The Bucknell Logical Systems C Handbook

Dr. Daniel C. Hyde
Computer Science Department
Bucknell University
Lewisburg, Pennsylvania 17837
hyde@bucknell.edu

February 8, 1993

Contents

1 Introduction

This manual describes the running of Logical Systems C programs on Bucknell University's Transputer network. The Transputer network consists of four Inmos B014 boards installed in a Sun/3 server called "hydra." The four boards are populated with T800 Transputers. The students access the Transputer network by way of the Department of Computer Science's Sun network which consists of about 40 Sun3s and Sun4s.

While some of the material is implementation dependent, much of the material would be useful to others using Logical System C in a Sun environment or an IBM PC compatable environment.

2 Using Logical Systems Parallel C

You will write programs in a parallel C developed by Logical Systems[1], which compiles on the Sun3s and loads onto the Transputer network. Extensions to the C language have been added for message passing between concurrent processes (much like in Occam). The extensions are in the form of library calls (See pages 177-185 in *Logical Systems C for the Transputer: Version 89.1 User Manual* available from Logical Systems.).

2.1 Running Ordinary C Programs on One Transputer

2.1.1 To Setup, Compile and Run

To run a C program on one Transputer, do the following steps.

1. Source the file `~logicalC/setup` (or put the contents in your `.cshrc` file).

   ```
   % source ~logicalC/setup
   ```

   Below is a copy of Bucknell's setup file:

   ```
   # setup file for Logical Systems C
   # path of files is /home/hydra/local/LogicalC/lsc89.1
   # Installed March 27, 1991 by Dan Hyde
   set path = ($path /home/hydra/local/LogicalC/lsc89.1)
   setenv TMP /usr/tmp
   setenv PPINC /home/hydra/local/LogicalC/lsc89.1/include
   setenv TLIB /home/hydra/local/LogicalC/lsc89.1/library
   setenv LINKNAME /dev/bxiv0
   # the next four lines are for aliases to allow a user to switch from
   # one board to another of the four B014 boards
   alias b1 'setenv LINKNAME /dev/bxiv0'
   alias b2 'setenv LINKNAME /dev/bxiv1'
   alias b3 'setenv LINKNAME /dev/bxiv2'
   alias b4 'setenv LINKNAME /dev/bxiv3'
   ```

2. To compile and link a C file called mine.c do the following on a SUN 3:

   ```
   % tcc mine
   ```

[1]Logical Systems, P. O. Box 1702, Corvallis, OR 97339

This should create a file with a `.tld` extension if there were no compiler errors.

3. To load onto one Transputer, do the following on the sun server `hydra`:

```
% ld-one mine cio
```

where `cio` is the host server interface program. This uses the `.tld` file to load the Transputer, start execution on the Transputer and interact with the screen and keyboard.

2.1.2 Further Details

You can redirect the input by

```
% ld-one mine cio < mydata
```

You can redirect output by

```
% ld-one mine cio > myout
```

To change between the four Transputer boards, type `b1`, `b2`, `b3` or `b4`. These are aliases in the `setup` file.

2.2 Running Concurrent C Programs on One Transputer

2.2.1 Using the LSC Concurrent Routines

To use the concurrent routines, insert the following "include" file:

```
#include <conc.h>
```

2.2.2 Sample Concurrent Program

Below is a Logical Systems C program which allocates three concurrent processes P1, P2, and P3 which all run on one Transputer. The program allocates three channels which are used to send integers from P1 and P2 to P3. P3 uses an ProcAlt to determine which channel has a message then a ChanInInt to read it.

