Developing parallel C programs for transputers

INMOS Technical Note 68

INMOS Limited

72-TCH-068-00

Contents

1 Introduction **4**

2 What is the C toolset? **4**
2.1 Introduction 4
2.2 Software toolset summary 4
2.3 Software design cycle - single transputer systems 5
2.4 Software design cycle - multiple transputer systems 7

3 Example problem description **8**
3.1 Introduction 8
3.2 What is configuration? 8

4 Using the Dx214 C toolset **9**
4.1 Introduction 9
4.2 C parallel processing library extensions 9
4.3 Parallel version on one transputer, all In C 10
4.4 Configuring a multi-processor version using icconf 17

5 Conclusions **27**

A Differences between 3L and icc concurrency library **27**

B Useful hints when writing C toolset programs **28**
B.1 What if the program will not run? 29

References **30**

1 Introduction

This document presents a cook book approach to writing parallel C programs for single and networks of transputers.

Using the IMS Dx214 series C toolset the user can write programs which contain many parallel processes, these processes can then be mapped onto a number of transputers using a method called configuration.

Although this technical note is aimed primarily at new users to the INMOS development systems, advanced users may find this useful to come quickly up to speed with the new IMS Dx214 C toolset. Also, users of the IMS D711 3L/INMOS toolset may find this note of interest as there is a section on how to convert existing IMS D711 C code over to the IMS Dx214 toolset.

All of the examples presented were developed using the following equipment:

- IMS D7214 C toolset for IBM-PC.
- IMS B008 Motherboard for IBM-PC.
- IMS B404 TRAM (Two were required for the network example).

2 What is the C toolset?

2.1 Introduction

The IMS Dx214 is a software cross development system for transputers, hosted on a variety of platforms e.g. PC, SUN3, SUN4 or VAX. The development system consists of a set of tools to enable users to write programs for single or multiple transputer networks.

In this section we shall look at the typical development cycle for code running on single or multiple transputers. A brief outline of each tool used at each stage is given. Later on in the document we shall show, by means of worked examples, exactly how to use the tools at each step. For now this serves as a guide to where the tools fit into the grand scheme of things!

2.2 Software toolset summary

The following is a brief list of all the tools provided in the toolset.

Tool	Description
icc	The ANSI C compiler.
icconf	The configurer.
icollect	The code collector.
icvlink	The TCOFF file convertor.
idebug	The network debugger.
idump	The memory dumper. Used by idebug.
iemit	The transputer memory configuration tool.
ieprom	The EPROM program formatter tool.
ilibr	The toolset librarian.
ilink	The toolset linker.
ilist	The binary lister.
imakef	The Makefile generator.
iserver	The host file server.
isim	The IMS T425 simulator.
iskip	The skip loader tool.

Table 1: Summary of toolset components

2.3 Software design cycle - single transputer systems

This section will take a look at the major tools and steps involved in developing software for a single transputer system.

Lets take a look at each of the steps:

Edit

The Edit phase consists of writing the source code for your program, this will include all of the source modules and header files. Any standard text editor can be used for the C toolset, e.g. MicroEmacs, Microsoft Word and even edlin.

Compile

The compilation phase consists of submitting the source code to the compiler. The compiler is called icc, which stands for Inmos C Compiler. It requires the name of the source file and which processor type you wish to compile to.

Linking

Linking is achieved using the ilink tool. This pulls together all of the object files and any libraries that have been created. When compiling for a single transputer then the whole of the C Run Time library is used to give access to the host services.

