Interface Translator Charm++ (ITC++) Quick Reference Manual

Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

December 4, 1998

(Note: *1. Many facilties in ITC++ are implemented as C/C++ macros. Many parameters that these macros take are classnames or methodnames. These parameters are denoted in this manual as of type* `name`.
2. In functions/macros that send messages, the user relinquishes ownership of a message after sending it; further access to the message field can cause runtime errors.
3. Branch Office Chare is an old name for chare groups.
)

(macro)
void new_chare(name `charetype`**, name** `msgtype`**,** `msgtype *msgptr`**)**
Deposits the seed for a new chare of type `charetype` in a pool of seeds, and returns immediately. The chare will be created later on some processor, and initialized by calling its constructor entry function with the message parameter specified to `new_chare`.
`charetype` is the chare type of the newly created chare.
`msgtype` is the message type which is accepted as a parameter by the constructor for `charetype`.
`msgptr` is a pointer to object of type msgtype which will be passed to the constructor for the newly created chare.

(macro)
void new_chare2(name `charetype`**, name** `msgtype`**,** `msgtype *msgptr`**,**
`ChareIDType *vhndl`**,** `int destpe`**)**
Deposits the seed for a new chare in a pool of seeds. The function fills in a value for `vhndl` and returns immediately. The chare will be created later on a processor as specified by `destpe`, and initialized by calling its constructor entry function with the message parameter specified to `new_chare`. `charetype` is the chare type of the newly created chare.
`msgtype` is the message type which is accepted as a parameter by the constructor for `charetype`.
`msgptr` is pointer to object of type `msgtype` which will be passed to the constructor for the newly created chare.
`vhndl` is a pointer to the handle of the newly created chare, of type `ChareIDType`, which is filled by the `new_chare` function before it returns. This handle is useful for sending messages to the chare, even though it has not yet been created on any processor. If no virtual handle is required to be returned, the parameter can be `NULL_VID`.
`destpe` : When a chare is to be created at a specific processor, the `destpe`, is used to specify that processor. If no particular processor is required, the parameter can be `CK_PE_ANY`.

(macro)

int new_group(name `boctype`**, name** `msgtype`, `msgtype *msgptr`**)**

Used to create a branch of the branched chare type `boctype` on every processor and execute the constructor entry function in that branch with the message parameter specified to `new_group`. The `new_group` operator should be called from the main entry of the main chare and it returns the group id of the newly created branched chare. `boctype` is the branch office chare type of the newly created branch office chare.
`msgtype` is the message type which is accepted as a parameter by the constructor for `boctype`.
`msgptr` is pointer to object of type `msgtype` which will be passed to the constructor for the newly created branch office chares.

(macro)

void new_group2(name `boctype`**, name** `msgtype`, `msgtype *msgptr`,
`int returnef`, `ChareIDType *returnhandle`**)**

Used to create a branch of the branched chare type `boctype` on every processor and execute the constructor entry function in that branch with the message parameter specified to `new_group2`. This version is used when branched chares are dynamically created (from outside `main::main`). 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 `int group` as its first field. `boctype` is the branch office chare type of the newly created branch office chare.
`msgtype` is the message type which is accepted as a parameter by the constructor for `boctype`.
`msgptr` is pointer to object of type `msgtype` which will be passed to the constructor for the newly created branch office chares.
`returnef` is The entry function to which the group id is to be returned.
`returnhandle` is chare which contains the entry function `returnef`.

(macro)

int GetEntryPtr(name `charetype`**, name** `entryfunction`**, name** `msgtype`**)**

Returns a handle to the specified entry function within the specified chare.
`charetype` is the chare type which contains the entry function `entryfunction`.
`entryfunction` is the name of the entry function to which a handle has to be returned.
`msgtype`: `entryfunction` accepts a parameter which must be a pointer to this message type.

(macro)

void CSendMsg(name `charetype`**, name** `entryfunction`**, name** `msgtype`,
`msgtype *msgptr`, `ChareIDType *charehandle`**)**

Used to send a message to a specified entry function within a specified chare. This call is asynchronous and non-blocking; it returns immediately after sending the message.
`charetype` : the chare type to which the message is to be sent.
`entryfunction` : The entry function to which the message is to be sent.
`msgtype` : the message type which is to be sent.
`msgptr` : pointer to object of type `msgtype` which will be passed to `entryfunction`.
`charehandle` : The chare which contains the entry function `entryfunction`.

