Abstractions for Continuous-Media Programming

Ralf Steinmetz
IBM European Networking Center
Tiergartenstr. 8, 6900 Heidelberg, Germany
Tel: +49 6221 404280, Fax: +49 6221 404450
E-mail: steinmet at dhdibm1.bitnet

J. Christian Fritzsche
Johann Wolgang Goethe-University of Frankfurt
Computer Science Department
Robert-Mayer-Strasse 11-15, 6000 Frankfurt, Germany
Tel: +49 69 7988373, Fax: +49 69 7988353
E-mail: fritzsch at rbiffm.informatik.uni-frankfurt.de

Abstract

This paper surveys different techniques for programming multimedia applications. To the notion of the authors, so far no work on multimedia programming as integral part of high level languages has been performed (as it may be, to treat media as types). Some new ideas and concepts in this direction are presented in this document.

1. Introduction

Work on multimedia computing and communications has been focused on the provision of suitable workstation and network components together with appropriate software technology. The HeiTS ("Heidelberg High-Speed Transport System") prototype under development at IBM ENC in Heidelberg is one of these systems [7]. Today's multimedia applications are usually programmed in conventional languages (such as C), augmented with hardware-specific multimedia libraries. Replacing any underlying continuous-media device, even with a functionally-equivalent component from another vendor, often requires reimplementing a substantial part of the application programs. Some applications may have been produced with tools either generating or providing the code to interface with the multimedia devices. In such a case, any replacement of the multimedia equipment requires major changes in the tools; new interfacing methods; and, at the least, regenerating the applications' executable code.

Thus, the following questions arise:

1. Why are multimedia application so hardware dependent?
2. How can this problem be overcome?

An analogy can be made to techniques for programming with floating-point numbers. The diverse hardware engines for performing floating point processing are also different in terms of architecture, instructions and interfaces. Sometimes RISC archi-

tectures or parallel processing are used. Nevertheless, only a few standard representation formats such as the IEEE format are used. Programmers use built-in functions of high-level languages (HLL) for their programming with real numbers. Any change in the hardware would rarely affect, e.g., a Fortran application program.

Compared to our multimedia environment we find relatively well-defined abstractions, usually HLL data types, within the programming languages. It is thus possible to hide the actual hardware from the application without any major decrease in performance.

The research community frequently approaches multimedia programming within object-oriented environments (see, e.g., [2; 3; 6; 16] at one workshop on multimedia [8]or [20] as further examples). We encompass a similar development in the presentation of communication functions to the applications. "Multimedia objects" allow for a fast integration of all kinds of very different capabilities and functions with the environment itself. Unfortunately the class hierarchies encountered are very dissimilar: Today, there is no consensus on a common or "the best" class hierarchy. A multimedia product developed with an object-oriented language is still the exception.

In this paper we describe various possible abstractions for continuous media, and as a new concept we propose to treat "multimedia" as an integral part of a HLLs.

2. Libraries as Programming Abstractions

Let us first consider the most common approach currently encountered: All continuous media processing is based on a set of functions packaged as a library.

In a computing system, each device is accompanied by a device driver and a library to control all the available functions. In DiME, we experimented with a wide assortment of audio and video devices attached to workstations. We found that the libraries are very different with respect to their degree of abstraction. Some can be regarded as an extension to the window system, others simply control collections of bytes to be passed as control block to the respective device.

As an example let us take some functions which support IBM's "Audio Visual Connection" (AVC):

```
acb.channel = AAPI_CHNA
acb.mode    = AAPI_PLAY
...
aud_init(&acb)        /* acb is the audio control block */
...
audrc = fab_open (AudioFullFileName, AAFB_OPEN, AAFB_EXNO,
                  0, &fab, 0, 0, 0, 0);

fork (START IN PARALLEL)
  aud_strt(&acb)
  displayPosition(RelativeStarttime, Duration)
...
acb.masvol = (unsigned char)Volume
audrc      = aud_crtl(&acb)
...
```

Libraries are very convenient at the operating system level, but, there is no consensus (and we assume there will never be one) about which are the most convenient functions for the various devices to be supported. As long as there is no proper operating system support for multimedia and no integration into programming environments, a variegated multitude of functional call interfaces will remain.