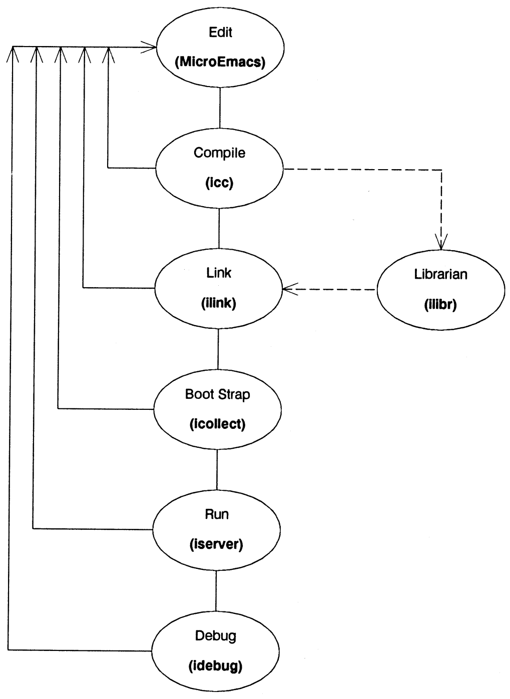

Figure 1: Typical software development route - single transputer systems

Boot Strap

To make the program run a transputer, a Bootstrap must be added. This is a small piece of code which is added to the front of your code and contains instructions to reset the transputer and get ready to load and run your program. After the load has occurred the bootstrap is overlayed and disappears. To add this boot strap we use the icollect tool.

Run

To load the program onto the transputer we use the iserver program. This program sits on the host and enables the transputer to access the hosts services, these include the file system, keyboard and screen.

Debugging

If the program failed to run correctly then we enter the debugging phase of development. The toolset provides an interactive single stepping debugger called idebug. The user can specify breakpoints and trace through the code.

2.4 Software design cycle - multiple transputer systems

In the previous section we saw how to develop programs using single transputers, this section will concentrate on developing programs for multi-transputer systems. Diagram 2 shows the development cycle:

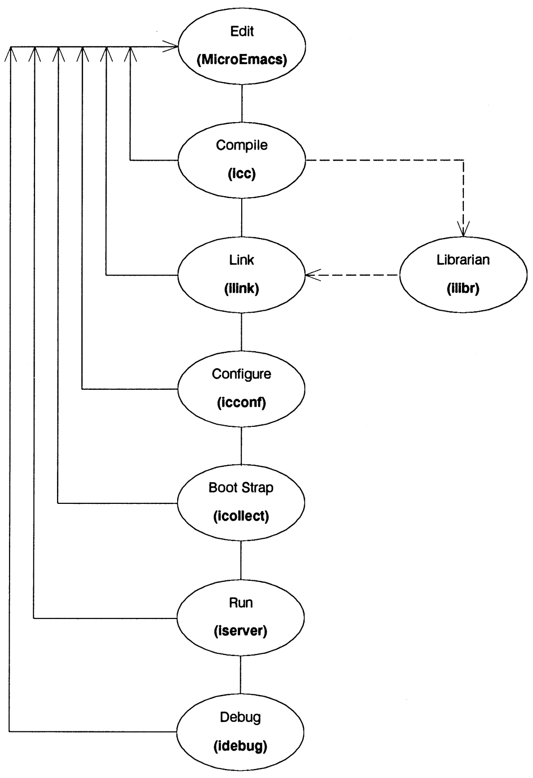

Figure 2: Typical software development route - multiple transputer systems

This development cycle is almost the same as for the single transputer systems, the exception is in the use of the configuration tool icconf.

Configure

Configuration is the step taken to map processes onto processors. The step consists of writing a configuration script which states how this mapping is to occur, we shall take a look at some example scripts later on. The tool for this step is called icconf.

3 Example problem description

3.1 Introduction

Throughout this document one simple example will be used. The system consists of the following components:

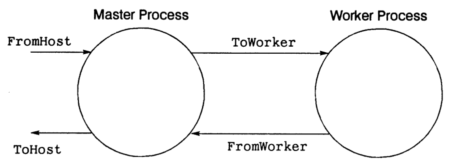

Figure 3: Example two process system

There are two parallel components, Master and Worker, these are called processes. They communicate with each other using channels, these are shown on the diagram as ToWorker and FromWorker.To access the host services, i.e. screen and keyboard, we use the FromHost and ToHost channels.

The two processes are written in C (we could have easily chosen another language e.g. Pascal, Fortran or Ada). The Master process takes input from the keyboard via stdin and passes it onto the Worker via the ToWorker channel. The Worker processes the data and passes it back to Master via the FromWorker channel. The Worker simply takes the keyboard characters and turns them into upper case characters, these are then displayed by the Master process.

