HAL open science

The Ircam Real-Time Platform And Applications

François Déchelle, Maurizio de Cecco

▶ **To cite this version:**

François Déchelle, Maurizio de Cecco. The Ircam Real-Time Platform And Applications. ICMC: International Computer Music Conference, 1995, Banff, Canada. pp.1-1. hal-01157142

HAL Id: hal-01157142

https://hal.science/hal-01157142

Submitted on 27 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Ircam Real-Time Platform And Applications

François Déchelle, Maurizio De Cecco

ICMC 95, Banff (Canada), 1995
Copyright © Ircam - Centre Georges-Pompidou 1995

Abstract

This paper describes the architecture of the FTS real-time executive, which is the framework for IRCAM's real time applications. FTS includes both classical real-time distributed kernel features and a message-based object system. FTS is a portable and configurable system, which exists now on several platforms which are described.

Introduction

As it has been already described in [Déchelle,F.], FTS is now an open and portable system. It is based on a client/server architecture. Its internal structure has a modular and configurable organization, suitable for a wide range of application, from rapid prototyping and algorithmic composition to embedded sound processing applications.

In this paper we describe the architecture of the system, from the point of view of the application writer, and we describe the main evolutions planned for the short and medium term future.

Architecture

FTS is organized as a library providing a number of services to application writers; some of the services relate more to resource handling and hardware interface, and they constitute the FTS kernel.

Other services implement a kind of "middle-ware", providing a reacher framework, were to build applications; at the moment they are the FTL DSP/vector computational engine, and the MAX object oriented layer.

The programmer is free to use some or all of the services, depending on the application; it can also link only the actually used services, reducing the executable size, making FTS more suitable for embedded and stand-alone applications.

It is worth noting that, with respect to previous distributed version, FTS do not require now that all the application code is included in a Max external object; the application writer are free and are have the resources to add modules directly on top of the FTS kernel, bypassing the object layer.

The FTS kernel

Kernel services essentially provide a high level interface to manage the resource provided by the underling hardware and/or OS (memory, CPU time, communication channels). In the following, we describe the kernel services from the simpler to the more complex.

Data Structure Library The data structure library provide the application writers with a number of basic data types and data structures, used in the FTS implementation and made availables for other purpose.

Atoms are tagged unions, that may store float, long and symbols (unique representation of strings).

Atom List are data structures keeping a editable list of atoms; they can be used in any case where a list of unknown size is needed.

Hash Tables provide mapping between symbols and any other data type, and can be used to implement local name-spaces.

Basic services Basic services include low level services as memory handling, error handling, time handling, and profiling support.

The memory handling module provide set of primitives to allocate and free memory; since FTS may run on machines with different and complex memory architectures (like the ISPW or the C40), and since in some case the kind of memory for a particular kind of data make a big difference in the resulting performance, this library define the concept of memory attribute, and allow the user to request the memory with the right attributes for the application.

The error handling module provide a standard way to represent errors, and to signal them between FTS procedures and to the client, using the event library (see below); FTS errors include declarative and textual representation of the error condition; the client is responsible to choose the way, if any, the user is informed of the error.

The time handling module provide a set of time related primitives, namely alarm clocks, timers and time gates, usable to implement different time based application services; the time is defined by objects called "clocks"; the time module implement "clocks" providing milliseconds and scheduling loop synchronous time, but the user can defined its own clock, for example in order to synchronize an application to an external MIDI time code.

The profiling module provide a set of primitives to collect execution time and statistics of different code fragments, and can be used to obtain detailed information about load distribution and charge in a FTS system.

Communications The FTS kernel cope with two kind of communication: client communication and interprocessor communication.

The Client Communication module provide primitives to communicate with FTS clients; the module support communication thru multiple clients at a time , include all the message parsing and unparsing functionalities and provide a "message" abstraction to the programmer; an application can install its own handler for certain kind of messages received from FTS; moreover, in order to make message sending safe in a real-time context, the module provide a way to reserve communication bandwidth, to avoid blocking message send. Client messages are coded using an architecture independent communication protocol.

