ATRテクニカルレポート表紙

[非公開]

TR-M-0058

Dressable Music:
a Musical System Creating a Parallel World

ロマイン　ルーブ　Romain Rouve
多田　幸生　Yukio Tada
前川　督雄　Tadao Maekawa
西本　一志　Kazushi Nishimoto
間瀬　健二　Kenji Mase
中津　良平　Ryohei Nakatsu

2000.12.27

ATR 知能映像通信研究所

Dressable Music : a Musical System Creating a Parallel World

Romain Rouve (*rouve@email.enst.fr*)
(ATR / ENST)

Yukio Tada (ATR), Tadao Maekawa (ATR),
Kazushi Nishimoto (ATR / Japan Advanced Institute
of Science and Technology, Hokuriku),
Kenji Mase (ATR), Ryohei Nakatsu (ATR)

December 27, 2000

Contents

List of Figures

List of Tables

1 Abstract

Playing a musical instrument is often restrictive. It requires a special stance or equipment. For example, it is hard to play piano, guitar or violin while walking. This project intends to propose a new type of musical instrument which is both simple and practical and can be used *anywhere, anytime and with or by anybody.*

The instrument is designed as clothing and can thus be worn. It plays music using MIDI and wireless transmissions. This allows anyone to play music more easily and lightheartedly than usual. It can be played with people in the streets as well as friends at home and even while walking with random people you meet. It opens a new kind of music depending on the feelings and the situation.

In this report, I will describe more precisely the project and introduce technical concepts used, like *the MIDI protocol* or *the ITRON specifications.* Then I will precise my work in the conception of the instrument and conlude with future evolutions of the project.

2 Introduction to the Project

2.1 An Active Entertainment

The first idea of the project is to create a musical instrument which is both is easy and practical to use. With it you can hear music but also play music whenever and wherever you like. You can play it in your home or while walking in the streets of a town you are visiting or even when you are going shopping. This is what is called "Active Entertainment" as opposed to an actual mobile music player such as MD or portable CD player which offer no opportunity to participate (and thus will qualify as "Passive Entertainment").

An important point is to make this instrument as mobile as possible but also to make its use very simple so that it would not be a pain to play for the person who is using it. However, this simplicity should not make the instrument totally obsolete and still interesting to use for confirmed musicians. In a way, it should be customizable according to the person using it.

Finally, the idea of a wearable instrument seems to solve these problems.

2.2 A Wearable Instrument

The musical instrument has the shape of clothes like pants, jackets or gloves. On these clothes touch sensors are fixed. When you touch the sensors it makes a sound using MIDI datas and a tone generator. Since you can wear it, you can take it with you everywhere and you can play it while walking. The instrument is really mobile and portable.

Moreover, the positions of the sensors on the clothes (especially on the jacket) can be customized so that a confirmed musician will find more possibilities than with the beginner version. For example, the simpliest version could have only the 8 basics notes of an octave and another version could have all the half pitches...

Besides, the customization is not limited to the positions of the sensors. Since the instrument uses a MIDI tone generator to produce sound, it is not limited to only one kind of instrument. The user has the possibility to change the sound of his instrument as he likes, among the sounds available in the tone generator. And nowadays MIDI tone generators have an impressive amount of choices.

Thus, the instrument, like clothing do, tells little about its owner. This one can choose the type of music and the way of playing it, according to his feelings and impressions of the moment. However, these ideas will be nothing if it is not possible to share your music as easily as you play. As a consequence, the project assumes to connect the users to each other via a wireless network using once more MIDI transfer protocol for the music data in the hope of creating a “musical communityware”.

2.3 Creating a Musical Communityware

The instrument will be able to communicate with servers on the streets that broadcast musical accompaniments. But it will also be able to communicate with other instruments and open “open session” with them to have a performance. This ability to communicate offers countless possible applications (figure 1 on page 6 shows a set of such possible applications).

For example, you can place several servers in town which will broadcast typical music of the town (you can imagine having new age music in Tokyo and more classical japanese music in Kyoto). Consequently, while visiting a town, you will see it but you can also hear it. Moreover, you can play music to the accompaniment of the town and enhance it with your impressions. Finally you can exhange with the server your performance and the accompaniment so that the music in the town will evolve according to the people living in it and so that you can listen to this music even when you get back home.

Another example is the possibility to have ad-hoc session at almost anytime depending on the random people you meet. Imagine you are playing a jazz theme on the street and you meet somebody who is playing the same kind of music. Then you can open a session and have a jam with him in the street although you barely known him. And people could also join your session if they like it and you can leave it if you see you'll be late for your date. By this way, new relationships could be created among people who haven't

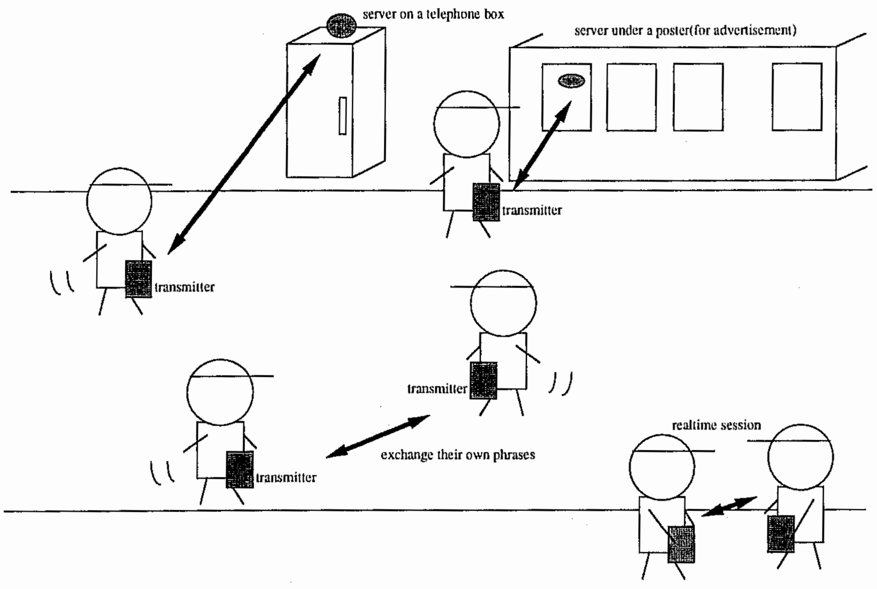
server on a telephone box
server under a poster(for advertisement)
transmitter
transmitter
transmitter
transmitter
exchange their own phrases
realtime session

Figure 1: Possible applications

talked to each other and, at the end, a whole musical communityware could break through.

