C and C Specification

Revision: 0.2

This document is owned by Cindy Spiller and is a combined effort with Ken Crouch.
It may be found in /p1/cm5/psim/doc/CandC.tex
Printed October 31, 1989

Contents

1 Introduction

C and C, or *CnC*, is the name given to C functions which provide access to the hardware in order to facilitate communication between the pe's and the scalar. This document specifies the C functions in the scalar, and the C functions in the pe (C and C) required to accomplish this communication.

1.1 Related Documents

The following documents are related and should be read in conjunction with this one:

- CM5 System Architecture Specification (currently danny's Darpa proposal addendum)
- Control network architecture: /p1/cm5/concepts/cn-architecture.tex
- data router architecture: /p1/cm5/doc/concepts/data-router.text
- CN Implementation Specification (/p1/cm5/doc/modules/cn.text)
- Diagnostics Network Specification (/p1/cm5/doc/modules/dn.text)
- NI Architecture (/p1/cm5/doc/concepts/ni-architecture.tex)

2 CnC Overview

2.1 Coding Conventions

CandC is considered the lowest level programing language of the CM5, or the assembly language. CandC functions and variables use capital letters for the beginning character in each word of a name. No underscores are used. Ex. RouterSend.

- Function names and variables use capital letters for first character in words of name.
- All pointer variables end in Ptr.
- Constants are all capital letters with underscores separating words.
- Typedefs are all capital letters with underscores separating words and a _T at the end.

2.2 Definition

CandC provides functions for the data router, combine, broadcast, global, and sync networks. Communication on these networks is controlled by the NI chip. CnC's main objective is to provide an abstract access to the control of this chip, yet be as efficient as possible. The users of CnC will be the Paris Library, the output of the LoC compiler, the Paris compiler, Fortran compiler, and kernel builders. CnC may also be made available to other users as the low level (assembly type) language.

Note that the term function will be used to express the action to be performed by CnC. However, this term is used loosely, and the action may be performed by a true C function interface, or whenever possible, by a macro.

CandC is provided as a library to be linked with users code. The header file `/pl/cm5/psim/current/CandC.h` specifies the constants required when linking with CandC.

CandC functions are called by the processors in a partition, called pe's, as well as by the scalar processor. The scalar processor is usually a large capacity cpu which is physically distinct from the pe's. Throughout this document, a description of the action performed will be dependent on whether the function is being called from a pe or from the scalar, or if it is the same when called by all processors. Whenever the term 'all processors' is used, this refers to all pe's and scalar. The term 'all pe's' refers to all the processors in the partition except the scalar.

There is another distinction regarding processor status. There can be a pe within the partition which obtains the status of 'broadcaster'. The term broacaster simply means that this pe has control of the broadcast network, and it can send broadcasts while everyone else can receive them. Generally the scalar will be the broadcaster, but this power can be relinquished to any pe in the partition. Of course this power can only be given to one pe at any one time. This pe will be refered to as the 'broadcaster'.

The scalar usually does not participate in contributing to combine operations, but receives the result of a combine operation which has been 'reduced'. However, any pe in the partition can be set up to not combine, yet have the reduce result sent to him. (More than one pe could be doing this? Would this make sense?)

3 Constants

- MAX_ROUTER_MSG_WORDS Maximum number of words which can be placed in the router fifo for one message. Current maximum is 6 words including the destination.
- MAX_BROADCAST_MSG_WORDS Maximum number of words which can be placed in the broadcast fifo for one message. Current maximum is 15 words.
- MAX_COMBINE_MSG_WORDS Maximum number of words which can be placed in the combine fifo for one message. Current maximum is 4 words.

- CHUNK_TABLE_ENTRIES number of chunk table entries. Currently set at 64.
- etc.

4 Variables

4.1 SelfAddress

int SelfAddress;

Physical address of this PE or scalar.

4.2 ScalarAddress

int ScalarAddress;

Address of scalar processor. This will be the same as PhysicalSelfAddress in the scalar.

4.3 PartitionSize

int PartitionSize;

Number of PE's in the partition.

