Applying Object Oriented Programming to Developing Programs
on CTRON Interfaces

Kenji Saito, Masao Tabata
Medium Scale System Department
Hitachi Software Engineering Co., Ltd.

Yukiko Sato
2nd Operating System Department
Software Development Center, Hitachi, Ltd.

Abstract
One of the objectives of the CTRON interface specifications is to promote software-reutilization for real-time applications by standardizing operating system interfaces. Although this approach does improve reusability by improving portability, means for gaining better composability has to be taken into consideration to achieve more advanced reutilization.

Object Oriented Programming (OOP) which supports encapsulation, polymorphism and inheritance is an appropriate programming paradigm for improving composability of software-modules.

This paper describes how we apply OOP to systems based on the CTRON specifications and mainly describes the C++ class libraries we have been designing. This paper also describes how we evaluate the performance, reusability and portability of programs that use the base class libraries.

1. Introduction
Reutilization is a proven way to improve productivity and reliability. It reduces the number of parts we must design from scratch in developing new products. It also makes the dependable parts of previous products available to us in new products.

In real-time applications, little emphasis has been placed on reutilization during system design. Performance has been the all-important quality, and the abstraction required to promote reutilization tends to deteriorate performance.

With advances in hardware, however, this notion of abstraction deteriorating performance is becoming irrelevant. We are getting enough processing power to achieve real-time response, while giving the necessary abstraction to systems. This growth of processing power also magnifies the capability of real-time systems and increases the needs of them. Thus, qualities such as productivity and reliability are becoming more important for real-time systems.
(2) Reutilization and high-level language interface

Even more beneficial software reutilization can be achieved by using high-level programming languages. Many high-level languages support abstractions of procedures and of data. Procedures and data, once abstracted, can be reused in other programs. This makes it possible for programmers to design by themselves reusable parts which can be shared among disparate programs. In most high-level languages, language specifications and their libraries abstract operating system interfaces so that these parts or whole programs can be shared among systems based on different operating system interface standards.

(3) Obstacles in conventional reutilization

The approaches we described above do reduce the numbers and sizes of programs to be designed from scratch and do improve the productivity and reliability of software products. But, these improvements do not seem to satisfy the requirements of today's industries. Two notable obstacles in the conventional reutilization of software are outlined below.

(a) Differences among operating system interfaces

Even though high-level language interfaces can hide differences among operating system interfaces, some application programs need to use operating system interfaces directly. For example, real-time application programs would need to do so to achieve higher performance and more often, would be required to use operating system interfaces not supported by most high-level languages such as task control interface to have minute control of their processing. Because existing operating system interfaces often differ in systems, it is rather difficult to reutilize such programs when they are not developed for the target systems. Standardization could be a solution, but requirements for operating system interfaces vary in application fields and it is impractical to standardize everything, since systems would contain many unnecessary functions and become too costly both in performance and in size.

In the CTRON interface specifications, this problem seems to be solved by letting users select configurations of extended OS's and subsetting the operating system interfaces [5,8] (or preparing profiles). But again, this makes it rather difficult to reutilize programs or parts of programs among systems based on different configurations of extended OS's, or subsets, or both.

(b) Lack of modularity in abstraction-units of high-level languages

In most high-level languages that support structured programming, the main unit of abstraction is procedure. The term "procedure" here includes units of programs such as procedures (Pascal, etc.), functions (Pascal, C, Lisp, etc.), predicates (Prolog, GHC, etc.), and so on.

These units all lack data abstraction. It has to be done externally from procedures. In languages like Pascal and C, data abstraction is supported in a form of data type definitions, and procedures must be designed so they deal with specific types of data. This means that any procedures are dependent on data types which they handle. Even worse, internal designs of data are always visible to any procedures. The facts above show that these abstraction-units are incapable of information hiding and they show poor composability.

2.2 Object oriented programming

(1) Object and encapsulation

To solve problem (b) which we presented in the previous section, OOP introduces a new kind of abstraction. As we described in the previous section, this procedure is the main unit of programs in languages which support structured programming. In OOP languages the unit of programs is called an "object".

An object is an abstraction of data; but it is unlike data type definition which defines data by its structure. Instead, an object defines data by operations which can be executed on it. For this, users of objects do not have to know their internal design. Objects can hide information appropriately and they show high composability. Figure 1 illustrates this concept called encapsulation.