From a practical point of view, considering the evolution of the telecommunication technologies, you can imagine plugging your instrument into a mobile phone that can perform all the operations it needs : customizing your instrument, generating the music, communicatig with other instruments...

3 Technical Description

3.1 The Client Part

3.1.1 hardware description

Figure 2 on page 7 shows the hardware block diagram of a client. It decribes the hardware implementation of a user instrument. The different modules are:

- *A control unit* (CPU and RAM). The CPU will have an ITRON operating System. ITRON has been chosen for its *real-time* abilities (see section 5 page 13).

- *An A/D convertor* which converts the signal of the sensors into MIDI commands (for example).

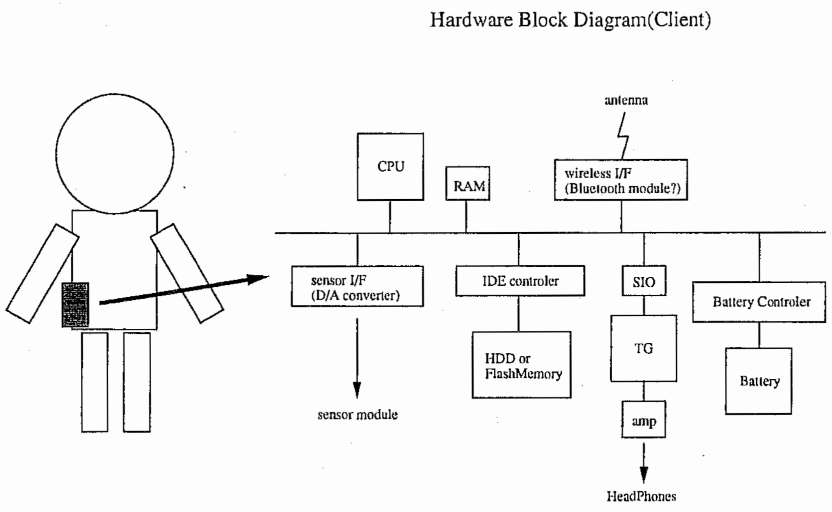

Figure 2: Client hardware block diagram

- *A memory storage utility.* That will allow the user to store his data. It could be either a hard disk drive or flash memory.
- *A sound generator.* Basicly, a MIDI tone generator and an amplifier.
- *A communication module* using for example the new Bluetooth protocol for transmitting the data to the server or the other users.
- *A battery pack,* of course.

3.1.2 software description

Figure 3 on page 8 shows the software block diagram of a client. It describes the different tasks an intrument should handle and how they interact.

A program remaps the input it gets from the sensor and sends it to the sequencer. With the help of this *remapper* and because of the properties of MIDI datas (see section 4 page 10) you can customize the instrument as you want.

Then a *sequencer program* mixs the sound it gets from the sensor with the one it gets either from the other users or a server (via the wireless comminucation controler), or from the phrase database of the user. The *sequencer* sends the melody to the *tone generator controler.*

The *phrase exhange controler* handles all the communications and exhanges with other users or servers. It also gets the phrases stored by the user via a *database manager that* controls the I/O on the memory drive.

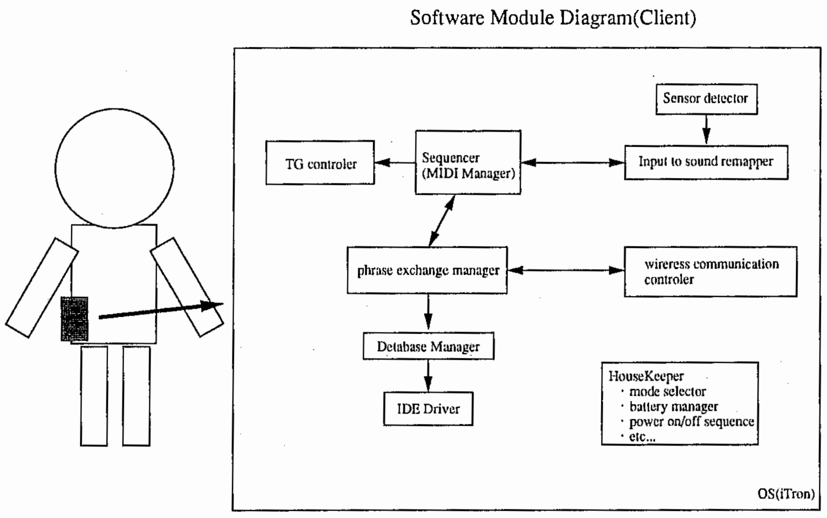

Figure 3: Client software block diagram

3.2 The Server Part

3.2.1 hardware description

Figure 4 on page 9 shows the hardware block diagram of a server. It decribes the hardware implementation of the server. Its structure is much simpler; it is only composed of:

- *A control unit* (CPU and RAM).
- *A memory storage* utility where the phrases will be stored. Unlike the memory of the client, this one must be much bigger. That is why a hard disk drive will be used.
- *A communication module,* using for example the new Bluetooth protocol for transmitting the data to the users.