The Interprocessor Communication module implement a number of primitives to allow communication between the different processor where FTS is running (on a multi-processor machine, of course); the IPC communication is substantially different from the client communication; IPC communication is synchronous, the data is exchanged exactly once for every scheduling loop; and the data representation use a machine dependent coding, for maximal performance.

IPC allow communication of bulk data (usually signals) and of discrete message (usually control information); also in this case application can declare new message types and declare handlers for those types.

Scheduling An FTS application consists of an infinite loop (called tick) performing a number of tasks, each implemented by a C function; this loop may be synchronized to some I/O device if some of the scheduled functions do blocking I/O on a time synchronized device, like a DAC.

FTS use a static scheduling; each module can require one or more functions being called each scheduling tick; the scheduling order is computed at boot time, based on declarative informations provided by the module themselves; this information is given using a provide/require model, where each function is declared to need or to satisfies some precondition given in a symbolic form.

An application can declare its own scheduled functions, using the provide/require mechanism to specify the right scheduling position with respect to the kernel and other application scheduled functions.

I/O library The FTS I/O architecture is based on two layers; an I/O library layer providing high level I/O functionality to the applications, and the device driver layer, providing a uniform interface between the FTS kernel and applications and the underling hardware (and possibly software) environment.

The I/O library provide Events, Audio Streams, Midi streams, Direct User messages (post) and Debug I/O.

Events are a mechanism used to communicate to a client the occurrence of a specific situation; events are used to signal errors, to perform graphic updates in graphic clients and so on; the client library provide support to install user event handlers on the client side.

Audio Streams are a high level mapping of audio I/O; identified by integers, and automatically handled by a scheduled function, provide mapping of streams id to physical device and synchronous I/O operations.

Midi streams are a similar mapping for Midi I/O; support mapping of MIDI streams id to physical device, and the installation of application parsers on a midi device.

A number of I/O functions to provide direct application printout are provided; the basic version is based on events, and the content is supposedly printed by the client in an application dependent way, and a debugging I/O functions, that provide direct printout in a platform dependent way, used mostly to debug new code.

Device Drivers FTS provide an uniform way to interface to the underling hardware and/or system software (when appropriate): the device API.

A device, in FTS terms, is an abstraction supporting a small number of i/o and control operations; the device API define character and vector based synchronous or asynchronous input/output operations; all the devices have the same API, but a device is free to don't implement some of the functions.

A device is created and controlled in a straightforward FTS version of the UNIX open/close/ioctl model; multichannel device are supported by the concept of "device class", that include a set of device corresponding, for example, to the multiple channels of the device.

The device API is simple enough to make the implementation of the devices for platform already proving audio I/O very easy; for example, around one day of coding and debugging for the SGI machine.

FTS ship with devices for the currently supported architectures; but new devices can be easily added by simply linking device libraries to the FTS kernel.

Configuration The new version of FTS is headed to support a broad number of platform and configurations, in different application contexts, from the rapid prototyping environment, to embedded applications. In order to support all the different cases without producing a specialized FTS source for every situation, FTS include a number of configuration oriented features and services.

First of all, FTS is organized around libraries; it is not anymore a single, monolithic executable; only the needed modules are linked with the applications.

Second, the whole FTS, and in general all the applications written for FTS, are organized in modules; a module is a programming abstraction, that declare a init and a shutdown entry point for a software

component; the application can install new modules just before starting FTS; or, FTS can be configured and scaled down just selecting which modules are actually included in the executable.

Third, FTS provide a generic configuration language (UCS), used to declare at start up time (or during the execution) the device configuration; the language is extensible, and new device are automatically recognized by the UCS parser. UCS allow to declare things like which serial line to use as MIDI i/o, which DAC line map to which audio streams, which device is the device used for clients communications and so on.

Finally, FTS allow the applications to specify an initial and hardwired set of client messages to execute at start up before activating the real client connection; this allow a part or all the configuration or to be hard-coded in the FTS executable if needed.

The application framework : The message system and FTL computation engine

The FTS message system

The FTS message system is a new definition and implementation of the Max/FTS message system as described in [Puckette,M. 1991].

Objects, messages and methods A FTS object is a *state* (some memory holding values) and a set of *connections*. A connection is a link between an *inlet* and an *outlet*, and is used for message sending. In a connection, the outlet is the source of the link and the inlet the destination.

