The Charm++ (4.5) Programming Language Manual

February 29, 2000

The Charm software was developed as a group effort. The earliest prototype, Chare Kernel(1.0), was developed by Wennie Shu and Kevin Nomura working with Laxmikant Kale. The second prototype, Chare Kernel(2.0), a complete re-write with major design changes, was developed by a team consisting of Wayne Fenton, Balkrishna Ramkumar, Vikram Saletore, Amitabh B. Sinha and Laxmikant Kale. The translator for Chare Kernel(2.0) was written by Manish Gupta. Charm(3.0), with significant design changes, was developed by a team consisting of Attila Gursoy, Balkrishna Ramkumar, Amitabh B. Sinha and Laxmikant Kale, with a new translator written by Nimish Shah. The Charm++ implementation was done by Sanjeev Krishnan. Charm(4.0) included Charm++ and was released in fall 1993. The current version of the Charm software (4.5), has been developed by Attila Gursoy, Sanjeev Krishnan, Milind Bhandarkar, Joshua Yelon, Narain Jagathesan and Laxmikant Kale.

Contents

1 Introduction **4**

2 The CHARM++ language **5**

2.1 Structure of a CHARM++ program 5

2.2 Basic CHARM++ constructs 5

2.3 CHARM++ Execution Model 7

3 Basic CHARM++ Calls **8**

3.1 Creating Chares 8

3.2 Creating Branched Chares 8

3.3 Sending Messages 9

3.4 Message Creation and Deletion 10

3.5 CHARM++ Exit Calls 11

3.6 Quiescence Detection 12

3.7 Other Charm++ Calls 12

4 Information Sharing Abstractions **14**

4.1 Read Only Variables and Messages 14

4.2 Accumulator Variables 14

4.3 Monotonic Variables 16

4.4 Write Once Variables 18

4.5 Distributed Tables 19

5 Other CHARM++ Features **22**

5.1 Message Packing 22

5.2 Prioritized Execution 23

A List of Keywords in Charm++ **27**

B Syntax changes from previous versions of Charm++ **28**

C Calling Charm++ methods from C++ files **29**

D Further Information **31**

D.1 Related Publications 31

D.2 Associated Tools 31

D.3 Contacts 31

1 Introduction

This manual describes CHARM++, an object oriented portable parallel programming language based on C++. Its program structure and execution model, language constructs and runtime system calls are described[1]. The current version of CHARM++ has some syntax changes as compared to the previous version. This manual overrides older syntax descriptions (such as those in the OOPSLA 1993 paper). Backward compatibility is not maintained in some cases with older syntax, please check Appendix B in this manual for specific details.

CHARM++ is an explicitly parallel language consisting of C++ with a few extensions. It provides a clear separation between sequential and parallel objects. The execution model of CHARM++ is message driven, thus helping one write programs that are latency-tolerant. CHARM++ provides specific modes for sharing information between parallel objects. Several dynamic load balancing strategies are provided. CHARM++ supports both irregular as well as regular, data-parallel applications. It is based on the CHARM parallel programming system, and its runtime system implementation reuses most of the runtime system for CHARM. This manual describes the constructs and syntax of CHARM++ and explains its features.

Currently the parallel platforms supported by CHARM++ are the TMC CM-5, nCUBE/2, Intel Paragon (OSF and Sunmos), IBM SP-1, a single workstation or a network of workstations from Sun Microsystems (SunOS or Solaris), IBM RS-6000 (AIX) and the HP 700 (HP-UX) series. There is also an uniprocessor-based multiprocessor simulator. CHARM++ programs can run without change on all these platforms. Please see the "Installation and Running Charm Programs" manual for details.

For a description of the C-based CHARM parallel programming system, please refer to the CHARM programming language manual and the tutorial introduction to CHARM. For details about installing, compiling and running Charm++ programs, please refer to the installation manual. For details about tools, please refer to the manual for each tool. See Appendix D for the addresses of our Internet sites and pointers to more information.

[1]For a description of the underlying design philosophy please refer to the following papers :
L. V. Kale and Sanjeev Krishnan, *"CHARM++ : Parallel Programming with Message-Driven Objects"*, in "Parallel Programming Using C++", MIT Press, 1995. (to be published).
L. V. Kale and Sanjeev Krishnan, *"CHARM++ : A Portable Concurrent Object Oriented System Based On C++"*, Proceedings of the Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), September 1993.

2 The CHARM++ language

2.1 Structure of a CHARM++ program

A CHARM++ program is structurally similar to a C++ program. Many components of a CHARM++ program *are* C++ code fragments [2]. The main syntactic units in a CHARM++ program are function and class definitions. A CHARM++ program can possibly be distributed across several source code files.

There are five disjoint categories of objects (classes) in CHARM++ :

- Sequential objects : as in C++
- Chares [3] (concurrent objects)
- Branched chares (a form of replicated objects)
- Shared objects (specific information sharing abstractions)
- Messages (communication objects)

2.2 Basic CHARM++ constructs

This section describes the syntax of message, chare and branched chare definitions. Please see Section 4 for a description of shared objects.

Messages : Message definitions are similar to class definitions in C++.

```
message class MessageType {
    // List of data and function members as in C++
} ;
```

A message pointer has type `(MessageType *)` as in C++.

Chare classes : Chares are concurrent objects. A chare definition has the form

```
chare class ChareName [: superclass names] {
    // Data and member functions as in C++
    // One or more entry function definitions of the form
 entry:
    ChareName(MessageType1 *MsgPointer)
        { C++ code block }
    void EntryPointName2(MessageType2 *MsgPointer)
```

[2] **The C++ code cannot, however, contain global or static variables.**

[3] Chare (pronounced **chär**, ä as in cart) is Old English for chore.

```
        { C++ code block }
} ;
```

Chares are concurrent objects encapsulating medium grained units of work. Chares can be dynamically created on any processor ; there may be thousands of chares on a processor. The location of a chare is usually determined by the dynamic load balancing strategy ; however, once a chare commences execution on a processor, it does not migrate to other processors. Chares do not have a default "thread of control" : the entry-function functions in a chare execute in a message driven fashion upon arrival of a message.

The entry function definition specifies a function that is executed *without interruption* when a message is received and scheduled for processing. Only one message per chare is processed at a time. Entry function are defined exactly as normal C++ function members, except that they must have the return value void (except the constructor entry function which may not have a return value) and they must have exactly one argument which is a pointer to a message. Currently entry functions cannot be overloaded (i.e. there cannot be two entry function of the same name in a chare definition).

