BCR Principles and User Guide

BCR is "Before CDS Redesign", or (C-1)(D-1)(S-1)
CDS is Cooperative Data Sharing

Index

Revision 1.0, describes BCR v0.7

Important License Terms: Warning, Agreement, and Disclaimer

This software and documentation is currently an experimental product, without any guarantee, warrantee, or source code, and license to use it is provided to you in accordance with the following terms, and with the expectation that significant bugs may still exist. Even if such bugs do not exist, the user should not infer or assume that it is well-suited to any particular purpose: Determination of such is entirely the user's responsibility. Users should consider themselves early adopters, and as such expect greater risks than someone using or testing a mature product. For example, this software runs its own background processes ("daemons") which open their own sockets (ports), initiate processes on remote machines, and use low-level UDP/IP communication which may not interact appropriately, or share resources fairly, with other software using the same networks (e.g. based on TCP/IP). Programs will often intentionally execute tight loops while waiting for events, thus potentially consuming significant CPU time while not performing useful work, and programs may not correctly terminate in some cases, thereby potentially consuming resources even when the user does not intend it to do so. At the very least, then, this software should probably be used only on known-insecure systems or "behind the firewall", and on computers and networks where risks from unforeseen circumstances (potentially including failure and/or corruption) are considered acceptable. While the creators have made reasonable attempts to minimize certain modes of failure, there is no guarantee that they have been successful, or that those particular modes are applicable to you. Your use of this software implies agreement to this license, your understanding and acceptance of these risks, and your agreement that Elepar will not be held liable for any loss or damages resulting from its use.

Your use of this software also implies that you will not attempt to determine internal operation of the compiled or executable code, through reverse engineering, disassembly, or other means, or to share that information with others if it is so determined, or to work around or otherwise thwart any protections or restrictions built into the software, e.g. those intended to keep the user from performing certain operations before he/she agrees to and/or purchases additional licenses.

Introduction

Cooperative Data Sharing (CDS) is an approach to programming computers, and most specifically collections (or teams) of computers. That is, it allows one program to be broken into pieces and spread among many different computers, and for those pieces to then communicate with one another to perform a task. There are other approaches to this, which go by names like "message passing" and "shared memory", but each of these approaches is optimized for specific kinds of computer architectures, networks, and sometimes applications. As a result, programmers using those approaches must often decide, even while designing a program, whether that program will execute on a single processor, a cluster of PCs, a high-performance parallel machine, a "computational grid", a geographically distributed "peer-to-peer" network, or some other platform. If the program moves to some other place using a different platform, or if the program doesn't move but the platform for which it was programed is phased out, a great deal of effort may be lost.

CDS is a more portable approach, created after carefully studying the advantages and disadvantages of these other approaches in varying situations. Although CDS plays the same sort of role as those other approaches, it in some sense transcends them and encompasses them. One way of considering CDS is as a single way of programming that can automatically switch between shared memory and message passing, depending on what is best at the time—i.e. based both on what the communicating entities need to do to the data and on how those entities are related on the hardware (e.g. on the same processor, on different processors that share memory, or on processors that must communicate over high- or low-latency networks). When you program in CDS, your program can look like message passing or like shared memory or both, but optimal flexibility and efficiency can often be obtained by making each part of your program use the style that works best *for that part*. This leads to a CDS-unique style which will be further described on these pages.

The specific programming interface described here is called BCR, which is a CDS approach, but is slightly simplified from the full CDS programming interface (at `www.elepar.com/CDS/`). Although not equal, BCS is *very* similar to an earlier version of CDS, called CDS1 (which was implemented at NASA Ames Research Center by the founder of Elepar), so "Before CDS Redesign" (BCR) is a fitting name. (Each letter in BCR also happens to be alphabetically just before the corresponding letter in CDS.) Because of BCR's similarity to true CDS, it serves as a good way to learn and experience CDS, and unless clearly stated otherwise, any general comments here that refer to CDS will also apply to BCR.

Background: Shared Memory and Message Passing

Terms like "shared memory" and "message passing" will be used liberally throughout this document. So that all readers can start from the same point, here is a brief description of these terms, which can safely be skipped by those already familiar with them.

Message passing is used to communicate between independent processing entities ("processes") which are assumed to share access to some sort of bidirectional communication channel (e.g. network, or in some instances, memory) between them. To move data over the channel, one entity (the "sender") states the desire to push a certain amount of data, currently residing in one or more particular memory areas, to a particular other entity, while that other entity (the "receiver") states the desire to take what comes in over the channel and store it in one or more particular memory areas. The data moving across the channel is called a message. If the sender initiates the push but the receiver is not waiting for the message, the message is usually queued (or "buffered") automatically and silently on the processor of the receiving entity until it is needed, but this will depend on the particulars of the message passing system being used, and sometimes on the particular options specified by the sender and/or receiver. If the receiver states its intent before the sender does, the receiver often just waits (or "blocks"), but this can also differ depending on the situation. To help ensure that the senders and receivers don't get mixed up and start crossing messages with other entities (e.g. on the same processors), it is common for each message to carry an additional name (or "tag" and "context") which both the sender and receiver specify, and which must match for the transfer to occur.

Shared memory is used to communicate between processing entities that can access common data—i.e. where the data stored in some or all of their address space is shared. (These entities may still be referred to as "processes", but are often called "threads", especially if they share all of their address space.) This approach is often restricted to hardware that has a very efficient (low-latency) connection between processing entities. Communication occurs whenever one processing entity modifies some of this shared memory, and one or more other entities read (access) it. In addition to actually writing and reading, the communicating entities must cooperate on when each is reading and/or writing, and to which area of memory. (Consider two people communicating in a shared room: Having both speak simultanously, and then listen simultaneously, doesn't work, they must agree on who is speaking when.) To accomplish this, the entities usually divide the memory up into regions, and associate each region with another (very small) region called a *lock*. The entities agree not to access the larger region unless they first acquire the lock, which means waiting until it set to some predetermined "unlocked" value, and then changing its value to "locked". Special locks are sometimes employed that allow multiple entities to read the memory at the same time, but if there are any readers or writers, stops any subsequent writers from acquiring the lock. (Newer hardware may also support "wait-free locking" in some cases, e.g. by not locking memory, but just sensing whether it was accessed by another entity when you didn't want it to be. This document will not refer to that approach, as its applicability is limited.)

Motivation Behind CDS

Message passing has significant advantages when the communicating entities are separated by a high-latency channel (i.e. it takes relatively long to move data between them), because

- the programmer can explicitly dictate when data should move across that channel
- the latency can be "hidden", in many cases, by moving the data to its destination even while the sender and/or receiver are doing other work
- the data moves in large chunks, so it can be "pipelined" (or transmitted in an assembly-line fashion) to minimize the communication overhead for each individual piece
- since data can be buffered on the receiving end, it is often unnecessary to get the sender and receiver to synchronize exactly (which can be difficult in that environment)

In a low-latency environment, shared memory is often better, because:

- the programmer isn't bothered with managing data movement
- there is no overhead associated with copying the data from place to place—each entity can just access the data where it sits
- in "demand-driven" cases, where the next entity that will need some particular data is not known until that entity needs the data, it can be much easier to program

The amount of latency that is tolerable for a shared memory programming model can vary depending upon the specific requirements or on the current state-of-the-art. For example, as processors get faster, any delay has a relatively greater effect. For this reason, even shared-memory systems have begun to try to integrate some features of message passing between computing entities and/or caches. At its most extreme (when latencies are expected to be pretty high), this approach is called distributed shared memory (DSM). These features may include:

- allowing the programmer or runtime system to specify a "home" entity for certain memory addresses, designating where they will be accessed most often, so if they are not specifically being used elsewhere, they will tends to move toward their home (possibly hiding latency)
- allowing the programmer to explicitly prefetch memory to an entitiy before it is actually used, again to hide latency
- transfering not just the data that is used, but also the data ("page") around it, to pipeline the transfer/overhead. (This can lead to a problem called "false sharing" when some of that other data is needed on another processor.)

If these approaches to accomodating high-latency while using a shared memory interface worked well, one could write a program using a single programming model, without first determining whether it would be executed in a high-latency or low-latency environment, or some combination (e.g. like clusters of SMPs). Alas, though adding these features to shared memory have been demonstrated to be somewhat effective in some cases, it is rarely as efficient as using message passing in high-latency environments, because the hybrid approach often relies on the runtime system to guess what (and how much) data will be needed where when.