5 Participating and Abstaining from network activity

Each processor in the partition is connected to the following networks: Broadcast, Combine, Router, and Global. The networks assume that since the pe is connected to the network, that it will participate in all activity on the network. If the pe wishes to not participate in some activity, it can set a bit to indicate which activity it wishes to abstain from.

The logical activities that a processor would wish to abstain from are the ones which require synchronization from all the processors before the operation will complete. If a processor does not plan to contribute, then it must set its abstain bit to indicate this so that the operation will complete without it.

The processor can abstain from receiving or sending to the broadcast network. However, only one processor (called the broadcaster) can be broadcasting at any time, so any processor enabling its abstain bit to allow sending broadcasts has to be coordinated with all the other processors.

The processors can abstain from contributing to the combine operation, and can abstain from receiving combine reduce results.

Since the router network does not require that every processor participate in an operation to complete, there are no abstain bits for the router network.

5.1 initial configuration

Note: add hardware reset state.

Initially, the abstain bits are set such that the scalar is the broadcaster, and all pe's will receive the broadcast. All the pe's will participate in the combine operations, and all the pe's and the scalar will receive reduce results.

5.2 changing participation

At various times after initialization, different processors will need to change their participation. These functions provide that capability. The activity that can be participated in or which can be abstained from include:

- NI_REDUCE_RECEIVE
- NI_BC_RECEIVE
- NI_COMBINE
- NI_SYNC_GLOBAL

```
ParticipateIn(activity)
int     activity;

Example:

ParticipateIn (NI_REDUCE_RECEIVE | NI_SYNC_GLOBAL);

AbstainFrom(activity)
int     activity;

Example:

AbstainFrom(NI_SYNC_GLOBAL);
```

5.2.1 EstablishBroadcaster

This function must be synchronized amoung all processors, therefore all processors in the partition (including the scalar) must execute this function before it will return. The processor whose address is specified will become the broadcaster.

5.2.2 OnlyPesCombine

Sets up the combine network so that only the pe's in the partition (not the scalar) contribute to the combine operation.

5.2.3 ScalarCombinesAlso

All pe's and the scalar contribute to the combine operation.

5.2.4 ReduceToScalarOnly

A combine reduce result is available in the scalar only.

5.2.5 ReduceToPesOnly

The combine reduce result is available to the pe's in the partition only.

5.2.6 ReduceToAll

The combine reduce result is available to all pe's as well as the scalar.

6 Global

Note: specify how supervisor gets bits, distinction between user and supervisor.

The global network or interface provides a bit which can be globally updated by every processor in the partition including the scalar. There are three bits available. The synchronous bit, and two asynchronous bits, one for the user and one for the supervisior. The asynchronous bits are updated almost instantly without the need for input from any other processor. The synchronous bit is updated only after every processor has contributed its bit.

6.1 Global Sync

The synchronous global facility provides a way for each processor in the partition to OR a bit with every other processor in the partition. Each processor presents a bit which the hardware then OR's with a bit from every other processor which is participating (not abstaining) in the sync. When all the participating processors have presented their bits, then the hardware indicates that the sync is complete. The result is available to be read.

6.1.1 OrGlobalSyncBit

This function takes a 32 bit quanity, of which only the least significant bit is relevant. This bit is presented to be OR'ed with the other processors' bits.

```
OrGlobalSyncBit(value)
unsigned int value;
```

6.1.2 GlobalSyncComplete

After OrGlobalSyncBit() is called, the hardware has to coordinate all the other processors to obtain the result. When this result is available GlobalSyncComplete will return true. It will return false until this result is available, and will become false again after the next call to OrGlobalSyncBit().

```
int GlobalSyncComplete()
```

6.1.3 GlobalSyncRead

Note: explain what happend if this is executed before sync complete .