If you know Occam, you can easily write Logical Systems C concurrent programs by using the constructs that correspond to Occam. For channels, you use the ChanAlloc, ChanIn and ChanOut functions. For concurrent processes you use ProcAlloc, ProcFree, ProcAlt and ProcPar functions. Of course, Logical Systems C is more flexible than Occam as you can allocate and free processes at run time, have recursive functions and dynamic memory allocation; all which are not possible in Occam.

```
/************************* file mine.c **************************/
/* Logical Systems C example with three concurrent processes. */
/* P1 and P2 each send an integer to P3.  P3 receives the integers and */
/* displays them. By Dan Hyde, July 24, 1992 */
#include <stdio.h>
#include <conc.h>
#define WS_SIZE 8192    /* work space size */

P1(Process *p, Channel *out)
{
    int y;
    y = 3;
    ChanOutInt(out, y);
}

P2(Process *p, Channel *out)
{
    int z;
    z = 7;
    ChanOutInt(out, z);
}

P3(Process *p, Channel *in1, Channel *in2)
{
    int which, i, x;

    printf("\nStarting Program.\n");
    for (i = 0; i<2; i++){
            which = ProcAlt(in1, in2, NULL);
            if (which == 0) {
                x = ChanInInt(in1);
                printf("in1 = %d\n", x);
            }
            else {
                x = ChanInInt(in2);
                printf("in2 = %d\n", x);
            }
        }
    printf("\nProgram termination.\n");
}
```

```
main()
{
    Process *p1, *p2, *p3;
    Channel *a, *b, *c;

                 /* allocate channels */
    a = ChanAlloc();
    b = ChanAlloc();
    c = ChanAlloc();

                /* allocate process p1, one parameter -- chan a */
    p1 = ProcAlloc(P1, WS_SIZE, 1, a);
                /* allocate process p2, one parameter -- chan b */
    p2 = ProcAlloc(P2, WS_SIZE, 1, b);
                /* allocate process p3, two parameters -- chans a and b */
    p3 = ProcAlloc(P3, WS_SIZE, 2, a, b);

    ProcPar(p1, p2, p3, NULL);

    ProcFree(p1);
    ProcFree(p2);
    ProcFree(p3);
}
```

2.3 Running on a Network of Transputers

2.3.1 Setting Board Topology

Before you can run on multiple Transputers, you must be using board 1 or board 4 and have set the topology of the Transputers' hardware links on the board as described on page 24 of *The Bucknell Transputer Handbook Using Inmos's Toolset*.

2.3.2 Loading a Network of Transputers

To load a network of Transputers, we use `ld-net` instead of `ld-one`. `ld-net` requires a network information file with an `.nif` extension. This file is a simple description of Transputer node numbers, the program to run on each and the connection of the links. See the section on `.nif` files on how to create them.

To run the below program, we use `ld-net` with the `test.nif` file.

```
% ld-net test
```

Notice that we do *not* need the `cio` parameter which we used with `ld-one`.

The "_node_number" (those are underlines!) variable holds the network address for the processor which is running the program. The declaration for "_node_number" is in `conc.h`.

Below is an example LSC program that runs on two Transputers. By using "_node_number," we distinguish at run time what code to execute. This allows us to write one program and load it on both Transputers. Or, we could have easily used two separate files. Notice that we did not need to allocate any channels as we used the predefined channels which correspond to the Transputer's links, e. g., LINK1IN, when we allocated the processes.

```
/* test.c -- Written by John Douglass */
/* Logical Systems C program to time communication between two Transputers */
#include <stdio.h>
#include <stdlib.h>     /* needed for rand function */
#include <conc.h>

#define WS_SIZE 8192
#define TICKSPERSEC 15625
#define NUM 100         /* number of messages to be sent */

process1 (Process *p, Channel *in, Channel *out)
{
    int x, y;
    int B[NUM];

    x = ChanInInt(in);         /* wait for start signal */
    SetTime(0);
    ChanIn(in, (char *) B, sizeof(B));
    y = Time();
    printf("\n Time for communication is %f seconds\n", (float) y / (float)
           TICKSPERSEC);
    printf("\n Received %d random numbers from process 2\n\n", NUM);
    for (x=0; x < NUM; x++)
        printf("\t%d\n", B[x]);

}

process2 (Process *p, Channel *in, Channel *out)
{
    int A[NUM];
    int i;

    for (i=0; i < NUM; i++)
        A[i] = rand();        /* generate NUM random integers */

    ChanOutInt(out, 1);       /* send signal to start timing */
    ChanOut(out, (char *) A, sizeof(A));
}


main()
{
    if (_node_number == 1) {
        Process *p1;
        p1 = ProcAlloc(process1, WS_SIZE, 2, LINK2IN, LINK2OUT);
        ProcPar(p1, NULL);
    }
    else {
        Process *p2;
        p2 = ProcAlloc(process2, WS_SIZE, 2, LINK1IN, LINK1OUT);
        ProcPar(p2, NULL);
    }
}
```