CDS takes a different approach. Instead of starting with either message passing or shared memory, its design was motivated by looking at the advantages of each in different environments (like those above) and working toward a common interface with all of those advantages. The result is a small set of more general primitive communication operations which, when used in certain ways or combinations, can be made to look like either message passing or shared memory, but which in themselves are more flexible than either.

There is an advantage (for both CDS and distributed shared memory) which extends beyond simple latency issues. Environments differ not only in the way processors are connected, but in how many processors there are. If computing entities running on the same processor can exchange data with very little overhead, then they can be viewed very much like subroutines or objects in a single entity, rather than as separate entities. In other words, the number of entities can effectively dynamically change (i.e. decrease) to accomodate the number of different processors, so a program is not so dependent upon the number of processors.

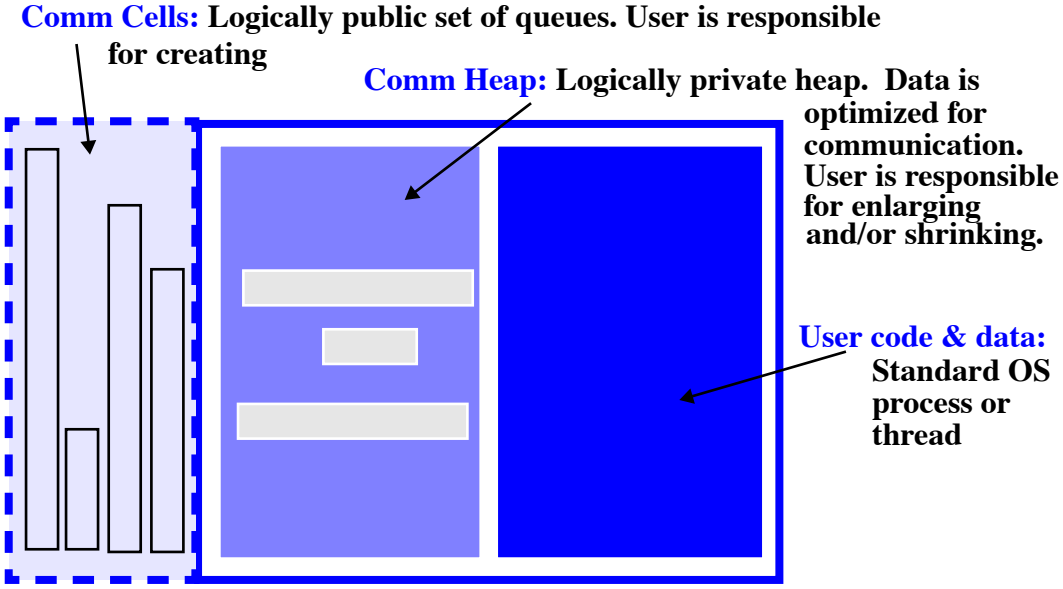
Features	CDS	DSM	MPI	SOCK	LINDA
Some data can be traded/shared in place (true 0 copy!)	**x**	**x**			
Consumer can pull (get) data from passive producer	**x**	**x**	**2**		**x**
Consumer can prefetch/prepull data to hide latency	**x**	**?**	**2**		
Producer can push (send) data to passive consumer	**x**		**x**	**x**	**?**
Data can be queued at producer waiting for pull	**x**		**x**	**x**	**?**
Pushed data can be made to overwrite previous value	**x**	**x**			**x**
Producer can retain access rights to communicated data	**x**		**2**		**x**
Producer can relinq access rights to communicated data	**x**	**x**	**x**		**x**
Dynamic memory allocation for shared memory	**x**	**?**			
Consumer can specify timeout for waiting	**x**	**?**			
Supports heterogeneous platforms	**x**		**x**		
Simplicity (~number of function + macro interfaces)	51	20	!!!	13	5

In the best of all worlds, programmers would not even be exposed to low-level communication approaches like shared memory, message passing, or CDS, instead always using higher-level languages and tools, but (a) a small step is better than no step, and (b) CDS can just as well be considered as a good way to build high-level programming tools, instead of providing a programmer interface.

The CDS Approach

CDS accomplishes its seemingly-conflicting goals in kind of a tricky way, which may not be evident until you hear the whole story. So, the best way to learn CDS is to just open your mind for a bit and forget about shared memory or message passing, or even efficiency. Those kinds of issues will be addressed in the next section, after explaining the basics of CDS.

Each CDS computing entity is similar in some ways to a thread, sharing information with other entities, and in some ways similar to a process, being independent, so here it will just be called a *CCE* (for "CDS computing entity"). A CCE declares itself as a CCE (allowing it to call CDS routines) by calling a special CDS **`init`** routine, but this will block until or unless the CCE has been *enlisted*. It can be enlisted by another CCE (calling the **`enlist`** routine), or by the user directly (using the **`enl`** program). If **`enlist`** cannot find an appropriate CCE blocked at **`init`**, it will try to create one, so enlisting a CCE will often serve both to create it, and then to unblock the **`init`** when it is executed. Each CCE is identified with an integer CCE ID, which is similar to a file descriptor in that the specific integer used to identify a CCE may depend not only on the CCE being identified, but on the CCE holding that identification. After a CCE has called **`init`**, it can find its own ID, and the ID of the CCE which enlisted it (both in global variables).

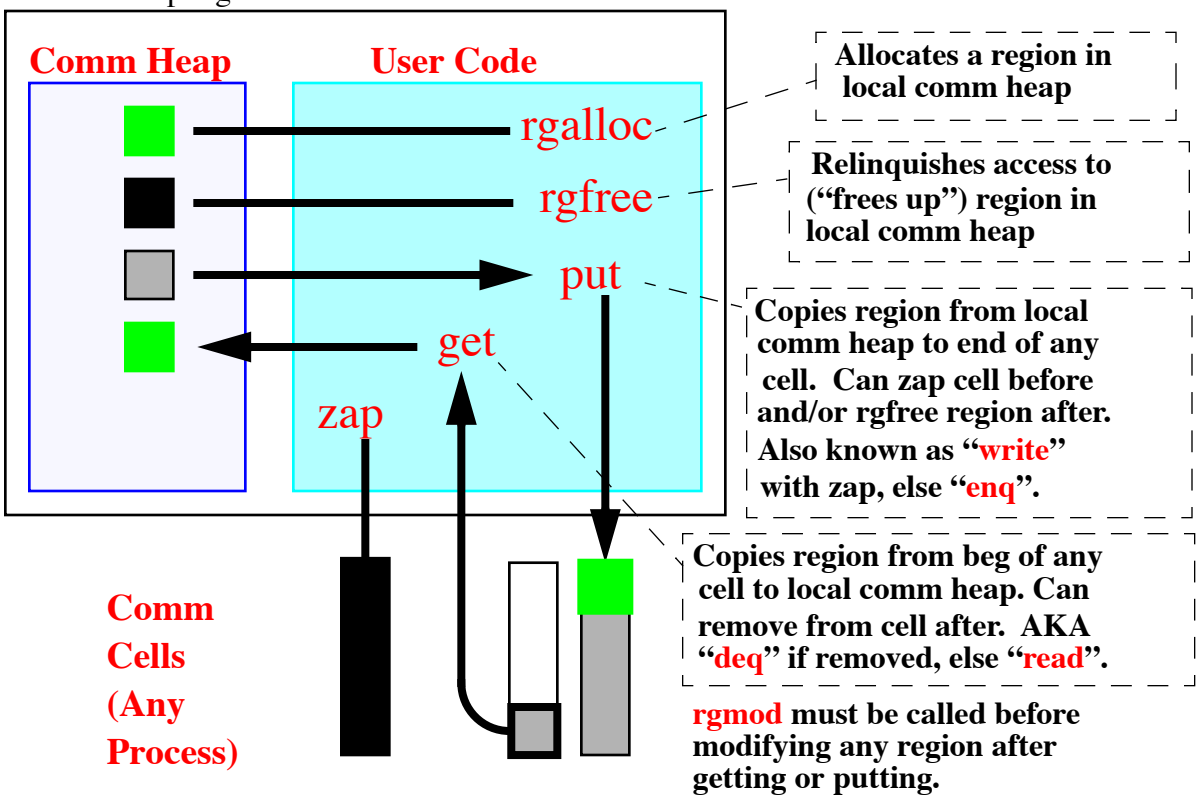


Each CCE possesses a logically-private *communication* (or *comm*) *heap*, which is just a special heap of memory that CDS knows about. A CCE allocates a memory region from this heap by using the **`rgalloc`** operation, which is analogous to the standard C **`malloc`** routine (e.g. it takes a length in bytes), but instead of returning the region's address like **`malloc`**, **`rgalloc`** returns a CDS *region ID*, which can be thought of (by the user) as a pointer to the region's address—i.e. the data can be accessed by using "`**rgid`" instead of just "`*rgptr`" (as one might use with traditional **`malloc`**). There is also an **`rgfree`** operation in CDS that is analogous to **`free`** in standard C, that effectively declares that the CCE will no longer access that region, as well as an **`rglen`** operation to determine the length of an already-allocated region, and an **`rgrealloc`** operation to try to change a region's length without moving it.