3.2.2 software description

Figure 5 on page 9 shows the software block diagram of a server. It describes the different tasks the intrument should handle and how they interact. Once again, it is much simpler. Since the server has only a phrase exhange function, the only modules it has are the *phrase exhange manager the database manager*, and the *wireless communication controler.*

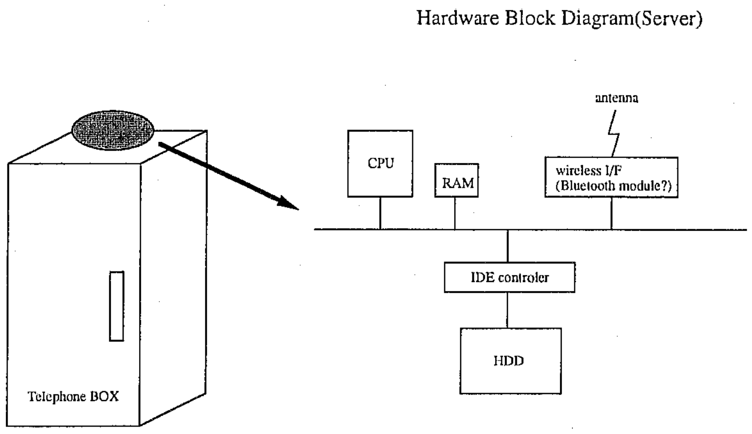
Hardware Block Diagram(Server)
antenna
CPU
RAM
wireless I/F (Bluetooth module?)
IDE controler
HDD
Telephone BOX

Figure 4: Server hardware block diagram

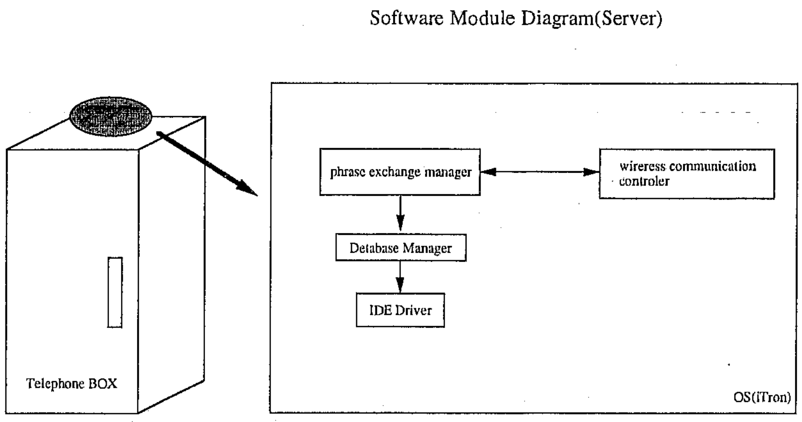

Figure 5: Server software block diagram

4 The MIDI specification : a short explanation

This section introduces basic concepts about the MIDI protocol. For more detailed explanations please refer to the MIDI specification, published and maintained by the *MIDI Manufacturers Association*[1] (MMA).

4.1 Introduction

MIDI stands for *Musical Instrument Digital Interface* and is a communications protocol that allows electronic musical instruments to interact with each other.

Much in the same way that two computers communicate via modems, two synthesizers communicate via MIDI. The information exchanged between two MIDI devices is musical in nature. MIDI information tells a synthesizer, in its most basic mode, when to start and stop playing a specific note. Other information shared includes the volume and modulation of the note, if any. MIDI information can also be more hardware specific. It can tell a synthesizer to change sounds, master volume, modulation devices, and even how to receive information. In more advanced uses, MIDI information can indicate the starting and stopping points of a song or the metric position within a song. More recent applications include using the interface between computers and synthesizers to edit and store sound information for the synthesizer on the computer.

The basis for MIDI communication is the *byte*. Through a combination of bytes a vast amount of information can be transferred. Each MIDI command has a specific byte sequence. The first byte is the *status byte*, which tells the MIDI device what function to perform. Encoded in the status byte is the *MIDI channel.*MIDI operates on 16 different channels, numbered 0 through 15. MIDI units will accept or ignore a status byte depending on what channel the machine is set to receive. Only the status byte has the MIDI channel number encoded. All other bytes are assumed to be on the channel indicated by the status byte until another status byte is received. The status byte is divided in two 4-bit nibbles. The channel is encoded in the low nibble (ie, the four less significant bits) and the function is encoded in the high nibble (ie, the four most significant bits).

Depending on the status byte, a number of different byte patterns will follow. For example, the *Note On* status byte tells the MIDI device to begin sounding a note. Two additional bytes are required, a pitch byte, which tells the MIDI device which note to play, and a velocity byte, which tells the device how loud to play the note. Even though not all MIDI devices recognize the velocity byte, it is still required to complete the Note On transmission.

The status byte has his Most Significant Bit set to 1. This bit is set to 0 for the additional bytes. Moreover, a status byte does not need to be

[1]http://www.midi.org

repeated if the next MIDI command is the same. Consequently, if there are more additionnal bytes than needed, it would be interpreted as several MIDI commands with the same status byte. So, a Note On byte can be followed by a pitch byte and a velocity byte and another pitch byte and velocity byte, etc...

4.2 Basic Midi Commands

Some of the functions indicated in the status byte are *Note On*, *Note Off*, *System* command such as *System Exclusive* (SysEx), *Program Change*, *Control Change*...

We have already seen the *Note On* command (0x9n [2]) which make the MIDI device play a note. The command to stop playing a note is not part of the Note On command; instead there is a separate *Note Off* command (0x8n). This command also requires two additional bytes with the same functions as the Note On byte. You can also stop playing a note by sending a Note On Command with a velocity of 0.

The *Control Change* byte (0xBn) allows you to change some settings of the synthesizer such as the volume or the balance of a channel. It is followed by two additionnal bytes: the controller type (volume, balance...) and its value.