Here is the `test.nif` file for the program.

```
; This is the test.nif file for the test of communications
; John Douglass 6/15/92
1, test, R0, 0,     , 2[1],  ;
2, test, R1,  , 1[2],     ,  ;
```

2.4 LSC Examples

The Logical Systems C system comes with several examples. The below code is a unmodified copy of `exam4.c` in the directory:

```
/home/hydra/local/LogicalC/lsc89.1/example
```

This directory which came with the compiler has other examples you may want to look at. We suggest you look at or try exam1.c, exam4.c to exam9.c and pipe4.nif.

```
/*  exam4.c file */
#include        <stdio.h>
#include        <conc.h>
main(){
Channel                         *chan;
        int                         id = 0;

        if(_node_number == 1)
                {
/*
 *        We are the root Transputer, inform the user as the
 *        various node addresses arrive.  Note that we listen for
 *        input from the bootstrap link also but this doesn't
 *        matter since it shouldn't generate any output!
 */
                printf("Root node ID # was %d\n",_node_number);
                printf("Booted from input channel: %p\n",_boot_chan_in);
                while(1)
                        {
                        chan = LINK0IN + ((int) ProcAlt(LINK0IN,LINK1IN,
                                LINK2IN,LINK3IN,0));
                        printf("\nMessage on input channel: %p\n",chan);
                        ChanIn(chan,(char *) &id,2);
                        printf("Message was node ID # %d\n",id);
                        }
                }
        else
                {
/*
 *        We are not the root, look at all input links and send
 *        any node addresses we get out the link we were
 *        bootstrapped on.
 */
                ChanOut(_boot_chan_out,(char *) &_node_number,2);
                while(1)
```

```
                        {
                        chan = LINK0IN + ((int) ProcAlt(LINK0IN,LINK1IN,
                                LINK2IN,LINK3IN,0));
                        ChanIn(chan,(char *) &id,2);
                        ChanOut(_boot_chan_out,(char *) &id,2);
                        }
                }
        }
```

Here is a .nif file for this example.

```
; Network Information File for exam4.c
; Dan Hyde, April 23, 1991
;
1, ex4, R0, 0, , 2[1],  ;
2, ex4, R1,  , 1[2], ,  ;
```

The above code demonstrates a simple approach to using C on multiple Transputers. We have one process running on each Transputer and we communicate on the link channels directly. This means we don't have to worry about creating new processes or creating new channels. _boot_chan_out and _boot_chan_in are the channels used to boot the Transputer. Therefore, you can be sure they are connected to a neighboring Transputer which is closer to the SUN host.

Notice that by using the node number variable we can write one program which we run on many Transputers. This may be wasteful of memory space but makes programming easier.

2.5 How to Make and Use .nif Files in Logical Systems C

The `.nif` extension on a file stands for *network information file*. This file gives information about Transputer links to the network program loader `ld-net` used by Logical Systems C. It is therefore very important that you create the `.nif` files with careful deliberation.

2.5.1 Form of `.nif` file

The file is organized as a sequence of node descriptions, one per line ending with a ";". The node description will take the following form:

```
node#, program name, parent node, link 0, link 1, link 2, link 3;
```

2.5.2 Field #1: Node Number

The node number can be a number of the user's choice. This number must be a positive integer and may range from 1 to 1000. Non–consecutive numbers may be used but they must be distinct. Node number 0 is reserved to represent the host interface (i.e. the `SUN`). By convention node number 1 should be the root Transputer in the network. The node number provided in the `.nif` file is available to the user during execution as the contents of the "_node_number" external integer variable defined in `conc.h`.