3.2 What is configuration?

Configuration is the process by which individual components of the system are mapped onto physical processors. For the simple upper casing example we have two parallel communicating processes. This can be run as two parallel processes on one processor or as two processes running on two separate processors. Figure 4 shows how the two processes are mapped onto two separate processors.

Each transputer has four links, for this simple example we shall only use two links for communication, these are marked on the diagram as LINK1 and LINK2. The LINK0 is connected to the host machine, this is the link by which the system is booted and all communication with the host is passed back through this channel. For our example, all of the keys and output messages are passed via LINK0.

Each INMOS link is bi-directional, this means that you can map one input channel and one output channel onto one physical link.

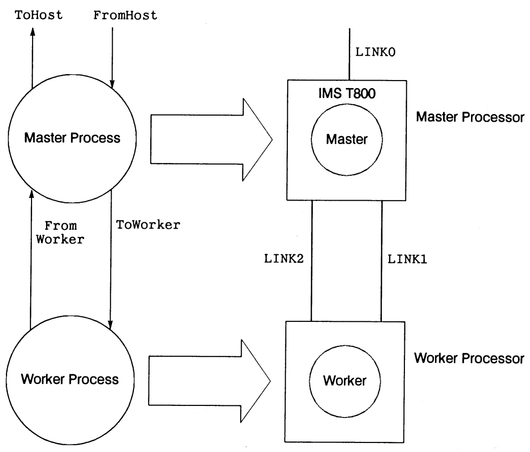

Figure 4: Example two processor system

4 Using the Dx214 C toolset

4.1 Introduction

In this chapter we shall use the IMS Dx214 C toolset to build and configure the upper case example.

We shall use the example program in the following different ways:

- Upper casing on a single transputer, all in C.
- Upper casing on two transputers using the icconf configuration tool.

4.2 C parallel processing library extensions

As well as being an ANSI standard C compiler, icc provides a rich set of parallel processing library calls to enable us to write parallel programs all in C, for more in depth coverage of these routines the reader is referred to [2]. The extensions are similar to the ones provided in the IMS D711 INMOS/3L Parallel C compiler [1].

4.3 Parallel version on one transputer, all In C

In this section we shall look at how we use the C parallel library functions to create a parallel program running on a single transputer. The best way to understand how these libraries work is by the use of an example, so here goes:

```
/*
-- ----------------------------------------------------------------
-- MODULE: Upper Casing Example All in C, using icc.
--
-- FILE : system.c
--
-- NAME : Richard Onyett (Santa Clara, RTC)
--
-- PURPOSE:
--   To create and run two parallel processes to perform the
--   upper casing function.
-- ----------------------------------------------------------------
*/

#include <stdlib.h>
#include <stdio.h>
#include <channel.h>     /* New headers for the channel IO */
#include <process.h>     /* New headers for the processes  */

/*
-- The Master and worker processes are in different files.
*/

extern void Worker( Process *p, Channel *FromMaster, Channel *ToMaster );
extern void Master( Process *p, Channel *FromWorker, Channel *ToWorker );

int main ( void ) {

  Process *WorkerPtr, *MasterPtr; /* Declare some processes */
  Channel *ToWorker, *FromWorker; /* Connect 'em up with channels */

  printf( "Upper Casing EXAMPLE - STARTS\n" );

  /*
  -- Allocate and initialise some channels.
  */

  if (( ToWorker = ChanAlloc()) == NULL ) {
    printf( "ERROR- Cannot allocate space for channel ToWorker\n" );
    exit( EXIT_FAILURE );
  }
```

```
    if (( FromWorker = ChanAlloc()) == NULL ) {
      printf( "ERROR- Cannot allocate space for channel FromWorker\n" );
      exit( EXIT_FAILURE );
    }

    /*
    -- Allocate the processes needed.
    */

    if (( WorkerPtr = ProcAlloc( Worker, 0, 2
                            , ToWorker, FromWorker )) == NULL ) {
      printf( "ERROR- Cannot allocate space for process Worker\n" );
      exit( EXIT_FAILURE );
    }
    if (( MasterPtr = ProcAlloc( Master, 0, 2
                            , FromWorker, ToWorker )) == NULL ) {
      printf( "ERROR- Cannot allocate space for process Master\n" );
      exit( EXIT_FAILURE );
    }

    /*
    -- Now we have some processes, lets run 'em all in parallel...
    -- This will not return until ALL of the processes have finished.
    */

    ProcPar( MasterPtr, WorkerPtr, NULL );

    /*
    -- All processes have successfully terminated, lets say so..
    */

    printf( "Upper Casing EXAMPLE - ENDS\n" );

    exit( EXIT_SUCCESS ); /* I'm Outta here */
  }
```