A more structured approach interfaces audio and video through "toolkits" (see e.g. [1; 2]). These toolkits are used

- to abstract from the physical layer
- to introduce client-server paradigms, i.e. to hide communication
- as interface for quality of service parameters

It is also known that toolkits facilitate the hiding of implementation aspects such as the process structure. Out of our experience this "encapsulation" of the basic implementation architecture is only possible at the expense of performance and within one type of system (e.g. only one operating system).

3. Multimedia-Specific Abstractions at System Level

Some dedicated abstractions, such as "time capsules" [9], are seen by a multimedia system as extensions to files. These extended files are used for the storage, exchange, and access of continuous media. Individual data items in a time capsule have a "life span" or "duration", in addition to being associated with an indication as to the type of data and the actual data.

This concept is easier to understand and more useful for video than for audio. In the case of full-motion video (25 frames per second), each frame has the duration of 40 msec. In a normal presentation, the read access is performed at this rate. For fast-forward, slow-forward, fast-backwards, etc., the presentation rate is changed. This can be accomplished in one of two ways:

1. The presentation duration of the data items (i.e., the video frames) can be altered. In the case of slow-motion video individual consecutive frames become valid for a longer duration.

2. The duration is not affected, but instead the selection of the segments to be delivered by the time capsule is influenced. At the fast forward mode some data items will be skipped, but the duration for each frame remains. In slow motion mode, frames may be delivered twice.

Herrtwich's work could be extended by taking into account the granularity of the data items (pixel, video frame, sequence) for time capsules. The change of rate should not be performed on a per-sample basis, but instead should be applicable to sequences of samples. With respect to the video presentation hardware, each video frame must still have the same duration; a change in the rate is not applicable. The same is true for audio. For intermediate processing this may not apply. A similar approach combining the data with the rate is presented in [6].

4. Abstract Data Types as Programming Abstractions

As we will use the answering machine example throughout this paper, let us specify this application with Abstract Data Types (ADT) which is one example of a formal specification. This formal definition - also called called abstract type definition - is an interface specification without any description of the internal algorithms.

Using the ADT definition, we encountered some severe restrictions: Communication and synchronization can not be expressed directly, and synchronization of multimedia data is essential for most multimedia applications. These applications also include a certain degree of parallelism which can only be expressed with an ADT by using some tricks, as shown in the following example. (The order of the functions' execution is not defined since this does not make sense in an ADT specification.)

The following example describes the answering machine which we used as a preliminar example to study and compare the various programming interfacing techniques:

```
ADT Answering Machine
operations:
        create          :                       -> CALL_LIST;
        toggle_state    :                       -> {.collect_calls,report_calls}.;
        answer          : CALL x CALL_LIST      -> BOOLEAN x CALL_LIST;
        play            : CALL_LIST             -> BOOLEAN x CALL_LIST;
        volume_control: INTEGER

semantics:
        create  = create_list();
        answer(call,call_list)
                = IF collect_calls
                  THEN((accept_call AND play_infomation_message AND
                        record AND disconnect),
                       enqueue(call,call_list))
                   ELSE (FALSE,call_list);

        play(call_list)
                = IF report_calls
                  THEN((play_voice_mail(head(call_list) AND
                        display_relative-position),
                       dequeue(call_list))
                  ELSE (FALSE,call_list);
```

Although the ADT definition of the answering machine gives a good preliminary idea of the actual application, it excludes the description of some essential features and thus, it is not a real substitute for the required multimedia programming abstractions. This formalism looks like a program, but, it does not define the communication required between the functions. State selection and volume control are possible at any time during operation; these operate in parallel. At this level of abstraction we cannot describe details such as how to play voice mail or how to determine the average volume level. Even the required synchronization between play_voice_mail and display_relative_position cannot be specified.

5. Object-Oriented Approaches as Programming Abstractions

In object-oriented environments – according to Wegner's definition [21; 14] –, multimedia programming is approached by the implementation and expansion of class hierarchies. However, very different types of such class hierarchies can be built:

An **application related class hierarchy** introduces abstractions conceived specificly for one application or a well defined set of applications and it's environment. This is the most commonly used approach and leads to the actual variety of class hierarchies.

