A Set of C++ Classes for Co-routine Style Programming

Bjarne Stroustrup
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Each activity, here called a task, has its own locus of control, a program to execute, and its own private data. Tasks can communicate by explicit sharing of data, by messages, or by data pipes.

This memorandum describes C++\(^7,8\) classes for a range of styles of multi-programming techniques in a single language, single address-space environment. Class \texttt{task} is a base class for representation of an activity in a multi-programmed system. A task can be suspended and resumed without interfering with its internal state. Class \texttt{qhead} and class \texttt{qtail} enable a wide range of message passing and data buffering schemes to be implemented simply.

The task system can be used for writing event driven simulations. Tasks execute in a simulated time frame presented by the variable \texttt{clock}, and objects of class \texttt{timer} provide a convenient and efficient facility for using the clock.
1 Introduction

Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Such activities, here called tasks, must be able to execute in parallel with each other and communicate through means convenient to the chosen style of task usage.

Facilities for multi-thread computation can be provided in the semantics of a language, as is done in Concurrent Pascal\textsuperscript{1} and Mesa\textsuperscript{2}, or a language without such facilities can be augmented using special run-time support systems and library functions, as has been done for BCPL\textsuperscript{6} and C\textsuperscript{3}. The use of C classes to implement tasks represents an intermediate approach pioneered by Simula67\textsuperscript{2}.

The tools presented here provide the basic facilities for several styles of multi-thread programming in a single language, single address-space environment. The underlying facility is a simple and efficient tasking system with non-preemptive scheduling. That is, a task will only be suspended on its own request, so no “system policy” can be enforced without the cooperation of all tasks. In contrast to pure co-routine systems, however, the task system provides a framework for processor sharing and communication between tasks.

The task system is intended for applications, like event driven simulations, where tasks are used to express a quasi-parallel structure for a single program. For this class of applications a concept of simulated time is implemented. A unit of simulated time can represent any amount of real time, and it is possible to compute without consuming simulated time. A few simple random number generating classes and a class histogram for data gathering are also provided. The task system is not intended for handling real parallelism of some underlying real-time system. Consequently, no facilities are provided to map interrupts and other real-time events into the concepts provided by the task system.

2 Tasks

The declaration of class task looks like this (the ellipsis ... is used, un-grammatically, to indicate where details not considered relevant to the discussion has been removed):

\begin{quote}
\texttt{The class object used in the declaration of class task is a simple base class used by all classes in the task system. It contains some of the pointers used by the task system's internal "house-keeping", and also a value indicating the type of the object. Class object is presented in appendix A.}

\texttt{The ellipsis ... is used (un grammatically) to indicate details not considered relevant to the discussion.}
\end{quote}
class task : public object {
    ...
    public:
    task(char* = 0, int = 0, int = 0);
    
    task* t_next;
    char* t_name;
    int rdsstate();
    long rdttime();
    void result(int);
    int result(task*);
    void cancel(int);
    void sleep();
    void wakeup();
    void delay(int);
    int preempt();
    void wait(object*);
    int waitvec(object*);
    int waitlist(...);
    void print(int);
};

A task is a locus of control, a virtual processor. It can only be used as a base class. A task executes the program supplied as a derived class’s constructor. The most basic feature of an object of class task is that it can be suspended and later resumed so that several tasks can run in quasi-parallel. Most class task functions are conditional or unconditional requests for suspension.

A task can be in one of three states:

RUNNING:
The task is executing instructions or it will be scheduled to do so without further intervention from other tasks.

IDLE:
The task is not RUNNING, but it can be transferred to the RUNNING state by some suitable action.

TERMINATED:
The task has completed its work. It cannot be resumed, but its result can be retrieved.

The class task function rdsstate() returns the state.