Another important status byte is the *Program Change* byte (0xCn). This requires only one additional byte: the number corresponding to the program number on the synthesizer. The program number information is different for each synthesizer, and the standards have been set by the MIDI Manufacturers Association (MMA). Channel selection is extremely helpful when sending Program Change commands to a synthesizer.

The *System* commands (0xFn) are very general commands and do not apply to any channel; instead the low nibble design tells which system command will be performed. For example the *Reset* command is 0xFF and does not need any additional byte.

The *SysEx* status byte (0xF0) is the most powerful and least understood of all the status bytes because it can instigate a variety of functions. Briefly, the SysEx byte requires at least three additional bytes. The first is a manufacturer's ID number or timing byte, the second is a data format or function byte, and the third is generally an "end of transmission" (EOX=0xF7) byte...

Table 1 on page 12 gives a quick overview of the different MIDI messages.

[2] Note that I'm using the C programming language convention of prefacing a value with 0x to indicate hexadecimal. *n* design the number of the channel.

MIDI Message	Status Byte	Data Byte	Data Byte
Note Off	0x8n	Note Number	Velocity
Note On	0x9n	Note Number	Velocity
Polyphonic Aftertouch	0xAn	Note Number	Pressure
Control Change	0xBn	Control Number	Data Information
Program Change	0xCn	Program Number	-
Channel Aftertouch	0xDn	Note Number	Pressure Value
Pitch Wheel	0xEn	Value1	Value2
System Commands	0xFn	...	...

Table 1: MIDI Messages

4.3 The Midi File Format

The *Standard MIDI File* (SMF) is a file format specifically designed to store the data that a sequencer records and plays (whether that sequencer is software or hardware based).

This format stores the standard MIDI messages (ie, status bytes with appropriate data bytes) plus a time-stamp for each message (ie, a series of bytes that represent how many clock pulses to wait before "playing" the event). The format allows saving information about tempo, pulses per quarter note resolution, time and key signatures, and names of tracks and patterns. It can store multiple patterns and tracks [3] so that any application can preserve these structures when loading the file.

The format was designed to be generic so that any sequencer could read or write such a file without losing the most important data, and flexible enough for a particular application to store its own proprietary, "extra" data in such a way that another application will not be confused when loading the file and can safely ignore this extra information that it does not need. The MIDI file format is a kind of musical version of an ASCII text file (except that the MIDI file contains binary data too), and the various sequencer programs are mere text editors all capable of reading that file.

However, unlike ASCII, MIDI file format saves data in *chunks*, groups of bytes preceded by an ID and size, which can be parsed, loaded, skipped, etc. Therefore, it can be easily extended to include a program's proprietary information. For example, maybe a program wants to save a "flag byte" that indicates whether the user has turned on an audible metronome click. The program can put this flag byte into a MIDI file in such a way that another application can skip this byte without having to understand what that byte

[3] A track usually is analogous to one musical part, such as a Trumpet part. A pattern would be analogous to all of the musical parts (ie, Trumpet, Drums, Piano, etc...) for a song, or excerpt of a song.

is for. In the future, the MIDI file format can also be extended to include new "official" chunks that all sequencer programs may elect to load and use. This can be done without making old data files obsolete (ie, the format is designed to be extensible in a backwardly compatible way).

5 What is ITRON ? : a short introduction to the ITRON Project

The ITRON[4] [5] [6] Project creates standards for real-time operating systems used in embedded systems and for related specifications. Since the project started, several ITRON real-time kernel specifications have been drawn up and offered them to the public.

In order to meet the requirements of real-time processing and embedded systems, the ITRON OS specifications were designed according to the following principles:

Allow for adaptation to hardware, avoiding excessive hardware virtualization in order for an OS to take maximum advantage of the performance built into the Microcontroller Unit (MCU) or other hardware. Specifically, the ITRON specifications make a clear distinction between aspects that should be standardized across hardware architectures (task scheduling rules, system call names and functions, parameter names, error code names...) and matters that should be decided optimally based on the nature of the hardware and its performance (parameter bit size, interrupt handler starting methods...).

Allow for adaptation to the application, which means changing the kernel specifications and internal implementation methods based on the kernel functions and performance required by the application, in order to raise the overall system performance. In the case of an embedded system, the OS object code is generated separately for each application, so adaptation to the application works especially well.

In designing the ITRON specifications, this adaptation has been accomplished by making the functions provided by the kernel as independent of each other as possible, allowing a selection of just the functions required by each application.

Specification series organization and division into levels. To enable adaptation to a wide diversity of hardware, the specifications are organized into a series and divided into levels.

[4]TRON is an abbreviation of "The Real-time Operating system Nucleus."

[5]ITRON is an abbreviation of "Industrial TRON."

[6]TRON and ITRON are names of concepts and projects aimed at developing a new computer system and environment; they do not refer to any specific product or products.

Provide a wealth of functions and provide primitives that are not limited to a small number. By making use of the primitives that match the type of application and hardware, system implementers should be able to achieve high runtime performance and write programs more easily.

The major feature of ITRON specifications is their adaptability which can be implemented even on compact hardware systems having extremely limited amounts of memory but also which can be optimized for the application. Of these specifications, the μITRON real-time kernel specification, which was designed for consumer products and other small-scale embedded systems, has been implemented for numerous 8-bit, 16-bit and 32-bit Microcontroller Units (MCUs) and adopted in countless end products, making it an industry standard in this field.

The detailed specifications of ITRON are available at the *ITRON Project Home Page*[7]

6 Description of the Work Done

In order to present a prototype during the *ATR Open House* (November 1st and 2nd 2001), the project has been simplified. It was decided to design three prototypes capable of playing in an open session with a background music played by a server. Each instrument sends the notes it plays to the server. The server mixes the notes from each instrument with the background music (thanks to a sequencer program) and then sends the whole melody to the instruments. I was in charge of a part of the sequencer program and of the design of two instruments: the *vest type* and the *pants type*.