(2) Message passing and polymorphism

Encapsulation requires a kind of communication method called message passing. Because objects are independent from one another, procedure call, which requires certain knowledge of receivers' internal design (name of the routine, etc.), is not applicable to communication between objects. Instead, objects request receivers in the form of messages to execute operations. Using message passing, it is possible for objects to receive the same messages, but act differently from one another. This idea is called polymorphism. Figure 2 shows an example of polymorphism.

(3) Class and inheritance

Because an object is an abstraction of data, a concept called class corresponds to conventional data type. One of the differences between classes and data types is that classes have definitions of operations besides definitions of data structures. But their major difference is that classes can be categorized to make a tree structure.* Classes placed below in the tree structure (derived classes) inherit operations and the internal design from classes placed above in the tree structure (base classes). Figure 3 is an example of this concept of inheritance.

* If a system supports multiple inheritance, in which objects can have more than one base class, the classes will form a network structure instead of a tree structure.
Fig. 1 Encapsulation

Fig. 2 An example of polymorphism
(4) Benefits
Encapsulation, polymorphism and inheritance benefit the productivity, reliability, extendibility, and maintainability of software products in these ways:
(a) Productivity
Appropriate information hiding by encapsulation allows programmers to compose existing objects and build new systems easily. Inheritance also enables programmers to define new objects slightly different from previous ones to adopt them into new systems. Polymorphism reduces the number of interfaces among objects and standardizes interfaces so that objects can become compatible and highly composable. It also frees programmers from memorizing a flood of procedure names and from remembering a mass of details when designing systems.
(b) Reliability
The advanced reusability described above allows programmers to reuse dependable objects without modifying them.
(c) Extendibility
Again, inheritance enables programmers to add new objects without modifying old ones.
(d) Maintainability
As described above, inheritance makes systems easy to extend. Encapsulation and polymorphism make source codes easy to read.
Also, because operating system interfaces can be described as abstractions of system resources (thus, described as abstractions of data), OOP languages are suited for abstracting operating system interfaces. Abstracting operating system interfaces with OOP languages makes it possible for programmers to take system resources as data types. It seems more natural than to take them as a set of procedures. It also enables programmers to define derived classes of system resources and virtually extend their systems' capabilities. This abstraction also solves the problem with differences among operating system interfaces by hiding what is particular to some operating systems and by extending/restricting their capabilities.
These benefits above would also apply to so called encapsulation languages[4] such as Modula-2 and Ada. However, these languages do not support inheritance and with these languages it is difficult to extend existing modules without modifying them or to express logical relationship (such as is-a relation) among modules. It is
also difficult, when abstracting operating system interfaces with these languages, to enable programmers to extend their systems' capabilities by adding new operations to system resources.

(5) Drawbacks
For all the advantages we described above, OOP still has the following disadvantages when applied to developing programs on CTRON interfaces:

(a) Inefficiency
High reusability is derived from the high degree of abstractness in using OOP languages and abstraction may still be an obstacle in achieving real-time response with currently available processing power. More specifically, the problems which may arise in using OOP languages are their inefficiency in message passing and in dynamic polymorphism.

(b) Obstacles in portability
Although programs written in OOP languages potentially promise high portability, it is not simply applicable to programs on CTRON interfaces, at least in the present situation, because the only valid programming languages now available for software circulation in the CTRON project are C, Ada, and CHILL[11].

For C++, however, these disadvantages above may become irrelevant.

2.3 Applying C++ language
(1) Features of C++
C++, one of the extensions of C language, supports some of the OOP features in the following ways:

(a) Encapsulation
C++ allows programmers to define classes. Classes in C++ are realized as structures of C language.

(b) Polymorphism
In C++, message passing is realized as function calls, and function names can be overloaded. Polymorphism in C++ is solved statically in compilation when normal member functions are used. Dynamic polymorphism, which is solved at the time of execution, is realized as virtual functions, a special mechanism equivalent to having pointers to functions as members of structures.

(c) Inheritance
C++ allows programmers to define classes as derived classes of one class or more than one class.

(2) Benefits
We have introduced C++ to our project because of the following advantages:

(a) Efficiency
As we described above, message passing in C++ is realized as function calls; this does not cost more than using conventional languages such as C. And in most cases polymorphism is solved statically. Dynamic polymorphism also is relatively efficient because using pointers to functions reduces the cost of searching required routines. Furthermore, C++ supports inline functions even when mechanisms such as function calls seemed too costly (the details are in 5.1 Using inline functions).

(b) Portability
Because most existing C++ compilers are able to output source codes in C, there should be few problems in portability of programs written in C++ in the CTRON project.