Setting up a process

This example runs two processes Master and Worker in parallel. The system.c file contains the set up and calls to the two processes. Each process in the system is declared by saying:

```
  #include <process.h>

  Process *WorkerPtr; /* Declare a process */
```

The process is defined as a function call i.e.

```
void Worker( Process *Ptr, Param1, Param2, ... , ParamN ) {
  /* Do some things */
}
```

The Process * parameter must always be supplied, it is needed so that the process can be executed by the system.

To create the process we must use the ProcAlloc function i.e.

```
WorkerPtr = ProcAlloc( Worker, 0, 2, ToWorker, FromWorker );
```

The parameters are as follows:

- `Worker` - The name of the function (process)
- `0` - Default workspace size (4Kbyte on a 32-bit transputer and 1 Kbyte on a 16-bit transputer).
- `2` - Number of parameters to be passed, in this case its two.
- `ToWorker` - First parameter, in this case a channel called ToWorker.
- `FromWorker` - Second parameter, in this case a channel called FromWorker.

Note that `WorkerPtr` is the pointer to our process (or NULL if no space), `ProcAlloc( Worker, 0, 2, ToWorker, FromWorker )` is the function that builds processes.

Running the processes in parallel

Now that we have set up the processes we must run them in parallel. To achieve this we use the `ProcPar` library call i.e.

```
ProcPar( MasterPtr, WorkerPtr, NULL );
```

The parameters to this are as follows:

- `MasterPtr` - The master process.
- `WorkerPtr` - The worker process.
- `NULL` - No More processes.

Channel communications

To enable communication between our two processes we must declare some channels. This is done by saying:

```
Channel *ToWorker, *FromWorker; /* Connect 'em up with channels */
```

We must now allocate some space for this channel by saying:

```
ToWorker = ChanAlloc();
```

This also initialises the channel.

The master process

The master process takes a character from the standard input channel and passes it onto the worker process. The code for this process is as follows:

```
/*
-- ---------------------------------------------------------------
-- MODULE: Simple Parallel C example.
--
-- FILE : master.c
--
-- NAME : Richard Onyett (RTC, Santa Clara)
--
-- PURPOSE:
--   To generate a stream of ascii characters on a channel and
--   pass them to the upper case worker process.
-- ---------------------------------------------------------------
*/

#include <stdio.h>
#include <stdlib.h>
#include <channel.h>
#include <process.h>

/*
-- Declare a procedure called MASTER
*/

void Master( Process *p, Channel *FromWorker, Channel *ToWorker ) {

  int c;
  int Going = 1;

  p = p;              /* Takes care of unused variable warning */
  /*
  -- Intro
  */

  printf( "Master C Process STARTING\n" );
  printf( "Enter some text and it will be upper cased\n" );
  printf( "\nTo quit type EOF (CTRL-Z)\n" );

  while ( Going ) {

    c = getchar();           /* Get a character from the stdin */
    ChanOutInt( ToWorker, c );           /* Pass to the worker */

    if ( c == EOF ) {
      Going = 0;
    } else {
      c = ChanInInt( FromWorker );   /* Get the character back */
      putchar( c );                   /* Output to the screen */
    }
  }

  printf( "Master C Process ENDING\n" );  /* Shake the needles */
                                           /* from your back */
}
```

This is the master procedure. A character is read in and passed along a channel to the worker process. Channel communication is done using the ChanOutInt and ChanInInt library calls.