The sequencer has been designed in three stages. The first one was to create a program that will read the MIDI commands in the input of the *YAMAHA MMP Core*, interprete it, and dump the interpretation on the output monitor running on the computer. During the second step, I had to create several programs that will write then read and play a MIDI file from a flash memory card. For this, I used a set of functions already implemented by YAMAHA. Finally, I was able to create a sequencer mixing the background music read from a flash memory card with the notes it received (MIDI commands) via the input by melting the former programs together. A detailed description of these programs is presented in section 7 page 15.

The instruments were designed after the sequencer implementation. The sequencer was then very usefull while mapping the sensors on the prototypes. A detailed decription of this step is available in section 8 page 17.

The CPU of the server was a *YAMAHA MMP Core* with an ITRON operating system (figure 6 on page 15 shows the hardware block diagram

[7]http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html

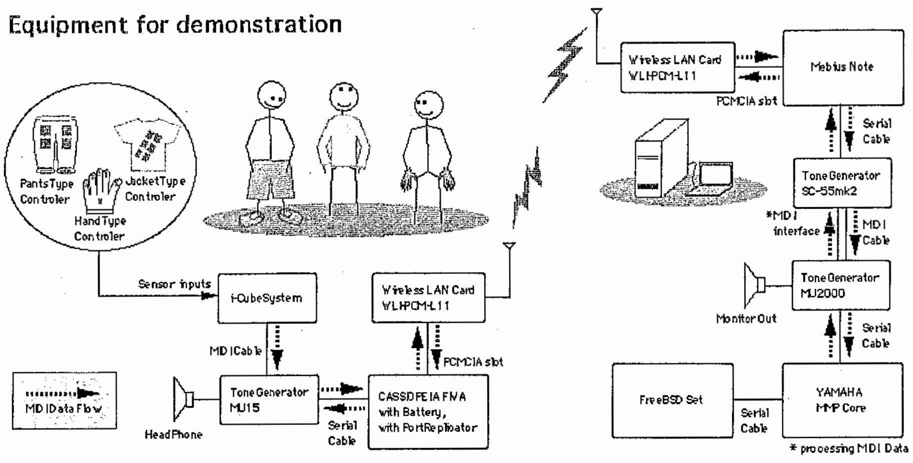

Figure 6: Server hardware block diagram for the Open House

of the prototype). All the programming has been made using the development kit of the YAMAHA MMP Core on a FreeBSD Set (ie, a compiler, a debugger, a monitor...) and the language of programming is ANSI C.

The people in charge of the project were (i)Mr. Tadao MAEKAWA, my supervisor, (ii)Mr. Yukio TADA, from YAMAHA temporaly working for ATR, and (iii)Mr. Kazushi NISHIMOTO.

7 Structure of the Server Program

As said in the former section, the conception of the server was done in three steps. Each step is described below. For a more precise description of each module presented in this section please refer to appendix B page 22.

7.1 Reading and Dumping Midi Data

The first step was to make a program that will read MIDI data in the input on the YAMAHA MMP Core and then dump the data on a monitor after its interpretation. The interpretion was actually a translation of the MIDI in to text command. For example the command *0x90 0x3C 0x64*, would be interpreted as *NoteOn 1 C3 100* [8].

In order to make the program more efficient for real-time sessions, two different *tasks* (or *process*) are working at the same time. The first one reads the data in the input and stores it in a ring buffer. The second one reads the data from this ring buffer, interprets it and then dumps the interpretation. The two task read and dump access to two different I/O devices. If these

[8]It should be read "a *Note On* command on channel *1* that play a *C3* note with a velocity of *100*".

tasks were implemented as one: when some trouble occured on one I/O device (for example a delay), it will spread to the other device, which may have a bad influence for the device connected to this I/O. Therefore it had to be divided it into two tasks using ringbuffer in order to prevent such a trouble beforehand

The ring buffer declared as a global variable. It is actually a table of 256 bytes (or *unsigned char* as declared in C). In order to implement this structure a package has been created (*ringbuffer*). This module offers three functions for (i)initializing the ring buffer, (ii)writing something in the ring buffer and (iii)reading something from the ring buffer. Moreover, this module implements a semaphore structure to control the critical I/O on the ring buffer. This structure works as a mere producer/consumer system.

Two other modules were implemented (*midiread* and *midiplay*), each one containing the necessary functions for the reading and the dumping task. All the main program has to do is to initialize the ring buffer, create and then start the tasks.

This step required a good understanding of the MIDI protocol for the interpretation of the data and a knowledge of the ITRON operating system specifications for the creation and the use of tasks and semaphores.

7.2 Writing and Dumping Midi Files

The next program was to make the YAMAHA MMP Core play a MIDI file that was written on a flash memory card. In fact, the MMP Core does not "really" play the music. It reads the *chunks* of a MIDI file (see section 4 page 10) and sends MIDI commands to a tone generator according to what is read from the chunks.

My work was really made easy by the use of a set of functions already implemented by YAMAHA that can (i)read and write data on a flash memory card and (ii)translate MIDI file data stored in a variable into MIDI command and send it to the tone generator (ie, the standard output). The first set of packages is called DBM (for DataBase Manager), the second one is called MMSAPI (MultiMedia Sequencer API). I created a program that write the MIDI file stored in a variable onto the flash memory card and then a second program that read this MIDI file and stored it into a variable so that it can then be played by MMP Core thanks to the functions already implemented.

7.3 The Sequencer

The final step was to conceive a sequencer that would mix background music and notes played by the three instruments. Actually, the sequencer does not collect the notes by itself. It receives them via its standard input (see Hardware Block Diagram, figure 6 page 15). The sequencer just mixes the

MIDI data from its standard input with the MIDI data "translated" from a MIDI file (the background music) and sends it to its output.