2.5.3 Field #2: Program to Download

Contains the name of the program to download to that particular node. By default, a file name extension of `.tld` is assumed if none is specified.

2.5.4 Field #3: Parent Node

This field contains both the number of the parent node and an indication about whether the "system" or "sub–system" output of the parent node controls the reset for this node. System output is denoted by an R and sub–system output is denoted by an S. No more than one node may be connected to the "system" or "sub–system" output of its parent node. Since node number 0 is defined to be the host interface, a field containing "R0" should be used with the root node to describe who resets it. It is important that there exist a link between a node and it's parent node.

2.5.5 Fields #4-#7: Link Connections

The last four fields in the line are used to specify which node each link is connected to. If a link is unconnected, its entry may be left blank (a comma is still required as a place holder). It is best to explicitly declare which link of the node you are connected to, this is done using the "[ ]" notation. For example:

```
1, zip, R0, 0, 2[1], 3[2], ;
```

In this example, the root node (1) is loaded with program "zip.tld", is reset by the system output of the host, and its link 0 is connected to the host, link 1 is connect to node 2 link 1, link 2 is connected to node 3 link 2, and link 3 is left unconnected.

2.5.6 Planning

Careful planning of the `.nif` file at the beginning will increase your understanding of the file and possibly prevent you from experiencing errors in your program which occur not in the code but in the way in which you set the network links. The first phase of this planning is to draw a diagram of your program showing the communication paths between your processes. In this example there are four processes, two which generate data and a third which merges this data and a fourth which prints it out.

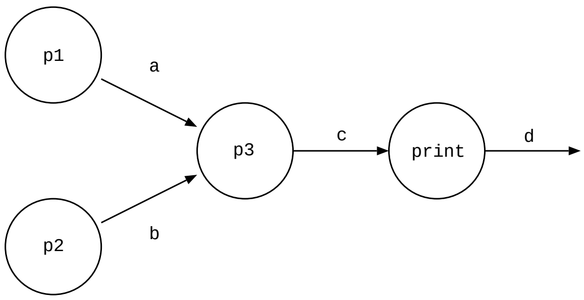

Figure 1: Process Diagram

Now carefully map this diagram to a diagram of Transputers which each have four links. Remember that the node one Transputer is the one which can do i/o, and that it's link 0, must go to node 0. Remember to set the topology of the Transputers.[2]

[2]see reference in Transputer Manual, page 24.

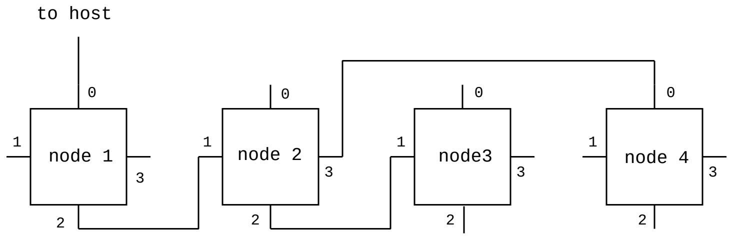

Figure 2: Transputer mapping

2.5.7 Example .nif File

The following file would be a sample `.nif` file for the above example.

```
; .nif file for the above diagram.
1, print, R0,    0,     , 2[1],     ;
2, p3   , R1,     , 1[2], 3[1], 4[0];
3, p1   , R2,     , 2[2],     ,     ;
4, p2   , S2, 2[3],     ,     ,     ;
```

2.6 Generating Random Numbers and Timing Programs in LSC

This section is for the student who needs either to generate random values in Logical Systems C or to time code which is written in Logical Systems C. For reference see *Logical Systems C for the Transputer: Version 89.1 User Manual*.