Along with a comm heap, each CCE can also have any number of *comm cells* (usually just referred to as "cells") which are visible to other CCEs. (Both the comm heap and the comm cells

are created using the **magrow** routine.) Each cell has an integer name (unique within that CCE), so any cell can be addressed by any CCE by naming the CCE that holds it and the cell number within that CCE. Each cell can hold any number of regions, in a queue-like fashion. There are four basic communication primitives that operate on cells:

1. **zap**, which takes a cell address (i.e. CCE name plus integer cell name), and deletes any regions which might be sitting in it.
2. **put**, which takes a region ID and a cell address, and produces a copy of that region into the cell at the end, optionally **zap**ping the cell first, and optionally **rgfree**ing the region after.
3. **putm**, which is the same as **put**, but takes a list of cell addresses and produces a copy into each of them.
4. **get**, which takes a cell address, and produces a copy of the first region from the cell into caller's comm heap (and optionally removes it from the cell), returning a region ID to the new comm-heap region.



So, that's how CCE's in CDS communicate: They create regions in their comm heaps using **rgalloc**, then shuffle and/or copy those regions between their comm heaps through cells using **put** and **get**, and eventually relinquish them using **rgfree**. There is one more basic rule covering communication: If you are going to change data in a region that you obtained with a **get**, or which you have already **put** somewhere since you called **rgalloc** (or **rgmod**), then you need to call **rgmod** (with the region ID) first. You don't need to tell it what you intend to change, just that you intend to change something in that region. (**rgalloc** may update the region pointer stored in the region ID, i.e. as `*rgid`, so care must be taken not to save an old value of this pointer into a temp variable across a call to **rgalloc**.)

Sometimes, the data that you need to move from one place to another will be sitting (or need to sit) somewhere in the CCE's program memory instead of in the comm heap. It is usually possible

to just copy the data to/from the comm heap in traditional ways (e.g. using the pointer in the region ID, and perhaps **memcpy**), but CDS provides three more flexible routines to help; one to copy data to a region from other memory (called **copyto**), one to copy from a region to other memory (called **copyfm**), and one to copy data between regions (called **copytofm**). All of these routines take a set of instructions called a *copy descriptor* (or sometimes likened to a "type"), expressed as a sequential array of integers, that allows data of different types and relative positions to be copied and packed or unpacked. Another routine, **copytosz**, tells how much space a **copyto** will use in the destination. For those somewhat-common cases when one needs to **rgalloc** a region (after first determining the size needed with **copytosz**), **copyto** data to it, and then **put** it into a cell, CDS offers a single routine called **send** which does all of these. Similarly, there is a **recv** routine that does the same as a **get**, a **copyfm**, and an **rgfree**.

The copy routines are required when moving CCE IDs to/from the comm heap, because CCE IDs are not stored as integers within regions. (They are actually much larger, e.g. 4 words.) The copy routines are also extremely helpful when translating data between heterogeneous processors. BCR facilitates this translation by endowing each region with an integer code (called an *archtype*) that tells how that data in the region is represented—i.e. the kinds of processors that represent their data that way. This code gets passed along with the region from one CCE to the next. If the CCE is running on a processor type that matches the archtype in a region, then it can safely access it directly (using the `**rgid` approach). Otherwise, the data should probably only be accessed using the copy routines, which always convert appropriately based on the archtype. The archtype for a region is set when it is **rgalloc**'d—i.e. the archtype defaults to the same as the processor creating the region, though this can be overridden by a second argument to **rgalloc**. The archtype for a particular region can be queried with the **rglen** routine. The archtype for the current CCE is available in a global variable, **archtype**, but the archtype corresponding to any CCE ID can also be determined with the **arch** routine (e.g. to be supplied to **rgalloc**).

Finally, each cell can be endowed with a handler (i.e. user-supplied subroutine) using the **handler** routine, to be invoked ("upcalled") whenever the number of cells exceeds a specified high- or low-water mark. Even if it is invoked, a handler will only run if the currently executing coroutine (i.e. handler or main CCE) calls a CDS routine, and has a lower priority than the handler. The priority of a coroutine can be set using the **priority** routine, and can be temporarily adjusted down using the **allow** routine.

How It *Really* Works

The previous section conveys very few advantages of CDS's advantages over shared memory in a low-latency environment, especially with all of that copying. In fact, CDS appears to require even more copying (and therefore more overhead) than message passing. The trick is that the above is a *logical* description, telling you how you can think about CDS when programming. The *physical* model underneath is (or at least, can be) quite different. Some people will find the physical model confusing, because all of the special cases and tricks can be complex, so always remember that you can rely on the simple rules in the logical model.

Four efficiency tricks rely on these differences between the physical and logical models:

1. In the logical model, each comm heap is private to its CCE. In the physical model, each comm heap is stored in shared memory, where it can be accessed directly from any CCE that the hardware allows (all such CCEs will be called a "CCE group" here), or from a special daemon process that helps access that memory for CCE's outside of the group.

2. The comm cells don't really hold regions that have been copied from the comm heap, they hold pointers to regions that are still sitting in the comm heap.

3. Related to 2, data is rarely copied from place to place. Instead of copying a region to or from a cell belonging to the same CCE group, a pointer to the region is just moved to or from the cell. Reference counts are kept with each region to keep track of how many pointers point to it. Data is only actually copied in three cases: when accessing a cell in a different group (so the data needs to be moved); when modifying (i.e. calling **`rgmod`** for) a region targeted by other pointers (as indicated by the internal reference count), and when the copy routines are called explicitly. (The second reason is sometimes called "copy on write".)

4. The **`send`** and **`recv`** operations can be heavily optimized. For example, they don't really need to go through the comm heap at all in some cases, because the region that it says it uses is never visible to the caller, anyway. The data can be moved directly between the communication channel and the CCE's memory addresses.

As a result of these, CDS has many similarities to shared memory. Different processing entities (like threads) communicate through some common memory area that they can all access, and are careful when accessing that area to ensure that they cooperate properly, and do not interfere with each others' intentions. The shared area is partitioned into regions which are managed separately—i.e. a computing entity obtains access to an entire region at one time, uses it as desired, and relinquishes it. Multiple entities can read the same region at the same time.

And, CDS also has similarities to message passing. A computing entity (the sender) specifies some data to move and the computing entity (the receiver) where it should moved to (because it will likely be used there). The sender and receiver use a common tag to ensure that they are both working on the same data. The data can be queued on the receiver until it is needed.