The program of the sequencer is obtained by merging the two former programs and adding some functionalities. Once again, the functions implemented by YAMAHA were useful. Indeed these functions create a new task for playing the MIDI file. Thus I had to make important changes only on the first program.

First, the main module of the first program which initialized the ring buffer and created and started the tasks was implemented in a new package called *midiplay*. It has two functions, (i)one making all the initializations (MIDI I/O and ring buffer) and creating the tasks, (ii)the other starting the tasks. Then I replaced the interpretation function of the dumping module into a remapper function that will allow me to configure and design the clothes.

As for the second program, the main module becomes a new package, *smpplay*.

Note: for the ATR Open House, this program has been embedded by Mr. Yukio TADA. The program plays a sucession of three music accompaniments repeated 4 times in an infinite loop. The music accompaniments have been composed by Mr. Yukio TADA and the midi files were not written on the flash memory card, but stored in three variables...

You can see in appendix A the functional block chart of the final program.

8 Clothes Design

During this part of my work, I had to decide about the mapping of the sensors on two instruments (the vest and the pants) and about the note and the sound each sensor would play.

The sensors were connected to an I-Cube System (A/D converter) which transforms the signals of the touch sensors into a *Note On* command. The I-Cube System was configured so that the couple note played and channel was unique. That is to say that if we know the note and the channel, we can dertermine which sensor has been activated and transform the command into any command we want thanks to the MIDI dumping program (see section 7 pageref 15).

8.1 Pants Type

When you play music with your pants, you are usually trying to play percussions. That's why it seems natural that the pants play the drum part of the music. The instrument has thus been conceived as a little drum kit.

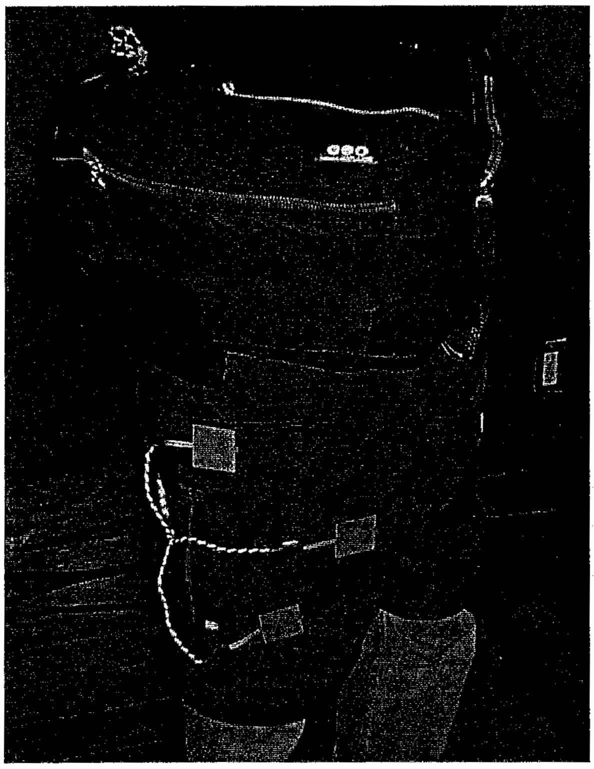

Figure 7: The pants prototype

Although it is more comfortable to play this instrument while sitting you can also play it while walking.

Detail wise, the pants are very simple. They can only play one type of instrument and the notes are always the same (see figure 11 on page 27 in the appendix C). The most tricky part was to find the correct percussions to put on the sensors.

8.2 Vest Type

The design of the vest is a little bit more complicated. Instead of a vest playing some disparate set of sounds, the vest has been designed so that it can play a real melody.

The vest has eleven sensors. Eight of them play notes (C, D, E, F, G, A, B, C) and the other ones are command sensors: *Octave Up*, *Octave Down*, *Change Instrument.* That means that the vest is not limited to eight notes of one instrument. It can play all the notes of several instruments (see table 3 on page 26 in the appendix C). Moreover, the dispersal of these sensors made them practical to use with the hands (see figure 10 on page 25 in the appendix C).

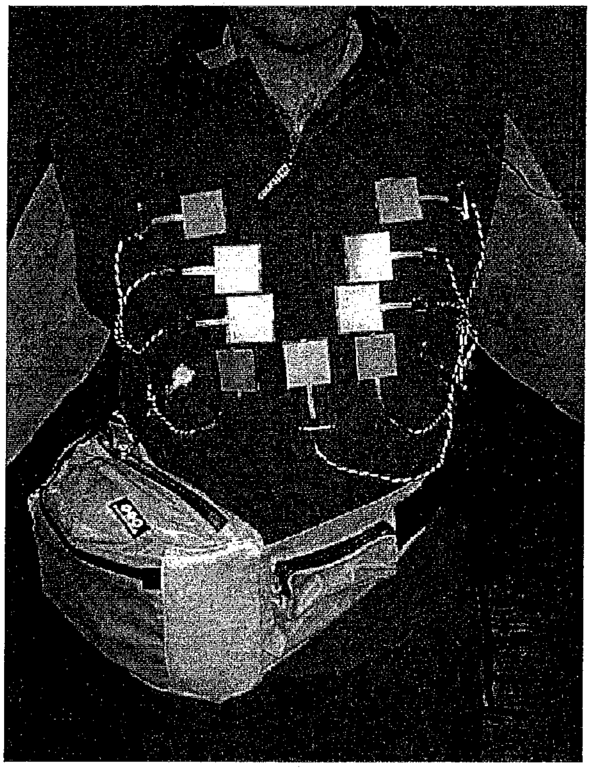

Figure 8: The vest prototype

More specificly, the *Octave Up* and the *Octave Down* commands actually perform two actions. They first send an *All Note Off* command (0xBn 0x7B 0x00) on the requested channel (n) to prevent any excessive duration of a note. Then they increment or decrement a variable used to calculate the note played by the vest in the remapping module. As for the *Change Instrument* command, it only sends a *Program Change* command on the requested channel (see section 4 page 10 for details on these MIDI commands).