Each chare instance is identified by a *handle* which is essentially a global pointer, and is unique across all processors. The handle of a chare of type `ChareName` has type (`ChareName` handle). The variable thishandle holds the handle of the chare whose entry function or public function is currently executing. thishandle is a public local variable of the chare object (it is inherited from the system-defined superclass for chares). thishandle can be used to set fields in a message. This mechanism allows chares to send their handles to other chares. The variable mainhandle holds the handle of the single instance of the main chare.

The programmer can construct inheritance hierarchies of chares ; however, currently private and protected base chare classes are not supported.

Branched chare classes : These are modeled after the "branch office chares" of CHARM. A branched chare is really a group of chares that has a branch (representative of the group) on each processor, with each branch having its own data members. Branched chares have a definition syntax similar to normal chares, except that they must inherit from the system defined class groupmember.

```
chare class ChareName : public groupmember [,other superclasses ] {
{   // Data and member functions as in C++
    // Entry functions as for normal chares
} ;
```

A branched chare is identified by a globally unique group id, whose type is (`ChareName` group). This id is common to all its branches. This id can be obtained from the variable thisgroup, which is a public local variable of the groupmember superclass. For branched chares, thishandle is the handle of the particular branch in which the function is executing : it may be used exactly as a normal chare handle.

Sequential objects, normal chares and other branched chares can access public members of the

branch of a branched chare *on their processor* by
`ChareGroupId[LOCAL]->DataMember` and
`ChareGroupId[LOCAL]->FunctionMember()`. Thus a dynamically located chare can call a public function of a branched chare without needing to know which processor it is itself on : the function call executes in the local branch of the branched chare.

Branched chares can be used to implement data-parallel operations easily. Messages can be broadcast to all branches of a branched chare as well as sent to a particular branch.

There can be many instances corresponding to a branched chare type : each instance has a different handle and its set of branches on all processors.

2.3 CHARM++ Execution Model

Every CHARM++ program must have a chare type `main`. There can be only one instance of this chare type, which executes on a single system selected processor. This must have the public function `main`. Execution of a CHARM++ program begins at this function. Typically, this function starts the computation by creating chares and branched chares. It can also be used to initialize shared objects.

CHARM++ program execution is terminated by the *CharmExit()* call. This call is not the same as exit() in Unix : it merely informs the runtime system that all computations on all processors must be terminated, and then returns to the user program: the programmer should make sure that no work is done after CharmExit() is called. The runtime system ensures that no messages are processed and no entry-functions are called after the currently executing entry-function completes. CharmExit() need not be called on all processors : it is enough to call it from just one processor at the end of the computation.

The user can specify an entry function (using the *CStartQuiescence()* call, see Section 4) to be called when the computation has become quiescent (i.e., when no processor is executing an entry function and all messages that have been sent have also been consumed). This entry function is useful in computations where the user cannot forsee when the program is going to become quiescent. It is also useful for computations which proceed in phases — the quiescence entry function can be used to start new phases of computation.

The basic unit of parallel computation in CHARM++ programs is the chare. A chare is similar to a process, an actor, an ADA task, etc. A chare is created by the newchare operator. At runtime, many chares may be active. Chares may send messages to one another. Conceptually, the system maintains a "work-pool" consisting of seeds for new chares (which are created when newchare calls are made), and messages for existing chares. The runtime system may pick multiple items, non-deterministically, from this pool and execute them. It will not process two messages for the same chare concurrently, but otherwise it is free to schedule them in any way.

3 Basic CHARM++ Calls

This section contains the basic system calls in CHARM++. These calls can be used to create new chares, send messages to existing chares, allocate memory and free memory. Various other calls for input-output and exiting from CHARM++ programs are also described in this section.

If a CHARM++ call returns a value, the value must be cast to the proper type.

3.1 Creating Chares

The operator *newchare* creates a new instance of a chare. The syntax of the call is :

newchare [(**&VirtualHandle** [, **DestPE**])] **ChareName(MsgPtr)**

where

ChareName: The type name of the chare that is to be created.

MsgPtr: A pointer to a message whose type must correspond to the parameter for the constructor entry function ChareName() in chare ChareName.

&VirtualHandle: The first placement parameter is a pointer to a chare handle, which is filled by the *newchare* operator. This optional argument can be used if the user desires to have a *virtual* handle to the instance of the chare that will be created. This handle is useful for sending messages to the chare, even though it has not yet been created on any processor. Messages sent to this virtual handle are either queued up to be sent to the chare after it has been created, or simply redirected if the chare has already been created. For performance reasons, therefore, virtual handles should be used only when absolutely necessary. Virtual handles are otherwise like normal handles, and may be sent to other processors in messages.

DestPE: When a chare is to be created at a specific processor, the second placement parameter DestPE is used to specify that processor. If no virtual handle is required to be returned, the first placement parameter can be *NULL_HANDLE*. Note that, in general, for good load balancing, the user should let CHARM++ determine the processor on which to create a chare. Under unusual circumstances, however, the user may want to choose the destination processor. If a process replicated on every processor is desired, then branched chares should be used.

The newchare operator deposits the *seed* for a chare in a pool of seeds and returns immediately. The chare will be created later on some processor, as determined by the dynamic load balancing strategy. When a chare is created, it is initialized by calling its constructor entry function with the message parameter specified to newchare. The newchare operator does not return any value.

3.2 Creating Branched Chares

Branched chares are also created with the newchare operator. The syntax of the operator is :

newchare [(ReturnEF, ReturnHandle)] ChareName(MsgPtr)

This call broadcasts the message `MsgPtr` to all processors, creates a branch of the branched chare on every processor and executes the constructor entry function in that branch with the message `MsgPtr` as parameter.

Branched chares are usually created in the main function of the main chare, in which case the *newchare* operator returns the group id of the newly created branched chare. When branched chares are dynamically created (from outside the main function), the group id is returned in a message to the entry function ReturnEF in the chare whose chare handle is ReturnHandle. The ReturnEF entry function must accept a message which has ChareName group as its first field.

The ReturnEF–ReturnHandle mechanism : Since all CHARM++ calls are non-blocking, calls that require remote computations cannot return values immediately. Instead, the user is required to specify a ReturnEF–ReturnHandle pair, which is like a *continuation*, to which the value is sent later in a message. This mechanism will be used in many calls.