BCR Application Interface

The BCR C-callable interface is defined by 38 routines and/or macros, 4 global variables (initialized by **bcr_init**), and several predefined constants. Some routines may set **bcr_errno**, documented in error.h. **Unless otherwise stated, the prefix bcr_ has been omitted in each case.**

CCE CONTROL & COMM AREA MGMT
enlist init cagrow cafree archtype

REGION OPERATIONS
rgalloc rgmod rgfree rgrealloc rglen rgwait rgwaitm

COMMUNICATION OPERATIONS
put get zap putm

COMMUNICATION MACROS
write enq writem enqm read deq

SHARED MEMORY OPS/MACROS
acqrl rlsrl acqwl rlswl wl2rl

COPYING /TRANSLATION OPS
copyfm copytofm copyto copytosz

MESSAGE PASSING COMPOSITE OPS
send recv sendm

HANDLER OPERATIONS
handler block allow priority

GLOBAL VARIABLES
cce enlistor cceord archtype

#DEFINED IN BCR.H
BLOCK PENDING FREE NOFREE (plus init flags, next column)

#DEFINED IN TYPES.H **(prefixed bcrT_, or b crT_1 for descriptor)**
CCE CHAR SHORT INT LONG LONGLONG FLOAT DOUBLE SKIP_FM SKIP_TO END NEST

CCE Control & Comm Area Mgmt

```
int bcr_init(int flags,
             char *ccename)
```

Declares the calling CCE to be a BCR CCE, and assigns it a name (currently ignored, but best equal to file name). Must be called before any other BCR routine. This routine will block until/unless the CCE is enlisted. flags is obtained by "OR-ing" together the BCR features required, from the following table:

TABLE 1. BCR Features

Name (in BCR.h)	Meaning
bcr_inorder	Make same-source regions arrive in order in dest cell
bcr_user_ca	Put comm heap in user space
bcr_errors	Send error regions on failure
bcr_reliable	Guarantee delivery
bcr_timeout	Allow positive timeout values on get
bcr_handlers	Allow user handlers (via "handler" routine)
bcr_gc	Use garbage collection when comm heap full
bcr_implemented	All of above implemented for this machine

Global variables are initiated by this call as follows: **bcr_enlistor** is set to the CCE ID of the enlisting CCE; **bcr_cce** is set to the new ID of the calling CCE; **bcr_archtype** is set to the archtype of this CCE; and **bcr_cceord** is set to the ordinal of the calling CCE (i.e. resulting from the cceord1 argument to **bcr_enlist**). The function returns the flags which were requested but were not available in this implementation (i.e. were not present in bcr_implemented).

If the bcr_user_ca flag is specified, any region ID rgid can be cast to type "rgid_t**" (where rgid_t is the type of the region) and used to efficiently and directly access the data within that region—e.g. using the expression **((rgid_t**)rgid). Since

bcr_rgmod, and other operations that implicity call it (i.e. **bcr_copyto**, and **bcr_copytofm**), may alter the value of `*((rgid_t**)rgid)`, care should be taken not to rely on this value to remain constant across calls to these functions.

```
int bcr_enlist(char *mach,
               int prcssr,
               int cceord1,
               char *obj,
               void **rgid,
               int nofree)
```

Enlists a new BCR CCE on machine `mach`, processor `prcssr`. If `prcssr` is negative, its absolute value represents the number of CCEs to be enlisted, on processors to be chosen by BCR. The first CCE is given a CCE ordinal (see **bcr_cceord**) of `cceord1`, and others are numbered sequentially after. `obj` is a slash-separated path specifying how to locate and/or invoke the CCE, and currently just represents the executable file on the specified machine containing the CCE. If `rgid` is non-zero, it is taken to be the ID for a region which is no longer than 64 bytes, and that region is put in cell 0 of the new CCE before its **bcr_init** completes. If `nofree` is zero, an implicit **bcr_rgfree** is performed on `rgid`.

```
int bcr_cagrow(int qbase,
               int nprivqs,
               int ninqs,
               int noutqs,
               int nioqs,
               int nrgns,
               int nbytes)
```

The size of the comm heap is increased by `nbytes` bytes. The number of comm cells in this CCE is increased by the sum of `nprivqs`, `ninqs`, `noutqs`, and `nioqs`. (In some future implementation, these may refer to the number of cells that are private, enq only, deq only, and public to other CCEs, respectively.) These comm cells are capable of holding a total of `nrgns` regions at any one time. If possible, the first comm cell is assigned cell ID `qbase`. `bcr_cagrow` returns the cell ID actually assigned to the first comm cell.

```
int bcr_cafree(int qbase)
```

The comm heap grow request which returned cell ID `qbase` is effectively revoked. The cell IDs (starting with `qbase`) and communication heap space allocated by that request may be made available for reuse by future comm heap grow requests.

```
int bcr_arch(int cceid)
```

Returns the architecture type for CCE cceid. This is used primarily to compare against the current architecture (in **bcr_archtype**) and/or provide as a second argument to **bcr_rgalloc**.

Region Operations

```
void **bcr_rgalloc(int len,
                   int archtype)
```

A region`len` bytes long, with architecture type `archtype`, is allocated within the comm heap, and a region ID for that region is returned. If `archtype` is 0, **bcr_archtype** is used.

```
int bcr_rgmod(void **rgid)
```

Informs BCR that the contents of region `rgid` will (might) be modified by the user. This routine may modify the location of the region (i.e. `*rgid`).

```
int bcr_rgfree(void **rgid)
```

Frees region `rgid`.

```
int bcr_rgrealloc(void **rgid,
                  int newlen)
```

Attempts to change the size of region `rgid` to `newlen` without performing a copy. Returns zero if successful.

```
int bcr_rglen(void **rgid,
              int *archtype)
```

Returns the length of region `rgid` in bytes. If `archtype` is non-null, it is set to the architecture type of the region.

```
int bcr_rgwait(void **rgid,
               int msec,
               int failfree)
```

This routine is used to check or wait for the effective completion of a prior **bcr_get** call which was invoked with an `msec` argument equal to **bcr_PENDING**. The routine blocks (i.e. does not return) until region `rgid` contains data that can be accessed by the user (in which case a 1 is returned), or *msec* milliseconds have elapsed (in which case a 0 is returned). In the latter case, if the `freefail` flag is true, then a call to **bcr_rgfree** is performed implicitly, and the previous **bcr_get** operation is guaranteed to be unsatisfied.

```
int bcr_rgwaitm(int nids,
                void ***rgids,
                int msec,
                int failfree)
```

This routine is used to check or wait for the effective completion of one or more prior **bcr_get** calls which were invoked with an `msec` argument equal to **bcr_PENDING**. `rgids` is an array containing `nids` region IDs, and this routine blocks (i.e. does not return) until at least one contains data that can be accessed by the user (in which case its index relative to 1 is returned), or *msec* milliseconds have elapsed (in which case a 0 is returned). In the latter case, if the `freefail` flag is true, then a call to **bcr_rgfree** is performed implicitly, and the previous **bcr_get** operation is guaranteed to be unsatisfied.

Comm Cell Manipulation

```
int bcr_put(int qlike,
            void **rgid,
            int cce,
            int cell,
            int nofree)
```

Adds a copy of region `rgid` to the end of comm cell `cell` in CCE `cce`. If `qlike` is zero, any regions in the cell before the new region are removed. If `nofree` is zero, an implicit **bcr_rgfree** is performed on `rgid`.

```
void **bcr_get(int qlike,
               int cce,
               int cell,
               int msec)
```

Places a copy of the first region from comm cell `cell` into the comm heap for the current CCE. If `qlike` is non-zero, that first region is removed from the cell. If the specified comm cell is empty, the routine will block for up to `msec` milliseconds for the cell to become non-empty. If `msec` is negative, the routine blocks indefinitely for the cell to become non-empty. This routine returns the region ID of the dequeued region. If no region is found, **NULL** is returned.If `msec` is **bcr_PENDING**, the routine always returns a region ID immediately whether or not the cell has elements but the data within the region must not be accessed until verified using **bcr_rgwait**.

```
int bcr_zap(int cce, int cell)
```

Removes all regions from cell `cell` in CCE `cce`.

```
int bcr_putm(int qlike,
             void **rgid,
             int ncells,
             int *cells,
             int nofree)
```

bcr_putm is identical to **bcr_put** and except that it can operate on many cells in many CCEs with one call. `ncells` is the number of cells, and `cells` is a list of 2*`ncells` integers interpreted as `(cce,cell)` pairs.

Communication Macros

```
#define bcr_write(rgid, cce,      \
                  cell, nofree)   \
        bcr_put(0,rgid, cce,      \
                  cell, nofree)
```

Add a region to a cell destructively.

```
#define bcr_enq(rgid, cce, \
                  cell, nofree) \
        bcr_put(1,rgid, cce, \
                  cell, nofree)
```

Add a region to a cell non-destructively.

```
#define bcr_writem(rgid, ncells, \
                  cells, nofree) \
        bcr_putm(0,rgid, ncells, \
                  cells, nofree)
```

Add a region to multiple cells destructively.

```
#define bcr_enqm(rgid, ncells, \
                  cells, nofree) \
        bcr_putm(1,rgid, ncells, \
                  cells, nofree)
```

Add a region to multiple cells non-destructively.

```
#define bcr_read(cce, cell, msec) \
        bcr_get(0, cce, cell, msec)
```

Copy a region from a cell non-destructively.

```
#define bcr_deq(cce, cell, msec) \
        bcr_get(1, cce, cell, msec)
```

Copy a region from a cell destructively.