The three control sensors use a different channel than the other sensors. This makes the distinction easier (particularly when programming the remapping module in the dumping program) and more practical when sending *All Note Off* and *Program Change* commands.

9 Conclusion

The design of the other instrument, musical gloves, the set up of the demonstration and the rest of the programming were made by Mr. Yukio TADA[9]. For the demonstration Mr. Kazushi NISHIMOTO was playing the jacket, Mr. Yukio TADA was playing the gloves, and I was playing the pants. Finally the prototypes worked quite well and people seemed interested when we showed them during the Open House, and several other presentations we have made (a french article has even been published on it[10]). However there is still lots of work to do until the project is completed.

As I said earlier, the prototype is really a simple version of the initial project. Only three instruments could play in a unique open session, and they were quite forced to participate. You couldn't switch on a button and play your own melody apart from the others. It would simply not work.

The next expansion would be to transfer the sequencer program in the instrument itself and make it autonomous. Then the phrase exhange module can be implemented. However, I see several problems that should be first resolved.

First of all, the MIDI specification as it is first defined is limited by sixteen channels. If you consider that the accompaniment background takes eight channels, this means that you can't have more than seven instruments playing together (assuming that you have a channel reserved for control commands and that each intrument takes only one channel). Nevertheless I have heard that there is some way to get more than sixteen channels, so this problem is not unsolvable.

Another problem is to define a sort of protocol for sessions where issues like opening sessions and channel attributions will be solved. The same is for the phrase exhange sessions...

Finally, the integration of the project on to mobile phone, although it sounds interesting, seems very ambitious but not impossible after all. The musical capabilities of modern mobile phones are quite restrictive but considering the evolution of telecommunications, we can hope that in the near future they will elvove enough.

[9]The communication module between the server and the instruments using the protocol UDP has been programmed by Mr. Tadashi TAKUMI (CSK Corp.)

[10]http://www.internetactu.com/archives/dossiers/atr/index.html

A Functionnal Block Chart

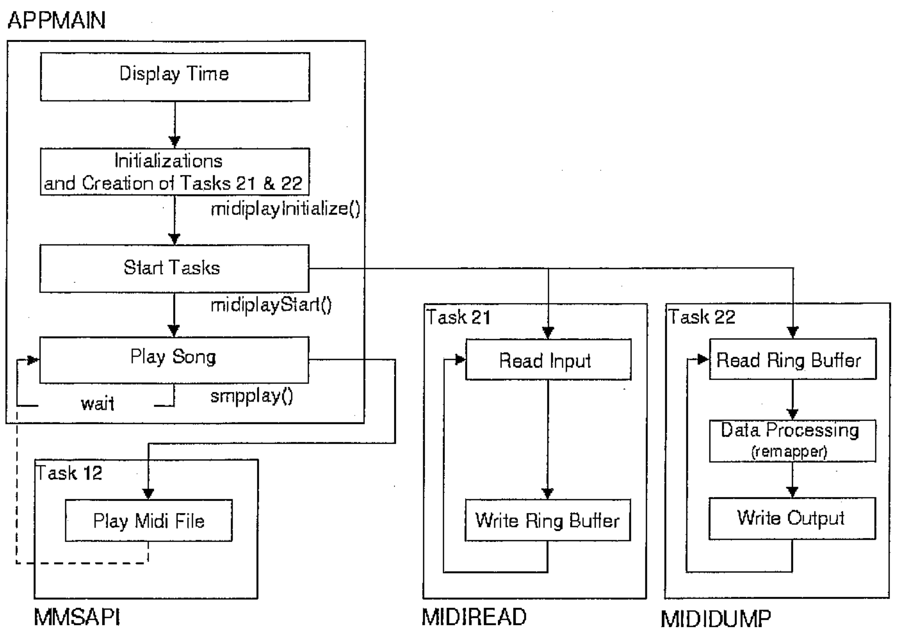

Figure 9: Global Chart

B Description of the Functions

B.1 *appmain.c*

This program contains the main function void *appmain*(void).

B.2 The Package *ringbuffer*

This module implements a simple ring buffer that will be used to store the data read on the input waiting for its processing. There are three functions:

***void* initialize_buffer(*void*):** which resets the ring buffer and create a semaphore to control the i/o access on the ring buffer.

***int* write_buffer(*unsigned char *data, unsigned char len*):** which writes a data in the ring buffer. It returns 0 if the operation has been sucessful or -1 otherwise.

> ***unsigned char *data*:** pointer to the data to be written in the ring buffer.
>
> ***unsigned char len*:** length of the data in byte.

***int* read_buffer(*unsigned char *data, unsigned char len*):** qwhich read a data from the ring buffer. It returns 0 if the operation has been sucessful or -1 otherwise.

> ***unsigned char *data*:** pointer to the buffer where the data that has been read will be stored.
>
> ***usigned char len*:** length of the buffer.

B.3 The Package *midiplay*

This package contains the functions of initialization and start for the MIDI I/O. There are two functions:

***ER* midiplayInitialize(*void*):** makes all the initializations for reading and dumping programs, ie, initializes MIDI I/O devices, the ring buffer and the tasks for reading and dumping. It returns a standard ITRON error code.

***ER* midiplayStart(*void*):** starts the reading and dumping tasks. It returns a standard ITRON error code.

B.4 The Package *midiread*

This module implements the function used by the task that reads midi data on the input :

***void* midiread(*void*):** read midi data on the input and store it on the ring buffer.

B.5 The Package *mididump*

This module implements the function used by the task that dumps midi data on the output (tone generator):

***void* mididump(*void*):** reads midi data from the ring buffer, process the data and dump it to the output.

B.6 The Package *smpplay*

This package contains the functions used to start playing the background music. Originally, this package reads a file from the flash card and play it once but it has been modified by Mr. Yukio Tada for the demonstration. Then it can play three music in an infinite loop, but these music are stored in variables. There are three functions:

***void* smpSetup(*UB const *pSetup*, *int nSetup*):** interprets Control Change and Program Change command and send it to the tone genrator.