The worker process

The worker process takes a character from an input channel and converts it to upper case, it then passes the new character back along an output channel to the master process. The code for the worker process is as follows:

```
/*
-- ------------------------------------------------------------
-- MODULE: Simple Parallel C example.
--
-- FILE : worker.c
--
-- NAME : Richard Onyett (RTC, Santa Clara)
--
-- PURPOSE:
--   To receive a stream of ascii characters on a channel and
--   convert them to upper case, whether they want to be or not!!
-- ------------------------------------------------------------
```

```
*/

#include <stdiored.h>
#include <ctype.h>
#include <channel.h>
#include <process.h>

/*
-- Declare the worker process.
*/

void Worker( Process *p, Channel *FromMaster, Channel *ToMaster ) {

  int key;
  int Going = 1;

  p = p;              /* Takes care of unused variable warning */

  while ( Going ) {

    key = ChanInInt( FromMaster );/* Get a key from the master */

    if ( key == EOF ) {
      Going = 0;
    } else {
      ChanOutInt( ToMaster , toupper( key ));  /* Pass it back */
    }
  }
}
```

Again, channel communication is done using the `ChanOutInt` and `ChanInInt` library routines.

Building the upper case example

The C components are built using the following makefile

```
#
# Module : Simple Parallel C Examples
# Name   : Richard Onyett (Santa Clara, RTC)
# File   : makefile
#

CC     = icc
CFLAGS = /t8
SRC    = master.tco worker.tco system.tco
LINK   = ilink /f startup.lnk $(SRC) /t8
BOOT   = icollect system.lku /t
```

```
system.btl: $(SRC)
        $(LINK) /o system.lku
        $(BOOT)

system.tco: system.c
        $(CC) system.c $(CFLAGS)

master.tco: master.c
        $(CC) master.c $(CFLAGS)

worker.tco: worker.c
        $(CC) worker.c $(CFLAGS)
```

To compile we say

```
$ make
```

This will invoke the C compiler (icc) and the linker (ilink), the output of the linker is then used to produce a bootable file which can be downloaded to the transputer.

Running the single processor version To run the program we type:

```
$ iserver /se/sb system.btl
Booting Root Transputer
Upper Casing EXAMPLE - STARTS
Master C Process STARTING
Enter some text and it will be upper cased

To quit type EOF (CTRL-Z)
a
A
hello
HELLO
^Z
Master C Process ENDING
Upper Casing EXAMPLE - ENDS
```

In detail,

- `iserver` - Invoke the host server program.
- `/se/sb system.btl` - Boot the file system.btl to the transputer and monitor the error flag.
- `Booting Root Transputer` - Message from the server as it starts.

4.4 Configuring a multi-processor version using icconf

Introduction

Now that we have the two process version working we shall map this onto two transputers using the configurer. The IMS Dx214 Toolset provides a C like configuration language to specify how the mapping is to occur. The configurer tool is called icconf, which stands for Inmos C Configurer. The physical system was shown earlier in figure 3. The reader should refer to this diagram as it will make this section clearer.

As before, intimate details of the configuration language syntax is ignored, the interested reader is referred to the IMS Dx214 ANSI C toolset User Manual [2].

Introducing some new tools

In section 2.4 we saw how to use the tools to develop a multi transputer program. The main difference between this development and the development for single processors is the use of the configuration tools to map the processes onto processors. The following diagram shows the steps presented in section 2.4, but shows the filename conventions used:

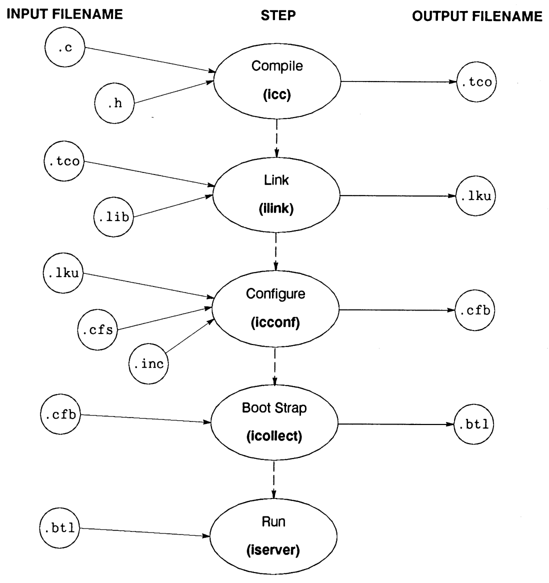

Figure 5: Filename conventions for multiple transputer development

The icconf tool takes, as input, a textual description of the transputer network, with a file suffix of .cfs, and outputs a configuration data file, with a .cfb suffix. This file is then passed into a code collector tool called icollect. The output from this tool is our bootable file that we can run on our network of transputers, using the iserver. The .inc files are predefined descriptions of transputer modules (TRAMS).

