Workstation should be a good platform for rapid experimentation with new ideas. In the ideal, musicians with only a user's knowledge of computers could invent and experiment with their own techniques for synthesis and control. The "let-me-help-you" approach to user interface design, in which the computer tries to hide the implementation-level details of a given synthesis or compositional algorithm, is unsuitable here, since it ultimately takes the computer out of the musician's control. It is better to invite the user to understand everything down to the level of an oscillator or a live control input. The level of user training required is lower, and the result better reflects the personality of the musician instead of the system.

One broad category of activity that we wish to encourage is the invention of new user interfaces, either by programmers or even by the "computer-literate musician." Work in this area has resulted in two graphical programming environments, MAX (Puckette 1991), and ANIMAL (Lindemann 1991b). These programs have placed fairly specific demands on the communication facilities between the CPs and the host. They also demand a great deal of flexibility from the CPs, which must support the incremental building and editing of a running application. This implies a heavy use of dynamic interconnection between objects, and also the ability to load subroutines dynamically. MAX and ANIMAL also bring the building-block structure of FTS to user level, using as metaphors the idea of assembly and interconnection of smaller objects into larger ones. The sections of this paper on MAX and ANIMAL will illustrate this.

Our desire for interactive, modular construction of musical applications, and for the integration of synthesis and control is consistent with a relatively straightforward multi-tasking approach to programming the IMW. It will be seen that we do not need all the usual ornaments of a real-time multitasking system; we can do without context switching between tasks or explicit mutual exclusion. The only communication facility needed in the underlying operating system is access to a real-time "port" mechanism to send datagrams from one processor to another with bounded latency.

Historical Background

Music languages in the "Music-N" style can be seen as very simple object systems. The input is usually divided into instrument definitions and a list of "note cards." In more current lingo, the note cards are instance-creation messages to the instruments, which are classes with exactly one method—*create-instance*. There is no return value and the "voice" which is created runs without further control (there are usually tricks for getting around this restriction – starting another note which changes a shared global variable, for instance). The "parameter fields" of the note card are arguments to the creation method.

This model is not well suited to situations in which some aspects of a sound are not defined at its beginning – that is to say, the majority of interesting situations. In as simple a case as a live keyboard performance, there is no way to predetermine the length of a note. The best answer to this problem that has been proposed so far is to consider the note as a process. This idea was partly formalized as part of the 4CED system (Abbott 1980), and more elegantly and completely in RTSKED (Mathews 1981). A "note" process can access a key-up event, for example, as a "trigger" which will cause it to turn off.

Many variations on the RTSKED idea have been proposed. The one major improvement of recent systems such as FTS over RTSKED has been that the process no longer has the burden of specifying the next thing or things it wishes to wait for; it merely waits until someone tells it what to do next. This makes it much easier to build structures which can do things in a non-predetermined order. Whether by coincidence or not, user interface design philosophy has moved in much the same direction in the last ten years; a good user interface does not wait for a specific input at a given time, but rather accepts anything the user wishes to do in any reasonable order.

Message-Passing Model

FTS occupies some number of real-time tasks (one task per CP in the case of the IMW), and defines an object system specifically for real-time music applications. In many respects it is much simpler than most object systems, but it provides a combination of services needed in the IMW that is not provided by other C-language message systems. An object in FTS resides in a single task, and all code accessing it must run in that task. Intertask communication takes place by message-passing.

The most distinctive feature of the FTS object system is that messages are objects which can be copied and stored, and whose arguments are typed. FTS can check the argument types of a message against the types taken by the receiving object's method for it. This is essential if one is passing a message to an object about which there may be no type information at compile time. The typing of message arguments also facilitates transmission across machine boundaries. For example, byte swapping is necessary when passing message arguments between the NeXT host and a CP; but it is essential to know the types of a message's arguments to byte-swap it properly.

An FTS message consists of a selector, which is a pointer to a symbol, and zero or more typed arguments. The fundamental operation defined for a message is to pass it to an object, as shown in Fig. 1. In its most dynamic (least pre-compiled) form, this takes place as follows. The caller assembles the arguments for the method into a contiguous data structure and calls FTS's message-passer. The message-passer looks up the receiving object's entry for the message's selector, in a table pointed to by the first slot of the receiving object's data structure. This entry contains a pointer to the object's method for that selector and an argument type template. The FTS message-passer checks that the types of the message arguments are the same as, or can be coerced to, the types in the message entry. If the conversion succeeds, the method is called with the coerced arguments.

The types of arguments for messages are chosen from: integer, float, pointer to symbol, pointer to FTS object. Each argument of a given method may be of one of these types, possibly defaultable (numbers default to zero and symbols default to the symbol whose name is the empty string; there is no default object pointer.) Alternatively, the receiving object may declare that a method should simply be passed the message structure itself as an argument, complete with type information, in order to take a variable argument list.

The receiving object can catch a message for which it has no method by declaring a method for the symbol named "anything"; the FTS message-passer, after failing to find a method for a given message, searches for an "anything" method and calls that if available. If the method search still fails, or if type-checking fails, an error results.