2.6.1 Random Numbers

Generating numbers in Logical Systems is very simple. First the user must include `stdlib.h` in his or her program. Defined within this library are three useful functions `srand()`, `rand()`, and `frand()`. The first of these functions takes an unsigned integer as a parameter. This function will set the seed for the pseudo-random number generator. The functions `rand()` and `frand()` return an integer random number and a floating-point random number in the range $0.0 \leq x < 1.0$, respectively.

2.6.2 Timing

Timing sections of code is essentially very easy in Logical Systems C. Two functions, `SetTime()` and `Time()`[3] are provided in order to help the programmer. The first, `SetTime` takes an integer as a parameter, this integer is what the timer in ticks for the current priority level will be set to. The second `Time` takes no parameters but instead returns the value of the timer (in ticks) for the current priority level. These two functions are defined in `conc.h` which must be included in the program.

The only other bit of information a person needs to know is that the timer runs at different rates for high (1 μS/tick) and low priority (64 μS/tick) processes. This means that there are 1,000,000 ticks per second and 15,625 ticks per second respectively.

[3]You must use a capital "T" on this function name as `time()` is a different function.

2.6.3 Sample Code

Here is an working example which demonstrates both random number generation and timing of code.

```
#include <stdio.h>
#include <conc.h>
#include <stdlib.h>

#define TICKSPERSEC 15625  /* for a low priority process */

main()
{
        int runtime;  /* will store the elapsed time */
        int A[100];   /* array for random values     */
        int result;   /* temporary result            */
        int i, j;     /* loop counters               */

        for (i=0; i < 100; i++)
             A[i] = rand();  /* store rnd number in A[i] */

        printf("\n About to reset the timer\n");
        SetTime(0);   /* reset timer to zero */
        for (i=0; i < 1000; i++)
             for (j=0; j < 100; j++)
                  result = A[j] * 5;
        runtime = Time();
        printf("The runtime is %f seconds\n", (float) runtime / (float)
               TICKSPERSEC);
}
```

2.7 Debugging LSC Programs

Output and input statements, e. g., printf and scanf, may only occur in the program running on the root Transputer. Also, all of the printf statements should appear in only one concurrent process.

When debugging concurrent LSC programs on one Transputer, one may insert printfs in any concurrent process as long as no other process tries to print. It is best to have a screen manager process which collects the debugging messages by way of a vector of channels and displays them on the screen.

Debugging on multiple Transputers is more difficult. Always debug a concurrent version of the program on one Transputer before moving the program to a network of Transputers. If you design your program with this move in mind, the move should be straight-forward.

When debugging programs on multiple Transputers, always double check your `.nif` file carefully. If your program hangs, concentrate first on getting the communication working between the Transputers. Once that is working, you can send debugging messages to the screen manager on the root Transputer.

2.8 Common Pitfalls in Logical Systems C

The following four sections were written by John Douglass of Bucknell University's class of 1992.

2.9 Introduction

This section documents some pitfalls which were discovered during a first experience with Logical Systems C. It assumes the reader has a basic knowledge of C and has available as reference Chapters 14, 15, and 16 of *Logical Systems C for the Transputer: Version 89.1 User Manual*. A previous knowledge of Occam is helpful but not required.

2.10 Student Misunderstandings

After having read Chapters 14 and 15 of *Logical Systems C for the Transputer: Version 89.1 User Manual*, I felt that I was ready to begin programming. This section of the Handbook is designed to help prevent you from making the same errors I did, mostly from not reading carefully enough (and to save you the hours of debugging which I took to find the errors).

2.10.1 The ProcAlloc Statement

The `ProcAlloc` statement[4] is used to allocate a stack frame so that a new process may be executed. A new process is allocated as follows (from LSC Manual):

```
#include <conc.h>
Process *ProcAlloc (func, sp, nparam, p1, p2, ..., pn)
     int (*func)();
     int sp;
     nparam;
```

`ProcAlloc ()` takes a pointer to a function that contains the code for the process. The parameter `sp` indicates the amount of stack space required for the process. Not being sure of what value to use, I used 8192 as it is used in all the LSC Manual's examples. Using 0 for the default value as suggested in the manual did *not* work.