(c) Reusability of previous environment
Because C++ is a super set of the C language, we could reutilize the environment we prepared for developing Hitachi's version of the CTRON kernel using C, such as include files, library functions and development tools.

For the same reason, we did not consider introducing Ada to our project though it supports encapsulation and polymorphism (but not inheritance as we explained before) and is an available language in the CTRON project.

(3) Drawbacks
Applying C++ still may have these disadvantages:

(a) Inefficiency
Virtual functions seem to cost more than just pointers to functions. Output source codes may be redundant as long as compilers generate them.

(b) Lack of libraries
To promote the reutilization of objects, OOP environments must have a collection of classes called a class library as the source of base classes. Existing class libraries in C++ are unsuited for real-time applications, or at least not designed assuming use of real-time operating systems.

This means that we have to develop a class library that is applicable to real-time applications by ourselves. We have to develop C++ run-time library functions using CTRON interfaces, too, to make our programs actually run on operating systems based on the interface specifications.

3. The base class library
Class libraries are collections of base classes from which we define derived classes; thus, we call them the base class libraries in our project.

We have set the following policies in designing them:

(1) Prepare necessary classes for various applications.
(2) Define classes for symbolic processing and text processing to standardize command interpreters and editor functions for application programs.
(3) Abstract every CTRON interface except ones that can be abstracted by ANSI C standard library functions[3] to achieve portability among systems based on different
interface subsets or on other interface standards. It should also be possible to define derived classes of system resources to extend systems' capabilities.

Because the necessary sorts of classes vary in application fields, we have been developing a number of libraries, each supporting a specific application field. Table 1 lists the libraries.

Table 1 List of the base class libraries

<table>
<thead>
<tr>
<th>Name of library</th>
<th>Applied field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base class library 1</td>
<td>Common among every application field</td>
</tr>
<tr>
<td>Base class library 2</td>
<td>Symbolic processing</td>
</tr>
<tr>
<td>Base class library 3</td>
<td>Text processing</td>
</tr>
<tr>
<td>Base class library 4</td>
<td>Real-time applications</td>
</tr>
<tr>
<td>Base class library 5</td>
<td>Network communications</td>
</tr>
</tbody>
</table>

Among these, base class library 1 and base class library 4 are almost complete. This chapter outlines these two base class libraries.

3.1 Base class library 1

(1) Objectives

Base class library 1 furnishes basic data structures that are used commonly among every application field. Its objectives include the following:

(a) To free programmers from troublesome operations and to let them deal with these operations through higher-level interfaces.

(b) To improve the reliability of programs by encouraging programmers to reuse dependable code.

(2) Contents

This library contains classes that abstract basic data structures such as lists, queues, stacks, arrays, binary trees, hash tables, and so on. Figure 4 illustrates the relations among the classes.

(3) Comments

Many of the classes in this library require programmers to define their derived classes before using them. For example, to use class `HashTable`, one must define its derived class specifying three entities of virtual functions including a hash function since class `HashTable` does not have entities of these functions.

These classes in OOP are called abstract classes. Abstract classes achieve high composability because they are not designed to be applied to specific applications.

3.2 Base class library 4

(1) Objectives

Base class library 4 furnishes the system resources available in typical real-time kernels. Its objectives include the following:

(a) To improve the productivity of real-time systems by defining system resources as classes and by letting programmers define derived classes of such classes to extend their class libraries and virtually extend their kernels' capabilities.

(b) To improve portability of real-time systems by abstracting all system calls provided by the CTRON kernel interface[10] (without deteriorating the performance) and by letting programmers redefine system resource classes when porting programs using the library.

(c) To improve the reliability of real-time systems by encouraging programmers to access system resources as data types.

(d) To extend/restrict functions of the CTRON kernel by adding new operations as high-level language interfaces.

(2) Contents

The library contains classes that abstract system resources such as tasks, event flags, semaphores, message boxes, memory pools, memory blocks, and so forth. These classes are implemented by using the C language binding of the CTRON kernel interface; and, system calls to control these resources are represented by member functions (mostly inline functions) of the classes. However, we did not design every member function to completely agree with the CTRON kernel system calls nor every class to represent resources in the CTRON kernel. Some of the classes and their member functions are designed in accordance with the ITRON2 kernel specifications[2,7]. The reasons for this are:

(a) Abstracting interfaces does not necessarily mean following them. To improve portability, it is better to hide what is particular to the CTRON specifications. Yet it is desirable to follow the features of TRON from an educational point of view.