A simple example of the use of tasks is where one task creates another to run in parallel with itself. Later the creator can obtain the result produced by the “secondary” task. For example, a task which counts the number of spaces in a string could be declared. First a class spaces must be declared.

struct spaces : public task {
    spaces(char*);
};

In the case of class spaces the declaration is trivial. It states that spaces is derived from class task so that each object of class spaces becomes an independently scheduled entity. The program for the task is provided by the constructor spaces.spaces(). This use of this constructor resembles the use of main() in a C program.
spaces(spaces(char* s))
{
    int i = 0;
    while (*s) if (*s++ == ' ') i++;
    return i;
}

This function counts the spaces in its argument string and return the result using the class task function result(). A task of class spaces can now be created and used like this:

spaces SS("a line with four spaces");
count = SS.result();

When a new task is created, like SS here, its constructor is called with the argument list provided, and the two tasks now run in parallel. The task function result() returns the value returned from spaces.spaces() by the call of the task function result(), that is, in this example the value 4. If a task calls result() for another task which has not yet completed it will be suspended waiting for that task to become TERMINATED. When that happens it will be resumed. A task waiting for another to complete is IDLE. If a task calls result() for itself it will cause a run time error†.

A task cannot return a value using the usual function return mechanism; it must use the class task function result(). This function puts the task into the TERMINATED state from which it can not be resumed.

3 Queues

A queue is a type of storage that is organized so that objects are retrieved from it in the order in which they were inserted into it. A queue has a head from which data is retrieved and a tail to which data is added. With a little elaboration this basic type of data structure makes an excellent inter-task communication facility.

There is a function put() which adds an object to the tail of a queue and a function get() which retrieves an object from the head of a queue. There is no "class queue" available to a user. Instead, the two classes qhead and qtail provide the services needed. This allows explicit separation between the source and the recipient of data. The declaration of class qhead looks like this:

† The handling of run time errors will be described below.
class qhead : public object {
...
public: 
  qhead(int =WMODE, int =10000); 
  object* get(); 
  int putback(object*); 
  int rdcnt(); 
  int rdmode(); 
  int rdmax();
  void setmode(int); 
  void setmax(int); 
  qtail* tail();
  qhead* cut();
  void splice(qtail*);
  void print(int);
};

A queue can be created like this:

qhead qh;

To obtain a qtail for an existing queue execute tail() for its qhead:

qtail* qtp = qh.tail();

The queue could now be used as a one way inter-task communication channel by giving its head and tail as arguments to two new tasks:

producer PP(qtp); 
consumer CC(&qh);

The producer task PP can now put() objects to its qtail (denoted by the pointer qtp) and the consumer task CC can get() those objects from its qhead (denoted by the pointer &qh). The class qtail function put() takes a pointer to a class object as argument, and the class qhead function get() returns such a pointer. Unless the user has specified otherwise a task executing put() will be suspended temporarily if the queue is full. When the queue becomes empty the suspended task is resumed. Similarly a task executing get() on an empty queue will be suspended until the queue becomes non-empty.

The objects transmitted through a queue must be of class object or derived from it. Class object is provided by the task system, and it is up to the programmer to define types of objects suitable for each application. Appendix A describes class object.

4 Example: A Server Task

As an example of the use of tasks and queues we will define a "server" task that receives requests for service in the form of messages on a queue, handles the requests and returns replies on other queues. One could define a class message as follows:

† The default maximum size for a queue is 10000. That is, the queue can hold up to 10000 pointers. It does not, however, pre-allocate space.
struct message : public object {
    int r_operation;
    int r_arg1;
    int r_arg2;
    qtail r_reply;
};

A message, that is an object of class message, describes an operation r_operation that is to be performed by the recipient of the message. Arguments for this operation can be passed as r_arg1 and r_arg2, and the result of the operation is to be returned as a message on the queue denoted by r_reply.

A task serving requests presented as messages on a queue can be defined as follows:

    class server : public task {
        server(qhead*);
    };

    server(server(qhead* in)
    {
        for (;;) {
            message* req = (message*) in->get();
            queue* reply = req->r_reply;
            int res = VALUE;
            int val;

            switch (req->r_operation) {
            case PLUS:
                val = req->r_arg1 + req->r_arg2;
                break;
            case MINUS:
                ...
            default:
                res = ERROR;
            }
            req->r_operation = res;
            req->r_arg1 = val;
            reply->put(req);
        }
    }

This style of server has proved useful in many contexts. In particular, it is the backbone of many "message-based systems". In this particular example a server, that is an object of class server, and the queue on which it depends can be declared:

    qtail* rq = new qtail;
    server* ser = new server(rq->head());

Other tasks can now send a request to this particular server through rq. For example:

    qhead reply;
    qhead* reply_to = reply.tail();
    message* mess = new message;

    mess->r_operation = PLUS;
    mess->r_arg1 = 1;
    mess->r_arg2 = 2;
    mess->r_reply = reply_to;

    rq->put(mess);
    mess = (message*) reply->get();
    if (mess->r_operation == ERROR) error();
5 More about Queues: Mode and Size

A qhead has a private variable mode that controls what happens when get() is executed on an empty queue. In EMODE this causes a run time error. In ZMODE it will cause get() to return the NULL pointer instead of a pointer to an object. In WMODE a task executing a get() on an empty queue will wait on that queue, that is become IDLE, until the queue becomes non-empty. Unless the user specifies the mode explicitly a qhead will be in WMODE. The qhead function setmode() can be used to reset the mode. The function rdmode() returns the mode of a qhead.