After GlobalSyncComplete() returns true, the result can be read. GlobalSyncRead returns a 32bit quantity which only the least significant bit is relevant.

```
int GlobalSyncRead()
```

6.1.4 GlobalSyncReadWhenComplete

After OrGlobalSyncBit() has been called, the user can call GlobalSyncReadWhenComplete() which will wait until the result is available, and then return the result.

```
int GlobalSyncReadWhenComplete()
```

6.1.5 GlobalSync

The entire global sync operation can be performed in one call. GlobalSync takes the 32 bit quantity, presents it to the hardware, waits for the operation to complete, and then returns the result.

```
int GlobalSync(value)
unsigned int value;
```

6.2 Global Async

This facility is like the Global Sync in that one bit is presented from a processor, OR'ed with the bits from every other processor, and the result is updated in all pe's. The major difference is that one or more pe's can present its bit, and the result is calculated without the the need for all processors in the partition to participate. Therefore there is no need for abstaining from Global Async, just do not participate.

6.2.1 OrGlobalAsyncBit

```
OrGlobalAsyncBit(value)
unsigned int    value;
```

This function takes a 32 bit quanity, of which only the least significant bit is relevent. This bit is OR'ed with the other processors' bits, however, the other processors do not have to all update this bit before the result becomes available.

6.2.2 GlobalAsyncRead

```
int GlobalAsyncRead()
```

When a processor presents its bit, the hardware must OR this with the bits previously presented by all the other processors. This takes a few cycles to perform, and there is no indication of when this is completed. If this bit is going to be written and read immediately (calling OrGlobalAsyncBit() and then immediately calling GlobalAsyncRead()), GlobalAsync should be called. GlobalAsyncRead should be used only when inspecting the current value of the global async bit.

6.2.3 GlobalAsync

Note: explain worst case cycles.

This function logically calls OrGlobalAsyncBit() and then guarantees (by waiting for worst case cycles) that the result must be valid, and returns the result.

```
int GlobalAsync(value)
unsigned int value;
```

7 Broadcast

The following routines allow broadcasting units of data between the broadcaster and all other pe's. Any pe or the scalar can acquire the control of the broadcast network and issue a broadcast. This must be synchronized, so that only one processor in the partition controls the network at any one time. All other processors in the partition can receive the message from the broadcast net. The processors can abstain from receiveing any broadcast messages if requested.

7.1 Broadcast basics

Any broadcast architecture functionality can be accomplished by applying a combination of the following functions (plus the abstain functions) into the proper algorithm. In order to use these functions, you must proclaim yourself an expert of the entire architecture of the CM5. These are the building blocks used to implement the macros and functions in the remainder of this section. It is highly recommended that the macros and functions be used instead of these. If the architecture changes, these functions will most likely become invalidated.

7.1.1 BroadcastStatus

```
int BroadcastStatus()
```

BroadcastStatus returns the value in the broadcast status register. Bit maps are provided to map out any combination of the values available in the status register. Also, macros are provided to obtain all the fields in the status register.

Bitmaps:

- NI_SEND_SPACE_P NI_SEND_SPACE_L
- NI_REC_OK_P NI_REC_OK_L
- NI_SEND_OK_P NI_SEND_OK_L
- NI_SEND_EMPTY_P NI_SEND_EMPTY_L
- NI_REC_LENGTH_LEFT_P NI_REC_LENGTH_LEFT_L

Macros:

- SEND_SPACE(status)
- RECEIVE_OK(status)
- SEND_OK(status)
- SEND_EMPTY(status)
- RECEIVE_LENGTH_LEFT(status)

Example of using these Macros:

```
int  LengthLeft;

LengthLeft = RECEIVE_LENGTH_LEFT(BroadcastStatus());
```

Example of using the bit maps:

```
#define RECEIVE_LENGTH_LEFT(status) \
  ((status >> NI_REC_LENGTH_LEFT_P) & ~(~0 << NI_REC_LENGTH_LEFT_L))
```

7.1.2 BroadcastPrivate

SUPERVISOR only!

Returns the value in the broadcast private register.