(macro)

void CSendMsgBranch(name `boctype`**, name** `entryfunction`**, name** `msgtype`**,**
`msgtype *msgptr, int bocid, int destpe`**)**

Used to send a message to an entry function within a particular branch(as specified by the processor number for that branch) of a branch office chare. This call is asynchronous and non-blocking; it returns immediately after sending the message.

`boctype` : the branch office chare type to which the message is to be sent.
`entryfunction` : The entry function to which the message is to be sent.
`msgtype` : the message type which is to be sent.
`msgptr` : pointer to object of type `msgtype` which will be passed to `entryfunction`.
`bocid` : The group id of the branched chare to which message is to be sent.
`destpe` : The processor number, to which the message is to be sent.

(macro)

void CBroadcastMsgBranch(name `boctype`**, name** `entryfunction`**,**
name `msgtype`**,** `msgtype *msgptr`**,** `int bocid`**)**

Used to send a message to an entry function within all the branches of a branch office chare. This call is asynchronous and non-blocking; it returns immediately after sending the message.

`boctype` : the branch office chare type to which the message is to be sent.
`entryfunction` : The entry function to which the message is to be sent.
`msgtype` : the message type which is to be sent.
`msgptr` : pointer to object of type msgtype which will be passed to `entryfunction`.
`bocid` : The group id of the branched chare to which message is to be sent.

(function)

void CRemoteCallFn(`int entryfunction`**,** `msgtype *msgptr`**,** `int destpe`**)**

This is used as a blocking call to an entry function in a remote chare. It causes the thread which made the call to sleep until the specified entry method has been executed on the remote chare.

`entryfunction` : The entry function which is to be called remotely.
`msgptr` : pointer to object of type `msgtype` which is accepted by the `entryfunction`.
`destpe` : The processor number, on which the remote function will be called.

(function)

void CRemoteCallBranchFn(`int entryfunction`**,** `msgtype *msgptr`**,**
`int bocid`**,** `int destpe`**)**

This is used as a blocking call to an entry function in a remote branch office chare. It causes the thread which made the call to sleep until the specified entry method has been executed on the remote branch office chare. `entryfunction` : The entry function which is to be called remotely.
`msgptr` : pointer to object of type msgtype which is accepted by the `entryfunction`.
`destpe` : The processor number, on which the remote function will be called.

(function)

void ChareExit(void)

The ChareExit call is used 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.

(function)
void CharmExit(void)
Similar to the regular Unix exit, except that this routine will kill all running processes non-preemptively 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).

(function)
void CStartQuiescence (`int entryfunction`, `ChareIDType charehandle`)
When quiescence occurs, this call will make the system send a message to the specified entry function within the specified chare. `entryfunction` : The entry function to which the message is to be sent. `charehandle` : The chare which contains the entry function `entryfunction`

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

(function)
int CMyPe(void)
Returns the processor number on which the call was made.

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

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

(function)
void *CAlloc(`int size`)
This is similar to the C malloc function. It needs to be used explicitly in Charm++ programs only when allocating space for message data that is to be freed by the system(i.e. when sending messages to other non-Charm++ modules).

(function)
int CTimer(void)
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).

(function)
int CUTimer(void)
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).

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

(function)
int CSpanTreeParent(`int pe`**)**
Returns the processor number of the parent(in the spanning tree) of the processor specified by `pe`.

(function)
int CNumSpanTreeChildren(`int pe`**)**
This returns the number of spanning tree children of the processor specified by `pe`.

(function)
void CSpanTreeChildren(`int pe, int *children`**)**
This is used to determine the children in the spanning tree for the processor specified by `pe`. The processor numbers for the children are returned in the integer array `children` which must be of sufficient size and allocated by the user.

(function)
void *pack(`int *length`**)**
If a message contains pointers and dynamically allocated data structures, the user must supply the associated pack and unpack functions. The pack function takes a 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. It is important that the user free heap allocated structures in the unpacked (original) message inside this function.

(function)
void unpack(`void *inmessage`**)**
If a message contains pointers and dynamically allocated data structures, the user must supply the associated pack and unpack functions. The unpack function takes a single parameter `inmessage`, which is a pointer to the message received. The packed message, `inmessage`, should not be freed in this function (the system after taking care of various necessary message parameters will free the original message).

(function)
void *new_packbuffer(`void *msgptr, int size`**)**
Inside the user-supplied pack function, this call is used to generate the contiguous message area in which the unpacked message can be packed. It takes two parameters; `msgptr`, a pointer to the message which is being packed, and `size`, the size of the message after it is packed.

