The 4X system, realized at IRCAM by C. Di Giorgi, is a powerful tool for the synthesis and temporal control of sounds and signals in real time. Recent developments in software for the 4X include ORPHEE, a mini-VM written in modula-2; ORFEO, a compiler of real-time control programs; MAX a set of real-time processes and graphic interfaces for the 4X patch language and FACSIMILE. This paper will describe these new developments in building a real-time application environment using the 4X.

**INTRODUCTION**

The 4X workstation consists of: a Sun host computer running UNIX, connected to a real-time 68000 VME computer, the 4X itself, and a DAC/ADC conversion module.

**ORPHEE DIAGRAM**

The real-time music system ORPHEE, developed by M. Finucane, is distributed between the VME and the host. The VME main part implements a Unix-like command language mainly used to manipulate files on local and remote storage devices and to run programs; provides such system services as input/output (to local and remote files), memory allocation and remote program invocation through Unix-like system calls, and services real-time input/output requests. The host's resident part services such remote requests as file manipulation, navigation through the file system and communications with host-resident processes.

**4X REAL TIME DIGITAL SIGNAL PROCESSOR AND AUDIO CONVERSION SYSTEM**

The 4X is a compiler of control programs which run on the VME 68000 and plug the 4X in real time during the execution of a given application. 4X was designed by R. Rowe and O. Konchilo, and implemented by R. Rowe. Applications have to date included programs for sampling fixed sounds, real-time FM modula-tion programs for the 4X and its environment, and musical compositions.
key consists of a set of language extensions to the C programming language and a runtime environment. The compiler is designed to support the description of functions, called actions, which will be executed in real-time and may be running in parallel, and to facilitate the interface of these actions with the 4x4 patch language and hardware peripheral environment. Many of the constructs in 4x are meant to simplify the coding of computational algorithms, and to enhance the ability of such algorithms to be controlled in a real-time environment.

MAT, realized by M. Pochman from MIT, is an implementation of a set of real-time process scheduling and communication ideas aimed at making it possible to design efficient systems that can be combined quickly and without a changing code. MAT is partly written in 4x and partly written in 4x language. MAT greatly enhances the development of new 4x applications, making it possible to use a 4x patch without writing a full-motion control program for it.

The 4x patch language of P. Firstbrook, described in another paper, has been equipped with a graphic interface for the 4x and MacIntosh workstations. The power of the 4x and the multiplexing of real-time control has led to a system of increasing complexity. The development of sophisticated graphic tools addresses this complexity by providing an elegant and efficient mechanism for generating both lower level code to be executed directly by the system, and program data that can be incorporated in user applications. In the case of the patch language, an interime is loaded into the 4x and executed. This interpreter reads commands sent through a link with the graphics station. On the screen, the user can manipulate symbols representing various 4x and high-level system monitors, connecting them to produce and control sound in real-time. Ensembles of such symbols can be grouped and given a name, to be used in other configurations. Once a patch has been developed, the source code for it can be output to a file.

A similar tool for the PACOM control console has been developed by Emmanuel Fornaro. The same communications link is used between the 4x and graphic host, which this time represents the device available to the user. This tool can accept the output of different devices to registers in the 4x controlling the patch, and immediately test the control on the console monitor. A range of values output by each device can be defined, as well as various types of filtering, which will be applied before input to the control value to the 4x. The combination of this tool with the graphic patch language described above provides powerful means of designing and developing patch programs for the 4x and real-time environments to control them.

ORPHEE A REAL TIME UNIZ0080 MONITOR (Univ of Edinburgh)

This chapter describes the programming environment of the 4x computer. It is composed of three main parts: a program, the VME (a 68000 VME-based system), and the host's (a Sun system) interface. It provides a running example of a set of programs developed under UNIX and used to control the 4x and MIR devices.

ORPHEE, the real-time monitor, is distributed between the VME and the host. The VME resident part (henceforth referred to as the monitor) implements a Unix-like command language mainly used as an interactive facility on the storage devices in a hierarchy of directory structure and, to run programs, provides such services as data transfer, general output, input and removal, file allocation and remote program invocation through Unix-like system calls; and services real-time interrupt requests from the 4x and other peripheral devices.

The host's resident part (henceforth called the server) services each remote request as file manipulations, navigation through its file system and communications with the host resident programs.

1.3.1 Boot

ORPHEE can be directly booted from any of the local disk or from the host.

1.3.2 Development Tools

This section lists the specific tools used to prepare a program to run on the system.

User programs are typically distributed between the 4x and the VME, and can be debugged on the host or another remote computer. The VME resident programs are written either in C or in an extension, called the 4x language, while those to be run on the 4x are written in a language called patch. In addition to such standard tools as editors, the following are available:

- Current compiler for VME programs written in C:
- C program compiler for 4x programs written in the patch language:
- C program compiler for VME programs written in an 4x language:

1.3.3 Monitor Commands

The monitor commands available at the keyboard are mainly used to manipulate local or remote files, and to navigate in the local or remote file system.

Commands issued at the terminal are of one of two kinds: local (i.e., executed by the monitor) or remote (directed to the host and executed by the server). Remote command names are preceded by an r.