A *message* is a *selector* which is a symbol (unique representation of a character string), and has arguments which are atoms, as described in the section ``The FTS kernel''. A message can keep the exact type and number of arguments, in order to do connection and run-time type-checking.

A method is a function which call is triggered by the reception of a message on an inlet. Thus, the definition of the methods of a class associates a method to a selector *and* an inlet number.

Below is the standard signature of a FTS method. It must be noticed that all the methods have the same signature :

```
void method(
  fts_object *object,         /* object which is the destination of the message */
  int winlet,                 /* number of the inlet on which message was received */
  const fts_symbol *selector, /* message selector */
  int ac,                     /* number of arguments */
  const fts_atom *at);        /* arguments array */
```

Classes and meta-classes A class keeps all the informations which are common to all the objects of the class : inlets and outlets numbers, association between methods and message selector plus inlet number, type-checking information. The inlets and outlets can be typed, the type of an inlet (outlet) is the list of the messages it can receive (send). This information is used to do connection-time type-checking and method lookup optimization.

This definition leads to a high factorization of data, but implies that all the objects of a class have the *same* number of inlets and outlets. This is the case for simple objects such as `integer, +`, ..., but is not for objects such as `trigger` (the object which sends its input message to all its outlets, from right to left, the number of outlets being the number of object creation arguments). Those kind of objects compute their number of inlets and/or outlets at their creation time. This is now supported by the *meta-class* abstraction.

A *meta-class* is a mechanism of dynamic instantiation of classes. To a meta-class is associated a base of

already instantiated classes. When creating an object of a meta-class, a class is first instantiated if necessary. The lookup in the meta-class class base uses the comparison with a discrimination function, which can for instance compare the number and types of objects creation arguments.

The class information protocol The FTS message system implements a class information protocol, which can be used by any client at any time to know the classes which are present in FTS. This protocol uses the FTS kernel event abstraction, described in I/O library section.

The class information protocol exports to the client the relevant information : number and types of inlets and outlets, but does not export the code of the methods.

The introduction of this class information protocol leads to a true client/server information, in which a client has to know strictly nothing about the classes which are present at a given time in the FTS server to which it is connected. This is an obvious advantage for the external object programmer : no client code has to be written.

Improvements The FTS message system is fully portable (100 ANSI C). It is faster and more compact than the original one. It is orthogonal and has no arbitrary limitations like number of inlets or restricted types of messages for inlets other than the first one.

The FTS message system has a small (25 functions) and documented API, which is guaranteed to work on all supported platforms. The API is strictly the same for ``internal'' and ``external'' objects, meaning that there is no longer the distinction between those two kinds of objects : any object can be dynamically or statically loaded.

The FTL vector computation engine

The former DSP engine, as described in [Puckette,M. 1991] was a set of vector computation functions and a ``DSP chain'' builder and interpreter. A ``DSP chain'' is a linear list of calls to the vector computation functions.

We introduced the FTL (for Faster Than Light) vector computation engine with the following goals :

- guarantee the portability of FTS DSP applications
- reach the maximal efficiency for a given platform
- provide a symbolic, thus readable and edit-able representation of code, which can also be loaded from a FTS client
- generalize the ``DSP chain'' to an interpreter with a set of primitive operations (the functions) and a set of flow control instructions (branch, loop, call, ...)
- support platform specific extreme optimizations

FTL provides an intermediate abstract machine model with a uniform and unique code generation interface, which is application and architecture independent.

A FTL program can either be interpreted by a default portable engine, or be compiled to a machine executable format, including direct binary code.