(b) ITRON2 provides simpler system calls. ITRON2 avoids introducing flags in parameters by separating system calls where necessary. It agrees with a principle of OOP to have simple yet direct operations.

(c) ITRON2's version of rendezvous is preferable to that of CTRON's. ITRON2 provides users with a unified way of handling objects by preparing ports as the system objects for rendezvous mechanism, while CTRON does not provide ports and makes the mechanism attached to tasks. ITRON2's version of rendezvous is independent from tasks, and it is preferable when building efficient systems based on the server/client computing model. It also performs higher-level functions such as forwarding ports.

The library also contains higher-level resources such as servers (derived from tasks), event queues and events. Interrupt and exception are not currently included in the library, but we believe that they can be described as special cases of events.

Figure 5 illustrates the relations among the classes.
Fig. 4 Relations among classes in base class library 1
A mailbox is equivalent to a message box in locate mode.
A message buffer is equivalent to a message box in move mode.

A local memory pool/block represents a memory pool/block with task attributes.
A memory pool/block here means one with system attributes.

Fig. 5 Relations among classes in base class library 4
Some of the features of this library are:

(a) Class SystemResource

Class SystemResource has a resource id as its protected data member. A resource id is a system tag type of data equivalent to identifiers acquired by CRE_XXX system calls. Because the CTRON interface specifications provide programmers with a uniform way of accessing system resources by using identifiers, we abstracted system resources accordingly.

(b) Constructors/destructors

The CTRON kernel interface has two ways of acquiring identifiers. One is to create resources, and the other is to obtain identifiers by using GET_XID system calls. Accordingly, the library has two ways of constructing system resource objects. One is to specify parameters necessary to create resources when declaring/allocating objects, and the other is to specify their names. So most classes belonging to this library have more than one constructor (Fig. 6). Constructors in C++ have the same names as classes and are called implicitly when declaring/allocating objects.

The destructors of the system resource classes judge if their identifiers were acquired by creating resources or by calling GET_XID system calls before destructing the objects. If the identifiers were acquired by creating the resources, the destructors will call DEL_XXX system calls to delete them. Destructors in C++ have names such as -ClassName and are called implicitly when declaring/allocating objects.

The source code in this paper do not represent the actual code of the base class libraries. They are simplified to help understanding.

(b) Constructors/destructors

The CTRON kernel interface has two ways of acquiring identifiers. One is to create resources, and the other is to obtain identifiers by using GET_XID system calls. Accordingly, the library has two ways of constructing system resource objects. One is to specify parameters necessary to create resources when declaring/allocating objects, and the other is to specify their names. So most classes belonging to this library have more than one constructor (Fig. 6). Constructors in C++ have the same names as classes and are called implicitly when declaring/allocating objects.

More than one object can represent a single system resource, one object in one task in usual cases. One of the advantages is that the objects can have information specific to each task. This enables us to virtually extend capabilities of system resources. A corresponding disadvantage is that extensions to the objects, when they deal with more than one task, may affect the system fatally. In this library, we prepared special classes whose objects only one task can have. Extensions made by adding operations to objects are restricted to cases where they do not affect other tasks.

(c) Default parameters

Some parameters of the system calls in the CTRON interface specifications have default values. In conventional C language binding, programmers must designate NULLP, or certain constants beginning with D_ as parameters when they want to omit them. This does not seem like such a good idea since what programmers want is to omit parameters, and not to specify them. In this library, we have applied default parameters in C++ to member functions that represent the system calls. The consequence is that the order of parameters does not always agree with the language binding; output parameters usually come first just like other TRON specifications such as ITRO[7]. An example of using default parameters is shown in Fig. 6, in the first constructor of class Semaphore.

(c) Default parameters

Some parameters of the system calls in the CTRON interface specifications have default values. In conventional C language binding, programmers must designate NULLI, NULLP, or certain constants beginning with D_ as parameters when they want to omit them. This does not seem like such a good idea since what programmers want is to omit parameters, and not to specify them. In this library, we have applied default parameters in C++ to member functions that represent the system calls. The consequence is that the order of parameters does not always agree with the language binding; output parameters usually come first just like other TRON specifications such as ITRO[7]. An example of using default parameters is shown in Fig. 6, in the first constructor of class Semaphore.

```cpp
class Semaphore : public SystemResource
{
public:
    Semaphore(e_reason_t &int_t name,int_t count,int_t attr = NULLI); // Represents CRESEM
    Semaphore(int_t name); // Represents GET_SID
    ~Semaphore(); // Represents DEL_SEM
};

CreateSemaphoreRoutine() // An example of creating semaphores by CRE_SEM
{
    e_reason_t return_code;
    Semaphore aSemaphore(return_code,SEMAPHORE_1,1); // Creates a semaphore
}

ReferSemaphoreRoutine() // An example of referring semaphores by GET_SID
{
    Semaphore aSemaphore(SEMAPHORE_1); // Refers a semaphore
    aSemaphore.wait(); // Waits for a signal
}
```