Shared Memory Operations

These operations treat a cell as a lock, and the region within the cell as being protected by it. All are equivalent to other operations, so may be implemented as macros in some cases.

```
void **bcr_acqrl(int cce,
                 int cell,
                 int msec)
```

Attempts to acquire read lock (corresponding to cell `cell` in cce `cce`) for `msec` microsecs. Equivalent to:

```
    bcr_get(0,cce,cell,msec);
```

Returns the ID of the newly-locked region, as though from `bcr_get`. Does not prevent the region from being locked again (with same lock), but ensures that writers do not interfere with readers.

```
int bcr_rlsrl(void **rgid)
```

Releases a read lock to the given region. Equivalent to:

```
    bcr_rgfree(rgid);
```

The return value is identical to that from `bcr_rgfree`

```
void **bcr_acqwl(int cce,
                 int cell,
                 int msec)
```

Attempts to acquire write lock (corresponding to cell `cell` in cce `cce`) for `msec` microsecs. Equivalent to:

```
    rgid = bcr_get(1,cce,cell,msec);
    if (rgid) bcr_rgmod(rgid);
```

Returns the ID of the newly-locked region, as though from `bcr_get`. Upon acquiring the lock, no further attempts to acquire the lock (by any CCE, for reading or writing) will succeed until it is relinquished.

```
int bcr_rlswl(void **rgid,
              int cce,
              int cell)
```

Relinquishes a write lock for region `rgid` to lock corresponding to cell `cell` in cce `cce`. Lock does not need to be the same as that from which region was acquired, but should be empty (i.e. write locked). Equivalent to:

```
    bcr_put(0,rgid,cce,cell, 0);
```

The return value is identical to that from `bcr_put`.

```
void **bcr_wl2rl(void **rgid,
                 int cce,
                 int cell)
```

Converts a write lock to a read lock, by effectively relinquishing the write lock and then acquiring a read lock, so descriptions for those operations apply. Equivalent to:

```
bcr_put(0,rgid,cce,cell, 1)
```

The return value is identical to that from **bcr_put**.

Copying/Translation Operations

Copy routines transfer user-space data to or from a region, as directed by a copy descriptor, while automatically performing translation as necessary to adhere to the region's architecture type code (archtype). The routines take a buffer in user space (denoted by an address and a length in bytes), an address in the comm heap (specified as a `region` and `offset`), and a copy descriptor—an integer array which is interpreted by BCR as a sequence of operations to follow, each of which may result in data being copied from the source to the destination, and/or skipped in one and/or the other. Copying terminates if either the source or destination is exceeded (i.e. attempts to go outside of the user buffer or the region in the comm heap), or if all instructions in the copy descriptor have been performed.

The copy descriptor is interpreted as a list of triples, each having the form `(adjust,type,repl)`, where:

`adjust` is an integer displacement, in bytes, to take place within the user area relative to the end of the last operation,

`type` denotes one or more data types (as described below) and flags. The flags denote whether fields of that type (those types) are to be copied (default), or skipped in the source (the **bcrT_SKIP_FM** flag) and/or skipped in the destination (the **bcrT_SKIP_TO** flag), as well as whether this is the last triple in the copy descriptor (the **bcrT_END** flag)

`repl` is a replicator, denoting the number of times to repeat the copy or skip(s) denoted by `type`.

The `type` represents either an atomic type or a complex type. An atomic type is a single data value, and is represented by one of the predefined constants **bcrT_CHAR**, **bcrT_SHORT**, **bcrT_INT**, **bcrT_LONG**, **bcrT_LONGLONG**, **bcrT_FLOAT**, **bcrT_DOUBLE**, or **bcrT_CCE**. A complex type is analogous to a record type, and is denoted by the **bcrT_NEST** flag plus an offset (in triples) to the beginning of the triple of a new copy descriptor denoting that complex type. (The intent is for this new copy descriptor to exist within the same integer array as the current one, but to begin after the last triple of the current one.) This complex type has the effect of inserting the new copy descriptor into the current one `repl` times. Any **bcrT_SKIP_FM** and **bcrT_SKIP_TO** flags on a **bcrT_NEST** apply to all of the triples within the new copy descriptor.

For convenience, pointers to some simple complete copy descriptors are provided in **types.h**, each containing just one field of a given type, and each having a name of **bcrT_1type**, where *type* is as before (e.g. **CHAR**, **INT**, **CCE**, etc.). For example, **bcrT_1INT** is defined as:

```
int bcrT_1INT[3]
   = {0, bcrT_INT|bcrT_END, 1};
```

To process each triple, `adjust` is first added to the user pointer, then the copy and/or skip(s) take place, performing any padding necessary (on both source and destination) to adhere to architecture-specific alignment restrictions.

Possible return values **copyfm** and **copyto** are:

- n >= 2: Copy descriptor was expended, and n-2 bytes of source were not copied (where n-2 might equal 0).
- 1: Copy descriptor not expended, all of source read
- 0: No translation available for archtype
- -2: Bad copy descriptor or other arg error
- n < -2: Copy descriptor not expended, but n-2 bytes of source were not copied because destination filled up.

```
int bcr_copyfm(int *copydesc,
               int repl,
               void **rgid,
               int offset,
               void *buffer,
               int buflen)
```

Copies data from region `rgid`, starting at byte `offset` within the region, to user space address `buffer` which is `buflen` bytes long, according to copy descriptor `copydesc` processed `repl` times.

```
int bcr_copyto(int *copydesc,
               int repl,
               void **rgid,
               int offset,
               void *buffer,
               int buflen)
```

Copies data to region `rgid`, starting at byte `offset` within the region, from user space address `buffer` which is `buflen` bytes long, according to copy descriptor `copydesc` processed `repl` times.

```
int bcr_copytosz(int *copydesc,
               int repl,
               int archtype,
               int offset,
               void *buffer,
               int buflen)
```

Identical to **`bcr_copyto`** with identical argument except that (a) no data is actually copied, and (b) the return values are different, denoting the amount of space which would have been consumed in the region had the copy occurred. This is used to determine the size of region required for a subsequent **`bcr_copyto`**. The `buffer` argument can be omitted (i.e. null), but if provided will be used to determine alignment only. No `rgid` argument is present, but if a non-zero `archtype` is provided in its place, it will be interpreted as the archtype of the associated region (which will rarely have any effect).

Possible return values are:

n >= 2: Copy descriptor was expended, some of source may not have been read, n-2 bytes used in destination (including original offset).
0:No translation available for archtype
-1: Bad copy descriptor or other arg error
n <= -2: Copy descriptor not expended, but n-2 bytes used in destination before source ran out.

Message Passing

The message passing routines are defined in terms of calls to other routines, but may be extensively optimized (i.e. not actually implemented that way). These are similar to message passing routines offered by other packages.

```
int bcr_send(char *buffer,
             int len,
             int *buftype,
             int repl,
             int cce,
             int cell,
             int qlike,
             int archtype)
```

A call to this routine is semantically identical to the following code:

```
size = bcr_copytosz(buftype, repl,
           0, buffer, len);
if (size < 0) size = -size;
if (size < 2) return -2;
rg = bcr_rgalloc(size-2,archtype);
if (!rg) return -1;
ret = bcr_copyto(buftype, repl,
           rg, 0, buffer, len);
if (ret <= 0) return ret;
if (bcr_put(qlike,rg,cce,cell,0))
           return -2;
return ret;
```

The routine is allowed to (effectively) temporarily extend the comm heap, thereby not returning a -1 even in cases where the comm heap may otherwise be full.

```
int bcr_sendm(char *buffer,
              int len,
              int *buftype,
              int repl,
              int ncells,
              int cells,
              int qlike,
              int archtype)
```

A call to this routine is semantically identical to the following code (with appropriate error checking):

```
  rgid = bcr_rgalloc(len, archtype);
  bcr_copyto(buftype, repl, rgid, 0,
             buffer, len);
  bcr_putm(qlike,rgid,ncells,cells,0)
  ;
```

```
int bcr_recv(char *buffer,
              int len,
              int *buftype,
              int repl,
              int cce,
              int cell,
              int qlike,
              int msec)
```