Passing a message in this way entails much more overhead than the object systems of C++ or ObjectiveC. The intention is to use it for user-built connec-

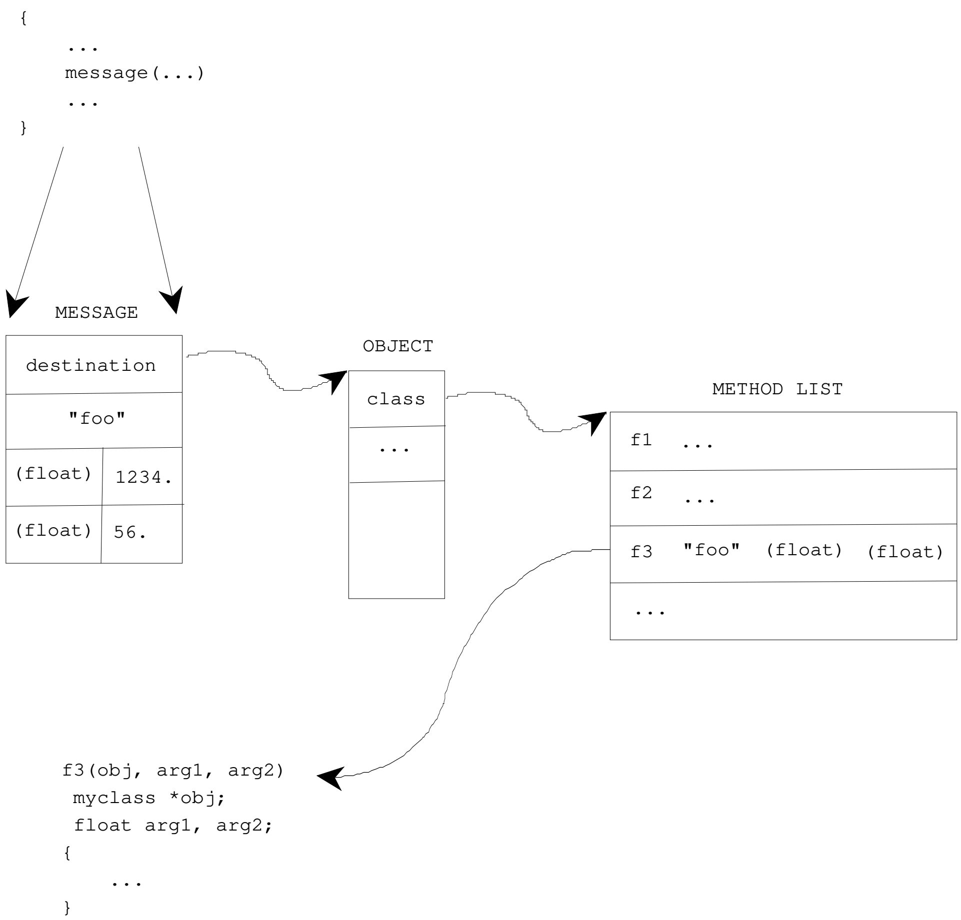

Figure 1: The message system.

tions, not for internal coding, for which, if message passing is needed, one can use C++. It is also possible to pre-fetch a method and pre-establish certain argument lists (those that consist of only one argument.) These features are used by the interconnection facility of MAX to reduce message-passing overhead to an acceptable level.

Message-passing is only defined to work between objects on the same FTS task. To pass a message to an object on a different FTS task, it is necessary to set up a remote message-passing channel, which is handled by the `remote_send` and `remote_receive` classes described below. It is left to the host application to either show remote message-passing explicitly, or to set it up implicitly when a connection is made across a machine boundary (of course, the ideal would be to hide the machine boundary altogether, but that is probably unrealistic.)

The FTS object system can be directed to install new classes dynamically, and (with some care) change a class's instance data structure and/or methods. This facility is needed to make the IMW environment extensible; it is used by the MAX and ANIMAL graphical editors. Implementing dynamic classes requires incremental linking and loading of program segments. Obviously, when a method is changed, other objects which may have pre-fetched it and/or prechecked argument types must be notified, and if the instance structure of a class is changed, it is then necessary to track down every existing instance of the class to bring it up to date. This cleanup is the responsibility of whoever changes the method (see, for example, the section on ANIMAL below).

To load an external object file, memory is allocated and the object file is linked, taking as defined the symbols provided in the FTS executable, and handing the linker the address of the allocated memory as the virtual base address of the code object to create. The object file is read into the memory of every CP at the same virtual address; special CPOS support is needed to allow a CP to allocate memory at a pre-specified address. External object files may not access symbols defined by other external files; anything that is shared by more than one external object file (such as the inlet/outlet feature used by all MAX classes) must be part of FTS.

The object file may contain several functions, but only one entry point. When FTS loads an object file to define a new class, the function at the entry point informs FTS of the instance structure and related data, and supplies all the methods which will belong to the class (usually functions defined in the object file), along with their selectors and argument types. This style of class definition is also used for the classes predefined by FTS. No extensions to the C language are necessary to support the object system; all definitions are made functionally. This message system is therefore compatible, in a restricted sense, with either Objective C or C++; to make a C++ class appear as an FTS class, for instance, one need only give FTS the information it needs to call C++ methods. The procedure for modifying a class that already exists is tailored to the needs of ANIMAL, and will be described in the ANIMAL section of this paper.

The dynamic type-checking capability of FTS allows one to create message-passing connections between objects at run time. The `inlet` and `outlet` classes are provided to support connections as they are defined in MAX; other types of connection, with different semantics, could easily coexist with this one, simply by defining new classes to implement them. The MAX experience has shown that the notion of dynamic message-passing connections is useful. Many musical algorithms can be described by interconnecting pre-existing objects; dynamic connection allows these algorithms to be prototyped without writing and compiling new code. The ability to create or change the classes which are connected in this way offers a "programming escape" for those operations which are more conveniently expressed in C than graphically, or in cases where the overhead of the connection mechanism is too great. The inlet/outlet mechanism is described in the section on MAX below.