3.3 Sending Messages

A message may be sent to a chare using the notation :

ChareHandle=>EntryFunction(MessagePointer)

This sends the message in MessagePointer to the chare whose handle is ChareHandle at the entry function EntryFunction, which must be a valid entry function of that chare type. This call is asynchronous and non-blocking ; it returns immediately after sending the message. The EntryFunction could also be a variable of type *EntryFunctionType*. Currently the Charm++ translator is not sophisticated enough to handle arbitrary expressions for ChareHandle : it is best to declare it as a variable in some enclosing scope.

A message may be sent to a particular branch of a branched chare using the notation :

ChareGroupId[Processor]=>EntryFunction(MessagePointer)

This sends the message in MessagePointer to the branch of the branched chare ChareGroupId which is on processor number Processor at the entry function EntryFunction, which must be a valid entry function of that chare type. This call is asynchronous and non-blocking ; it returns immediately after sending the message.

A message may be broadcast to all branches of a branched chare (i.e., to all processors) using the notation :

ChareGroupId[ALL]=>EntryFunction(MessagePointer)

This sends the message in MessagePointer to all branches of the branched chare ChareGroupId at the entry function EntryFunction, which must be a valid entry function of that branched chare type. This call is asynchronous and non-blocking ; it returns immediately after sending the message.

Note that the programmer relinquishes control of a message after sending it : further access to the message field can cause runtime errors.

3.4 Message Creation and Deletion

Messages are allocated using the C++ new operator as usual :

MessageType *msgptr = new [((int *)sizes_array)] MessageType [(constructor arguments)] ;

In Section 5.1 we explain how messages can contain arbitrary pointers, and how the validity of such pointers can be maintained across processors in a distributed memory machine. However, we also provide the user a keyword VARSIZE to declare messages with *single dimensional* arrays whose sizes may be known only at run time when the message is allocated. [4] The translator generates code to handle the pointers (so as to maintain validity) for such messages. The optional placement parameter sizes_array to the new operator is a pointer to an array of integers, and is used to specify variable sized messages. An example of usage of a variable sized message appears below:

```
message class MSG {
public:
    int size1, size2;
    float field1[VARSIZE];
    float field2[VARSIZE];
} ;
...
...
int sizes[2];

sizes[0] = Size1;
sizes[1] = Size2;
MSG *msg = new (sizes) MSG ;
msg->size1 = Size1;
msg->size2 = Size2;
for (i=0; i<Size1; i++)
        msg->field1[i] = i;
for (j=0; j<Size2; j++)
        msg->field2[j] = j;
```

[4] The array element type must not be a pointer type, i.e., an array of pointers cannot be a variable sized array. To use messages with multi-dimensional arrays and arrays containing pointers, see Section 5.1.

In the above example a message with two single dimensional variable sized arrays has been declared. The sizes array is used to specify the sizes of the arrays in the order in which they appear in the message declaration. Notice that the user should maintain the sizes of the arrays (as done with size1 and size2 fields in the message, in this example) if such information is needed at a later stage.

Allocating prioritized messages :

For allocating space for messages that have **integer** or **bit-vector** priorities, an extra placement argument is required for the new operator :

MessageType *msgptr = new ((int)PrioSize [, sizes_array]) MessageType ;

The operator allocates a message of type MessageType and a priority field of size PrioSize bits. This operator is different from *new* in that additional information is kept by the system to facilitate the handling of the message with priorities by other system routines. Any data to be used as a prioritized message and passed to system routines MUST be allocated using this syntax. See Section 5.2 for more details on different types of priorities and how to access them. The usage of the optional parameter sizes_array is same as that of non-prioritized variable sized messages.

When a message is sent to a chare, the programmer relinquishes control of it ; the space allocated to the message is freed by the system. When a message is received at an entry point it is not freed by the runtime system. It may be reused or deleted by the programmer.

Messages can be deleted using the standard C++ delete operator.

3.5 CHARM++ Exit Calls

void ChareExit()

The *ChareExit* call is used to to release any memory allocated for the local variables in the chare. It does not stop program execution at the point of call ; hence the user must ensure that no access is made to chare-local variables after the *ChareExit* call. At present the runtime does not handle messages sent to chares that have exited by making a call to *ChareExit*; these messages constitute runtime errors, and may lead to unpredictable behavior.

void CharmExit()

Similar to the regular Unix *exit*, except that this routine will kill all running processes and then exit itself. The *CharmExit* routine does not stop execution of the program in the calling entry point immediately. It simply tells the system that after the current entry point finishes execution, it should take care of any housekeeping activities required, and ask all other running processes to terminate (again, non-preemptively).

3.6 Quiescence Detection

The call *CStartQuiescence* permits a user to detect quiescence. The syntax of the call is :

CStartQuiescence (&(ChareName::EntryFunction), ChareHandle)

Quiescence is defined as a state in which no processor is executing an entry point and all messages that have been sent have also been consumed.

When quiescence occurs, the system will send a message to the entry point EntryFunction in the chare of type ChareName whose handle is ChareHandle. The message is of type QUIESCENCE_MSG, which is predefined by the system in the file `quiescence.h`. After quiescence is detected, the user can start the next phase of the computation, and if necessary, detect quiescence again by making another *CStartQuiescence* call. This can be done indefinitely many times if required.

3.7 Other Charm++ Calls

The following calls provided commonly needed functions such as terminal I/O, timers, etc in a portable manner.

int CNumPes()
returns the total number of processors on which the program is being run.

int CMyPe()
returns the processor number on which the call was made.

int CPrintf(format [, arg]*)
This call is used for atomic terminal output. Its usage is similar to printf in C.

int CScanf(format [, arg]*)
This call is used for atomic terminal input. Its usage is similar to scanf in C.

int CTimer()
returns the current value of the system timer in milliseconds. The system timer is started when the program begins execution. This timer measures process time (user and system).

int CUTimer()
returns the current value of the system timer in microseconds. The system timer is started when the program begins execution. Note that on some platforms such as workstation networks, the granularity of the system timer may be more than a microsecond, in which case this call returns the most accurate value available. This timer measures process time (user and system).

FunctionRefType CFunctionNameToRef(function-name)
This converts a global function pointer to a reference which can be passed across address spaces. This call is useful on machines where function pointers are not the same across address spaces.

CFunctionRefToName((FunctionRefType)function-ref)
This converts a function reference back to a function pointer.

int CSpanTreeRoot()
returns the processor number of the root of the spanning tree into which all processors are organized.

int CSpanTreeParent(int proc)
returns the processor number of the parent (in the spanning tree) of processor proc.

int CNumSpanTreeChildren(int proc)
this returns the number of spanning tree children of processor, proc.