Let us now focus on the view of objects in a more physical sense: Let us associate **devices with objects** as done, e.g., in the DiME project [17]. Common methods should be usable in a device-independent fashion. Synchronization methods are applicable to many devices and may be mapped onto, e.g., start/stop/on/off operations. Some devices may comprise various media: e.g., a computer-controlled VCR or Laser Disc Player (LDP) are storage devices combining video and audio. Within multimedia systems an abstract type definition of devices such as cameras and monitors can be provided. However, nothing is said about the actual implementation. It turned out to be a rather difficult task to define a common interface across several similar audio or video, input or output devices as shown in the following example:

```
class media_device
{char* name;
 public:
    void on(), off();
};/* end media_device*/

class media_in_device :
public media_device
 {private:
        DATA data;
  public:
        refDATA get_data();
 };/* end media_in_device */

class media_out_device :
public:
        void put_data(refDATA dat);
 };/* end media_out_device */

      class answering_machine :
public media_device
 {private:
       list  my_list; // class for ADT  list
       media_in_device  recorder;
       media_out_device message_for_caller,
                        message_from_caller;
       refDATA information; // text a caller hears
       void display_position ();
```

```
    public:
         void answer()
                {message_for_caller.on();
                 message_for_caller.put_data(information);
                 message_for_caller.off();
                 recorder.on();
                 my_list.enqueue(recorder.get_data());
                 recorder.off();
                }
         void play()
         {

            message_from_caller.on();
                 message_from_caller.put_data(my_list.head());
                 display_position();
                 message_from_caller.off();
                 my_list.dequeue();
                }
     };/* end answering_machine */
    main(){};
```

The concept of devices as a class hierarchy offers the possibility of parallelism by a simple parallel execution of the methods. Synchronization is not defined in this hierarchy and must be provided from elsewhere. Multiple inheritance was often required in implementing the answering machine.

Initial concepts of DiME were based on a **data flow principle** with sources, sinks and intermediate processing components. Similar approaches are recently discussed in [2](see the comet's node types) [6](sinks, sources and filters) [18] (module with variable number of input and output data channels). This "Lego" model allows to assemble the data flow path by chaining the object or connecting input ports to the respectice output ports of other objects.

The **media class hierarchy** is a special data type structuring method which defines classes correspondent to the different attributes of the individual kinds of media. The following class hierarchy is extracted from the code (see Appendix A) and denotes only a part of the whole hierarchy:

```
media
  audio
    music
      opus
        note
          sample
    speech
      ...
```

```
visual
  video
    image
  animation
   ...
  text
  ...
raw data
...
```

We defined methods 'get' and 'put' for these classes. Related to the discussion about granularity of media, we introduce a second kind of relationship, apart from the 'is-a' hierarchy of classes. The new relationship is the 'is-sequence-of' relationship to model the granularity. This offers the chance to define synchronization in terms of granularity.

A unique property of multimedia objects is their lifetime as discussed in [17] and denoted by [6] as **active objects**: The processing is performed during as long as the connection exists or data is transferred even if no method (apart from a "new" and/or "init") is invoked. Typical methods are "play" and "stop". Gibbs' multimedia programming environment is extended towards the user-oriented interface by a "scripting" language with constructs for parallel, sequential and, e.g., repetitive processing ("a > > b", "a&b", "n*a").

Communication-oriented approaches incorporate **objects in distributed environments** by explicitely defining classes and objects for communication. In [3] information, presentation and transport classes are distinguished. Information encapsulated by the information objects can generate presentation objects to be played or displayed. Information objects can also be converted into transport objects for the purpose of communication, and transformed into presentation objects afterwards (see [3] for the complete state transition graph). We could imagine this model to be extended by a storage class as information is processed/coded differently for communication, presentation as well as storage purposes. And, storage formats are essential as they rely on, e.g., database, CD- and compression specific coding (plain CD-ROM ISO 9660, CD-ROM XA, DVI, CD-I, ... formats).

Another approach we studied is known as **application models**: Applications are derived from a generic application **class hierarchy**. We can either derive models from the basic functions of the applications [20] or understand media as perception, storage, transmission, and presentation media (see MHEG). There are three fundamental combinations of these media: The first we call "live presentation", which means that a live scene is perceived, the data are transmitted and then presented, e.g. a live TV broadcast that you watch on your TV at home. The second is a "recording medium": a combination of perception and storage media. The third combination is a (re-)play medium that presents stored data. From these three classes more specialized classes can be derived, e.g. video observation, video recorder, audio player, and projectors for film or slides.