Calculation of signals (periodic streams of samples, either of sound or continuous controls) requires communication bandwidths too large to be handled by the message-passing mechanism on a sample-by-sample basis. Objects which do signal computation, called "signal objects," resort to a special mechanism to schedule their computations and transmit information between themselves. Each signal object has a particular duty-cycle action which is carried out regularly to calculate a new set of signal outputs, assuming the existence of new data on all signal inputs. The signal information is carried as vectors of floating-point samples, and the duty cycle is the vector size divided by the sample rate of the inputs and outputs. In its current state, FTS makes the restriction that all signal calculation on a given processor take place at the same duty cycle, and that all signal vectors have the same length. This duty cycle is taken as the "tick," the fundamental unit of time in FTS.

A DSP handler object, global to each processor, maintains a list of signal processing actions to be carried out on each tick, shown in Fig. 2. Each action in the list corresponds to some signal object's duty-cycle method, which is called with pointers to the signal inputs and outputs, as well as other pertinent information kept by the DSP handler, as arguments. Special signal objects are defined to send signals between processors, to and from DACs and ADCs, or to and from soundfiles, which are kept on the host. Signal objects can send and receive messages other than the duty-cycle message; thus, from a control standpoint there is nothing special about them.

Real-Time Behavior

All messages in FTS have a *logical time*, which is kept globally. The logical time increases in regular, discrete increments, each equal to one tick, or one DSP duty cycle. While an object is servicing a message at a given logical time, any message it sends to another object (which must be in the same FTS task) arrives at the same logical time. Physical outputs are arranged to have the minimum jitter possible with respect to this logical time; in other words the

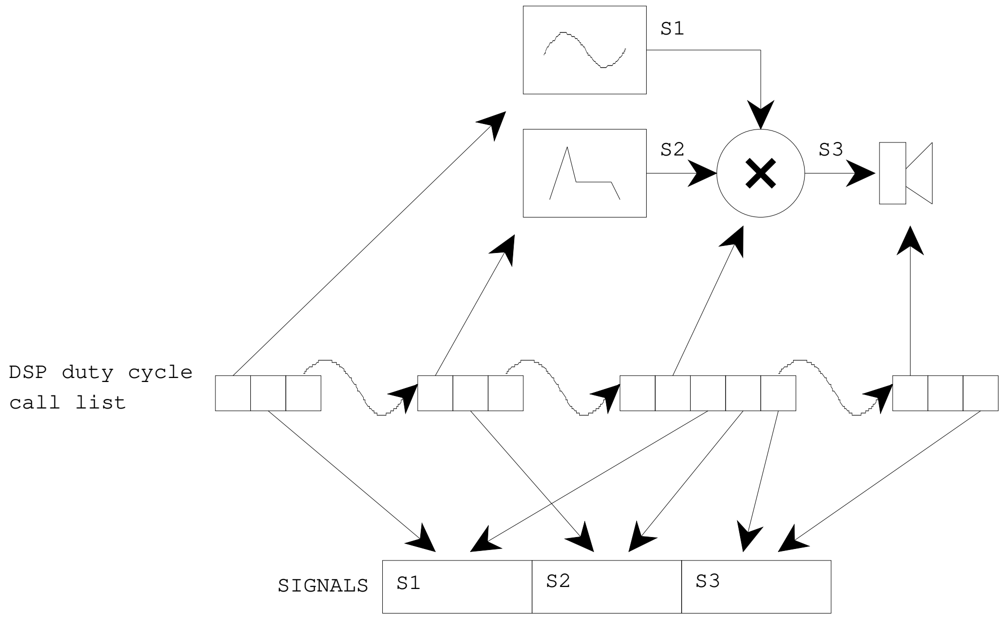

Figure 2: The DSP duty cycle for a simple network.

difference between a real output and the logical time at which it was requested is allowed to vary as little as possible. In the case of sound, this jitter is the jitter of the A/D/A clock and in the case of output to the serial port, it is usually dominated by the pile-up of queued output messages. Output to the NeXT host is quite jittery because of the non-real-time character of the NeXT itself.

Messages originate in four distinct ways: as a result of asynchronous on-board I/O completion (i.e., the serial port); after a timeout (an event indicating the end of a time delay); from another FTS task; or from the host. Messages incur a delay in going across machine boundaries which is reflected in logical time; external input of other sorts (i.e., from the serial port or the host) is stamped with the time of arrival, with roughly the same amount of jitter as in the corresponding outputs.

The FTS task and its relation to the rest of the world is shown in Fig. 3. The task's inputs all appear as time-tagged queues. Except for the serial input queue and the timeout queue, they all share the same structure, which is shown in Fig. 4. This general queue structure treats messages and sound differently. In each queue slot (the contents of a queue for a specific tick) there is a subqueue of messages, of variable length, and a prearranged number of signal buffers. In the sound input queue, the message part is empty.

The serial input queue contains time-stamped MIDI messages. FTS objects may arrange to be notified either for every MIDI Byte that arrives, or only for a certain class of standard MIDI messages. The timeout queue contains callback requests; an FTS object which has placed a request in the queue may later cancel it or change its scheduled time.

For each tick, the FTS carries out its (message and DSP) duty cycle as follows. The task empties out, in sequence, the message contents of each of its queue slots for that tick, passing each message to its destination. (In the case of the serial port and timeout queues, this is not an FTS message pass but a prearranged function call.) Before FTS processes the tick, it waits until all the queue slots for that tick have been filled, i.e., until the task or device which fills each queue has promised that no more information will be added to the queue slot for that tick. The queue slot associated with sound input is processed last; instead of looking in the (empty) message portion of the slot, FTS runs the DSP duty cycle for that tick.