* The source code in this paper do not represent the actual code of the base class libraries. They are simplified to help understanding.
(d) Class MemoryPool, LocalMemoryPool, MemoryBlock, and LocalMemoryBlock

These classes are used when programmers want minute control of memory allocation/deallocation or when they have to allocate message blocks specified in the CTRON specifications. Otherwise, we recommend that programmers use C++ default operator `new` to allocate memory, and operator `delete` to free memory. Implementation of these operators is described in 4.

(e) Hiding the differences among interface subsets

We have lessened by abstraction the differences among interface subsets. The kernel on which we developed this library is based on the $\mu$C subset, and does not contain semaphore functions. However, we can implement class Semaphore in our library. The secret to this is that the idea of semaphore can be realized by using other task synchronization and communication mediums such as message boxes (though the realized semaphores may have some deficiencies) as shown in Fig. 7.

Because some of the system calls supported by the C subset and not by the $\mu$C subset can be realized by using only system calls supported by the $\mu$C subset, we could enhance our kernel with some of the C subset features such as semaphores, serially reusable resources, and CREATE_AND_START_TASK (which actually is realized as one of the constructors of class Task) as the easiest example. It is also possible to implement functions of larger subsets such as rendezvous and selective message receive (though we did not implement CTRON's version of rendezvous as we described before).

This makes porting programs among different interface subsets more effective in the following ways:

(i) Programs on smaller subsets, if they use functions of larger subsets implemented by the library, can benefit from using the same functions provided by the kernels without being modified when they are ported to larger subsets. In some cases, this may improve their performance.

(ii) Programs on larger subsets will become more adoptable to systems on smaller subsets.

```cpp
class Semaphore : public SystemResource
{
public:
    Semaphore(e_reason_t &return_code,int_t name,int_t count,int_t attr = NULL) // V operation
    {
        error_t error_information;
        return_code = CRE_MBX(name,NULLID_MAX_MESSAGE_QUEUE_SIZE,
                              ON,D_PRIORITY,(flag_t)attr,&resource_id,&error_information);
        // Creates a message box instead of a semaphore
    }
    Semaphore(int_t name)
    {
        error_t error_information;
        GET_BID(name,&resource_id,&error_information);
        // Acquires the identifier of created message box instead of a semaphore
    }
    e_reason_t signal(int_t count = 1) // P operation
    {
        SND_MSG(...);
        // sends a null message to created message box instead of signaling a semaphore
    }
    e_reason_t wait(int_t count = 1,long_t time_limit = NOLIMIT) // P operation
    {
        REL_MSG(...);
        // receives a null message from created message box instead of waiting for a signal
    }
}; // Count must be 1 in this implementation.
```

Fig. 7 An example of realizing semaphore on the $\mu$C subset

112
(f) Hiding the differences among operating system interface standards

This abstraction of system resources may make it easier to port programs to systems based on different real-time operating system standards.

One example is portability among different TRON specifications. As we described before, some parts of our library are designed in accordance with the ITRON2 specifications, and it should not be difficult to rewrite the library to actually work on the ITRON2 kernel. One of the major differences between the ITRON2 specifications and the CTRON specifications is that the objects are identified by their ID numbers in the ITRON2 specifications. However, ID numbers can be substituted by resource names in the CTRON specifications. To realize this and to encourage programmers to specify names of resources, resource names are made mandatory in our library. Of course they can be omitted by specifying NULL for the names, as the conventional C language binding requires.

As another example, one of the existing operating systems based on the CTRON specifications has shown that it is possible to implement the subset of the CTRON kernel on top of operating systems with sufficiently rich sets of capabilities. Therefore, it must be possible to develop what is equivalent to our library on such real-time operating systems. This approach would have better performance than actually implementing the CTRON kernel for reasons we will present in 5.1 Using inline functions.