***UB const *pSetup*:** pointer to the command string

***int nSetup*:** length of the command string

***void* smpPlaySong(*UB const *pSmf*, *UB const *pSetup*, *int nSetup*, *int nTimes*):** makes four metronome clicks and then plays the song contained in *pSmf nTimes* times with the setup commands contained in *pSetup*.

***UB const *pSmf*:** pointer to the song (MIDI file format)

***UB const *pSetup*:** pointer to the setup string

***int nSetup*:** length of the setup string

***int nTimes*:** number of times the song will be repeated

***int* smpplay(*void*):** makes the initializations for the MMSapi and starts playing three songs four times in an infinite loop. It uses three variables that contains the midi files data. These variables are affected in the libraries *data_fusion.h*, *data_dancepop.h* and *data_foxtrot.h*. It returns -1 if an error occured or 0 otherwise.

B.7 The Package *play_hand*

This module contains the remapper function for the hand type instrument. It has been programmed by Mr Yukio Tada for the demonstration. There are two functions:

***void* phdTGSetup(*void*):** sets the configuration of the tone generator for the controller.

void **phdRemapEvent(*****int nCh*****,** ***int nNote*****,** ***int on*****,** ***unsigned char aBuf[]*****):**
the remapping fuction itself.

- ***int nCh*****:** number of the channel of the command.
- ***int nNote*****:** note that has been played (the note and the channel determine which sensor has been touched...)
- ***int on*****:** 1 if it's a Note On command, 0 otherwise
- ***unsigned char aBuf[]*****:** buffer where the new command will be stored.

B.8 The Package *dbm*

This package contains the functions to be used for writing and reading files. It has been implemented by YAMAHA.

B.9 The Package *mmsapi*

This package contains the functions for playing a midi file. It has been implemented by YAMAHA.

C Mapping of the Sensors

C.1 Vest type

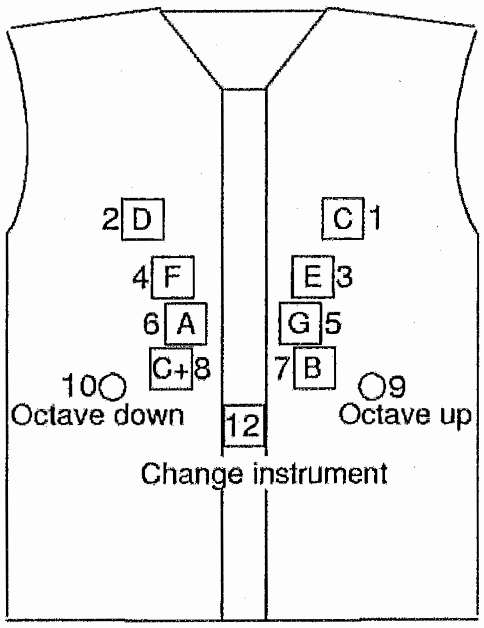

Figure 10: Vest type : sensors number and function

Vest sensor	1	2	3	4	5	6	7	8
I-cube sensor	1	2	3	4	5	6	7	8
MIDI Channel used	5							
Note played by I-cube	C3	D3	E3	F3	G3	A3	B3	C4

Vest sensor	9	10		12				
I-cube sensor	9	10	11	12	13	14	15	16
MIDI Channel used	6				7		8	
Note played by I-cube	C3	D3	E3	F3	C3	D3	C3	D3

Table 2: Sensor mapping for the vest

Instrument Number	Instrument
1	Grand Piano
2	Jazz Guitar
3	Distorsion Guitar
4	Finger Bass
5	Violin
6	Orchestral Harp
7	Strings 1
8	Synth Brass 2
9	Baritone Sax
10	Flute
11	Whistle
12	Voice Lead
13	Crystal
14	Shamisen
15	Koto
16	Melodic Tom

Table 3: The different instruments available on the vest

C.2 Pants type

Pants sensor	1	2	3	4	5	6	7	8
I-cube sensor	1	2	3	4	5	6	7	8
MIDI Channel used	3							
Note played by I-cube	C3	D3	E3	F3	G3	A3	B3	C4

Vest sensor								
I-cube sensor	9	10	11	12	13	14	15	16
MIDI Channel used	4							
Note played by I-cube	C3	D3	E3	F3	G3	A3	B3	C4

Table 4: Sensor mapping for the pants

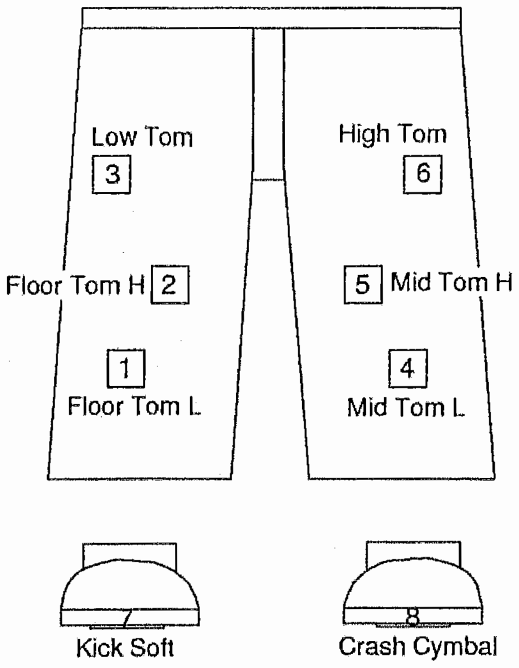

Figure 11: Pants type : sensors number and function

References