As the tick is being processed, the task can in turn send messages or signals to other tasks (or, indeed, to itself). Messages may be sent sporadically; a signal is sent, on each tick, to a particular signal buffer in the appropriate queue slot. Sound output to DACs is treated as if the DACs associated to a board were a separate task. Midi output is by subroutine call, implicitly time-stamped.

Each FTS task is assigned a latency d, a positive number of ticks about which we can make the following assertion: assuming that the task's input queues are all filled on time, the duty cycle for any given tick t will be finished by real time t+d. This latency defines the jitter in calculation time which we

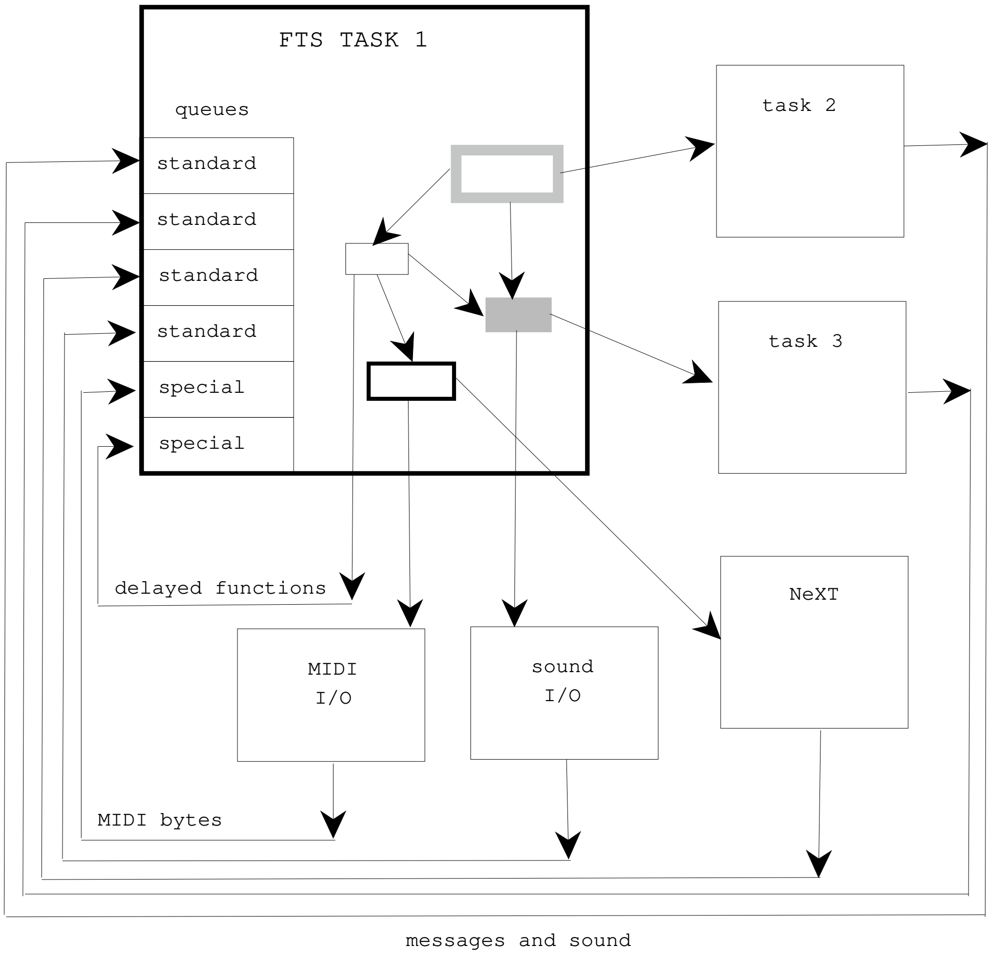

Figure 3: An FTS task and its external communications.