7.1.3 BroadcastSendFirst

```
BroadcastSendFirst(length, msg)
int      length;
unsigned int    msg;
```

7.1.4 BroadcastSendWord

Send this word onto broadcast send fifo.

```
BroadcastSendWord(msg)
unsigned msg;
```

7.1.5 BroadcastReceiveWord

Receive a word from the broadcast receive fifo.

```
int BroadcastReceiveWord()
```

7.2 Broadcasting messages resetricted to fifo length

If the broadcaster has some data which can be sent out in one message, i.e. is less then MAX_BC_MSG_WORDS and is already constructed, then it can use BroadcastSend to send the data. After assuring that it has control of the network, the broadcaster issues a BroadcastSend to broadcast a unit of data, and the pe's execute a BroadcastReceive to receive the unit of data being broadcast.

7.2.1 BroadcastSend

BroadcastMsg accepts a message of at most MAX_BC_MSG_WORDS and returns when the entire message has been successfully sent into the network.

```
BroadcastSend(msg, length)
void            *msg;
int             length;
```

7.2.2 BroadcastReceive

```
BroadcastReceive(msg, length)
void *msg;
unsigned length;
```

BroadcastReceive reads the length available in the network, up to MAX_BC_MSG_WORDS, and deposits the data into the message space pointed to by msg in the order read from the network.

7.3 Broadcasting C types

These broadcast functions are more efficient if the unit being broadcast is of the type: int, unsigned int, float, or double.

7.3.1 BroadcastRead*

BroadcastRead* is called by the pe's to accept a data unit from the broadcast network. The scalar must be executing a Broadcast* to inject this unit of data into the broadcast network.

int BroadcastReadInt()

unsigned int BroadcastReadUint()

float BroadcastReadFloat()

double BroadcastReadDouble()

int (*BroadcastReadFunction())()

7.3.2 Broadcast*

BroadcastInt(data)
int data;

BroadcastUint(data)
unsigned int data;

BroadcastFloat(data)
float data;

BroadcastDouble(data)
double data;

BroadcastFunction(func)
(*func());

7.4 Queueing broadcast words of arbitrary length

NOTE: this whole section may or not exist.

If large amounts of data are to be sent, or if the broadcaster wants to construct the message a word at a time, then the following functions can be used. These functions allow the broadcaster to initiate a broadcast operation of arbitrary length and keep queueing more words of data to the broadcast network (QueueBroadcastWord or QueueBroadcastMsg) until a FinishQueueingBroadcast is called. The broadcaster need not worry about hardware fifo lengths. FinishBroadcast will not return until all the queued message words have been sent into the broadcast network. If the queue starts to become too large, QueueBroadcast* will block until some of the queue has cleared out.

7.4.1 QueueBroadcastMsg

A call to QueueBroadcastMsg will initiate the queueing operation, if one is not currently initiated. If this call initiated the operation then this message will be inserted into the begining of the queue. Otherwise, the message will be put in the next available space in the queue. This will return immediately if there is room in the queue to insert this message. It will block however if the queue is getting too large. It will return after enough of the queued messages have been sent into the network to insert this new message.

7.4.2 QBroadcastWord

This can be called to initiate the queue, or subsequently, and it can be called alternatingly with QueueBroadcastMsg. This is a more efficient way to send one word or 32 bits of information. It blocks similiary to the QueueBroadcastMsg.

7.4.3 FinishQueueingBroadcast

This function will not return until all queued messages have been successfully sent into the broadcast network.

7.4.4 ReadBroadcastQ

This function will return when the length of words specified have arrived from the broadcast network. It places the received words in the array pointed to by msg in the order that they were read from the network.

7.4.5 ReadBroadcastQWord

This function will return immediately if a word (32 bits) is available from the broadcast queue. If one is not available, it waits for one.

8 Router

The router network functions are higher level routines which correctly incorporate the Network Interface (NI) accessors to perform the steps necessary to perform a router command.

This section needs more work. More functions are to be defined.

8.1 Router basics

Any router architecture functionality can be accomplished by applying a combination of the following functions (plus the abstain functions) into the proper algorithm. In order to use these functions, you must proclaim yourself an expert of the entire architecture of the CM5. These are the building blocks used to implement the macros and functions in the remainder of this section. It is highly recommended that the macros and functions be used instead of these. If the architecture changes, these functions will most likely become invalidated.