As mentioned above a queue also has a maximum size. This can be reset using the function setmax(), and read using the function rdmmax().

The mode and maximum size for a queue can also be specified when the queue is created. For example:

```c
qhead Q1(EMODE,10);
qhead GP2 = new qhead(EMODE,64*1024);
```

The public part of the declaration of class qtail is similar to that of class qhead. The two classes complement each other, and together they provide a representation of the general idea of a queue:

```c
class qtail : public object {
   public:
      qtail(int =WMODE, int =10000);
      int put(object*);
      int rdspace();
      int rdmmax();
      int rdmode();
      void setmax(int);
      void setmode(int);
      qhead* head();
      qtail* cut();
      void splice(qhead*);
      void print(int);
};
```

A qtail's mode controls what happens on queue overflow in the same way as qhead's mode controls what happens on queue underflow. For example, when a task executes put() on a full queue where the qtail is in WMODE, then that task will be suspended waiting for a get() on the head. The mode of a qhead or a qtail can be inspected by rdmode() and changed at any time by setmode(). The modes of a queue's qhead and qtail need not be the same.

Similarly the maximum number of objects which can be on a queue can be examined by rdmmax() and changed by setmax(). Decreasing the max below the current number of objects on the queue is legal. Doing this simply implies that no new objects can be put() on the queue until the queue has been drained below the new limit.

The qhead function rdcoun() returns the current number of objects in a queue, and the qtail function rdspace() returns the number of objects which can be inserted into a queue before it becomes full.

The qhead function putback() puts its argument back at the head of the queue, that is
qhead qh(WMODE, 10);
object* oo = qh.get();
qh.putback(oo);
oo = qh.get();

will assign the same object to oo twice. Putback() has proved to be a useful function in many
systems in the past, and it also allows a qhead to operate as a stack. When putback() is used,
the task executing it competes for queue space with tasks using put() on the queue’s tail. A
putback() to a full queue causes a run time error in both EMODE and WMODE. In ZMODE
it returns NULL.

6 More about Tasks

When a task is created it can be given three arguments. The first is a character string pointer
which is used to initialize the class task variable t_name. This name can be used to provide
more readable output and does not affect the behavior of the task. The string denoted by the
pointer will not be copied. The t_name is used by the debugging aids and error reporting func-
tions described below. The other two class task arguments are tuning parameters and will be
described below. If an argument is NULL a system default will be used. For example, we could
have given each server task a name like this:

class server : public task {
    ...
    server(char* qhead);  
    ...
};

void server.server(char* name, qhead* in) : (name, 0, 0)
{
    ...
}

server my_name_is_fred("fred", qhp);

The class task function sleep() suspends the task unconditionally without specifying what is
supposed to cause it to be resumed. The function wakeup() can be used to resume it.

The class function cancel() puts a task into the TERMINATED state and sets the return
value just like resultis(). However, cancel() does not invoke the scheduler.

The pointer
task* this task;

denotes the currently active task. If no tasks have been created its value is 0. It is illegal to assign
to this task. The use of this task enables the class task functions to be used from extern
functions without explicit passing of the current task’s this pointer.

The pointer
task* task_chain;

is the start of a chain of all tasks. In the following loop t points to every task in turn:

for (task* t=task_chain; t; t=t->t_next) ;

It is not possible to have only one task. Therefore, when the first task is created in a program
another task is implicitly created. Main() acts as its constructor, and its name is "main". It can
be suspended and resumed like any other task. Please remember that a return from main() ter-
minates a C program. If the "main" task should be terminated when there are other tasks
which should be left running, then resultis() can be used. For example,

this task->resultis(0);

can be executed in main(). The program will then run until no more tasks are or can become
RUNNING.
It is undefined what happens if a task's constructor returns. Always call \texttt{result()} instead of \texttt{return}, and never just "drop out of the bottom" of such a constructor. Unless a task's new function contains an infinite loop so that it will never terminate place a call of \texttt{result()} at the end of its body.