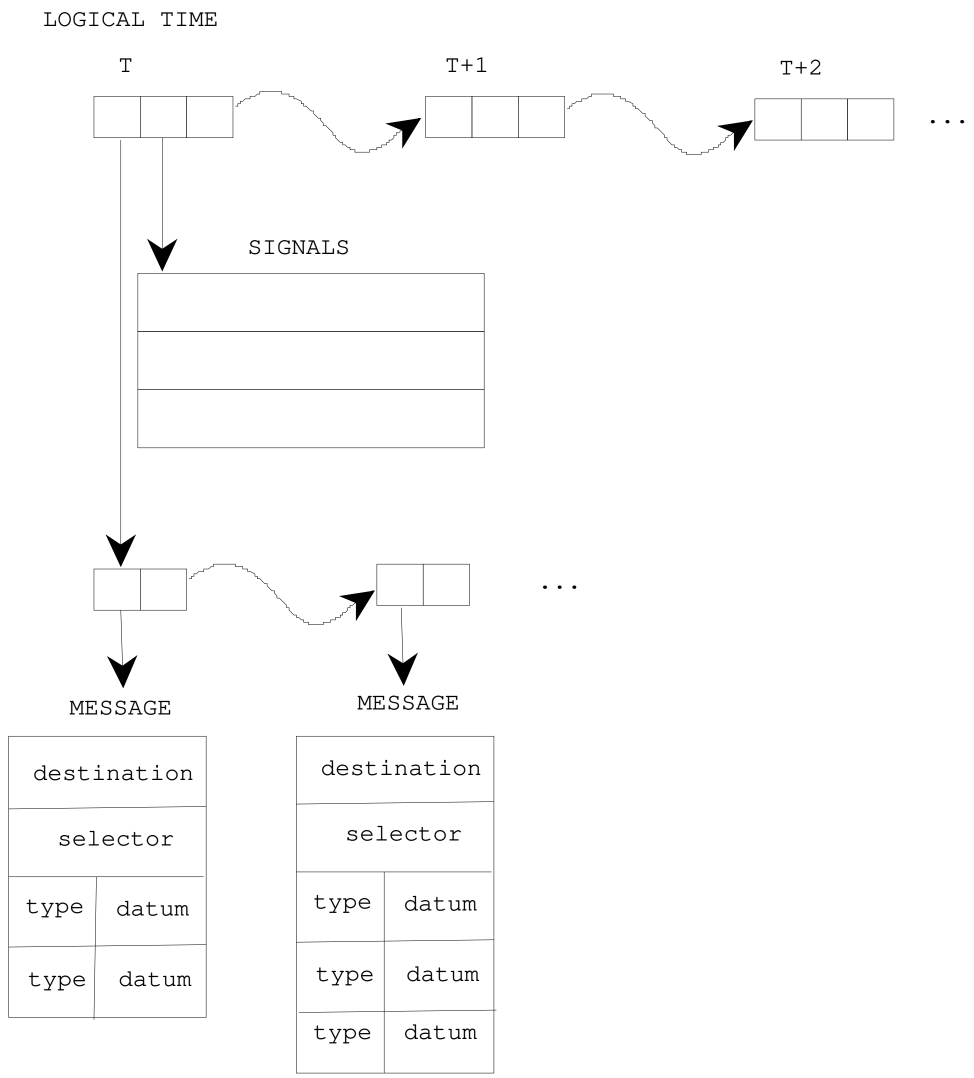

Figure 4: The structure of a standard queue.

will arrange to absorb, so that there is no uncertainty about the time at which an operation takes effect. This absorption is done by time-stamping all outputs of the task at t+d. That will determine the logical time at which another task will respond to a message, or the real time at which physical output will start. The assertion that the tick t will be finished by time t+d at the latest implies that these output messages, time-stamped to t+d, will all arrive on time.

Figure 5 shows the relation in a task between real and logical times, assuming the case in which each tick becomes runnable only at its corresponding real time. This is the worst case, assuming all other tasks keep their own deadlines. The Figure shows a pile-up of computation; whenever a tick is not finished at the moment the next one becomes runnable, the task starts to get behind. The assertion of the latency of the task is that we will not get so far behind that we cross the rightmost diagonal line.

A task can run at latency d if, in every interval [s,t) of logical time, the processing required for all the ticks in the interval does not exceed d+t-s. If one or more task gets late, there is still hope that the lateness will not propagate to an output, but there is no evident way to take advantage of this to loosen the latency specification for a given task.

The FTS approach to real-time multiprocessor programming differs from the standard approach in which tasks frequently compete for resources controlled by some exclusion mechanism. This competition makes the timing of the execution of a given task heavily dependent on the state of other tasks; it can be hard simply to avoid deadlock. On the other hand, the reliance of the setup described here on messages between tasks puts each task at the mercy of all others in another way – it cannot regulate the number of messages, and hence requests for actions, that might fall in a given time period. Thus the latency that a task can achieve depends on what the other tasks are doing.

The interdependence of tasks, diagrammed in Fig. 6, shows that the propagation delay of a message accumulates the latency of each task boundary it crosses. Any intertask loop incurs a similar delay. Loops within a single task, if they involve DSP, incur a delay equal to the vector length (this is the main reason we wish in the future to maintain variable DSP vector lengths; longer vectors can be more efficient, but certain DSP loops require short delays.)

Communication with the Host

The host acts as a front end and as a disk server for FTS. As a front end, its role is to maintain a representation of certain objects (on FTS tasks) as graphic objects on the host. A graphic object may interpret mouse and keyboard input as requests to send messages to a corresponding CP object, and/or change its appearance to reflect changes in the state of the CP object.

When a host user interface application wishes to create an object, it supplies a unique key by which the object is identified. The host can use the key to pass any message to the object. Since the arguments of the message are typed,