A call to this routine is semantically identical to the following code (with appropriate error checking):

```
  rgid =bcr_get(qlike,cce,cell,msec);
  bcr_copyfm(buftype, repl, rgid, 0,
             buffer, len);
  bcr_rgfree(rgid);
```

Handlers

```
int bcr_handler(int cell,
                void (*han-
                dler)(int),
                int water_mark,
                int activep,
                int blockp)
```

Registers a handler, `handler`, for comm cell `cell` in the current CCE. If `water_mark` is non-positive, it represents a low-water-mark, so the handler will be requested whenever the cell contains at most |`water_mark`| regions when `bcr_deq` is performed (before removing the region). If `water_mark` is positive, it represents a high-water-mark, so the handler will be requested whenever the cell contains at least `water_mark` regions after a **`bcr_enq`** is performed. An invocation of the handler is given a priority of `activep` while it is executing (or waiting to execute) and `blockp` while it is blocked (e.g. while waiting for a **`bcr_deq`** operation).

```
int bcr_priority(int activep,
                 int blockp)
```

Alters the priority of the calling handler invocation to be `activep` while it is executing (or waiting to execute) and `blockp` while it is blocked (e.g. while waiting for a `bcr_deq` operation).

```
int bcr_allow(int priority)
```

Allows requested handlers with priority of at least `priority` to be invoked within the current thread by temporarily altering the priority of the calling handler invocation to be `priority`. If priority is negative, the active priority of the calling routine is used.

```
int bcr_block()
```

Effectively performs `bcr_allow(0)` forever. A call to this routine never returns, and is used when all further processing for the current CCE is to be performed by handlers.

Coding Hints

If you are very familiar with message passing and/or shared memory, and perhaps have already written your software to use one or the other, the quickest way to make use of CDS/BCR is probably to first use the CDS analogies (e.g. the **`send`** and **`recv`** routines, or the "locking" macros), and then (over time) to determine when adherence to those traditional paradigms interferes with the optimal efficiency and/or understandability of the program, converting the individual communications over time as necessary to a more "CDS-native" style. This section will not discuss the conversion process, in part because not everyone will start there. It will, however, discuss this "CDS-native" style. Many of the aspects described here are used in programs present in the `test` directory (e.g. `ring.c`).

Basics

Planning a CDS program is similar to planning any other sort of concurrent program—i.e. figuring what computation can be occurring in parallel with what other, and what data needs to move between the pieces. It differs even here, though, because the programmer is not so bound to matching the number of computational pieces—CCEs—to the number of processors. In this sense, it is more instructive to think of CCEs as "portable threads" than as processes like one would use in a message-passing program. (Similar to the way "real" threads may physically share some data space, these CCE's logically share access to that data, which may or may not be implemented using physical sharing.) In some sense, then, the more CCE's, the more flexibility, since the larger number of processors that can be productively accomodated, but as in all things, this guideline has its limits, especially in the current BCS version. (Future versions are likely to let CCEs be implemented literally as OS threads, thus minimizing scheduling overhead.)

The next uniqueness in CDS programming is in specifying which data (i.e. regions) will be used where (i.e. in which CCE) and when. In traditional message passing, the programmer thinks in a "push" mindset, where each data producer must know who will consume. In traditional shared memory, the programmer thinks in a "demand" mindset, and cannot provide this information about the eventual consumer even if s/he knows it. In CDS, the programmer provides the information s/he knows about the consumer's location when s/he knows it. If the producer doesn't know who will consume, the data is generally **`put`** into a local cell, to minimize the likelihood that the **`put`** operation will result in latency, even if the **`get`** from the consumer eventually does. Even in this case, if the consumer realizes that it will need the data but doesn't need it *right now*, it can issue a `PENDING` get, which will effectively prefetch the data. (In this case, the consumer must issue an **`rgwait`** before actually accessing the region.) Of course, maximum latency-hiding opportunities still result from putting the data directly to its eventual destination, if this is known when the **`put`** operation is performed.

Message passing programmers will often stop there, thinking only about how much data needs to be moved where and when. The CDS programmer gets extra points for data that doesn't need to be moved to or from the region in order to be used, and for regions that only need to be read by some CCEs (at least for some period of time). These will provide maximum opportunity for communication without copying, and for having multiple CCEs accessing the same region at the same time. Trade-offs are definitely involved even still—e.g. does it make more sense to put logically-related data into two separate regions, just because some of it will be modified often while other rarely? It will most certainly depend on the program and probabilities of encountering latency.

For those programs that benefit greatly by having the data in their private (non-comm heap) memory, perhaps in an unpacked/scattered form, they should use the **send** and **recv** operations whenever they apply (instead of the corresponding primitive operations) since the combined operations provide the most opportunity for optimization within BCR (though the current BCR version does not highly optimize these operations).

This leaves the very basics of programming, and managing the resulting complexity. For the most part, managing cell numbers is very similar to managing tags in a message passing program. That is, a given cell usually accomodates a particular kind of region, though there are cases when a single cell can be used as a collection point for many different kinds of regions which are identified programmatically by data within each region. When using a shared-memory paradigm (e.g. using the macros), each cell corresponds roughly to a lock, but its location also represents a "home" for the covered region. This, unfortunately, has many of the same sorts of disadvantages that home processors in DSM systems—i.e. the data may end up in a different place than you know it well be needed next. For this reason, if a shared region is known to go through different phases which are processed in different CCEs, it is often productive to assign it several cells ("locks"), each designating an upcoming phase and each local to the CCE which will be processing the data in that phase.

Heterogeneous Platforms

It is possible to write CDS programs to run on collections of machines having heterogeneous data representations. The simplest approach is to never access data in place on the regions, but instead to always use **copyfm** and **copyto** (or **send** and **recv**) when accessing that data, thus translating the data if necessary, but this dispenses with most or all of the efficiencies offered by CDS (e.g. when conversion is not required). On the other hand, writing all of your code to be conditional on whether or not conversion is required can be very complex and error-prone. In most cases, a middle ground is possible. After the region comes in, it is easy to determine whether conversion is necessary, and if so, **copyfm** the data to a buffer in user space and create a pointer to it. If conversion is not necessary, create a pointer (in the same variable) to the region itself. From that point on, all access is through the pointer, regardless of how the pointer was created.

To see how this works, consider routines **xptr** and **xupdt**, as defined here (and in `test` directory file `hetero.c`), both which take a region ID (`rgn`), a copy descriptor for that region (called here `rgntyp`), a buffer large enough to hold the native region data (`rgntmp`), and the size of that buffer (`rgntmpsz`):

```
void *xptr(void **rgn, int *rgntyp, void *rgntmp, int rgntmpsz)
{
       int archtype;
       bcr_rglen(rgn, &archtype);
       if (archtype == bcr_archtype) {
           return *rgn;
       } else {
           bcr_copyfm(rgntyp, 1, rgn, 0, rgntmp, rgntmpsz);
           return rgntmp;
       }
}
```

```
void xupdt(void **rgn, int *rgntyp, void *rgntmp, int rgntmpsz)
{
      int archtype;
      bcr_rglen(rgn, &archtype);
      if (archtype == bcr_archtype) {
        bcr_copyto(rgntyp, 1, rgn, 0, rgntmp, rgntmpsz);
      }
}
```

These are used as follows. After a CCE **get**s a region to access, it must first call **rgmod** if it plans to change the data in the region, then (in any case) can call **xptr** to get a pointer to the region data, at which time data conversion will be silently performed if necessary. If the CCE ultimately updates the region data (through that pointer), **xupdt** should be called (with the same arguments) before **put**ting the region into a cell, allowing the data to be silently converted back to the archtype of the region, if necessary. If the process does not **rgfree** the region during the **put**, it can continue using the existing pointer—unless it might modify the region data, in which case it must call **rgmod** and **xptr** again. Note that this "**xptr**" approach does not work well if `rgntyp` contains some CCE IDs, unless all of them are at the end of the region and the caller doesn't need to access them, since CCE IDs must be converted through the copy routines to be useful, and the room taken by a native CCE ID in the region will be very different than the size of a converted CCE ID in the `rgntmp` buffer. When CCE IDs do need to be accessed, hybrid approaches can sometimes be developed, e.g. by handling the CCE part of the region independently from the non-CCE part.