CSpanTreeChildren(int proc, int *children)
this is used to determine the children in the spanning tree for a processor, proc.

4 Information Sharing Abstractions

Messages are too general and low level a mechanism for information sharing between chares. Moreover, communication overheads can be optimized if the pattern of information sharing is known. Hence CHARM++ provides specific abstract data types, referred to as *specifically shared variables*, for sharing data amongst chares. They provide as efficient an implementation for sharing data as the underlying machine can provide.

4.1 Read Only Variables and Messages

Read Only variables and messages are used to share information which is obtained only after the program begins execution, and which does not change subsequently. These variables can be accessed from any chare on any processor ; the cost incurred is that of a local reference. Read only variables can be declared only at file scope. Read only declarations have the following form :

readonly Type readname;
readonly MessageType *readmsgname;

In the first declaration, the variable readname is declared to be a read only variable of type Type. In the second declaration, the variable readmsgname is declared to be a read only message of type MessageType. Pointers are not allowed to be read only variables, unless they are pointers to message types, in which case the message will be correctly initialized on every processor.

Read only variables and messages can be assigned values only in the main function of the main chare. The value of a read only variable is the value it has at the end of the main entry point. The assignment to a read only variable is like a normal variable assignment. In case of read only messages, the user must allocate space for the message using the *new_message* call and initialize the message.

Read only variables can be used after initialization like normal variables. Currently, the compiler does not strictly disallow assignments to read only variables ; it is the user's responsibility to make sure that no assignments occur.

4.2 Accumulator Variables

The accumulator type is like a counter – it can be used to accumulate results from chares when the result of the accumulation is needed only once (perhaps, at the end of computation). It is optimized for fast update : the cost incurred is that of a local reference.

The accumulator abstract data type is a class with only the following members :

- one data member : a message containing the data area of the accumulator data type
- a constructor. This must allocate space and initialize the above message to the zero for the user defined metric the accumulator is used for. The constructor has exactly one argument : a message which is used to carry the initialization data to all processors.

- an Accumulate() function : this operator "adds" to the accumulator variable in the user defined metric. It may have arbitrary arguments. Currently, the code for the Accumulate() function must be within the accumulator definition.
- a Combine() function : this operator "combines" values from two processors. It has exactly one argument which is a message of same type as the data member. This function is only called by the system. The Accumulate and Combine operators must be commutative-associative.
- a CollectValue() system defined function : this can be called by the user to obtain the final result of the accumulation. It has two arguments : a chare handle and a pointer to an entry point in that chare which accepts a message of the same type as the accumulator's data member. The accumulator result is returned to that entry point by the system. This function is "inherited" by all accumulator objects, so it should not be listed in the accumulator definition.

```
accumulator class AccType {
    MessageType1 *accmsg ;
        // The data member
public: AccType(MessageType2 *) ;
        // Constructor, to be defined by user
    void Accumulate ( /* arguments */ ) ;
        // To be defined by user
    void Combine (MessageType1 *) ;
        // To be defined by user
} ;
```

Figure 1: Structure of an accumulator abstract data type

An accumulator variable is initialized using the *newaccumulator* call.

newaccumulator [(ReturnEP, ReturnHandle)] AccType(msg)

The above *newaccumulator* call creates an accumulator of type AccType. The call results in its constructor being invoked on every node with a single parameter msg, which is of type MessageType2. The *newaccumulator* call returns the handle of the newly created accumulator if the call has been made inside the main entry point of the main chare. Otherwise optional parameters can be used to receive the handle at the entry point ReturnEP (which is specified as &(ChareName::EP)) of the chare having handle ReturnHandle. The message received can be defined by the user to have any structure, except the first field must be of type AccType handle. A note of caution: the value returned by the *newaccumulator* call in any place other than main may not be a valid accumulator handle.

Any chare can "add" to an accumulator variable by calling the *Accumulate* method.

acchandle->Accumulate ([arg [, arg]*])

The final value of an accumulator can be obtained by calling the *CollectValue* method.

acchandle->CollectValue (&(ChareName::EntryPoint), charehandle)

This call results in the message associated with the accumulator acchandle being sent to the to the chare charehandle (of type ChareName) at entry point EntryPoint. The method can be called only once for each accumulator instance; later usages will result in an error being reported.

In Figure 2, `A1` is declared to be an accumulator, which is an array of integers. The two operators are `Accumulate` and `Combine`; `Accumulate` adds to elements inside the accumulator, and `Combine` takes an accumulator variable as a parameter and combines it (element by element) with the current value of the accumulator.

```
message class MSG1 {
public:  int a[MAX];
} ;

accumulator class A1 {
    MSG1 *msg;
public:
    A1(MSG1 *initmsg)
    {
        msg = new MSG1 ;
        for (int i=0; i<MAX; i++)
            msg->a[i] = initmsg->a[i];
    }
    void Accumulate (int y, int z)
    {
        msg->a[y] += z;
    }
    void Combine (MSG1 *othermsg)
    {
        for (int i=0 ; i<MAX; i++)
            msg->a[i] += othermsg->a[i];
    }
} ;
```

Figure 2: Example of an accumulator definition

4.3 Monotonic Variables

A monotonic variable is similar to an accumulator in the sense that it is like a "counter". However, it differs from an accumulator variable in that it's value may be accessed frequently during the course of the computation. Also, it's value has to increase monotonically in some metric. Monotonic variables are especially useful in branch-and-bound computations, where every chare needs to know the current best bound, and this bound might change if someone found a *better* bound. Repeated supply of the same bound would have the same effect as supplying it once.