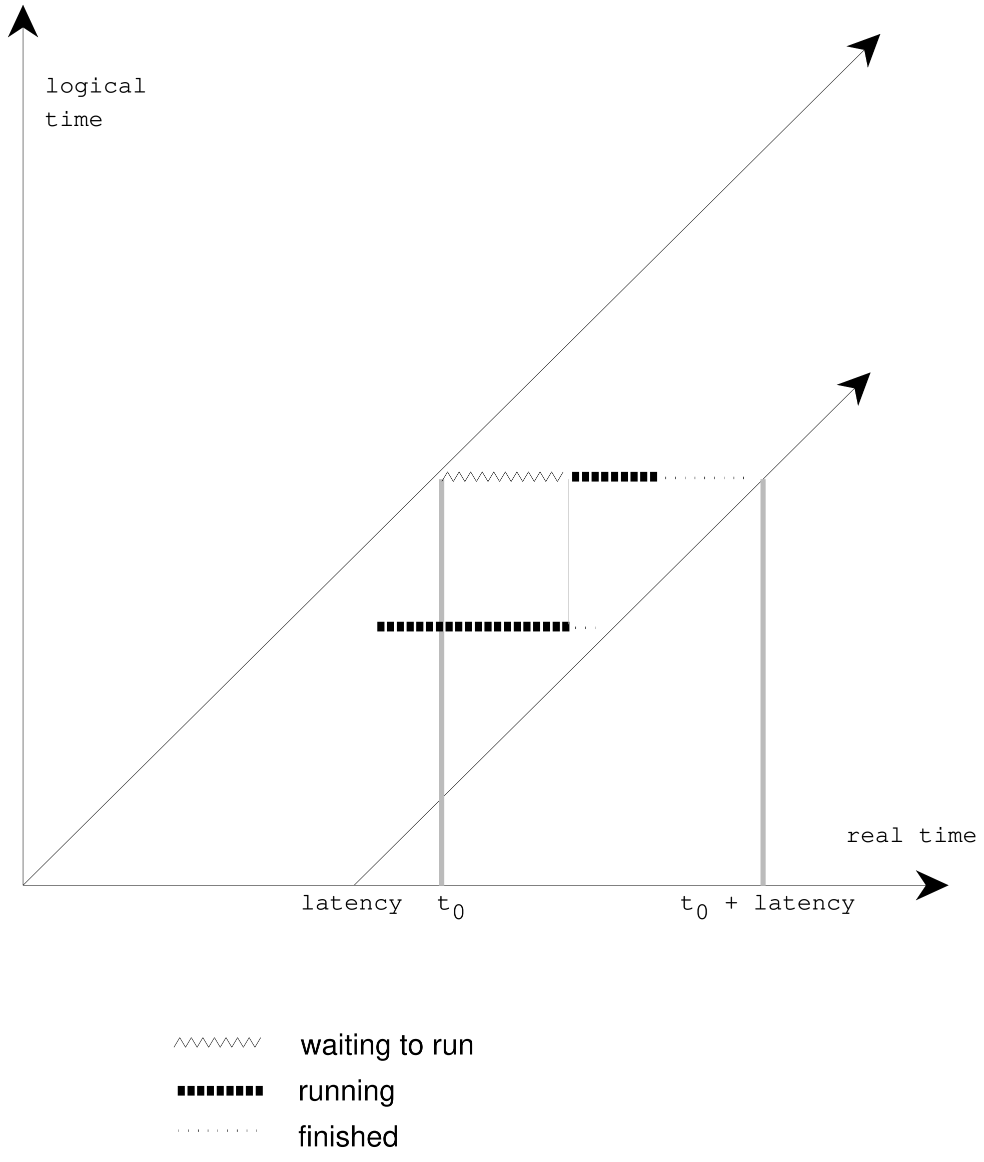

Figure 5: Logical versus real time in a single processor.

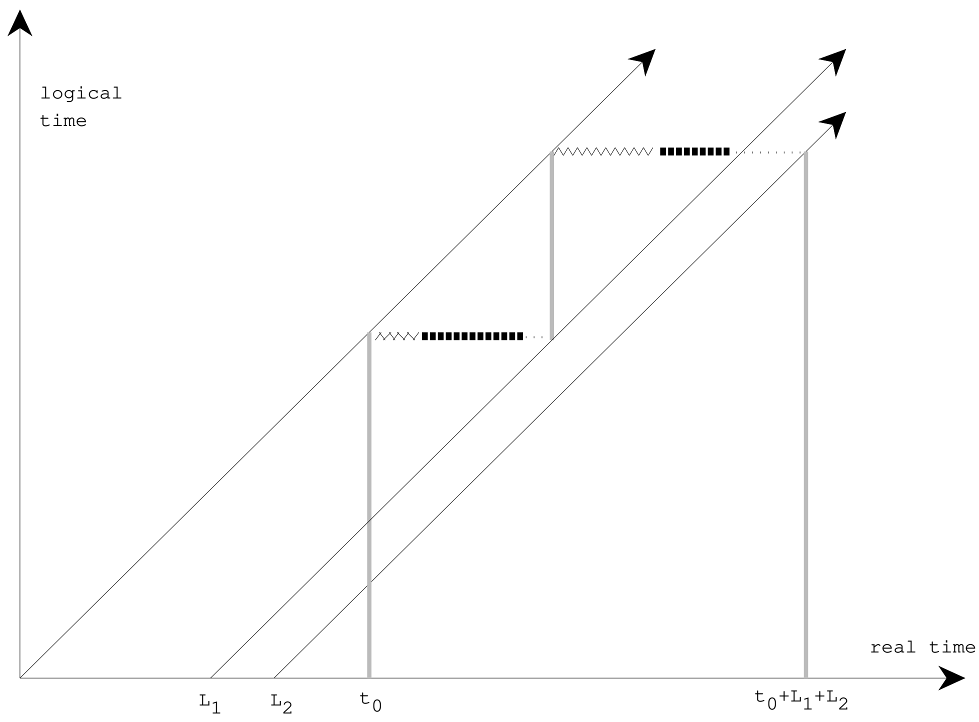

Figure 6: The accumulation of latency through two processors.

FTS can automatically perform any needed data translations. Integers and floating-point numbers are byte-reversed, and symbols are passed as strings and reconverted to symbols (that is, a unique address containing the given string and a possible binding) on a CP. One particular CP, the *master*, carries out the translation of a key to a CPU identifier and a local memory address, and also the generation of symbols from strings; all traffic between the host and the CPs is routed through this master CP. In the case of a symbol, any binding as seen on a given CP must point to an object on that same CP; hence, if a new symbol is created by the master CP, copies of it are created on all other CPs at the same virtual address.

Messages from the CP to the host are handled by the *host queue* mechanism. If any messages flow from the CP object back to the graphical one, the connection established must be bidirectional. (The host must at least send the CP messages to open and close the connection; the host is always the initiator of a connection between a graphical object and a CP object.) To receive messages from a CP object, a graphical object sends the CP object pointers to the graphic object, an update function, and an exception function. The CP object then enqueues calls to the update function on the host.