For example, is the list command executed locally, while is the show (remote) list command. This also applies to commands such as one whose name is not preceded by a period during its execution (even if it is found on the host), while if it is preceded by a period it will be executed by the shell on the host.

Remote commands, being executed by the remote shell, can manipulate only remote files. Local commands can manipulate local files, and in most cases, also remote files.

2.4 File Names and Directories

Users can reside either on the local or remote file system. The local one is commonly composed of 2 drives, 0 and 1, drive 0 being a fixed disk dedicated to the system, and drive 1 removable cartridge whose contents are left to the discretion of its owner.

The remote file system is the one supported on the host.

The local file system supports hierarchical directories, the root directory being named . The root is always a current (local) working directory, displayed by the pwd command and altered by means of the cd command.

A file name is a non-repeating: naming of acyclic characters. These built-in commands, and monitor system calls which manipulate files may need the name of one or more input or output files. These files may, in turn, be indirectly located on the local devices or the host.
2.6 Local Commands

In much the same way as the Unix shell, these commands fall into the following categories:
- built-in, executed by code wired in the machine
- shell commands consisting one or more primitive commands
- executable code files containing code compiled by cc, c89, c90, etc.

Continued file: executable code, though executed locally, can exist on the host.

The built-in commands implemented at date are:

```
sh shell
ps print file on standard output
ch change local working directory
cd change local working directory
ls list directory
compress create a compressed local disk
uncompress unpack a new file
cp copy a file to a new file
rm display file system map
ls display directory map
chmod change file attributes of the command
kill exit the process
exit exit from the monitor to the VI "EDIT" environment.
```

```
check check the local file system consistency
find find commands for a specific command
ls list one or more files in currently mounted local device
mkdir create a directory in the current working directory
df display a list of the device mounted on the file system
df display a list of the device mounted on the file system
mb display a list of the device mounted on the file system
ps display the password of a local disk
pwd display current path or name working directory
rm remove one or more files or directories
```

```
type print the contents of standard input into the file
```

2.7 Remote Commands

Remote commands are a subset of Unix commands, that are shipped by the monitor to the server, which then executes them. They currently include commands such as:

```
sh, date, ls, rm, ps, pss, ps, psw, ps, size, vi.
```

Note that the local versions of ls and cd can act on remote files (but since they can act locally, they will be slower than the corresponding remote commands).

3. System Calls and User Libraries

Most of the system calls available in a user program running on the VME are byte-code which are shipped for use by the users of the file. The library functions (e.g., stream-oriented input-output) in a user program. They also allow for manipulation of local and remote files, as well as other local machine services, and currently include:

```
sh
```
To obtain control over a predefined fixture, the user has access to an axis word readable fixture. The state of this word is updated by the action. Instruct to reference the number of the fixture currently being executed: Instruction are extracted from 0 to the number of fixtures - 1. Further, one can activate the intrinsic position of outlet higher up in the tree structure with the syntax instance, where a number for the last one wants to go. Therefore instance states refers to the printer instance number, instance number to the printer's instantiation.

It is also possible to determine the number of occurrences in a fixture for strides since its creation through the keyword "register". This can be used to determine the current state of a fixture and to access information about its components. However, the action to register a fixture, which is accessible through the event occurrence, is only described for instance: Count [0] Count [1], and so on. Count is distinct for each instance of any action, and can be assigned a new value by any user. It is possible for a user to use Count to indicate an array for instance: Count is reset to zero when an instance is created or deleted. This unique instance number is used to identify the action that created the instance.

Actions are triggered, collected, and registered using a set of conditions recognized by the language. For instance, the action "Time" makes the declaration:

Action declares (Simple condition) and to make this action that includes in-5 by-5 statement:

Next (condition)

These conditions apply to various devices and expressions in the action environment. The available keywords at present include:

- Button, Panel, Key, Keyset, Switches, Switches, Switches, Time, N-1, and so on.

As an example, button refers to the button state of a button that has been scheduled or registered with the key numbers. As another example, Keyset refers to the state of a set of buttons. The action "Time" is a condition that can be used to determine the time and the actions that occur within a specific interval of time. It is a time-based condition that is used to control the execution of actions within a specified time frame.

The interface with the push button is accomplished through the manipulation of a button file, generated by the button language, which traces the state of the push button and the execution of the button file followed by the next action. This file contains a set of action statements that permit the programmer to use the state-symbolic interface for the action data memory locations as a push button language.

A wide range of scales and raw scales are provided, which allow the manipulation of various external features of objects, either, or over scales. Such values, modified in a 'Time' statement, can be equal to end from the time scale and update the human and the mouse interface.

Other features provide the description of component elements, pitch, and display which can include real-time calculations. A full range of library routines and utilities complete the key system.

key, which were built in June of 1981, is new for most of the applications developed for the 4X machine, ranging from text processors to medical simulations. The purpose of this system is to make up for the loss of user interaction; since it is an execution of Q-dep which does not exist in the language can be copied directly and added to the language itself. This generation and function of a given technique have been thoroughly mapped. Most methods exist, but that key system will continue to grow, and that effect of development will not be entirely annihilated.