(function)
void CSetQueueing(`msgtype *msgptr, int queueingtype`**)**
Used to specify the order in which messages are to be processed. `msgptr` : pointer to the message whose processing order is to be specified. `queueingtype` : This can take on one of the following predefined integer constant values : `C_QUEUEING_FIFO` `C_QUEUEING_LIFO` `C_QUEUEING_IFIFO` `C_QUEUEING_ILIFO` `C_QUEUEING_BFIFO` `C_QUEUEING_BLIFO`. The last four options involve the use of

priorities, and a priority field must be attached to the message's buffer while allocating it. `MAXINT` is the lowest priority, and `NEGATIVE_MAXINT` is the highest priority.

(function)
unsigned int *CPriorityPtr(`msgtype *msgptr`)
Returns a pointer to the priority part of the message buffer pointed to by `msgptr`.

(function)
unsigned int CPrioritySizeBits(`msgtype *msgptr`)
Returns the size in bits of the priority part of the message buffer pointed to by `msgptr`.

(function)
unsigned int CPrioritySizeBytes(`msgtype *msgptr`)
Returns the size in bytes of the priority part of the message buffer pointed to by `msgptr`.

(function)
unsigned int CPrioritySizeWords(`msgtype *msgptr`)
Returns the size in words of the priority part of the message buffer pointed to by `msgptr`.

(function)
void CPriorityConcat(`msgtype *msgptr`, `msgtype *msgptr`, `unsigned int delta`)
This function concatenates the priority bits from the message pointed to by `msgptr`(which must have a valid priority) with enough bits from `delta` to fill the priority buffer in the message pointed to by `msgptr`(note that this priority buffer must be allocated for at least as many priority bits as in the message pointed to by `msgptr`). The extra bits are taken from the least significant end of `delta`.

(function)
`msgtype` ***new(MsgIndex(name** `msgtype`**),** `int priosize`) `msgtype`
This operator must be used to allocate the space pointed to by any message pointer variable which has been declared as readonly. `msgtype` specifies the type of the message for which space is being allocated, and `priosize` is an optional argument. of type int, used to specify the size of the priority field if any.

(macro)
void ReadInit(name `var`**)**
This macro is used to initialize the value of any readonly variable `var`.

(macro)
`type` **ReadValue(name** `var`**)**
This macro is used to access the value of any readonly variable `var` of type `type`.

keyword
varsize
This is a keyword used to declare messages having arrays whose sizes may be known only at run time when the message is allocated. When allocating space for such messages, the new operator must be passed an extra parameter which is a pointer to an array of integers. The entries filled in this array should specify the sizes of the variable-sized arrays within the message.

(member)
ChareIDType thishandle
This is a public member of every chare, of type `ChareIDType`, inherited from the system-defined

superclass for chares. It holds the handle of the chare whose entry function or public function is currently executing.

(member)
ChareIDType mainhandle
This is a pre-defined variable, of type `ChareIDType`, which holds the handle of the single instance of the main chare.

(member)
int thisgroup
This is a public local variable, of type int, inherited from the system-defined groupmember superclass. It holds the globally unique group id of the branch office chare to which the currently executing branched chare belongs.

(type)
chare_object
This is the system-defined superclass from which all chare classes defined by the user must inherit.

(type)
comm_object
This is the system-defined superclass from which all messages defined by the user must inherit.

(type)
groupmember
This is the system-defined superclass from which all branch office chares defined by the user must inherit.

(function)
int MsgIndex(name `msgtype`)
This macro is used when allocating space for a message object. Its usage is as shown below. The second form shown is used when a priority field has to be attached to the message. The size of this priority field is specified as the optional argument `priosize`, which is of type int.
msgtype *msgptr = new (MsgIndex (msgtype)) msgtype;
msgtype *msgptr = new (MsgIndex(msgtype), priosize) msgtype;

(function)
`boctype` ***CLocalBranch(name `boctype`,`int bocid`)**
This macro is used by sequential objects, normal chares or other branched chares to access public members of the branch of a branched chare on their processor. Its usage is as follows :
CLocalBranch(boctype, bocid)-¿datamember
CLocalBranch(boctype, bocid)-¿functionmember()
`boctype` : the branch office chare type of the branched chare whose members (`datamember` or `functionmember`) are to be accessed. `bocid` : the unique group id of the branched chare whose members are to be accessed.

keyword
chare
Keyword in the interface file denoting a chare class.

keyword

group
Keyword in the interface file denoting a chare group.

keyword
message
Keyword in the interface file denoting a message class.

keyword
packmessage
Keyword in the interface file denoting a user-packed message.

keyword
readonly
Keyword in the interface file denoting a readonly variable.

keyword
threaded
Keyword in the interface file denoting an entry method which is executed in a separate user-level thread.