(g) Extension/restriction of operating system capabilities

We also extended the functions of some of the system resources. For instance, class TaskAttachedMailBox, which represents concepts similar to task attached mailboxes in the CTRON specifications, lets programmers push back received messages or leave received messages in its message queue. These extensions are achieved by having an instance of class FifoQueue as a private data of class TaskAttachedMailBox.

These extensions, however, work only when a single task is receiving messages from a message box; thus, the usage must be restricted. The reason we designed class TaskAttachedMailBox or other classes named TaskAttachedXXX as derived classes of system resources is to realize this restriction. The objects of these classes cannot be created by tasks other than those which created the resources; thus the resources cannot be referred by more than one task as objects that have the extended operations. This shows that inheritance is a powerful mechanism in extending systems' capability since it realizes extensions and defines their applicable objects at the same time.

(h) Abstraction of tasks

Class Task is an abstraction of tasks in the CTRON kernel interface and prepares every system call for task management which requires a task identifier.

Class SelfTask, which represents the calling task of system calls, is a special case of task and is defined as a derived class of class Task. It only defines static member functions, which in C++ means functions executable without any instances, as representations of system calls which do not require task identifiers. This corresponds to the fact that system calls which only act on the calling task, such as SLP_TSK, do not require any task identifiers. Thus, member functions that represent these system calls must not require any objects. It is possible, however, as the kernel interface permits, to create an instance of class SelfTask and call member functions of class Task to execute operations designed to act on other tasks on the calling task by the C++'s inheritance mechanism (Fig. 8).

Class Server also is a special case of task and is defined as a derived class of class Task. It represents tasks driven by specific events. In application fields to which the CTRON interfaces are originally designed to apply, most tasks will belong to this category. This class must therefore be very carefully designed to achieve high composability and reliability.

```
ExampleOne()
{
    SelfTask::sleep();  // Sleeps (call without target object)
}

ExampleTwo()
{
    SelfTask me;        // Declaration of SelfTask object
    me.wakeup();        // Wakes itself up
}
```

Fig. 8 Using class SelfTask
(i) Abstraction of events

To realize class Server as a highly composable part of programs, we had to design an abstraction of events. A server task may receive events from other tasks running on the same processor, either through a message box or a port, or from other tasks running on other processors through networks. Preparing different kinds of servers for each way of receiving events will deteriorate the composability. Thus, we aimed to invent a uniform way of dealing with events.

We considered the following ways of achieving uniformity:

(i) To develop a distributed real-time kernel is one way of doing it. In such kernels, the communication method inside a processor and the one among processors are usually unified to achieve location transparency.

While we are happy to have CTRON's version of distributed kernel, there are some obstacles in realizing it. One of them is that communication control is designed as one of the extended OS entities in the CTRON specifications and building a distributed kernel will cause considerable modification in the CTRON interface system. Thus, the idea might not be fruitful.

(ii) Another way is to build an outer kernel on top of the CTRON kernel to provide higher layers with functions of distributed kernels. This approach is found in the CTRON2 specifications[6], and has potential for the CTRON interface system.

(iii) What we adopted in our project, however, is to abstract events by using an OOP language. We designed two classes which represent concepts related to events. Class EventQueue is an abstract class which can be used as a representation of message boxes, ports, network communication paths, etc. The most important abstraction -- the abstraction of mediums -- is achieved in this class. Class Event is derived from class Array, and has entries for event handlers as its elements. It also has a virtual function named classify that classifies received events and returns event numbers. Class Event calls one of the event handlers according to the event number acquired from the classify function. We designed the class to handle events in two ways: to wait for events one-by-one or to perform an event loop.

We believe that it is possible to consider interrupt and exception as special cases of events; but, this needs further study.

(3) Comments

This library proposes solutions to some of the problems that the current C language binding of the CTRON kernel interface has as follows:

(a) Encapsulating identifiers

The current language binding does not have any means for protecting identifiers of system resources though they must be treated as constants. This library abstracts the method of indicating resources by making them appear as data types and hides the identifiers from outside the objects. This also improves portability of programs using this library to systems based on other operating system interface standards since using identifiers to indicate resources may be particular to CTRON (ITRON2 shows an example of other ideas).

(b) Encapsulating error information

C programmers are used to discriminate errors by return values of functions. Error information as the last parameter of system calls in the current language binding, which is a direct transcription of the language-independent specifications, has four integers: error class, error reason, and two system-dependent values. They tend to be left unused except error reason which can be acquired as return values of the system calls; accessing the system-dependent values would be an obstacle in porting, and callers of the system calls would naturally know error classes since they indicate the managers that provide the system calls. In this library we removed error information from parameters of the member functions that represent system calls; but means for acquiring the information is prepared as another interface to be used in system-dependent modules such as operation administration and maintenance control.