The task system does not provide a garbage collector. It is left to the programmer to ensure that pointers to deallocated store are not used.

7 Waiting

Functions like \texttt{task.result()}, \texttt{qhead.get()}, and \texttt{qtail.put()} each provide a way of waiting for one single specific event to happen. More general facilities are sometimes needed. The class \texttt{task} function \texttt{wait()} provides a way of waiting on an arbitrary object. For example, if \texttt{taskp} is a pointer to a task then

\begin{verbatim}
  wait(taskp);
\end{verbatim}

will suspend the task executing it until the task denoted by \texttt{taskp} becomes TERMINATED.

Each class derived from class \texttt{object} which is ever going to be "waited on" must have some rules associated with it specifying under which conditions a task executing a \texttt{wait()} for it will be resumed. The rules for class \texttt{task}, for class \texttt{qhead}, and for class \texttt{qtail} have been stated.

The conditions for wakeup are reflected in state changes in the objects, and are not just transitory unrecovered signals. For example, if a task executes a \texttt{wait()} for a non-empty \texttt{qhead} it will immediately continue, that is the condition for returning from a \texttt{wait()} for a \texttt{qhead} is that the queue is non-empty, not a brief state change from empty to non-empty. Rules of this type simplify programming considerably by eliminating race conditions.

The class \texttt{task} functions \texttt{waitvec()} and \texttt{waitlist()} suspend a task waiting for one of a list of objects, for example to wait for messages to arrive on one of a number of \texttt{qheads}. \texttt{waitlist()} takes a list of object pointers terminated by a zero as argument, for example:

\begin{verbatim}
  qhead* q1;
  qhead* q2;
  short who = waitlist(q1,q2,0);
\end{verbatim}

will suspend the task executing it until either \texttt{q1} or \texttt{q2} is non-empty. If either is non-empty when \texttt{waitlist()} is executed the task will continue immediately.

The value returned is the position in the list of the object that caused the return from the wait, that is if \texttt{q2} caused the task to resume the value \texttt{who} will be assigned to \texttt{who}. Positions are numbered starting from 0. \texttt{waitlist()} can take any number of arguments. The degenerated example

\begin{verbatim}
  waitlist(0);
\end{verbatim}

causes unconditional suspension of the task executing it without any guarantee of later resumption. It is equivalent to \texttt{sleep()} and \texttt{wait(0)}.

Please note that one should not assume that because \texttt{waitlist()} returns a particular value indicating one object as the cause of resumption none of the other objects are "ready". The value returned by \texttt{waitlist()} only indicates what is known to have happened, and it does not exclude other independent possibilities. On the other hand, even if \texttt{waitvec()} indicates a particular object, that object cannot in all circumstances be assumed to be "ready". For example, two tasks could be taking objects from the same \texttt{qhead}, each using \texttt{waitvec()} to wait for several objects. If \texttt{waitvec()} returns with an indication that the queue has become non-empty, then this does not guarantee that the queue is still non-empty.

Because every class in the task system allows non-blocking examination of the conditions which might lead to suspension using the three \texttt{wait} functions, the value returned by \texttt{waitvec()} can always be ignored. The information it conveys can always be obtained by direct inquiry. In many cases, however, the value returned can be trusted and used to write simpler, more efficient programs.

\texttt{Waitvec()} takes the address of a vector holding a list of object pointers, for example:
object* vec[] = { q1, q2, 0 };
short who = waitvec(vec);

is equivalent to the previous example.