The master process

To enable the processes to be configured onto processors, we must add some code to form a process interface. Each process must have a main wrapped around it. The C code for the Master process is as follows:

```
/*
-- -----------------------------------------------------------------
-- MODULE: Simple Parallel C example.
--
-- FILE : master.c
--
-- NAME : Richard Onyett (RTC, Santa Clara)
--
-- PURPOSE:
--   To generate a stream of ascii characters on a channel and
--   pass them to the upper case worker process.
-- -----------------------------------------------------------------
*/

#include <stdio.h>
#include <stdlib.h>
#include <channel.h>
#include <misc.h>

int main( void ) {

  Channel *ToWorker;
  Channel *FromWorker;

  int c;
  int Going = 1;

  /*
  -- Access the external configuration channels.
  */

  FromWorker = (Channel *)get_param( 3 );
  ToWorker   = (Channel *)get_param( 4 );

  /*
  -- Intro
  */

  printf( "Master C Process STARTING\n" );
  printf( "Enter some text and it will be upper cased\n" );
  printf( "\nTo quit type EOF (CTRL-Z)\n" );

  while ( Going ) {

    c = getchar();           /* Get a character from the stdin */
    ChanOutInt( ToWorker, c );           /* Pass to the worker */

    if ( c == EOF ) {
      Going = 0;
    } else {
      c = ChanInInt( FromWorker );   /* Get the character back */
      putchar( c );                  /* Output to the screen */
    }
  }

  printf( "Master C Process ENDING\n" );  /* Shake the needles */
  exit_terminate( EXIT_SUCCESS );            /* from your back */
}
```

The worker process

The code for the worker process is as follows:

```
/*
-- ------------------------------------------------------------------
-- MODULE: Simple Parallel C example.
--
-- FILE : worker.c
--
-- NAME : Richard Onyett (RTC, Santa Clara)
--
-- PURPOSE:
--   To receive a stream of ascii characters on a channel and
--   convert them to upper case, whether they want to be or not!!
-- ------------------------------------------------------------------
*/

#include <stdiored.h>
#include <ctype.h>
#include <channel.h>
#include <misc.h>

int main( void ) {

  int key;
  int Going = 1;

  Channel *FromMaster;
  Channel *ToMaster  ;
```

```
    FromMaster = (Channel *)get_param( 1 );
    ToMaster   = (Channel *)get_param( 2 );

    while ( Going ) {

      key = ChanInInt( FromMaster );/* Get a key from the master */

      if ( key == EOF ) {
        Going = 0;
      } else {
        ChanOutInt( ToMaster , toupper( key ));  /* Pass it back */
      }
    }
  }
```

Notice the use of the `get_param` function to connect to the external configuration channels.

Connecting to the external configuration channels

Using the IMS D711 configuration language, we have used the following code to connect to an external configuration channel (the Dx214 still supports this method for compatibility):

```
  #define OUT_CHAN 2
  #define IN_CHAN  2

  int main( int argc, char* argv[], char* envp[],
            Channel* in[], int inlen,
            Channel* out[], int outlen ) {
    int c;

    /* Put out the official number on channel 2 */
    ChanOutInt( out[ OUT_CHAN ], 38 );

    ... other statements

    /* Pull in a INT on channel 2               */
    c = ChanInInt( in[ IN CHAN ] );
  }
```

For icconf we use the `get_param(param_number)` function call. This returns a pointer to the external configuration channel connected to param number, we shall see later on how we obtain the value for param number. So to obtain a connection to the outside world we write:

```
  Channel *ToWorker;      /* Declare a local channel */
```

```
/*
-- Connect to the external config channel
*/

ToWorker = (Channel *)get_param( 4 );
```

It is worth mentioning that the configuration channels zero and one are reserved for by the C run-time library.

Writing a configuration script

Now we are in a position to look at the configuration script required to build the example on two processors.