The monotonic abstract data type is a class with only the following members :

- exactly one data member : a message containing the data area of the monotonic data type

- a constructor. This must allocate space and initialize the above message to the zero for the user defined metric the monotonic is used for. The constructor has exactly one argument : a message which is used to carry the initialization data to all processors.
- an Update() function : this operator "adds" to the monotonic variable in the user defined metric. It must be idempotent, commutative, associative and monotonic. It has exactly one argument which is a message of same type as the data member. It should return 1 if the value of the variable is actually updated, and 0 otherwise. Currently, the code for the Update() function must be within the monotonic type definition.
- a MonoValue() system defined function : this can be called by the user to obtain the current value of the monotonic variable. It has no arguments. The value returned is a pointer to a message of same type as the data member. This function is "inherited" by all monotonic objects, so it should not be listed in the monotonic definition.

```
monotonic class MonoType {
    MessageType1 *monomsg ;
        // The data member
public: AccType(MessageType2 *) ;
        // Constructor, to be defined by user
    int Update (MessageType1 *) ;
        // To be defined by user
} ;
```

Figure 3: Structure of an monotonic abstract data type

An monotonic variable is initialized using the *newmonotonic* call.

newmonotonic [(ReturnEP, ReturnHandle)] MonoType(msg)

The above *newmonotonic* call creates a monotonic variable of type MonoType. The call results in its constructor being invoked on every node with a single parameter msg, which is of type MessageType2. The *newmonotonic* call returns the handle of the newly created monotonic if the call has been made inside the main entry point of the main chare. Otherwise optional parameters can be used to receive the handle at the entry point ReturnEP (which is specified as &(ChareName::EP)) of the chare having handle ReturnHandle. The message received can be defined by the user to have any structure, except the first field must be of type MonoType handle. A note of caution: the value returned by the *newmonotonic* call in any place other than main may not be a valid monotonic handle.

Any chare can update a monotonic variable by calling the *Update* method.

monohandle->Update ((MessageType1 *) msg)

The current value of a monotonic variable can be obtained by calling the *MonoValue* method.

MessageType1 * currentvalue = monohandle->MonoValue ()

This call results in the message associated with the monotonic monohandle being returned. This value may not be the globally (i.e., over all processors) current value of the variable, but it is guaranteed to *eventually* be the globally current value. However, the value returned will always be monotonically greater than or equal to the values returned by previous calls. The cost of a MonoValue call is that of a local procedure call.

In Figure 4, `GMax` is declared to be a monotonic, which contains a single integer. The functionality this monotonic implements is equivalent to multiple concurrent non-blocking global "max" operations, with the global maximum being continously updated and available on all processors.

```
message class MSG1 {
public:  int x;
} ;

monotonic class GMax {
    MSG1 *msg;
public:
    GMax(MSG1 *initmsg)
    {
        msg = new MSG1 ;
        msg->x = initmsg->x;
    }
    int Update (MSG1 *updatemsg)
    {
        if (msg->x < updatemsg->x) {
            msg->x = updatemsg->x;
            return(1) ;
        }
        return(0) ;
    }
} ;
```

Figure 4: Example of a monotonic type definition

4.4 WriteOnce Variables

Write once variables are similar to read only variables, except that they are dynamically created. Write once variables may be thought of as untyped global objects which can be initialized once from anywhere in the program, and then referenced from all processors, with the cost of a reference being that of a local procedure call. A write once variable handle has type writeonce handle. The system call that creates a write once variable is *new_writeonce*.

void new_writeonce(void *DataPtr, int DataSize, ReturnEP, ReturnHandle)

This routine creates a write-once variable of size DataSize which has the data pointed to by DataPtr. A handle denoting the write once variable is passed back to the user at the entry point ReturnEP of the chare whose handle is ReturnHandle. The message type declared by the user for receiving the write once variable handle is irrelevant as long as the first field in the struct is a variable of type writeonce handle.

The user may access the write-once variable data on any node using the DerefWriteOnce method :

```
void *dataptr = whandle->DerefWriteOnce()
        writeonce handle whandle;
```

The above call returns an untyped pointer to the actual data itself. Note that it is the user's responsibility to pass the handle (using messages or any other mechanism) to any chare that wishes to access the write once variable.

4.5 Distributed Tables

A Distributed Table is a set of entries, each entry being a record with an integer key and an untyped data field. Access to the entries are provided through the non-blocking calls *Insert*, *Delete* and *Find*. These user accessible functions of the distributed table class are defined by the system as :

```
class table {
public:
        void Insert(int key, void *data, int size_data, ReturnEP, ReturnHandle, int option) ;

        void Delete(int key, ReturnEP, ReturnHandle, int option) ;

        void Find(int key, ReturnEP, ReturnHandle, int option) ;
} ;
```

Each of the table class functions can result in a message of type TBL_MSG being sent to a specified entry point. TBL_MSG has the following structure :

```
message class TBL_MSG {
public: int key;
        void *data;
} ;
```

In each of the functions a message is sent as a reply only if the parameters ReturnEP and ReturnHandle are valid values. The user can choose to have no reply messages sent by setting the value of ReturnEP as NULL_EP and the value of ReturnHandle as NULL_HANDLE.

The key must be an integer. An element is said to match a request if the values of the key in the element and the key supplied by the request match. The matching element is said to have

defined data, if the data field of the element has a non-null value. A distributed table `tablename` is declared as follows:

#include "dtable.h"

```
table tablename;
```

This declaration must be at file scope, similar to read only variables. Currently, distributed table objects can only be statically declared, there are no table handles and there is no facility to create a table instance dynamically (i.e., there is no "new_table" call).

The table class functions are described below.

tablename.Insert(int key, void *data, int size_data, ReturnEP, ReturnHandle, int option) ;

The *Insert* call is used to insert the element with key key and data data of size size_data into the distributed table tablename. In each of the following cases a message msg of type TBL_MSG is sent back to the entry point ReturnEP in the chare whose handle is ReturnHandle. The key and data fields of the message msg are set as described below :

1. *matching element already present and data is defined :* The key and data fields of msg are the same as those of the matching element.
2. *matching element already present but data is undefined :*
   - If data is not NULL, then the data field of the matching element is defined to be data, and the key and data fields of msg are key and data, respectively.
   - If data is NULL and option = $TBL_WAITFORDATA$ then the Insert call is suspended until data is available, and the key and data fields of msg are key and the available data, respectively. Otherwise if option = $TBL_NOWAITFORDATA$ then msg is sent immediately with the key and data fields as key and NULL, respectively.
3. *matching element not already present :* An element is inserted with the key and data fields as key and data. The data is considered undefined if data = $NULL$. If option = $TBL_WAITFORDATA$ then the Insert call is suspended till data is available, and the key and data fields of msg are key and the available data, respectively. Otherwise if option = $TBL_NOWAITFORDATA$ then msg is sent immediately with the key and data fields as key and NULL, respectively.

tablename.Delete(int key, ReturnEP, ReturnHandle, int option) ;

The *Delete* function is used to delete the element with key key from the distributed table tablename. If a matching element was found a message of type TBL_MSG is sent to the entry point ReturnEP at ReturnHandle with the key field as key and the data corresponding to that of the deleted element. Note that as in the case of *Insert* the message is sent only if ReturnEP and ReturnHandle are valid. If a matching element was not found a message of type TBL_MSG is sent to the same address, if the option field is *TBL_REPLY*. The data field in this message is $NULL$, and the key field is key. No message is sent back if option is *TBL_NOREPLY*.