Related to the inheritance within all above mentioned concepts are the advantages of *polymorphism*, i.e. the same function call or method can be applied to different objects. We may use "play" with e.g. audio and video data, there will be different implementations to perform this command, the data may be a file in the local file

system or some audio/video sequence on a remote file server. Within the object-oriented framework "play" may be defined in various classes, according to the object to perform this operation the respective method is selected. Apart from code reuse, this concept is very useful considering the ease of system's use reducing the complexity of the various underlying systems and approaching uniqueness.

Our examples and the actual implementation were done in C++, but the results are not at all dependent on this specific language. For the future we see a coexistence of various class hierarchies with complex interrelationships. Since this complexity in not easily manageable, let us shift our focus to conventional HLLs.

7. Programming Abstractions within High-Level Languages

In procedural HLLs, multimedia functions may be issued by a set of uniform, i.e. relatively device-independent function calls. This leads to a certain desirable abstraction and assures better programming style and productivity. However, programs must be able to manipulate multimedia data very efficiently. Thus, in a procedural HLL the program will typically directly access multimedia data structures or control the attached processing engines via device-dependent system calls. In Appendix B we included some of the C-Code (it is C with C++ notation) of the answering machine. In the following example, we just show typical programming statements making use of a similar notation to OCCAM 2 [11; 19] (derived from CSP –Communication Sequential Processes– [10] as the language for programming transputers [22]) due to it's simplicity and inherent parallelism.

```
a,b REAL;
ldu.left1, ldu.left2, ldu.left_mixed AUDIO_LDU;
...
WHILE ...
  COBEGIN
    PROCESS_1
      input(micro1, ldu.left1)
    PROCESS_2
      input(micro2, ldu.left2)
  ldu.left_mixed := a * ldu.left1 + b * ldu.left2;
END_WHILE
...
```

For HLLs, an alternative to libraries is to consider **the media as data types**, e.g. a data type for video. In the case of text, a character would be the "atomic element" (bits and bytes aside). A program would manipulate characters: They can be copied, compared with others, deleted, generated, retrieved from a file, stored somewhere, be part of a data structure, etc. Why not permit the same operations on continuous media LDUs (to the extent that it makes sense)? Viewing media as a data type, we can distinguish different approaches based on the granularity of the media to be addressed by the functions (e.g. pixel, whole picture or sequence of pictures; audio sample or audio block). So far we experienced:

- If these sequences are too small, e.g. individual audio samples, real-time processing becomes difficult (DSP algorithms must be generated, changed, enhanced [15]). If the granularity is too coarse, individual items are not longer accessible.

As a practical solution, the programming capabilities should be restricted (i.e. pixel manipulation for DCT or FFT are not the domain of HLLs).

- The meaning of the operators "+", "-", etc. is not only media dependent but also application specific: The addition of two video pictures may be a superposition (with some transparent color) or just and addition of the luminance values. An agreement on the common interpretation is required.

- The compiler generated heap size is restricted (for efficiency purposes chaining of audio/video LDUs is not practical). Careful allocation and manipulation of buffer space can be reached through a system wide homogeneous buffer management and/or applying dedicated optimizing steps for the code generation.

```
file_h1 = open (MICROPHONE_1, ...)
file_h2 = open (MICROPHONE_2, ...)
file.h3 = open (SPEAKER, ...)
...
read (file_h1)
read (file_h2)
mix (file.h3  file_h1, file_h2)
activate (file_h1, file_h2, file.h3
...
```

Instead of extending the notion of data types we could try to follow the approach of looking at **continuous media streams as files**. By opening files we associate the physical files with file names and, the program uses file handles. In our case we will associate a device generating or consuming continuous media with a file name. Read and write functions describe what will happen if data items are available. By a seek function we could position at individual items, but in the following we will typically consider sequences of such items. Often such continuous media may also be derived from a source like a microphone or camera, in such a case a seek function will not be applicable. This is similar to discrete data derived from a keyboard. This approach is very convenient in UNIX environments because there devices are often treated as files at the application programming interface. We could then extend the notion of a device to Leung's "active devices" [12]: All file related functions are applicable and in addition a device could be activated and deactivated. The activation means that the actual data transfer is initiated and it is stopped by issuing the deactivation command. Less operations (than in the case of "media as data type") are applicable as natural extension to the file system.

```
  PROCESS cont_process_a:
    ...
    On_message_do
      set_volume ...
      set_loudness ...
      ...
  ...
[ main ]
pid = create (cont_process_a)
send (pid, set_volume, 3)
send (pid, set_loudness)
...
```