To free either the CP object or the host object involved in a bidirectional connection, the connection must first be broken; otherwise update messages that have already been buffered between the CP and the host might arrive for the (already deleted) object. This is the reason for the exception message, which the CP object enqueues on closing the connection. The host object is guaranteed that, once the exception function is called, no more update messages will occur through that path. In order to free the host object, a message is passed to the CP to close the connection, and the host object then waits for a callback to its exception function; typically it sets a "zombie" flag for the meantime to warn it not to respond to CP update messages which arrive in the interval.

The case where messages are passed only from the host to the CP is simpler; the host can create the object, send messages to it, and destroy it with no danger. Messages arrive at the CP in the same order they are sent from the host so no message to the object will arrive outside its lifespan.

An example of a bidirectional connection is a "changing value" on the CP, a datum of constant size which is tracked by the host. It is not necessary for the host to be notified of every single change in such a datum; it need only have a recent value of it. On the CP, every time the value changes, the update function is enqueued for the host. If a call to the update function is enqueued before the CP has sent the host a prior one, the new one replaces the old one. The amount of memory needed is thus bounded and can be allocated in advance, and an object whose state changes quickly need not swamp the host with updates. Overriding an already-enqueued update does not change its position in the update queue; hence, updates in the host are roughly round-robin.

More complicated situations are handled as they arise. For example, a

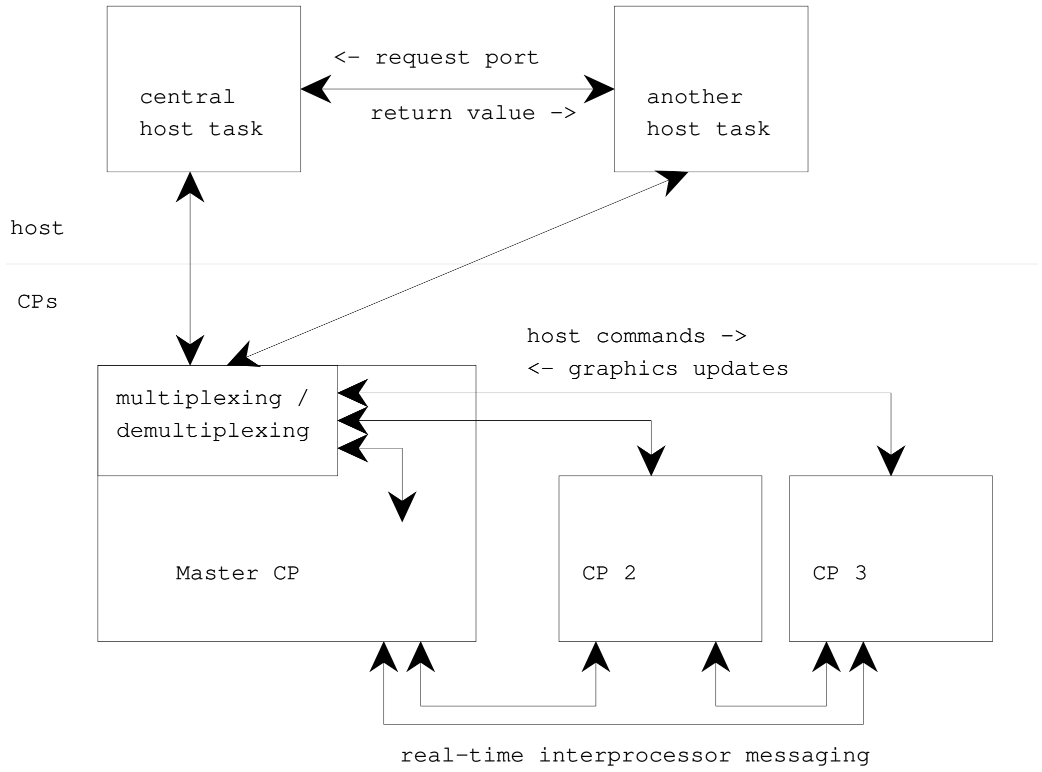

Figure 7: Communication between host tasks and CPs.

variable-sized ordered list (such as a "sequence") might be viewed and edited from the host, and "played" or "recorded" from the CP. In situations like this, the graphics and CP objects must implement a protocol on top of the host queue mechanism. This has been done for the simple case of a list all of whose elements have the same size. The FTS/System Interface

FTS sets up one task on each CP, and chooses one CP task to be the master, through which all host/CP communication is routed. FTS also sets up a host task which controls the allocation of CPs and through which other host applications set up ports of communication to the master CP. The host is also responsible for servicing file I/O requests from CPs, notably real-time soundfile access.

The ports of communication among FTS tasks and between them and host tasks are provided by CPOS, the CP operating system (Viara 1990.). FTS sets up ports between each pair of FTS tasks (in both directions) for real-time message-passing, and each FTS task except the master gets a port back to the master for queued updates to the host. Each host task gets ports to and from the master FTS task, obtained through the central host task as shown in Fig. 7. The port to the CP is used to create and pass messages to objects on a CP and the returning port multiplexes all host queue messages to that particular host task.

A host application needing to access FTS must message the central host task, which returns a pair of ports to the requesting task. The master CP is also notified of the existence of the new ports. The host/CP port is included in the list of ports the master CP selects on to receive messages from the host; each object can ask, when it is being created, for an identifier for the port where returning messages are to be sent.

Utilities Provided by FTS FTS provides software packages to do archiving and recovering of objects, to automate inter-CP message passing, and for soundfile and host queue access. An object can be archived by passing it a `save` message with a pointer to a `binbuf,` a sort of stream it can write formatted messages onto. To recover the object, the contents of the stream are evaluated as a list of messages.