(c) Reducing the number of parameters

The current language binding has many parameters in one function, which would not be approved by most C programmers. In this library we removed the identifiers and error information from the parameters and separated member functions that represent system calls with flags which switch functions in their parameters. We also introduced default parameters.

Above solutions do not necessarily indicate that the current C language binding should be changed. Some of the problems are impossible to be solved by using just C and we suggest that the direct transcriptions of the language-independent specifications are kept to be used where minute control is necessary.

This library lets programmers themselves define derived classes of system resources. This makes it possible for programmers to extend the function of the CTRON kernel, just as we did by defining classes such as class Server.

Also, those derived classes that programmers define will become reusable parts of real-time systems. This improves the productivity and the reliability of real-time systems.
4. Implementing the C++ run-time library functions

Because C++ has richer functions than C, it requires a larger run-time library. However, since most of the necessary functions are provided by the C compilers, we had to implement only functions which are peculiar to C++.

This chapter explains the implementation of the C++ run-time library functions using the CTRON interfaces.

4.1 Implementing the operator \texttt{new}

Probably the most important run-time library function is default operator \texttt{new}. Because operator \texttt{new} allocates memory, the easiest and most efficient way of implementing the operator is to use the ALC\_MEM system call of the kernel; and that is what we did.

Implementing operator \texttt{new} by using ALC\_MEM, however, has the following problems:

1. Operator \texttt{new} must know the memory pool id from which it allocates memory blocks.
2. Because operator \texttt{delete} only takes the location of the memory as its parameter, operator \texttt{new} must set a memory block id somewhere detectable by operator \texttt{delete} to enable it to call FRE\_MEM.

The former problem can be solved by creating a memory pool when starting programs, and maintaining the memory pool id as a global variable. This apparently disables programmers from minute control of memory allocation/deallocation; but, class \texttt{MemoryPool}, \texttt{LocalMemoryPool}, \texttt{MemoryBlock}, and \texttt{LocalMemoryBlock} are available whenever minute control is necessary.

The latter problem can be solved by a method which is generally adopted in systems which support dynamic memory allocation. The operator \texttt{new} allocates memory by using ALC\_MEM the size of one system tag larger than specified. Then the operator places the memory block id at the top of the allocated block, and returns the location the size of one system tag higher than the top (or lower than the top depending on the hardware architectures) as illustrated in Figure 9.

4.2 Implementing the operator \texttt{delete}

The operator \texttt{delete} is realized by using FRE\_MEM. The operator takes the location of the memory as its parameter, and calls FRE\_MEM specifying the location the size of one system tag lower (or higher depending on the hardware architectures) than the parameter and its contents as the memory block id to free.

4.3 Implementing other run-time library functions

Other run-time library functions, such as functions to realize virtual functions and program exit routine, are also implemented.

5. Improving performance

Programs written in C++ may be more efficient than ones written in other OOP languages, but their performance is still inferior to ones written in C. Yet there are several well-known ways to improve the performance of programs written in C++, such as:

1. Use inline expansion for small member functions.
2. Use virtual functions as little as possible.

This chapter explains both ways of improving the performance of programs written in C++.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig9.png}
\caption{Memory allocated by the operator \texttt{new}}
\end{figure}
5.1 Using inline functions

Inline functions are similar to macro definitions supported by a preprocessor command of C, but are superior to them in the following ways:

1. Inline functions have no side effects.
2. Inline functions have function prototype declaration.
3. Inline functions can have local variables.
4. Inline functions can be member functions.

Inline declaration of functions, however, is only a hint to compilers in C++. This means that in some cases some compilers do not generate inline codes. To keep this from happening, we reduced the nesting level of inline functions by eliminating some of the member functions to refer to private data and we made the data public or protected in our libraries. This, of course, goes against the principles of OOP. Nevertheless, by restricting this to cases when only public member functions both set and refer to this data, we believe we can confine the violation to such a level that it affects neither the reliability nor the maintainability of the libraries.

By using inline functions comprehensively, we can reduce the overhead of abstractions considerably.