Protocols for continuous media involve time-dependent processing. If we consider the lifetime of a process to be equivalent to the lifetime of the respective connection between the source and the sinks, then another way of incorporating media processing in the HLL is to look at **continuous media as processes**. Creation of the process identifies and reserves the respective physical devices. The interface to continuous media is through IPC. For example, the transfer of continuous media-data can be controlled by issuing signals or messages. The continuous-media process itself determines what actions should be carried out. So far we experienced that there is no one, single approach to be the solution. Note, this report discusses on-going (and not completed) research.

8. Conclusion

Our first, very preliminary, impression was that there exists very little or none work in this area of HLL abstractions for continuous media. And, too many different object-oriented approaches exist. This impression was correct.

Let us come back to the two questions posed within the introduction:

1. *Why are multimedia applications so hardware dependent?*
   The novelty and the diversity of device-specific multimedia functions are the main reasons for the insufficiency of actual programming abstractions for continuous media. Receiving data from a multimedia device and controlling the device (which may be a a camera) is different than simply receiving characters from a keyboard. There is an increase in complexity. Also, different devices (a camera and a microphone or, just two different CD-based storage devices) differ in their functions.

2. *How can this problem be overcome?*
   With proper multimedia specific operating system extension AND programming abstractions, this insufficiency can be tackled. In this paper we discussed and proposed some approaches for continuous media abstractions. However, we do not believe that any one approach is universally applicable? Nevertheless, we doubt that there is one single solution. HLLs, e.g., require enhancements and we are presently experimenting with some of these abstractions.

Summarizing our view of programming multimedia systems: Applications may directly invoke libraries or even call device drivers, a more convinient approach is to use toolkits or even HLLs or object-oriented environments. We see a coexistence of this various approach with a lack of research and development in some of them (e.g. HLLs). We are currently investigating in more detail the integration into HLLs.

Martin Zimmermann supported our practical work with many valuable advices concerning object-orientation and C++, Manny Farber devoted considerable time for commenting and improving the quality of the whole work. Thank you.

References

[1] *David Anderson, Ramesh Govindan, George Homsy.* **Abstractions for Continuous Media in a Network Window System;** Technical Report UCB/CSD 90/596, UC Berkeley, Sep. 1990.

[2] *David Anderson, Pamela Chan;* **Toolkit Support for Multiuser Audio/Video Applications;** 2nd International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Nov. 18-19, 1991.

[3] *Gerold Blakowski;* **Concept of a Language for the Description of Transport and (Re-)presentation Properties of Multimedia Objects;** (in German) Informatik Fachberichte no.293, Spriger-Verlag, 1991, pp.465-474.

[4] *Magdalena Feldhoffer;* **Communication Support for Distributed Applications;** International IFIP Workshop on Open Distributed Processing, Berlin, Oct. 1991.

[5] *E. Fiume, D. Tsichritzis;* **Multimedia Objects;** in: Active Object Environments, D. Tsichritzis (Ed.), University of Geneva, June 1988, 121-128.

[6] *Simon Gibbs, Christian Breiteneder, Laurent Dami, Vicki de May, Dennis Tschichritzis 91;* **A Programming environment for Multimedia Applications;** 2nd International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Nov. 18-19, 1991.

[7] *D. Hehmann, R.G. Herrtwich, R. Steinmetz:* **Creating HeiTS: Objectives of the Heidelberg High-Speed Transport System.** GI-Jahrestagung, Darmstadt, Oct. 1991.

[8] **2nd International Workshop on Network and Operating System Support for Digital Audio and Video;** Heidelberg, Nov. 18-19, 1991, Proceeding to appear as Lecture Note on Computer Science, Springer Verlag, 1991.

[9] *Ralf Guido Herrtwich;* **Time Capsules: An Abstraction for Access to Continuous-Media Data;** IEEE Real-Time Systems Symposium, Orlando, December 5-7, 1990, pp.11-20.

[10] *C.A.R. Hoare;* **Communication Sequential Processes;** Prentice-Hall International, 1985.

[11] *Inmos Limited;* **Occam Programming Manual;** Prentice-Hall International, 1988.