`nparam` specifies the number of words of space to allocate off the stack space initially for parameters to the function. `p1, p2, ..., pn` is the list of parameters. The name "nparam" is misleading as it is the number of words needed and *not* the number of parameters. For example, a double variable takes two words. From my experience, integer variables and channels take one word. Arrays of integers and channels also take one word as only a pointer is passed.

A sample invocation of this statement:

```
PRINT(Process *p, int a, int b)
{
     int c;
     c = a + b;
     /* print c here */
}
```

[4]See Chapter 14, page 178 and Chapter 16, page 272.

```
Process *print; /* process pointer */
x = 5;
y = 4;
if ((print = ProcAlloc(PRINT, 8192, 2, x, y)) == NULL )
     exit(1);
ProcPar(print, ..., NULL);
x = 4;
y = 5;
ProcPar(print, ..., NULL);
```

This invocation will allocate stack space for the process `PRINT` and assign it to the process pointer `print`. You then start the process by using the `ProcPar` or the ProcRun command.

The important thing to note here is that the execution of the `ProcAlloc` statement binds the actual parameters a and b to the current values of the formal parameters x and y. In the above example, both the first and second executions of `ProcPar` will send the `PRINT` process $x = 4$ and $y = 5$ even though their values have been modified before the second call. One solution to this problem is to invoke `ProcAlloc` again after the modification of x and y. However this solution produces unnecessary overhead.

A better solution to this problem is to use *call by reference*. In C this means that we must place an & before x and y in the `ProcAlloc` statement and a $*$ before instances of a and b in the PRINT function. For example:

```
PRINT(Process *p, int *a, int *b)
{
     int c;
     c = *a + *b;
     /* print c here */
}

Process *print; /* process pointer */
x = 5;
y = 4;
if ((print = ProcAlloc(PRINT, 8192, 2, &x, &y)) == NULL )
     exit(1);
ProcPar(print, ..., NULL);
x = 4;
y = 5;
ProcPar(print, ..., NULL);
```

This invocation binds the addresses of x and y and the $*$ before the a and b dereferences these. Now we can modify x and y and reinvoke `print` and still get the correct values.

2.10.2 ProcRun vs. ProcPar

Another error which I experienced was my incorrect use of `ProcRun`. There are some subtle differences between the way in which `ProcPar` and `ProcRun`[5] which I did not pick up when I read the manual. Take for instance the following code:

```
Process *print; /* process pointer */
x = 5;
y = 4;
```

[5] See Chapter 14, pages 178-179 for use of `ProcPar` and `ProcRun`.

```
if ((print = ProcAlloc(PRINT, 8192, 2, &x, &y)) == NULL )
     exit(1);
ProcPar(print, ..., NULL);
```

`ProcPar` here is used to invoke an instance of the process `PRINT`. Important to note is that this statement will block execution of the calling process until all of the procs in the list have completed. This is different from `ProcRun`, which takes only one process as an argument. `ProcRun` will spawn a process and then allow the continuation of the calling process. This difference can lead to hard to track down errors, for instance:

```
Process *print; /* process pointer */
x = 5;
y = 4;
if ((print = ProcAlloc(PRINT, 8192, 2, &x, &y)) == NULL )
     exit(1);
ProcRun(print);
ProcFree(print);
```

In the above code an instance of the process `PRINT` is spawned by the `ProcRun` execution. However the calling process then continues and executes the `ProcFree` statement which will free the process pointer. This can cause very serious errors. To avoid this error, use the `ProcPar` statement with one process instead, for instance:

```
ProcPar(print, NULL);
ProcFree(print);
```

This will spawn an instance of the print process and then block execution of the calling process until the print process has completed its execution.