8.1.1 RouterStatus

RouterStatus returns the value in the router status register. Bit maps are provided to map out any combination of the values available in the status register. Macros are provided to use extract the fields in the status register.

These bitmaps include:

- NI_SEND_SPACE_P NI_SEND_SPACE_L
- NI_REC_OK_P NI_REC_OK_L
- NI_SEND_OK_P NI_SEND_OK_L
- NI_REC_LENGTH_LEFT_P NI_REC_LENGTH_LEFT_L
- NI_REC_LENGTH_P NI_REC_LENGTH_L
- NI_DR_REC_TAG_P NI_DR_REC_TAG_L
- NI_DR_SEND_STATE_P NI_DR_SEND_STATE_L
- NI_DR_REC_STATE_P NI_DR_REC_STATE_L

Macros:

- SEND_SPACE(status)
- RECEIVE_OK(status)
- SEND_OK(status)
- DR_ROUTER_DONE(status)
- RECEIVE_LENGTH_LEFT(status)
- RECEIVE_LENGTH(status)

- DR_RECEIVE_TAG(status)
- DR_SEND_STATE(status)
- DR_RECEIVE_STATE(status)

Example of using these Macros:

```
int  Tag;

Tag = DR_RECEIVE_TAG(RouterStatus());
```

Example of using the bit maps:

```
#define DR_RECEIVE_TAG(status) \
  ((status >> NI_REC_TAG_P) & ~(~0 << NI_REC_TAG_L))
```

```
int RouterStatus()

companion functions:

int RouterStatusLeft()
int RouterStatusRight()
```

8.1.2 RouterSendFirst

Send this word onto router send fifo.

```
RouterSendFirst (tag, length, dest_pe)
unsigned        tag;
int             length;
unsigned        dest_pe;
```

8.1.3 RouterSendWord

```
RouterSendWord(data)
word    data;

companion functions:

RouterSendWordLeft(data)
word    data;

RouterSendWordRight(data)
word    data;

RouterSendWord(data)
word    data;

RouterSendDouble(data)
word    data;

RouterSendDoubleLeft(data)
word    data;

RouterSendDoubleRight(data)
word    data;
```

8.1.4 RouterReceiveWord

```
int RouterReceiveWord()
```

Receive a word from the router receive fifo.

```
RouterReceiveWord()

companion functions:
```

```
RouterReceiveWordLeft()

RouterReceiveWordRight()
```

8.2 Router functions and macros

These following functions are the ones recommended to be used. These functions combine the router basic functions in the correct algorithm to perform the given task.

8.2.1 SetCountMask

The router only counts messages which tags have been set in the count mask register. This function provides a way to ensure that your router messages will be accounted for when doing a RouterEmpty. (This may only be available to the supervisior in the future). Currently, tags 8 through 15 are available for the user.

```
SetCountMask(tag)
unsigned int tag;
```

8.2.2 RouterMsgToReceive

Returns true if there is a message to receive from the router network.

```
int RouterMsgToReceive()
```

8.2.3 RouterSendMsg

This function sends word_length words that are pointed to by source_base, and sends them to the specified destination processor. This function takes care of chopping up large data streams into packets that fit into the send fifo, and sequencing them to be re-assembled at the receiving end. RouterReceiveMsg must be called in the destination processor to receive this message and re-assemble it.

```
RouterSendMsg(dest_proc, source_base, word_length, tag)
void    *source_base;
int     word_length;
unsigned dest_proc;
unsigned tag;
```

8.2.4 RouterReceiveMsg

This function receives fifo length packets which were sent by RouterSendMsg, and asembles them back into the original message. It returns when the word_length specified has been received.

```
void RouterReceiveMsg(base, word_length)
void *base;
int word_length;
```

8.2.5 StartEmptyingRouter

This function indicates that the user is going to stop sending messages, and wait until the router network is emptied of all messages. The processor will receive all messages still in the network. This is a synchronization function, therefore requires that all processors in the partition call this function before the emptying can complete. The router empty operation uses the combine network. Any combines issued after a StartEmptyingRouter will not complete until after RouterEmptied returns true.

```
StartEmptyingRouter()
```