[12] *W. H. Leung, G. W. R. Luderer, M. J. Morgan, P. R. Roberts, S.-C. Tu;* **A Set of Operating System Mechanisms to Support Multi-Media Applications;** Proc. Intern. Seminar on Digital Comm., Zurich, Mar. 1988, pp. 71-76.

[13] *T.D.C. Little, A. Ghafoor;* **Synchronization and Storage Models for Multimedia Objects;** IEEE Journal on Selected Areas in Communication, vol.8, no.3, Apr. 1990, pp. 413-427.

[14] *O.M. Nierstra.* **A Survey of Object Oriented Concepts;** SIGMOD record, vol. 18, no. 1, March 1989.

[15] *L.R. Rabiner, L.W. Schafer;* **Digital Processing of Speech Signals;** Prentice Hall International Inc., 1978.

[16] *Lillian Ruston, Gordaon Blair, Geoff Coulson, Nigel Davies;* **A Tale of Two Architectures;** 2nd International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Nov. 18-19, 1991.

[17] *Ralf Steinmetz, Reinhard Heite, Johannes Rückert, Bernd Schöner;* **Compound Multimedia Objects - Integration into Network and Operating Systems;** International Workshop on Network and Operating System Support for Digital Audio and Video, International Computer Science Institute, Berkeley, Nov. 8-9, 1990

[18] *Daniel Steinmberg, Josh Sirota, David Berry;* **A Multimedia Application Programming Interface Paradigm;** 2nd International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Nov. 18-19, 1991.

[19] *Ralf Steinmetz;* **Occam 2: The Programming Language for Parallel Processing;** (German), Hüthig Verlag, Heidelberg, 1988.

[20] *Ralf Steinmetz, Thomas Meyer;* **Modelling Distributed Multimedia Applications;** to appear at IEEE Int. WS on Advanced Communications and Applications for High-Speed Networks, München, March 1992.

[21] *Peter Wegner;* **Dimensions of Object-Based Language Design;** Proceedings of OOPSLA'87, October 4-8, 1987.

[22] *Colin Whitby-Strevens;* **Transputers - Past, Present, and Future;** IEEE Micro, vol.10, no.6, December 1990, pp.16-19 & 78-82.

Appendix A: Part of the answering machine's C++-Code with Media as Class Hierarchy

```
class   media
        {private:
                // Attribute Set
         public:
                refDATA get();
                void    put(refDATA dat);
                refAttributes information();
        };/* end media */

class   audio :
        public media
        {private:
        //>>>> local Attributes
         public:
        //>>>> public methods
        };

class   music :
        public audio
        {private:
        //>>>> local Attributes
         public:
        //>>>> public methods
        };

class   opus :
        public music
        {private:
          sequence_of note;
          //>>>> local Attributes
         public:
          //>>>> public methods
        };

class   note
        ...
class   sample
        ...
```

```
class   speech :
        public media
        {private:
          //>>>> local Attributes
         public:
          //>>>> public methods
        };

class   discourse :
        public speech
        {private:
          sequence_of sentence;
          //>>>> local Attributes
        //public:  ....
        };

class   sentence :
        public discourse
        {private:
           sequence_of word;
         //public: ....
        };

class   word
        ...
class   phonem
        ...

class   visual :
        public media
        {private:
           //>>>> local Attributes
        //public: ....
        };
        ...
```

Appendix B: Extract of our C-Code (in C++ notation) of the Anwering Machine

```
//Answering_Machine
#include <string.h>
#include <stream.h>
enum state {collect_calls,report_calls};
```

```
//functions:
state select_state ()
       {/* collect_calls or report_calls*/
        char selectionff14";
        int i = 0;
        cout<<"select operation state: collect_calls or report_calls\n";
        cin>> selection;
        while (selectionffi++");
        cout<<"your choise:"<<selection<<"."<<i<<"\n";
       return (selectionff0" == 'r'/*eport_calls "*/);};
int volume_control;
state sele;

//operating:
main()
{
       sele = select_state();
       switch(sele)
       {
       case collect_calls:
              {/*accept_call;
               play_infomation_message;
               record..determines average volume level..;
               enqueue_call;
               disconnect*/
               cout<< "collecting\n";
               break;
              }

       case report_calls:
              {//play_voice_mail..adjusts volume.. and
               //display relative position in parallel
               cout<<"reporting\n";
               break;
              }
       default: cout<< "none\n";
       }
}
```