tablename.Find(int key, ReturnEP, ReturnHandle, int option) ;

The *Find* function is used to determine if the record with key key exists in the distributed table tablename. In some of the following cases message msg of type TBL_MSG is sent to the entry point ReturnEP at ReturnHandle. Again the message is sent only if ReturnEP and ReturnHandle are valid. The values of the key and data fields in msg are described below :

1. *a matching element exists and data is defined :* The key and data fields of msg are those of the matching element.
2. *a matching element exists but data is NULL :* The following actions are taken depending on the value of option :
   - If option = $TBL_WAIT_AFTER_FIRST$ or option = TBL_ALWAYS_WAIT, then the Find request is suspended till such time as the data is available, when the message, msg, is sent then with key and data fields as key and the available data, respectively.
   - If option = TBL_NEVER_WAIT then the key and data fields of msg are key and $NULL$, respectively.
3. *a matching element did not exist :* A table entry is created for the element with the key, key, and data as NULL. The following actions are taken depending on the value of option :
   - If option = TBL_ALWAYS_WAIT, then the Find request is suspended till such time as the data is available. The message, msg, is sent then with key and data fields as key and the available data, respectively.
   - If option = TBL_NEVER_WAIT or option = $TBL_WAIT_AFTER_FIRST$ then the key and data fields of msg are key and $NULL$, respectively.

In all the above calls if data is NULL, then the size of the data must be 0.

5 Other CHARM++ Features

5.1 Message Packing

In many programs the data structure to be passed in messages may be large and complex. As a simple concrete example, consider a dynamically created array being passed in a message. On a shared memory system, the message need only store a pointer to the base of the array and its size. However, on message passing systems (without a shared address space), a pointer is not valid across processors. So the whole array must be copied in each message. A chare may also want to send a dynamically created data-structure, such as a graph or a tree. Again, it must be copied (or "packed") into a contiguous structure without pointers before sending it in a message.

CHARM++ encourages a programming style that counters the unpredictability of available work by creating many small chares in the hope of being able to distribute them as needed. So in a message passing version of CHARM++, many chares that are created are not actually sent out to any other processor, but executed locally. When the system is saturated (i.e., all the processors have sufficient work), this happens to most chares. This behavior is desirable because it offers the flexibility of responding to load fluctuations as they arise.

In this context, packing every message in a format suitable for across-processor transmission seems quite wasteful. Moreover, such packing is clearly unnecessary and wasteful on shared memory machines. It would be preferable to pack only those messages that actually leave address-space boundaries, leaving other messages free to contain pointers. This poses two problems. First, the programmer doesn't know which one of the created chares (or messages) will end up going to another processor, as load distribution is dynamic and entirely the responsibility of CHARM++. Second, CHARM++ cannot pack an application specific message, since its structure is unknown.

The CHARM++ language provides a solution to this problem by postponing packing until it is absolutely necessary. Messages are allowed to contain pointers as in shared memory machines. However, if the runtime system decides to send a message to another processor, it calls an appropriate function in the *user* program to pack the message in a contiguous space and eliminate any explicit pointers. Conversely, before a message received from outside is scheduled for execution, the system calls another function to unpack the message.

The *pack* and *unpack* functions are associated with the message type. If the message is just a flat record, and does not need any packing, the user does not need to define any *pack* and *unpack* functions. However, if the message contains pointers and dynamically allocated data structures, the user must supply the associated *pack* and *unpack* functions.

Message Packing and varSize : When variable sized arrays, specified as varSize, are used in a message, the system automatically generates corresponding *pack* and *unpack* routines. Hence the varSize construct may not be used in a message type if the user defines the *pack* and *unpack* routines for that message type.

The structure of a message with *pack* and *unpack* methods is as follows :

```
message class MessageType {

    <Data member declarations>

    void *pack (int *length)
        <Pack-Function-Body>

    void unpack (void *in)
        <Unpack-Function-Body>
} ;
```

The *pack* routine takes one parameter: `length`, a pointer to the length in bytes of the packed message, which is set inside the *pack* routine. It returns a pointer to the packed message. Inside the *pack* routine the user allocates the packed message area of length `length` with the call *new_packbuffer()*. Then the user unravels all the pointers in the message, and initializes the contiguous packed message area. The *unpack* routine takes one parameter: `in`, the packed message. Inside the *unpack* routine the user re-creates the structure with pointers and sets the fields in the unpacked message.

It is important that the user free heap allocated structures in the unpacked (original) message in the *pack* routine, but the packed (original) message should not be freed in the *unpack* routine (the system after taking care of various necessary message parameters will free the original message). The *new_packbuffer* call is used to generate the contiguous message area in which the unpacked message can be packed. The syntax of the *new_packbuffer* call is as follows:

```
void * new_packbuffer(void *msg, int size)
```

The *new_packbuffer* call takes two parameters, msg, which is the unpacked message, and size, which is the size of the allocated pack buffer message. This is the only call that should be used to allocate packed messages – it is important because the call also transfers system relevant information from the unpacked message to the packed message.

5.2 Prioritized Execution

If one does not specify otherwise, Charm++ will process the messages you send in roughly FIFO order. For most programs, this behavior is fine. However, some programs need more explicit control over the order in which messages are processed. Charm++ allows you to control queueing behavior on a per-message basis.

The simplest call available to change the order in which messages are processed is `CSetQueueing`.

void CSetQueueing(message, queueingtype)
MSG_TYPE message;
int queueingtype;

where queueingtype is one of the following constants:

C_QUEUEING_FIFO
C_QUEUEING_LIFO
C_QUEUEING_IFIFO
C_QUEUEING_ILIFO
C_QUEUEING_BFIFO
C_QUEUEING_BLIFO

The first two options, `C_QUEUEING_FIFO` and `C_QUEUEING_LIFO`, are used as follows:

```
MsgType *msg1 = new MsgType ;
CSetQueueing(msg1, C_QUEUEING_FIFO);

MsgType *msg2 = new MsgType ;
CSetQueueing(msg2, C_QUEUEING_LIFO);
```

When message `msg1` arrives at its destination, it will pushed onto the end of the message queue, as usual. However, when `msg2` arrives, it will be pushed onto the *front* of the message queue.