8.2.6 RouterEmptied

Returns true when all processors have called StartEmptyingRouter, and all messages in the router have been received by the appropriate processor (the router is empty).

```
int RouterEmptied()
```

9 Combine (Scan)

The Combine interface allows the pe's to perform a given pattern and combine operation on a unit of data. The allowable patterns are: SCAN_FORWARD, SCAN_BACKWARD, SCAN_REDUCE, and the combines are: OR_SCAN, XOR_SCAN, ADD_SCAN, UADD_SCAN, and MAX_SCAN. For example, this amounts to the pe's OR'ing their data with the data from everyone else in the partition. The result of this operation can be sent to the scalar by using the SCAN_REDUCE combiner.

The combine interface allows these operations to be performed on up to MAX_COMBINE_MSG_WORDS. The operations are performed as though the data is one long word which is length * 32 bits long.

The Scan network allows the pe's to perform forward and backward scans, and it allows the scalar and/or all the pe's to receive the reduce of any of these operations.

9.1 Combine basics

Any combine architecture functionality can be accomplished by applying a combination of the following functions into the proper algorithm. In order to use these functions, you must proclaim yourself an expert of the entire architecture of the CM5. These are the building blocks used to implement the macros and functions in the remainder of this section. It is highly recommended that the macros and functions be used instead of these. If the architecture changes, these functions will most likely become invalidated.

9.1.1 CombineStatus

CombineStatus returns the value in the combine status register. Bit maps and macros are provided to map out any combination of the values available in the status register.

Bitmaps:

- NI_SEND_SPACE_P NI_SEND_SPACE_L
- NI_REC_OK_P NI_REC_OK_L
- NI_SEND_OK_P NI_SEND_OK_L
- NI_SEND_EMPTY_P NI_SEND_EMPTY_L
- NI_REC_LENGTH_LEFT_P NI_REC_LENGTH_LEFT_L
- NI_REC_LENGTH_P NI_REC_LENGTH_L
- NI_COM_OVERFLOW_P NI_COM_OVERFLOW_L

Macros:

- SEND_SPACE(status)
- RECEIVE_OK(status)
- SEND_OK(status)
- SEND_EMPTY(status)
- RECEIVE_LENGTH_LEFT(status)
- RECEIVE_LENGTH(status)
- COMBINE_OVERFLOW(status)

Example of using these Macros:

```
int  Overflow;

Overflow = COMBINE_OVERFLOW(CombineStatus());
```

Example of using the bit maps:

```
#define COMBINE_OVERFLOW(status) \
  (status & (1 << NI_COM_OVERFLOW_P))
```

9.1.2 CombineSendFirst

```
CombineSendFirst(op, dir, length, msg)
int op, dir, length;
unsigned msg;
```

9.1.3 CombineSendWord

Send this word onto combine send fifo.

```
CombineSendWord(msg)
unsigned msg;
```

9.1.4 CombineReceiveWord

Receive a word from the combine receive fifo.

```
unsigned CombineReceiveWord()
```

9.2 Combine functions and macros

9.2.1 CombineSegmentStart

```
SetSegmentStart(value)
int     value;
```

This function specifies to the network that this processor starts a new segment.

9.2.2 CombineSend

CombineSend takes a maximum of MAX_COMBINE_MSG_WORDS, sets up the send first register, and then pushes length words onto the fifo. It then injects the fifo into the network, and returns when the message has been sent.

```
CombineSend(combiner, pattern, data, word_length)
int combiner, pattern, word_length;
void *data;
```

9.2.3 CombineReceive

```
CombineReceive(result)
void *result;
```

CombineReceive copies the data out of the combine receive fifo, and places it into the receive buffer provided. The caller is expected to know how long this data is and what type it is. This function can be called from the scalar or pe.