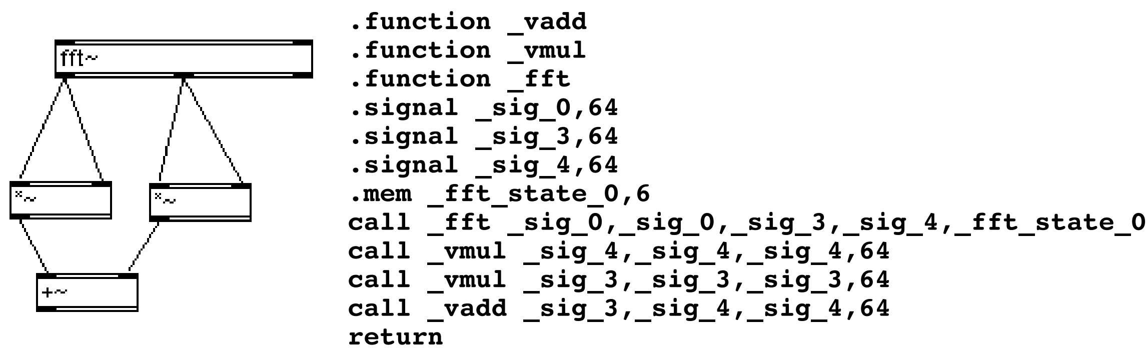

Figure 1 gives an example of an FTL program, generated from a Max patch doing DSP.

Figure 1: A MAX patch doing DSP and its associated FTL program

Applications

Max

Max is the main tool to prototype and develop applications based on the FTS message system. Max is now based on the FTS client/server communication protocol, and uses the FTS class information protocol to obtain informations about FTS objects; this means that external object writers do not need to compile and link the external object code in Max, making development simpler, especially in an heterogeneous client server environment.

Work is in progress to extend the Macintosh version of Max to make fully use of the FTS client/server protocol; this will allow the use of Max Macintosh as a complete development environment for FTS, and the integration of Macintosh Max and FTS applications.

Specific DSP algorithms

Work is currently under progress to implement in FTS IRCAM's Inverse-FFT additive synthesis algorithm [Rodet,X.], and to include in the FTS the latest result of the Ircam research in sound processing and synthesys.

Clients

Not every client server environment is based on client interface; this is why we are developping ``Min'', a command line based purely textual client for FTS, that allow booting FTS, loading patch and also controlling and editing a patch from a command line (DOS or Unix). The client is oriented to scripts based and embedded applications and is also very useful when porting FTS to a new platform. It exists on all platforms supporting FTS.

Platforms

FTS is now available in beta test on a number of platforms; depending on the platforms, different I/O system and client environment are available.

- Ircam Signal Processing Workstation : consists of a NeXT Cube with up to three ISPW card; run with NextStep 3.1 or 3.2 (note that NextStep 3.3 is not supported on the Cube); provide the complete environment, including Max.
- C40 based PC : consists of a ISA PC equipped of one or more Ariel PC-Hydra board (with four

TMS320C40 processor each), and one Nagra I/O board, including 8 AES/EBU stereo digital channels in input and 8 in output, and RS422 and MIDI (thru an adapter) connections. Supported OS are NextStep 3.2 and 3.3, and MSWindow/DOS. Max is available only on NextStep. Linux support may be added in the future.

- Unix workstations : supported workstations are now the SGI machines and Dec Alpha machines; on the SGI, the native audio system is supported (up to 4 channels in and 4 out, depending on the system); on the Dec Alpha, it use the AudioFile audio protocol. A Motif version of Max is available.

Note that, using a network connection, any version of Max can communicate with any version of FTS running on an other machine.

Porting to other platforms is easy, especially if a native audio I/O system is available; contact the authors in case of a specific interest for a specific platform.

References

[Lindemann,E.] Eric Lindemann, François Déchelle, Michel Starkier, Bennett Smith. *The Architecture of the IRCAM Musical Workstation*. Computer Music Journal, Fall 1991, 15(3): pp. 41-50.

[Puckette,M. 1991] Miller Puckette. *FTS : A Real-Time Monitor for Multiprocessor Music Synthesis*. Computer Music Journal, Fall 1991, 15(3): pp. 58-68.

[Déchelle,F.] François Déchelle, Maurizio De Cecco, Miller Puckette, David Zicarelli. *The Ircam ``Real-Time Platform'': Evolution and Perspectives*. ICMC 94, Aarhus.

[Rodet,X.] Adrian Freed, Xavier Rodet, Philippe Depalle. *Synthesis and control of Hundreds of Sinusoidal Partials on a Desktop Computer without Custom Hardware*. ICMC 93, Tokyo.