8 System Time and Timers

The long variable clock measures simulated time. It is initialized to zero. It is illegal to
assign to clock.
The task function delay suspends a task for a specified time. That is,

long t = clock;
delay(n);
actual_delay = clock - t;

will assign the value n to actual_delay. Delay() is useful for representing service delays in
simulations. While a task is delayed in this way its state is still RUNNING, but it will not be
affected by the actions of other tasks except if cancel() or preempt() is used on it.
Delay(n) makes an IDLE task RUNNING so that it will start executing at time clock + n.
The class task function preempt() makes a RUNNING task IDLE and returns the number
of time units left of its delay. Applying preempt() to a IDLE or TERMINATED task causes a
run time error. This function is useful when tasks are used to represent processes in a system with
preemptive scheduling and delay times are used to represent the time used by executing processes.
The value returned by preempt() allows the preempted task to be re-started with a new delay
time which is a function of the delay time at the time of preemption. For example:

int time_left = other_task->preempt();
other_task->delay(time_left + 10);

A timer provides a facility for implementing time-outs and other time dependent phenomena.
Class timer has this declaration:

class timer : public object {

public:
...  
timer(int);
int rstate();
int result();
void reset(int);
void cancel(int);
void print(int);
};

A timer is quite similar to a task with a constructor consisting of the single statement
delay(d); that is, when a timer is created it simply waits for the number of time units given to it
as its argument, and then wakes up any tasks waiting for it.
A timer's state can be either RUNNING or TERMINATED. This state can be inspected by
using rstate().
A common use of timers is to wait for a task and a timer. For example, one can wait for the
completion of a task handling an input operation and also on a timer. The timer ensures that the
waiting task will eventually be resumed even if the input operation is never completed†:

† In a quasi-parallel system this will only be true provided no infinite loop without task system calls exists. Such a loop
constitutes an error that only a system with true parallelism can recover from.
timer* tt = new timer(15);
short res = waitlist(io_ptr, tt, 0);

switch (res) {
    case 0: /* normal completion of i/o */
        ...
        break;
    case 1: /* time out occurred */
        ...
        break;
    default:
        error(IMPOSSIBLE);
}

The class timer function result() is very similar to task.result(). They differ only in that the value returned by timer.result() is undefined unless cancel() was used. In the same way timer.cancel() is identical to task.cancel().

The function reset() re-sets the timer delay to the value of its argument. This makes repeated use of timers possible. A timer can be reset() even when it is TERMINATED.

A unit of simulated time can be used to represent any unit of real time. Only use of delay() causes the clock to advance.

9 More about Queues: Cutting and Splicing

One of the most convenient and powerful ways of using tasks involves tasks defined to do a transformation on a data stream. Such a task is called a filter. It reads its input from one queue and writes its output onto another queue. Tasks at the "other ends" of these queues tend to view these queues plus the filter as one entity. The data source simply sees an output queue that is being emptied at some rate, and the task at the receiving end sees an input queue being filled. In other words, a task sees only its input and output queues and cares little about the "internal organization" of the programs that operate on the other ends of those queues.

For example, one task could produce a stream of lines of characters, that is objects of class line, and another expect an input stream consisting of words, that is objects of class word. A filter that handles the conversion could be defined and used like this:

```c
struct line_to_word : public task {
    line_to_word(qhead*, qtail*);
    word* next_word(line*);
};

line_to_word.line_to_word(qhead* in_q, qtail* out_q)
{
    for(;;) {
        word* w;
        line* l = in_q->get();
        while(w = next_word(l)) out_q->put(w);
    }
}
```

```c
define line_q = new qhead(WMODE, 10);
define word_q = new qtail(WMODE, 50);
```

```c
producer* prod = new producer(line_q->tail());
consumer* cons = new consumer(word_q->head());
line_to_word* filt = new line_to_word(line_q, word_q);
```

In this way the filter filt is programmed into the path between cons and prod using two queues to separate filt's input from its output.

This is a fairly static use of a filter. Often one would like to insert a filter into an existing data path. For example, a macro-based text formatting program could be organized as a sequence of
filters - each doing its small part of the common task. First some filters re-arrange the input into a form suitable for the formatter proper, then the "input independent" formatter does its job producing output of a standard form, and last some output filters adjust this output to a form suitable for physical output. The task filter is an example of such a filter. In this scenario it would be useful to have each macro defined as a filter which the formatter proper inserts just in front of itself when the macro expansion is needed and which removes itself when it is not needed any more. Assuming that data streams are represented by queues, this can be achieved by using the class qhead functions cut() and splice().

When the task formatter recognizes a call to the macro "foo" it creates a new task of class macro to handle a macro of type FOO and diverts its own input through it. This is done by first "cutting" the input queue to create a place to insert the new filter, and then creating the filter giving it the new qhead and qtail as arguments:

```c
qhead* newhead = input_queue->cut();
quail* newtail = input_queue->tail();
macro* f = new macro(FOO, newhead, newtail);
```

Cut() splits the queue to which it is applied into two. Newhead, the pointer returned from cut(), denotes the qhead for the original queue and has the same mode as the original qhead. The original qhead is now attached to a new empty queue with the same max as the original.