9.2.4 Combine

This simply does a CombineSend followed by a CombineReceive.

```
Combine(combiner, pattern, data, length, result)
int combiner, pattern, length;
void *data, *result;
```

10 Synchronous messages between Scalar and Pe's

10.1 Sending from Scalar to one PE

Write*ToProcessor is called from the scalar and results in this unit of data being sent to the specified pe. The pe's must be executing a Read*FromScalar simultaneously in order to maintain synchronization. Note that all the pe's will be executing this function, but only the specified one will accept the value. The contents of the variable 'value' will remain unchanged in the pe's which were not recipents of the write.

10.1.1 WriteMsgToProcessor

This function sends an arbitrary message of length words to the specified processor. It requires type casting of the value to (void *) so that the router can push length words onto the fifo, regardless of the actual type of the variable value.

```
WriteMsgToProcessor(Pe, Value, Length)
```

```
word     Pe;
void           *Value;
int Length;
```

The remaining write to processor functions take a known type and send it. These are macros which execute faster as the length is not required, and no type casting is necessary. WRITE MSG TO PROCESSOR could be used for all these functions instead.

10.1.2 Write*ToProcessor

```
WriteIntToProcessor(Pe, Value)
unsigned int    Pe;
int             Value;


WriteUintToProcessor(Pe, Value)
unsigned int    Pe;
unsigned int    Value;

WriteFloatToProcessor(Pe, Value)
unsigned int    Pe;
float           Value;

WriteDoubleToProcessor(Pe, Value)
unsigned int    Pe;
double          Value;
```

10.1.3 ReadMsgFromScalar

```
ReadMsgFromScalar(MsgPtr, MaxLength)
void *MsgPtr;
int MaxLength;
```

10.1.4 Read*FromScalar

```
Note: change these to return value;

ReadWordFromScalar(Value)
```

```
int       Value;

ReadUwordFromScalar(Value)
word      Value;

ReadFloatFromScalar(Value)
float     Value;

ReadDoubleFromScalar(Value)
double    Value;
```

10.2 Sending from one PE to Scalar

Read*FromProcessor is called by the scalar and results in this unit of data being received from the specified pe. The pe's must be executing a Write*ToScalar to maintain synchronization.

10.2.1 ReadMsgFromProcessor

```
unsigned int *ReadMsgFromProcessor(Pe, Length)
word Pe;
int Length;
```

10.2.2 Read*FromProcessor

```
int  ReadWordFromProcessor(Pe)
word Pe;

unsigned int  ReadUwordFromProcessor(Pe)
word Pe;

float ReadFloatFromProcessor(Pe)
word Pe;


double ReadDoubleFromProcessor(Pe)
word Pe;
```

10.2.3 WriteMsgToScalar

```
WriteMsgToScalar(Value, Length)
```

```
unsigned int    *Value;
int Length;
```

10.2.4 Write*ToScalar

```
WriteWordToScalar(value)
int     value;

WriteUwordToScalar(value)
word    value;

WriteFloatToScalar(value)
float   value;

WriteDoubleToScalar(value)
double  value;
```

A Example Code

Code is seperated into seperate files to split scalar code from the pe code. The following file is the scalar code.

combine_test.c

```
The following file is all of the pe code
combine_test_pe.c
```

B Compiling CandC for execution on simulators and hardware

It is highly recommended to run code on the simulator until it works correctly there. In fact, that is the only choice you have right now!

The simulators are being designed such that code written in CandC will not have to be modified in any way to run on the simulator, and also run on the hardware. The only difference in preparing code to run on the simulator versus preparing code to run on the hardware is to re-compile the code with a different CandC.h, and then link it with the appropriate library. The step of re-compiling with new CandC.h is only necessary because some of the CandC functions are macros.

Compiling the two example programs to run on the process simulator. This simulator provides a C/Saber debug environment for the scalar as well as the pe's in the partition.

```
cc -O -o combines combines.c -lpsim

cc -O -o combines_pe combines_pe.c -lpsim
```

It is more likely that the code will not run correctly at first and will have to be debugged. To debug this code efficiently, you must be familiar with Saber C. Copy saber init file /pl/cm5/psim/testsuite/.saberxwi to your working directory, invoke saber, and load in the two files. You can type 'run' now, and a partition of four pe's will be established. See /pl/cm5/psim/doc/psimtutorial for more on this.