The other four options involve the use of priorities. To attach a priority field to a message, one needs to set aside space in the message's buffer while allocating the message. To achieve this, the size of the priority field in bits should be specified as a placement argument to the `new` operator, as described in Section 3.4. Although the size of the priority field is specified in bits, it is always padded to an integral number of `int`s. A pointer to the priority part of the message buffer and its size can be obtained with these calls:

unsigned int *CPriorityPtr(msg)
MSG_TYPE msg;

unsigned int CPrioritySizeBits(msg)
MSG_TYPE msg;

unsigned int CPrioritySizeBytes(msg)
MSG_TYPE msg;

unsigned int CPrioritySizeWords(msg)
MSG_TYPE msg;

There are two kinds of priorities which can be attached to a message, these are termed "integer priorities" and "bitvector priorities".

Integer priorities are quite straightforward. One allocates a message, setting aside enough space in the message to hold the priority, which is an integer. One then stores the priority in the message. Finally, one informs the system that the message contains an integer priority using `CSetQueueing`:

```
MsgType *msg = new (C_INT_BITS) MsgType;
*CPriorityPtr(msg) = prio;
CSetQueueing(msg, C_QUEUEING_IFIFO);
```

The constant `C_INT_BITS` is defined to be the number of bits in an integer, usually `(sizeof(int)*8)`. (Remember, the size of the priority is specified in bits). The constant `C_QUEUEING_IFIFO` indicates that the message contains an integer priority, and that if there are other messages of the same priority, they should be sequenced in FIFO order (relative to each other). Similarly, a `C_QUEUEING_ILIFO` is available. Note that `MAXINT` is the lowest priority, and `NEGATIVE_MAXINT` is the highest priority.

Bitvector priorities are somewhat more complicated. Bitvector priorities are arbitrary-length bit-strings representing fixed-point numbers in the range 0 to 1. For example, the bit-string "001001" represents the number $.001001_{\text{binary}}$. As with the simpler kind of priority, higher numbers represent lower priorities. Unlike the simpler kind of priority, bitvectors can be of arbitrary length, therefore, the priority numbers they represent can be of arbitrary precision.

Arbitrary-precision priorities are often useful in AI search-tree applications. Suppose we have a heuristic suggesting that tree node N1 should be searched before tree node N2. We therefore designate that node N1 and its descendants will use high priorities, and that node N2 and its descendants will use lower priorities. We have effectively split the range of possible priorities in two. If several such heuristics fire in sequence, we can easily split the priority range in two enough times that no significant bits remain, and the search begins to fail for lack of meaningful priorities to assign. The solution is to use arbitrary-precision priorities, aka bitvector priorities.

To assign a bitvector priorities, two methods are available. The first is to obtain a pointer to the priority field using CPriorityPtr, and to then manually set the bits using the bit-setting operations inherent to C. To achieve this, one must know the format of the bitvector, which is as follows: the bitvector is represented as an array of unsigned integers. The most significant bit of the first integer contains the first bit of the bitvector. The remaining bits of the first integer contain the next 31 bits of the bitvector. Subsequent integers contain 32 bits each. If the size of the bitvector is not a multiple of 32, then the last integer contains 0 bits for padding in the least-significant bits of the integer.

The second way to assign priorities is only useful for those who are using the priority range-splitting described above. The root of the tree is assigned the null priority-string. Each child is assigned its parent's priority with some number of bits concatenated. The net effect is that the entire priority of a branch is within a small epsilon of the priority of its root.

To achieve this, the following call is available:

void CPriorityConcat(msg1, msg2, delta)
MSG_TYPE msg1;
MSG_TYPE msg2;
unsigned int delta;

Where `msg1` must be a message containing a valid priority, and `msg2` must be an allocated message buffer with at least as many priority bits as `msg1`. The priority bits from `msg1` are concatenated with enough bits from `delta` to fill the priority buffer in `msg2`. The extra bits are taken from the least-significant end of `delta`.

It is possible to utilize unprioritized messages, integer priorities, and bitvector priorities in the same program. The messages will be processed in roughly the following order:

- Among messages enqueued with bitvector priorities, the messages are dequeued according to their priority. The priority "0000..." is dequeued first, and "1111..." is dequeued last.
- Unprioritized messages are treated as if they had the priority "1000..." (which is the "middle" priority, it lies exactly halfway between "0000..." and "1111...").
- Integer priorities are converted to bitvector priorities. They are normalized so that the integer priority of zero is converted to "1000..." (the "middle" priority). To be more specific, the conversion is performed by adding 0x80000000 to the integer, and then treating the resulting 32-bit quantity as a 32-bit bitvector priority.
- Among messages with the same priority, messages are dequeued in FIFO order or LIFO order, depending upon which queuing strategy was used.

A final warning about prioritized execution: Charm++ always processes messages in *roughly* the order you specify, it never guarantees to deliver the messages in *precisely* the order you specify. However, it makes a serious attempt to be "close", so priorities can strongly affect the efficiency of your program.

A List of Keywords in Charm++

The following is the complete list of keywords in CHARM++ :

1. chare
2. message
3. entry
4. handle
5. group
6. thishandle
7. thisgroup
8. mainhandle
9. ALL
10. LOCAL
11. VARSIZE
12. accumulator
13. monotonic
14. readonly
15. writeonce
16. =>

The following is the complete list of system operators and calls in CHARM++. Currently no user defined functions may have one of these names.

1. newchare
2. newgroup
3. newaccumulator
4. newmonotonic
5. new_writeonce
6. new_packbuffer
7. CMyPe
8. CNumPes

9. CPrintf
10. CScanf
11. CTimer
12. CUTimer
13. CFunctionRefToName
14. CFunctionNameToRef
15. CSpanTreeRoot
16. CSpanTreeParent
17. CNumSpanTreeChildren
18. CSpanTreeChildren
19. CStartQuiescence
20. ChareExit
21. CharmExit

B Syntax changes from previous versions of Charm++

The syntax of some Charm++ constructs and calls is slightly different from previous versions : backward compatibility has not been maintained in many cases. This section gives the old and new syntax for each case.