2.10.3 ChanIn and ChanOut

`ChanIn` and `ChanOut` read/write *byte* streams. This is an important point to emphasize. For example, the following code would not send out the first 8 integers of the list.

```
int list[10]; /* initialize to some values */

x = 8;
ChanOut(out, (char *) list, x);
```

The code sends out the first two integers in the list. This is because each integer is 4 bytes. To correct this we should use the following statement:

```
ChanOut(out, (char *) list, 4 * x);
```

To send the whole list we can use the `sizeof()` function. For example:

```
ChanOut(out, (char *) list, sizeof(list));
```

Similarly, we must be careful when we use `ChanIn`.

2.11 Logical Systems C Quirks

The following are particular quirks to Logical Systems C running on Transputers which I found unusual and undocumented.

2.11.1 Arrays in Sequential Processing

One of the first problems I encountered when using Logical Systems C for a simple sort program (i. e., no concurrent processes), was the allocation of stackspace for the main. I had defined in the main function a local array A of integers to be of size 2000. Being used to programming on machines with virtual memory this was perfectly normal. However this caused my program to just lock up on the Transputers. After eliminating all other possibilities, it was determined that the problem was the program was running out of memory. I theorize that the stackspace was being stored in the Transputer's on–chip memory and the array exceeded the 4 KBytes allowed.

To solve this problem I simply moved the declaration of the array to make it global. This meant the array A was now being stored in the global stack space. Another solution is to make A static, this also has the effect of storing A not in the stack space but in the global area.

2.11.2 Array Problem Revisited

After having solved my array memory problem, I went on to splitting my sequential sorting program into a concurrent version which used two `sort` processes and a `merge`. The `sort` processes would send the results of sorting half a list to the `merge`, which would merge the two halves into a final list. These two `sort` processes were invoked from one process declaration. For example:

```
Sort1(Process *p, Channel *out, Channel *in)
{
    /* Intent: sorts values and sends them to Merge */
    int i;  /* loop index                      */
    int n;  /* number of elements to generate */
    static int A[MAX];

    n = ChanInInt(in);           /* receive value from merge  */
    GenArray(A, n);              /* generate n rnd elements   */

    ChanOutInt(out, 1);          /* send start timer to merge */
    Sort(A, n);                  /* sort the arrays           */

    for (i=0; i < n; i++)        /* send values out           */
        ChanOutInt(out, A[i]);
    ChanOutInt(out,  -1);        /* send done signal          */
}
main
{
    Channel *a, *b, *y, *z, ...;
    Process *sort1, *sort2, ...;

    /* allocate channels here */

    /* allocate process sort1, two parameters -- chans a and y */
    if ((sort1 = ProcAlloc(Sort1, WS_SIZE, 2, a, y)) == NULL)
```

```
        printf("No Memory for process 'Sort1'\n");

    /* allocate process sort2, two parameters -- chans b and z */
    if ((sort2 = ProcAlloc(Sort1, WS_SIZE, 2, b, z)) == NULL)
        printf("No Memory for process 'Sort2'\n");

    ProcPar(sort1, sort2, merge, NULL);
}
```

This however led to inconsistent results. At times the program would sort and merge the list correctly, at others it would have elements which were incorrectly placed. The problem here was the use of the *static* array. The solution from last section was the cause of the problem this time. Since the function `Sort1` is instantiated in both `sort1` and tt sort2 and the declaration is `static`, the processes share the same storage for array A.

The proper solution here is just to remove the `static` from the declaration. This will not create a memory problem as before since the memory used for the process is the stack space created when you do the `ProcAlloc` command.

The problem here is actually of a much "broader scope—any local function which may be called by two or more concurrent processes must be 're–entrant.' To insure that a function is re–entrant it should be written with no static or global variables and no I/0 memory/registers. In other words, ALL write variables must be volatile or stack based."[6]

2.12 Conclusion

Many of the problems I encountered when using Logical Systems C were simple misunderstandings. It is best to read the documentation *carefully* before trying to program. The above errors are just the problems which I had the most difficultly with and are certainly not all the pitfalls that you will encounter.

[6] This is from a e-mail message by Scott Cannon, Department of Computer Science, Utah State University, who helped answer a question about LSC.