Starting Up The Program

To start up a CCE "cleanly", the first thing to realize is that no other CCE can possibly access the cells in a new CCE unless they know its ID, and they can't know of its ID until it (or a CCE it has told) tells them. CDS does not give that ID to any CCE except for the new CCE itself, not even the enlistor. Likewise, the new CCE cannot inform any other CCE of its ID unless it knows *their* ID. Since a new CCE always knows its enlistor's ID, one common user-enforced protocol is for the enlistor to **get** a region (the so-called "birthcry") from the new CCE (in one of the enlistor's cells) containing the new ID. This **get** should have a timeout value specified to account for the case where the new CCE dies before it produces the birthcry. (If the enlistor can do productive work between the **enlist** and the **get**, it may be able to completely hide CCE creation overhead.) The options can be broadened by having the enlistor provide the ID for one or more other CCEs within the enlistment message itself. In either case, the new CCE is anonymous between the time that it calls **init** and the time that it **put**s its ID into another CCE's cell, so the intervening interval is precisely when it should call **magrow** to create its comm area (i.e. comm heap and comm cells): Waiting until after it performs the **put** runs the risk of other CCEs attempting to access the cells before they exist. Note that the comm heap can become fragmented over time, so to ensure that sufficient space will be available for any particular region, it should be created twice as large as the data that is actually required.

The property that one CCE cannot access cells within another without the other's ID can be used to implement information hiding—i.e. by allowing one CCE to have the ID of another only if it has good reason to access its cells. Since the enlistor and new child will often learn each others IDs during the enlistment process, there are limits to this, but it works to a degree. The **enlist**

operation is more powerful and flexible than many processes will need. Specifically, it is usually very unwise to hard-code machine names into your program. It is usually more productive to encapsulate the enlistment process into a separate routine—e.g. one which reads the machine names from a file or obtains them from a resource manager. This will also help accomodate potential future changes in the **`enlist`** arguments in future versions of BCR. See the `startup.c` and `readstrings.c` files in the `test` directory for examples.

Handlers

Handlers can be used as sort of a poor man's interrupt mechanism, or as a means for allowing emergency action to occur when a particular cell gets too empty or too full. They can even facilitate a complete demand-driven programming methodology, where the main program effectively stops (using the **`bcr_block`** operation), and all subsequent activity occurs within handlers.

A primary trick (and restriction) in the use of handlers is to understand that the appropriate conditions on the cell only cause the handler to be requested, not started. The handler is only started if the request is noticed by BCR, and the priority of the request is at least as high as the priority of the current execution. At this writing, BCR only notices requests if it blocks (or has the potential for blocking), so the `activep` priority actually has little meaning, but if overhead can be kept low, such checks may be installed at every call to BCR.

One of the most interesting operations is to install a handler with a low-water mark of 0, meaning that the handler will be requested only if a get is performed on the empty cell. The handler can therefore wait until a **`get`** has been issued to **`put`** a region into the cell to satisfy it.

Installing BCR

The easiest way to try out BCR is to put the `bcr.tar.gz` file into your home directory on each machine you will run it on, decompress it, and untar it—i.e.

```
gunzip bcr.tar.gz
tar -xvf bcr.tar
```

This will create a directory called BCR on that machine with subdirectories named `test`, `bin`, `include`, and `lib`. `lib` holds the BCR library `libbcr.a`, `bin` holds the BCR daemon `enl`, `include` holds the files `bcr.h`, `types.h`, and `error.h`, and `test` holds some test software and a `makefile` for building it.

Please read the README file and the LICENSE file. Using any of this software implies that you agree with the terms listed in these files, so if you don't like them, please delete all and stop.

```
lpr BCR/README
lpr BCR/LICENSE
```

Nothing needs to be (or should be) moved from these locations to use BCR—e.g. by default, this is where BCR will expect to find the `enl` program/daemon. To change this assumption, or several other ones (e.g. where to find the program to open a new window (`xterm`), to start a new process (`rsh`), or ...), look at the `bcrrc` file description in a subsequent section. You can also simplify matters by adding the `BCR/bin` directory to your path.

Compiling and Linking a BCR Program

When a BCR program is compiled, it must have access to the `bcr.h` include file, and when it is loaded it must have access to the `libbcr.a` library. The easiest way to see what is required is to view the sample source files and makefile in `BCR/test`. It is recommended that you try compiling these by changing to the `BCR/test` directory and running the `makefile`:

```
cd BCR/test
make
```

Depending on your system, you may need to change the compiler or other tools referenced in the makefile, as indicated in comments. No errors or warnings should result in a correct execution.

Running a BCR Program

A BCR program can be started (i.e. enlisted) in two different ways:

1. Just starting the first (or "root") CCE from the shell (i.e. by typing the name of the program)

2. Running the enl program, providing the name of the file containing the root CCE as an option.

In either case, the user (i.e. terminal) ends up attached to the `enl` daemon, with the new user process running as its child, by the time the **`init`** call has completed within that process. In the first case, the program must be initiated from the user's home directory, or the `BCR/bin` must be set permanently within the user's path, because BCR will (by default) expect to find the `enl` program/daemon at the relative location `BCR/bin/enl` (though this can be changed by using the `bcrrc` file). The first approach is obviously the simplest, but the `enl` program allows some additional options that can be specified (e.g. for debugging), discussed below. Also, in the first approach, the OS process ID (not CCE ID!) of the program changes when it calls **`init`**, but in the second case it doesn't, which could theoretically be important in some special applications. Regardless of how the program is started/enlisted, it can be aborted (usually!) by simply killing the process or daemon (e.g. with a control-C). By default, if any process terminates abnormally, the entire program will terminate, including any daemons which have started up.

When it starts, BCR assigns a name to the entire program, constructed from the name (or ip number) of the machine it started on and the OS process ID of the `enl` daemon on that machine. As the BCR program runs, each machine involved in the execution will automatically get its own daemon, and each daemon will get a daemon log file with the name `bcr.xxx.log` in its `/tmp` directory (or other location if changed in `bcrrc`), where *xxx* is the BCR-generated program-wide name. If something goes wrong, this file is often a good place to look. BCR can be configured (through the "startscript" and "endscript" variables in `bcrrc`) to move these log files to a more directory (e.g. on the first machine) as the daemons terminate, to make them more easily accessible.

The root CCE is the only one which should depend on having access to stdin, stdout, and stderr files. These are closed for other CCE's, though the CCE itself is free to again re-open them as desired.

Using `enl`, it is possible to initiate a BCR program so that, each time a new CCE starts up, a new X window is opened, and the new CCE is left in an initial state within that debugger. (Use of this feature may be problematic in secure environments.) To do this, the enl -d and -D switches are used to specify the debugger to be used, and the display upon which to open the windows, respectively. For example, to run the `nring1` test program from the user's home directory, using the `gdb` debugger and displaying the windows on the main display of machine `foobaz`, one might use the command:

```
enl -d gdb -D foobaz:0 BCR/test/nring1
```

Running the `test` Programs

There are (at least) three programs in the BCR/test directory which may be worth investigating and/or executing. The most complex and indepth of these is `ring`, which creates a ring of CCEs on processors specified by the user (in a file), then passes regions among them, both in an "all-to-all reduction" fashion and around the ring, printing various statistics on latency and throughput. Another, consisting of "`thrup`" and "`thrut`" ("through put" and "through take") simply blasts a large number of large regions from one CCE (`thrup`) to the other (`thrut`). Another, `timeout`, simply allows several **`get`**s to timeout and then determines how close the actual time taken is to that requested.

All of the issues described in the previous section (Running a BCR Program) apply, so unless the user has added `BCR/bin` to their path, these programs should be invoked from the user's home directory.

`ring`

Before running `ring`, the user should create a file named `machines` in the directory from which the program will be invoked (e.g. the user's home directory), and fill it with machine names (or IP addresses), one per line (with an effect described shortly). Each of these machines must have BCR installed in the home directory, with `ring` present and compiled in the `test` directory.

The program will first request and accept 4 integer inputs from the user:

- The number of CCEs to create (in the ring). The first CCE (i.e. that requesting this input) will become one of these, so one fewer CCE than the number entered will be enlisted (invoked). Each CCE will be enlisted on the next sequential machine listed in the `machines` file, starting at its beginning and rewinding it if more CCEs are requested than there are machines on the file. It is perfectly legal to repeat machine names on that file.
- The minimum and maximum region size to be passed around the ring, log base 10. Region sizes will vary by factors of 10, so an input of 0 and 3 for these values result in regions of size 1, 10, 100, and 1000 being passed.
- The number of times to pass each region around the ring.