Put()'s to the original qtail will therefore place objects on the filter's input queue, and get()'s from the original qhead will retrieve objects from the filter's output queue.

The result of these operations has been to insert a filter with an input and an output queue into a queue without changing the appearance of that queue to anyone using it, and without halting the flow of objects through that queue. In our example the macro expansion filter foo will get() the input which would otherwise have gone to the formatter, interpret it as macro arguments, and output the expanded input as its output.

The filter can be removed again by splicing its input and output queues together with splice():

```c
newhead->splice(newtail);
```

Splice() deletes the qhead to which it is applied, the qtail given to it as an argument, and the queue denoted by that qtail. If the splice() operation causes an empty queue to become non-empty or a full queue to become non-full all tasks waiting for such a state change are resumed.

Deleting the filter completes the cleanup:

```c
delete filt;
```

Typically a filter would remove itself when its task was completed, because the task that inserted it would not be programmed to be aware of the presence of the filter it inserted. The sequence of operations which enables a task to remove itself without a trace is:

```c
cancel(any_value);
delete this;
```

This will work because cancel() does not imply immediate suspension, only a guarantee that the task cannot be resumed.

The qtail functions cut() and splice() are similar to qhead's, but they operate on the other end of the queue.
10 Encapsulation

Passing information between tasks through queues can lead to rather tedious repetitive (and therefore error prone) packing and unpacking of information into messages. Simple encapsulation techniques can be used to relieve the programmer of this. For example, by adding a constructor to the class message the server example could be re-written thus:

```java
struct message : public object {
    int  r_operation;
    int  r_arg1;
    int  r_arg2;
    qtail* r_reply;
    message(int op, int a1, int a2, qtail* rp)
        { r_operation=op; r_arg1=a1; r_arg2=a2; r_reply=rp; }
};
```

```java
rq->put( new message(PLUS, 1, 2, reply_to) );
message* mess = (message*) rply->get();
if (mess->r_operation == ERROR) error();
```

Furthermore, because the message queues obviously are meant to hold only message objects a specific message queue could be defined and used:

```java
struct mqhead : public qhead {
    message* get() { return (message*) qhead.get(); };
};
```

```java
struct mqtail : public qtail {
    int put(message* m) { return qtail.put(m); };
};
```

The use of mqtail.put() ensures that only class message objects are put on the queue, and no type cast is needed when class message objects are taken from the queue using mqhead.get(). For example:

```java
mess = rply->get();
```

Because the body of mqtail.put() is present in the class mqtail declaration calls of mqtail.put() will be expanded inline. This ensures that using a mqtail is no less efficient as using a qtail directly. In many cases some error handling can also be handled by the derived put() and get() functions.

An alternative solution is to provide the server class with functions which handle the packing:

```java
class server : public task {
    qhead*  inp;
    public:
        server(char* name) : (name) { inp=new qtail(WMODE,100); }
        int  plus(int, int, mqtail*);
        int  minus(int, int, mqtail*);
};
```

```java
int server.plus(int arg1, int arg2, mqhead* rqt)
{
    inp->put( new message(PLUS,arg1,arg2,rq) );
    message* mess = rqt->head()->get();
    int x = mess->r_operation;
    delete mess;
    return x;
}
```

so now the server task can be requested to perform services like this:
mqtail qq;
server SS("plus_and_minus");
int two = SS.plus(1, 1, &qq);
int ten = SS.minus(12, 2, &qq);

For large programs this style of inter-task communication promises not only increased clarity; but also increased efficiency. The message queue interaction may, where necessary, be transparently replaced by a specially tailored inter-task communication facility.

11 Histograms and Random Numbers

To ease data gathering class histogram is provided.

```c
struct histogram {
    int  l, r, nbin;
    int* h;
    long sum;
    long sqsum;
    void histogram(int=16, int=0, int=16);
    void add(int);
    void print();
};
```

A histogram consists of nbin bins h[0] ... h[nbin-1] covering a range [l:r] of integers. The function add() adds one to the correct bin for its integer argument. The sum of the integers added is maintained in sum, and the sum of their squares is maintained in sqsum. If an argument to add() is outside the range [l:r] the range is adapted by either decreasing l or increasing r. The number of bins remains constant so the size of the range covered by a bin is doubled each time the size of the range [l:r] is. The print() function prints out the numbers of entries for each non-empty bin.