The configuration script is held in a file called upc.cfg

```
/*
-- MODULE: Configuration script file for a simple two processor
--                                                      program
-- NAME : Richard Onyett (RTC, Santa Clara)
--
-- PURPOSE:
--   To configure (using INMOS C configuration language) the
--   UPPER CASE program.
*/

/*
--
-- Declare all of the physical hardware in the system. We have
-- 2 x T800 TRAMSs in ours.
--
*/

T800 (memory = 2M) Root ;
T800 (memory = 2M) Slave;

/*
--
-- Tell the world how these two are intimately bonded
--
*/

connect Root.link[0], host;
connect Root.link[3], Slave.link[0];

/*
--
-- Describe the interface between our processes
```

```
--
*/

/*
-- Master task
--                      ---------
--           fs -----> |         | -----> ToWorker
--                     |         |
--           ts <----- |         | <----- FromWorker
--                      ---------
--
*/
process (stacksize = 1k, heapsize = 50k,
         interface ( input fs,          output ts,
                     input FromWorker, output ToWorker )) Master;

/*
-- Worker/Slave task
--                      ---------
--    FromMaster -----> |         |
--                     |         |
--    ToMaster   <----- |         |
--                      ---------
--
--  */
process (stacksize = 1k, heapsize = 50k,
         interface ( input FromMaster, output ToMaster )) Worker;

/*
--
-- Describe how things are wired up
--
*/

input   from_host;
output  to_host ;
connect Master.fs, from_host;
connect Master.ts, to_host;
connect Master.ToWorker,   Worker.FromMaster;
connect Master.FromWorker, Worker.ToMaster  ;

/*
--
-- Pull in the "real code"
--
*/

use "master.lku" for Master;
use "worker.lku" for Worker;
```

```
/*
--
-- Place the processES onto processORS
--
*/

place Master on Root;   /* Run Master on the first transputer  */
place Worker on Slave;  /* Run Worker on the second transputer */

place from_host on host;        /* Wire up to the HOST machine */
place to_host   on host;
place Master.fs on Root.link[0];
place Master.ts on Root.link[0];

/*
--
-- Configured for IMS B008
--
*/

place Master.ToWorker   on Root.link[3];
place Master.FromWorker on Root.link[3];
place Worker.ToMaster   on Slave.link[0];
place Worker.FromMaster on Slave.link[0];
```

Declaring the physical hardware

We must declare all the transputers nodes in the network. In detail,

- `T800` - Type of processor.
- `memory = 2M` - Describe amount of memory available on this processor.
- `Root` - Name of the transputer node.

Showing physical interconnect

We must describe how the links of our root and slave processors are connected. The root node is connected by its Link 0 to the host computer.

In detail,

- `connect Root.link[0], host` - Sign up for the host services.
- `connect Root.link[3], Slave.link[0]` - Connect root node to Slave node.

Interface description

This section describes how our process is joined to all others in the system.We pass in channel parameters and data concerning the size of the stack and heap required.

In detail,

- `process` - Configuration keyword.
- `stacksize = 1k` - Size of the stackspace.
- `heapsize = 50k` - Size of the heap.
- `interface` - Configuration keyword. Here comes the interface description.
- `input fs` - Pass input channel called fs.(Channel parameter 1).
- `output ts` - Pass input channel called ts.(Channel parameter 2). (The first two items in the interface description for a process communicating with the host must be the from server (fs) and to server (ts) channels, in that order.)
- `input FromWorker` - Pass input channel called FromWorker.(Channel parameter 3).
- `output ToWorker` - Pass output channel called ToWorker.(Channel parameter 4).
- `Master` - Name of this process.

This interface section is how channel parameters are passed into the C processes (although the parameters do not have to be channels). The textual declaration of the channels determines which number should be used in the `get_param` call.

WARNING! There is no checking that the parameter number is wired to the correct channel.

Wiring things up

The next section wires up the soft channels to the processes on the transputer nodes.

In detail,

- `connect Master.fs, from host` - Connect to the server on the host.

- `connect Master.ts, to host`
- `connect Master.ToWorker, Worker.FromMaster` - Connect the Master to the Worker.
- `connect Master.FromWorker, Worker.ToMaster`

Pulling In the real code

To pull in the actual compiled and linked code we use the use directive.