The program first reads the machine names, then starts the CCEs using the `start_ring` routine in `startup.c` and reports the time this takes. (If option REDUCE is defined, it then performs a reduction all-to-all operation using a parallel prefix on the ring.) It then just passes regions around the ring as requested by the inputs, timing each. Note that the statistics that it reports are "effective", rather than actual. That is, if regions are passed between CCEs on the same processor, then it is counted as though all of the bytes move from one to the other, even though all that really happens (internally) is that a pointer is added and removed from a cell.

th`rup and thrut`

The user starts `thrut` (the "taker"), and provides the name of the machine on which to start `thrup` (the "putter"). `thrut` then starts `thrup`, and `thrup` puts several large regions into a cell in `thrut` without waiting for any intervening communication from `thrut`. `thrup` determines how long this takes, and `thrup` reports it. This program is intended to give a feeling for bandwidth.

`timeout`

Extremely simple program which tries to get regions (which will never arrive) and then reports how low long it took for them not to arrive (after timing out).

The BCRRC Options

Each time a BCR daemon (i.e. enl) runs and/or init is called, BCR looks for configuration opions in two places: `/usr/lib/bcrrc` (no dot!), and in "`.bcrrc`" (with dot!) in the user's home directory. If both are present, the options in the first will be read, and then those in the second will be read, perhaps overriding them. Any of the options not found will be assigned default values, described here.

Each option in these files must be specified in the form

```
keyword value
```

The possible keywords, and the meaning and default value for each, is described here. Note that, for those defaulting to a commonly-available program, replacing it with another program will only be successful if the arguments for the new program are similar to the default one.

`xtermpath`

The path of the program BCR should use to open a new window. The default is `/usr/X11R6/bin/xterm`, except on Solaris systems, where it is `/usr/openwin/bin/xterm`

`enlistpath`

The path of the `enl` program. Please note that this is used when initiating the daemon on other machines, so things are not going to work well unless `enl` is in the same place on every machine (at least relative to the user's home directory). The default is `BCR/bin/enl`

`rshpath`

The path of the program BCR should use to start a process on another machine. The default is `/usr/bin/rsh`, and some sites may find this problematic because of security concerns. Under some circumstances, it may be possible to specify an `ssh` program here, but one must be cognizant that BCR is not currently configured to expect any sort of password-oriented security to be involved when starting a new process.

`tempdir`

The directory where BCR will expect to create its daemon log file, as well as certain other temporary files which are likely to have the same name across all machines (i.e. shared-memory pools) that BCR accesses. BCR attempts to delete all files after each execution, except for the log file which will remain unless explicitly managed by the endscript (below). Default is `/tmp`

`startscript`

The name of a program or script to be executed by the `enl` daemon each time it starts up. It will be provided with four arguments:

1. a zero (0)
2. the BCR program-wide name (used to build the name of the daemon log file)
3. the IP address of the machine on which the program was started
4. a 1 if this machine where the program was started, else 0

A common function for this script will be to do nothing, unless the last argument is a one, in which case it creates a directory in `/tmp` to collect log files from the endscript on other machines

(e.g. see `logscript` in the `/BCR/test` directory). If this option is given the value `*` (i.e. asterisk), it will be taken to mean that no startscript should be initiated. The default value is `*`

`endscript`

The name of a program or script to be executed by the `enl` daemon just before it shuts down. It will be provided with the same four arguments as for the startscript (if any), except the first will be a one (1) instead of zero, allowing the same script to be used for both the startscript and endscript. A common function for this script will be to delete the daemon log file from the `/tmp` directory after either inspecting it or copying (remote copying) it to a directory on the root machine (e.g. created by the startscript on that machine). See `logscript` in the `/BCR/test` directory for how this might work. If this option is given the value `*` (i.e. asterisk), it will be taken to mean that no endscript should be executed. The default value is `*`

`ifacesuffix`

A string to be appended to the end of any machine name specified in a call to **`enlist`**. This is provided in the expectation that such suffixes will often designate different interfaces, using different networks, to the same machine, so this can be used to easily experiment with different network types. A value of `*` (asterisk) designates that no string will be added. The default value is `*`

`packwords`

The maximum number of 4-byte payload words to (not including BCR header data) to include in a UDP packet. The number of words will be smaller if the receiving machine uses a smaller value. Default is 4092.

When Things Go Wrong: A Peek Under the Covers

The BCR daemons on each machine do their best to watch over the execution of the BCR program on their machine, and to communicate with each other when necessary to spread the word when things go wrong (or a program terminates normally) so that things can be shut down cleanly. However, when something goes wrong (a bug?) within the daemon itself, various trash may be left around, and/or the program may (at least appear to) refuse to terminate. Cleaning up the mess is often not fun, especially since it may be scattered over several machines. (Cleaning may not even be necessary in some cases.) This section will outline what kinds of trash may be left around and how to clean it up.

Some signs that something went wrong:

- Your program doesn't stop normally, or with control-C
- Your program has apparently terminated (or been killed), but not normally. Check any:
  - A file in `/tmp` has a name that starts with "`bcr.`" but does not end with "`.log`"
  - A user process corresponding to your BCR program is running, but no process named `enl` is running.
  - The `ipcs` command shows unusual shared memory (shm) segments (i.e. which can't be explained by other operations).

Processes

The three kinds of processes which constitute a BCR execution are the user processes constituting the CCEs (which should be easily recognized), the BCR daemon (called `enl`), and sometimes `rsh` processes (which are created when this `enl` daemon starts daemons on other machines, and usually hang around until the other daemon completes). The one `enl` daemon per program per machine is the parent of all of the user and `rsh` processes for that program on that machine, so if the daemon is not running but one or more of the other processes are, then something is definitely amiss, and the processes have been orphaned. An rsh process will also signify that a daemon is still running on another processor as well.

The `enl` daemons form a tree topology, and each knows which daemon started it (i.e. its ancestor in the tree). In the normal shutdown sequence, when a daemon sees any sort of problem (e.g. one of its child daemons or processes dies, or its parent daemon reports a problem), then the daemon tries to kill all of the user processes on the machine, and inform all of the daemons that it started (its children) that there is a problem. The daemon itself tries not to die unless all of its children have died, so if some process anywhere along the way does not die, it's as though a domino didn't tip, and the whole thing can lock up. Guessing which domino is the problem.

To clean up, it is recommended to begin with the main machine, where the program was started, and if the daemon is running, kill it, first normally, then with signal 9:

```
kill procno
kill -9 procno
```

If the first kill works, it will probably take the rest of the program with it. If not, then you may need to kill the other processes manually, again first normally and then with signal 9. If you kill an `rsh`, it may very well propogate the kill signal to the corresponding daemon on the remote processor and shut down that branch of the tree normally. Otherwise, you must repeat this there.

Shared Memory Segments

BCR typically creates one shared memory segment on each machine (the "machine segment"), plus one each time new comm area is needed (e.g. with each call to **`magrow`**). These are usually cleaned up automatically when the daemon finishes. They can usually be seen (i.e. while the program is running, or if it didn't terminate correctly) with a command called `ipcs`. Depending upon the OS (and version) that you are running, there may also be many many segments that have nothing to do with BCR. BCR segments have no special qualities to identify them, though the OS segments usually have some very predictable size (e.g. 32K).

It may not hurt to just leave extra shared memory segments lying around, but that will depend upon the OS and its configuration. Segments can usually be removed with the `ipcrm` command, but the way to do so will also depend on the OS and version. (It often requires a switch like `-M` or `shm`, plus some ID that can be extracted from the `ipcs` report.)

Temp Directory (/tmp)

BCR creates two files in the `/tmp` directory (or whatever directory has been specified as the temp directory in the BCRRC file). They are the daemon log file and the daemon exec file. Both have names starting with `bcr.name`, where `name` is the BCR-assigned program name, but the exec file has no further suffix, while the log file has the suffix "`.log`".

The presence of the log file does not indicate a problem, as it is left in `/tmp` by default. The exec file is deleted by the daemon when it terminates. If it is left behind from a termination problem, it will rarely cause a problem (since there is unlikely to be another program with exactly the same name that would conflict with it), but it can be safely deleted.