In most simulations some form of random number generation is needed. The generators provided here are intended to help the developer of a simulation to get started and to provide a paradigm for generators of more suitable distributions.

```c
class randint {
    /* uniform distribution of positive integers and floats */

    public:
        void seed(long);
        randint(long s =0) { seed(s); };
        int draw();
        float fdraw();
};
```

The following program shows the use of class randint. The ints returned by draw() are uniformly distributed in the interval [0::largest_positive_int]. The floats returned by fdraw() are uniformly distributed in the interval [0:1].

```c
main()
{
    randint ir;

    for (register i=0; i<100; i++)
        printf("i=%d f=%f ", ir.draw(), ir.fdrow());
}
```

Each object of class randint provides an independent sequence of random numbers. The seed() function can be used to reinitialize a generator. The draw() function uses the same algorithm as the C library rand(). Using class randint, generators for other distributions are easily programmed. Note that erand.draw() calls log() from the math library, so a program using it must be loaded with -lm.
struct urand : public randint {
    /* uniform distribution in the interval [low:high] */
    int low, high;
    urand(int ll, int hh) { low=ll; high=hh; };
    int draw();
};

struct erand : public randint {
    /* exponential distribution with mean "mean" */
    int mean;
    erand(int m) { mean=m; };
    int draw();
};

12 Implementation Details

The following sections contain many implementation-dependent details. The implementation
described is the version for a VAX running UNIX. Implementation-dependent information is
unfortunately often necessary to allow tuning and ease debugging.

13 Task Stack Allocation

The two arguments mode and stacksize allow the user to guide the system's handling of the
task. Their exact interpretation is implementation dependent. Users who are not interested in
implementation details and/or want a more portable program should set them both to zero. The
system will then choose (hopefully reasonable) implementation-dependent default values.

The stacksize argument indicates the maximum amount of stack storage that the task is
allowed to use. Using more is an error. It will be expressed in a unit of store suitable for stack
allocation on the host system. The stack is the one which is supported by the standard compiler
and operating system.

The mode provides additional information: The value SHARED indicates that the stack space
should be taken from the stack space of the parent task, that is the task which created the new
task. Where SHARED stacks are used the active part of the stack is copied to a save area when a
task is suspended, and copied back when it is resumed. Since stack locations are not dedicated to a
single task pointers to local variables should not be passed to other tasks. The time needed to
suspend and resume a task with SHARED stack is approximately proportional to the amount of
stack space actually used at the time of suspension.

If, on the other hand mode is DEDICATED then a new and separate stack area is allocated,
and no copying of stack space will occur.

14 Scheduling

Functions of a system class, known as the scheduler, are invoked as the result of any function
of class task which causes the suspension of a running task, and may be invoked by any function
from the standard classes described here. The scheduler selects the next task to run. When the
scheduler finds no more tasks to run it examines the pointer variable exit_fct, and if this is
non-zero the scheduler will call the function denoted by it.

Whenever clock is advanced the scheduler examines the pointer variable clock_task. If
this denotes a task, then that task will be resumed before any other task. The clock_task must
be IDLE when resumed by the scheduler. The class task function sleep() is useful to ensure
this.
15 Debugging and Tuning Aids

The task system has been designed under the assumption that a typical use of tasks may involve hundreds of tasks and need tuning to achieve an acceptable time-space tradeoff. The task of debugging such a system can safely be assumed to be non-trivial.

Classes were used in the implementation of the task system largely because they allow the scope of data and functions to be explicitly restricted to the object to which they belong. This allows better type checking of a multi-threaded program than could be achieved by a function-based implementation. The classes which constitute the task system were designed to allow quite strong type checking of programs using them.

A number of run time errors are detected by the task system. For example it is illegal to delete a queue on which a task is waiting. When such a run time error is detected the task system function task_error is called with the number of the error and the this pointer of the object which caused the error as arguments. Appendix B is a list of run time errors. Task_error() will in turn examine the pointer error_fct, and if this is non-zero call the function denoted by it with a copy of its own arguments. Otherwise task_error() will call the system function exit() with the error number as argument.