Inter-CP communication is handled through a pair of objects, `remote_send` and `remote_receive`. The `remote_send` object (of which there may be several corresponding to one `remote_receive`), can be given an extra delay beyond that which is implied by the machine boundary being crossed. Any message a `remote_send` receives is sent on to the corresponding `remote_receive` which sends it to a prearranged client. The `remote_receive` keeps track of the number of existing `remote_send`s so that it can delay being freed until all messages it might receive have arrived. Soundfile I/O is provided by a circular-buffer mechanism. The FTS scheduler periodically checks the status of all known soundfile I/O buffers and starts asynchronous disk I/O to service the "hungriest" one. Subroutines are provided for a soundfile user to synchronize with the I/O.

A *host queue element* or `qelem` controls host queue access, and contains the buffer space for a given object's slot in the queue. The client object tells the qelem when it needs to send an update; if an update is already pending this has no effect, but otherwise the `qelem` inserts itself into the host queue. When the `qelem`'s turn comes to be sent to the host, the `qelem` calls the client back to get the latest value to be sent, and formats and sends the message to the host.

Example: Using FTS from MAX

The MAX program (Puckette 1991), originally written for the Apple MacIntosh computer, has been ported to the IMW. A set of signal-processing objects has been written for MAX to allow patches to mix signal generation with control.

The basic connection mechanism of MAX, inlets and outlets, has been adopted without modification to connect the image objects on the CPs. Inlets and outlets are only defined to work between objects on the same processor.

All "patchable" objects in MAX (i.e., the objects that can be manipulated on the screen) maintain a list of inlets and one of outlets. Each outlet maintains a list of all connected inlets; "connecting" an outlet to an inlet means "putting the inlet in the outlet's list." The object owning the outlet can then pass any FTS message to it, which the outlet passes to the inlet, which passes it on to the receiving object (after modifying it to identify which inlet received it).

On the IMW, a box in MAX gives rise to two objects, one on the host and one on a CP. Whenever a connection is made or broken between two objects on the host, the corresponding change is made on the appropriate CP. When the user originates a message through the mouse or keyboard, the message is passed to the CP object instead of the NeXT object, so the host objects never do any message-passing themselves.

Indicators in MAX all fall under the easy case in the above discussion of CP-host communication, so it is straightforward to support graphical updates that follow the state of the patch on the CP. Certain objects require more work, notably the standard object `table` and the experimental object `explode` (Puckette 1990), which maintain a vector and a list, respectively. These are currently dealt with manually; in the table or explode editor, "get" and "send" buttons light up whenever the host and CP versions of the data get out of sync; the user can then choose either to copy up or down to resynchronize them. A more automatic mechanism can easily be envisaged, but has not yet been tried.

At the time of this writing, only a single FTS task may be accessed from MAX; (Puckette 1991) describes an easy extension that could be made to MAX to take advantage of multiprocessing.

Example: Using FTS from ANIMAL

Two classes, `animal` and `a_class` (i.e., "animal class"), have been written in FTS to support the ANIMAL environment (Lindemann 1991b). Here, `animal` will refer to the FTS class and "ANIMAL" to the host program. An instance of `animal` has a fixed and a relocatable part. The relocatable part holds the instance structure generated by ANIMAL. Any instance of animal belongs to an instance of `a_class`, which has methods to add or remove instance variables and methods for all the `animal` instances belonging to it. These editions are propagated to the FTS class of the `animal`; thus, `animal`s respond to standard FTS messages as specified by ANIMAL.

A function is provided to mark an animal instance "dirty," which enqueues a host update via the qelem mechanism. The host may add methods to an `a_class` to allow it to update the contents of the `animal`, for instance as a result of mouse motion or typing.

If an animal has a method for the selector `tick,` the method will be called at the DSP duty cycle. If it has one for `midi` it will be called for each incoming MIDI byte. Animals may arrange timeouts via virtual clocks in the same way any other FTS object does.

Conclusion

A reasonably simple message-passing system and interprocessor communication protocol can be defined to fill the real-time processing needs of such graphical programming environments as MAX and ANIMAL. Message delays between processors are non-transparent, as is the mapping of real-time programs onto the available processors. Since scheduling dependencies between objects are controlled explicitly via message-passing, we can avoid introducing MUTEXes, context switching, and the like. The ability of the IMW to do signal processing and "control computations" in the same processor make possible the very close cooperation between the two that real-time musical applications demand.

References

Abbott, C. 1980. "The 4CED Program." Proceedings of the 1981 International Computer Music Conference. San Francisco: Computer Music Association, pp. 278-304.

Favreau, E. et al. 1986. "Software Developments for the 4X Real-Time System." Proceedings of the 1986 International Computer Music Conference. San Francisco: Computer Music Association, pp. 369-373.

Lindemann, E. et al. 1991. "The Architecture of the IRCAM Music Workstation." Computer Music Journal 15(3): 41-49.

Lindemann, E. 1991. "ANIMAL – a Rapid Prototyping Environment for Computer Music Systems." Computer Music Journal 15(3): 78-100.

Mathews, M. and J. Pasquale. 1981. "RTSKED, a Scheduled Performance Language for the Crumar General Development System." Proceedings of the 1981 International Computer Music Conference. San Francisco: Computer Music Association, p. 286.

Puckette, M. 1990. "Amplifying Musical Nuance." Journal of the Acoustical Society of America 87(supplement 1): p. S39.

Puckette, M. 1991. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." Computer Music Journal 15(3): 68-77.

Viara, E. 1991. "CPOS: A Real-Time Operating System for the IRCAM Music Workstation." Computer Music Journal 15(3): 50-57.