5.2 Avoiding dynamic binding

Although using virtual functions improves the composability of objects, the mechanism is rather costly. We have set the following policies in designing our libraries to avoid using virtual functions as much as possible:

1. Use normal member functions when compatible objects need not coexist in one system.
2. If the functions do not need "this", which in C++ represents an implicitly declared target object, make them pointers to functions.
3. If the above conditions do not apply, use virtual functions.

Using pointers to functions instead of virtual functions adds extra coding in constructors, but does not affect composability.

6. Evaluation

This chapter explains how we evaluate our libraries with the current specifications. Results presented here are about base class library 1, representing common data structures.

6.1 Evaluating performance

We prepared two kinds of test programs to evaluate performance of base class library 1, using classes FifoQueue and BinaryTree. We wrote programs which perform equivalent functions in C, and compared their processing times. For enqueue/dequeue we prepared a high-speed version in C that uses a macro with a side effect. For binary sorting we prepared in C++ a version that uses virtual functions violating the design policies outlined in 5.2 Avoiding dynamic binding.

Table 2 shows the ratios of processing times among programs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Not optimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enqueue/dequeue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++ : C</td>
<td>1.06 : 1</td>
<td>1.02 : 1</td>
</tr>
<tr>
<td>C++ : (high speed)</td>
<td>1.27 : 1</td>
<td>1.35 : 1</td>
</tr>
<tr>
<td>C : (high speed)</td>
<td>1.20 : 1</td>
<td>1.33 : 1</td>
</tr>
<tr>
<td>Binary sorting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++ : C</td>
<td>1.08 : 1</td>
<td>1.07 : 1</td>
</tr>
<tr>
<td>C++(with V.F.) : C++</td>
<td>1.19 : 1</td>
<td>1.12 : 1</td>
</tr>
<tr>
<td>C++(with V.F.) : C</td>
<td>1.28 : 1</td>
<td>1.20 : 1</td>
</tr>
</tbody>
</table>

V.F.: virtual function

The results show that without virtual functions, performance will deteriorate less than 10% (except in cases of the side effect). In C++, the result that the program without using virtual functions shows higher reusability and that the number of non-reusable lines seems to be the same whether coded in C++ or C suggests that the measure used here is probably inappropriate. Other aspects such as ease of use or effects of inheritance such as the number of shared lines must be considered.
6.3 Evaluating portability

Because we have not ported programs using our libraries to any other systems yet, we cannot evaluate the portability quantitatively; however, we can evaluate it by showing how parts dependent on hardware architectures, operating systems, and compilers are encapsulated, as follows:

(1) Hardware architecture-dependent part
Our current libraries do not include hardware-dependent parts. It is easy to encapsulate such parts if any by defining compatible objects. However, some C++ compilers may output source code in C that include code dependent on specific hardware architectures.

(2) Operating system-dependent part
Classes belonging to base class libraries 4 and 5, and libraries we are going to design for abstraction of other operating system interfaces encapsulate operating system dependent parts. Differences among error codes are hidden by using enumerators.

(3) Compiler-dependent part
Our C++ compiler, which works on Hitachi's workstations, outputs source code in C based on the K&R specifications, except that identifiers generated are often longer than eight letters, and sometimes even longer than 31 letters. This problem is solved by using a tool we have developed for the operating system validation test of our kernel, which translates identifiers longer than specified length into shorter ones.

We have already compiled our libraries with a number of C++ compilers through implementing them. We have found that some inline functions are not expanded inline by some compilers due to a specific type of coding. This is solved in the current version of our libraries as shown in 5.1 Using inline functions.

7. Future work
The following must be done to improve our method:
(1) Find appropriate measures for gauging reusability.
(2) Quantitatively analyze portability, especially among different interface subsets and interface standards.
(3) Abstract interrupt and exception.
(4) Abstract the extended OS interfaces.
(5) Implement real-time application programs using our libraries.

8. Summary
Object Oriented Programming (OOP) is introduced to solve two notable problems caused by the conventional reutilization of software products: differences among operating system interfaces and lack of modularity in abstraction-units. The former is solved by encapsulating dependent parts on operating systems and the latter by encapsulation, polymorphism and inheritance.

Abstraction of real-time kernel is achieved in accordance with the CTRON and the ITRON2 kernel interfaces, as a class library written in C++. Encapsulating dependent parts on operating system interfaces improved portability among different interface subsets and interface standards. Inheritance showed possibilities of virtually extending systems' capabilities.

The deterioration of performance due to a high degree of abstractness is minimized by using inline functions and avoiding virtual functions. The deterioration rate is well under 10%.

References