6 THE MAX REAL-TIME CONTROL SYSTEM

(D. Porter)

MAX (named in gratitude to Max Mathews) is an implementation of a set of real-time process synthesis and communications ideas aimed at making it possible to design elements of a system which can be connected quickly and without changing code. This is an implementation of a process, which provides a process, which can be used to control a process, which provides a process, which can be used to control a process, which provides a process, which can be used to control a process, which provides a process, which can be used to control a process,

communications between the objects, which are the MAX called control processes (CPs), a one to "button", which are messages with associated times. The button is sent to the MAX scheduler, which sends it to the destination CP at the button's "time". If more than one button's "time" times have arrived, the scheduler delivers the button with the earliest "time".

Letters are transmitted as NMF symbols, so that it is easy for a letter to a button to a control process, that's the intention of any action.

The lack of data transmission implies the greatest possible flexibility in configuring CPs.

These MAX, the user specifies the use of CPs he desires in a configuration file which MAX reads at run time. A simple such file could contain:

4s by 5y

6s by 5y

which allocates one CPs named "a", which writes, reads, and produces events and another named "b", which performs for resources exchange and configure this from as well as to other CPs. This configuration is feasible enough to act as an object push during.

Further more monogram "set", MAX's output, and variables available, thus, for example, the configuration:

4s by 5y

20s by 6s by 0.0

as both the first configuration permits writing output 6s a message and "a", "set", "a", which assigns a value to a 100 events and a parameter which controls the output of another register. Since the last line of 3CPs performs functions such as "if", "then", event detection, timed occurrence of (which may themselves be written and may vary, a wide range of functions). Since all instructions to the CPs are in the form of letters, the event can be any language: the action specified might etc. turn associative agents even with another action.

ICMC36 Proceedings 372
Although MAX is written under MS-DOS, it is not currently easy to combine CP/M with MAX with Actions in Key; this is the subject of current work.

We are planning to add a "key" to CAT and a "key" to MAX for allowing a user to write his own key applications with MAX or CAT.

In addition, a miniconference of MAX in Key is planned for 1978, the same schedule of CAT is not be used, but the lectures will be replaced by links connecting the arguments of a message of the form "delivery" of the letter will be done by a "send" of the message at the approximate time.

5 DEVELOPMENT OF A GRAPHIC INTERFACE FOR THE MAXLGRAMELANGE (P. Poulakakos)

The increasing power of digital signal processors and the multiplication of real-time controls (keyboards, sequencers, customized controls, etc.) has led to the realization of workstations which are more and more complex. Usually, different elements have specific programming demands even though an efficient man-machine interface should use a more unified approach to different components of the system. In the case of the IX, classical workstations, programming the system can be separated into two parts: first, the definition of the instrument or signal processing algorithm with the "patch" language, and the second, being the control of the instrument in real-time, using preprogrammed inputs, through a "control" language.

One of the most difficult points of such a system, for a novice user, is programming and debugging an algorithms composed of equal processing elements (scaling, sampling, filtering, and the like) which machine thinks as the IX requires.

To solve this problem, we have designed and implemented a mouse-driven, bitmap graphics interface for MAX, incorporating the IX. The interface consists of a graphic command language, a graphic editor, and a graphic debugger.

The graphic command language makes a link between the graphic world and the ASCLL text world. It makes a translation between the "patch" language and the bit-mapped images in both directions. It allows the user of commands for IX and bit-loaded binary files. Figure 1 shows the different commands available in the language.

The graphic editor permits the manipulation, creation, and symbolic manipulation of signal processing modules, which we will harbor forth in "functions," as an iconic form. The icon symbols generate primitive addition, subtraction, multiplication, \textit{etc.}, and other functions implicitly defined. The icon allows icons on the screen and makes connections by drawing lines between them. These connections symbolize the flow of data between functions. The editor supports the hierarchical management of its function library. Figure 2 shows the organization of an oscillator by means of language primitives. Figure 3 shows the creation of an oscillating function, also. Figure 4 shows an description of an oscillator tank, using pre-defined functions.

The graphic debugger is primarily a tool for interactive testing and debugging with the IX. After a patch has been compiled and loaded into the IX, we can examine the patch, graphically, traverse its hierarchy, i.e., follow the connections of each icon in the patch, and at any point select its, visualize, or manipulate any of its data using a graphic window.

In conclusion, we have developed a set of graphic tools which provide an elegant solution to the problems outlined in the introduction. The multirho in real-time was made possible by the development of software distributed between the VME-68000 and a host computer.

Two versions of the interface described have been developed. First, implemented on a Macintosh, permits programming and is in a patch in real-time. The Mac provides lexical, syntactic and semantic analysis, and sends commands to an interface running on VME-68000 which is composed of a macro-language of commands, a macro-code optimizer, a loader, and several editors. The suspension is in development and uses a SUN 2/390 and VME-68000. This version will have two options: the first will all programming and testing in real-time, the second will only do testing of compiled patches, but these patches will have undergone better optimizations.

373

ICMC 86 Proceedings