Processor Identification:
Old : int myprocessornumber = CMyPeNum() ;
New : int myprocessornumber = CMyPe() ;

Number of processors:
Old : int totalnumberofprocessors = CMaxPeNum() ;
New : int totalnumberofprocessors = CNumPes() ;

Message Definition :
Old : message MessageType { ...fields... } ;
New : message class MessageType { ...fields... } ;

Message Allocation :
Old : MessageType *m = (MessageType *)new_message(MessageType) ;
New : MessageType *m = new MessageType ;

Message Deallocation :
Old : delete_message(messageptr) ;
New : delete messageptr ;

Chare Creation :
Old : new_chare(ChareType, &(ChareType::EntryFunction), messageptr) ;
New : newchare ChareType(messageptr) ;

Branched Chare Class Definition :
Old : branched chare class BChareType { ...fields... } ;
New : chare class BChareType : public groupmember { ...fields... } ;

Branched chare creation :
Old : new_branched_chare(BChareType, &(BChareType::EntryFunction), messageptr) ;
New : newgroup BChareType(messageptr) ;

Branched chare identifier type :
Old : BChareType handle h ;
New : BChareType group h ;

Obtaining branched chare identifiers :
Old : MyBChareType handle h = thishandle ;
Old : MyBChareType group h = thisgroup ;

Accumulator object creation :
Old : new_accumulator(AccType, messageptr) ;
New : newaccumulator AccType(messageptr) ;

Monotonic object creation :
Old : new_monotonic(MonoType, messageptr) ;
New : newmonotonic MonoType(messageptr) ;

C Calling Charm++ methods from C++ files

This section describes how to call functions inside Charm++ modules from C++ files.

The following header files from the Charm++ runtime system need to be included (in the same order) :
chare.h
globals.h
c++interface.h

Chare class declarations that need to be exported to C++ files should have the following modifications :

- chare class `ChareType` { ... } ; is exported to C++ as
  class `ChareType` : public `_CK_Object` { ... } ; where `_CK_Object` is defined in the run-time system.
- The access specifier entry must be changed to public.

A message class declaration message class `MsgType` { ... } ;
is exported to C++ as class `MsgType` : public `comm_object` { ... } ;

Of course, keywords and tokens specific to Charm++ cannot be used in C++ files.

Pointers to chares are NOT the same as chare-handles (a handle is a global pointer). In order to pass a pointer to a chare object to a C++ file, the programmer must obtain the pointer through this, from a method inside the chare class. This pointer can be passed to a C++ function in a C++ file, and used to call functions inside the chare object.

D Further Information

This section lists some important documentation and how it may be obtained as well as how to obtain the Charm/Charm++ system itself.

D.1 Related Publications

1. For starters, *Reports and Publications: Parallel Programming Laboratory* catalogs all of the major papers written about Charm/Charm++ and its associated tools.
2. An overview of some of the issues involved in parallel programming and the Charm approach to addressing those issues is discussed in *Parallel Programming with CHARM: An Overview.* Also discussed are some of the more recent projects that have been developed by the Charm group.
3. The report *Tutorial: Introduction to Charm* is also available with the distribution of Charm, which introduces Charm via a series of examples.
4. For a design philosophy of the language, see the report *The Design Philosophy of Chare Kernel Parallel Programming System*, University of Illinois Department of Computer Science Report No. UIUCDCS-R-89-1555, November 1989.
5. The *Dagger Language Tutorial* gives further information on Dagger, which can be used to express dependences among messages and computations.

D.2 Associated Tools

Some tools which may also be of interest follow. See the Charm distribution sites (Section D.3) for information on obtaining them.

Dagger is a coordination language which helps to express dependences among messages and computations.

DagTool is a visual frontend to Dagger.

Projections is a performance visualization and feedback tool. The system has a much more refined understanding of user computation than is possible in traditional tools. Thus it can provide much specific feedback. Future versions will be able to provide recommendations/suggestions for improving performance.

D.3 Contacts

The system is available (for research use and evaluation) by anonymous ftp, from a.cs.uiuc.edu, under the directory pub/CHARM. The files there include a manual, installation scripts, a README file that explains how to install the system on your machine, object files for supported machines, and many of the papers described above.

The above files are also available via the Charm Gopher server which runs on *dagger.cs.uiuc.edu.*

Further information about Charm/Charm++ is available via the World Wide Web server for the Charm group (accessible using **Mosaic**) at **http://charm.cs.uiuc.edu**.

If retrieval of a publication via these channels is not possible, please send electronic mail to **woodwort@cs.uiuc.edu** or postal mail to:

Joyce Woodworth
Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, IL 61801

While we can promise neither bug-free software nor immediate solutions to all problems, Charm++ is a stable system and it is our intention to keep it as up-to-date and useable as our resources will allow by responding quickly to questions and bug reports. To that end, there are mechanisms in place for contacting Charm users and developers.

Discussion about Charm/Charm++ occurs through the Charm news group, **uiuc.cs.charm**, and the Charm mailing list. To join the Charm mailing list, send an electronic mail message to **listserv@cs.uiuc.edu** which contains the single line **subscribe charm <full-name>** (where your name is used in place of <full-name>). After receiving acknowledgement of your subscription to the list, mail to **charm@cs.uiuc.edu** will reach all subscribers of the list.

Finally, send bug reports to **charmbugs@cs.uiuc.edu**.

Index

=> 9

Accumulate 15

accumulator 14

ALL 10

branched chare 5, 6

CFunctionNameToRef 13

CFunctionRefToName 13

chare 7
    branched 5, 6

ChareExit 11

CharmExit 11

C_INT_BITS 25

CMyPe 12

CNumPes 12

CNumSpanTreeChildren 13

CollectValue 15

CPrintf 12

CPriorityConcat() 26

CPriorityPtr() 24

CPrioritySizeBits() 24

CPrioritySizeBytes() 24

CPrioritySizeWords() 24

C_QUEUEING_BFIFO 24

C_QUEUEING_BLIFO 24

C_QUEUEING_FIFO 24

C_QUEUEING_IFIFO 24

C_QUEUEING_ILIFO 24

C_QUEUEING_LIFO 24

CScanf 12

CSetQueueing() 25

CSpanTreeChildren 13

CSpanTreeParent 13

CSpanTreeRoot 13

CStartQuiescence 7, 12

CTimer 12

CUTimer 12

Delete 20

DerefWriteOnce 19

distributed tables 19

entry function 5, 6

CSetQueueing() 24

Find 21

function
    entry 5, 6

handle 6
    virtual 8

Insert 20

LOCAL 7

main 14, 15, 17

mainhandle 6

message 5, 9, 11

monotonic 16

MonoValue 17

newaccumulator 15

newchare 7, 8

newmonotonic 17

new_packbuffer 23

new_writeonce 18

Once
    Write 18

pack 22, 23

readonly 14

ReturnEF-ReturnHandle 9

tables
    distributed 19

thishandle 6

unpack 22, 23

Update 17

VARSIZE 10

varSize 22

virtual handle 8

Write Once 18

writeonce 18