When returning from task_error() after executing an error_fct which returned rather than using exit() the task system will re-try the operation which caused the error (provided that error_fct could have affected the condition which caused the error). For example, a put() to a qhead will be re-trying because the user's error_fct might have either caused the get() function to be used on the queue, or used chmax() to allow more objects to be inserted into that queue. Note that allocation operations using the new operator which failed due to lack of free store will be re-trying because some kind of garbage collection may have been implemented in error_fct by the user.

Beware of infinite loops.

All task system classes have a function print() which can be used to print the contents of their objects on stdout. A print() function takes an int argument indicating the amount of information to be printed. Print(0) gives the minimum amount of information, print(VERBOSE) rather more, and print(CHAIN) will call print() for objects on lists associated with the object with its own arguments. The print() argument constants can be combined by the or operator. For example

    this_task->print(VERBOSE);
    run_chain->print(VERBOSE|CHAIN);

will verbose every non-TERMINATED timer and every RUNNING task. For tasks information about the run time stack is printed by print(STACK). If the function hwm() has been called print(STACK) will also give an estimate of the maximum amount of stack space ever used by the task, the stack's "high water mark". For tasks that share a stack, the high water mark printed will be the high water mark of most greedy task. For example, information describing stack usage for all tasks can be printed by:

    task_chain->print(STACK|CHAIN);

The output of the print() functions is implementation-dependent and hopefully self-explanatory.

16 Overheads and Performance

The store used for representing a class object in addition to the user specified data is:

| object | 3 words |
| timer  | 5 words |
| task   | 16 words + stacksize |
| queue  | 10 words (including the qhead and the qtail) |

The time needed to execute some of the task system functions are approximately:
procedure call + return 1 unit
task suspend + resume 9 units (using result())
put 2 units
get 2 units
wait, waitvec, or waitlist 3 units

The last four actions can all cause a task to be suspended. When this happens add 6 units of time. The task system uses about 8K bytes of store for program and data.

17 Acknowledgements

The task system is in many ways a descendant of A.G. Fraser's set of C functions described in reference 3. M. D. McIlroy acted as "midwife" for many parts of the design.
18 References

[1] Brinch-Hansen, Per
The Architecture of Concurrent Programs
Prentice Hall 1977.

SIMULA Common Base Language

[3] Fraser, A. G.
C Language Routines for Multi-Thread Computation
Bell Telephone Laboratories Internal Memorandum 1979.

The C Programming Language

[5] Lampson, Butler W. and Redell, David D.
Experience with Processes and Monitors in Mesa

[6] Richards, Martin
The BCPL Programming Manual
The Computer Laboratory, University of Cambridge, England
(April 1973).

[7] Stroustrup, Bjarne
The C++ Programming Language - Reference Manual
In this volume.

[8] Stroustrup, Bjarne
A C++ Tutorial
In this volume.

Bell Telephone Laboratories (January 1979).
Appendix A: Objects

The task system as described above is implemented using a lower level of abstraction based on the direct use of the class object. Class object can also be used as a base for other (user defined) abstractions, but beware, it is an implementation tool that is not intended to be used directly.

Class object is a base class for all classes in the task system and also the most basic facility for inter-task communication. The declaration of class object looks like this:

```c
class object {
    olink* o_link;
    public:
        object(int =0);
        ~object();
        short o_type;
        object* o_next;
        void remember(task*);
        void forget(task*);
        void alert();
        void print(int);
};
```

The task system implements objects of type TASK, QHEAD, QTAIL, and TIMER.

A task can be added to the set of tasks "remembered" by an object by executing remember() and a task can be removed from this set by executing forget(). Executing alert() has the effect of transferring all IDLE tasks remembered by the object to the RUNNING state. A task can be "remembered" by several objects or several times by the same object without any bad effects. Forget() will insure that its argument is not "remembered" any more, and it causes no bad effects when used for an object that does not "remember" its argument task. No record is kept of how many alert() operations have been executed on an object. Alert() does not cause an object to forget() tasks. Executing a remember() does not suspend a task. Applying alert() to an object that does not remember any tasks is legal, but has no effect. Caveat emptor!

The class object functions remember(), forget(), and alert() provide a simple, efficient, but unstructured and therefore error-prone, communication mechanism.

The declarations for the task system classes can be found in <task.h>.