- `use "master.lku" for Master` - Pull in the master process.
- `use "worker.lku" for Worker` - Pull in the worker process.

Placing the processes onto processors

The final stage is to place all of our processors onto a transputer node and place the soft channels onto physical transputer links.

- `place Master on Root` - Run Master on the first transputer.
- `place Worker on Slave` - Run Worker on the second transputer.
- `place Master.fs on Root.link[0]` - Connect up the server channels.
- `place Master.ts on Root.link[0]`
- `place Master.ToWorker   on Root.link[3]` - Connect up the Master using channel 2.
- `place Master.FromWorker on Root.link[3]`
- `place Worker.ToMaster   on Slave.link[0]` - Connect up the Worker using channel 1.
- `place Worker.FromMaster on Slave.link[0]`

Building the example

Now we have configuration script written we must build the whole lot! To do this we use the following make file

```
#
# Module : Simple Parallel C Examples
# Name   : Richard Onyett (Santa Clara, RTC)
# File   : makefile
#

CC     = icc
CFLAGS = /t8
OBJS   = master.tco worker.tco
LINK1  = ilink /f startup.lnk master.tco /t8 /o master.lku
LINK2  = ilink /f startrd.lnk worker.tco /t8 /o worker.lku
CONFIG = icconf upc.cfg
BOOT   = icollect upc.cfb

upc.btl: $(OBJS) master.lku worker.lku
        $(CONFIG)
        $(BOOT)

master.lku: master.tco
        $(LINK1)

worker.lku: worker.tco
        $(LINK2)

master.tco: master.c
        $(CC) master.c $(CFLAGS)

worker.tco: worker.c
        $(CC) worker.c $(CFLAGS)
```

The configurer is run by

```
$ icconf upc.cfs
```

In detail,

- `icconf` - Invoke the configurer.
- `upc.cfs` - The name of the configuration script.

The final stage, to collect all of the code, is done by

```
$ icollect upc.lku
```

Running the mufti-processor version

This is done exactly as described previously.

5 Conclusions

This document has taken at look at the various methods for writing parallel C programs. The user has the choice between a program written and configured completely in C or using a Mixed language approach by wrapping the C programs in occam. All of the methods allow the user to initially test the system on a single transputer, once this is working then a configuration script is written and the system is parcelled out onto multiple transputers.

A Differences between 3L and icc concurrency library

Table 2 provides a comparison between the, IMS D711 and IMS Dx214 concurrency libraries.

icc	**3L**		**icc**	**3L**
channel.h	chan.h		ProcGetPriority	thread_priority
Channel	CHAN		PROC_HIGH	THREAD_URGENT
ChanAlloc	Not available		PROC_LOW	THREAD_NOTURG
ChanInit	chan_init		process.h	timer.h
ChanReset	chan_reset		ProcAfter	timer_delay
ChanIn	chan_in_message		ProcWait	timer_wait
ChanInChar	chan_in_byte		ProcTime	timer_now
ChanInInt	chan_in_word		ProcTimeAfter	timer_after
ChanInTimeFail	chan_in_message_t		ProcTimePlus	Not available
ChanInChanFail	Not Available		ProcTimeMinus	Not available
ChanOut	chan_out_message		semaphor.h	sema.h
ChanOutChar	chan_out_byte		SemAlloc	sema_alloc
ChanOutInt	chan_out_word		SemInit	sema_init
ChanOutTimeFail	chan_out_message_t		SemWait	sema_wait
ChanOutChanFail	Not Available		SemSignal	sema_signal
process.h	thread.h		SEMAPHOREINIT	Not Available
Process	Not Available		Not available	sema_signal_n
ProcRun	thread_create		Not available	sema_wait_n
ProcPar	Not Available		Not available	net.h
ProcParList	Not Available		Not available	net_send
ProcAlloc	thread_create		Not available	net_receive
ProcInit	thread_create		Not available	par.h
ProcRunHigh	thread_create		printf	par_printf
ProcRunLow	thread_create		free	par_free
ProcReschedule	Not Available		malloc	par_malloc
ProcParam	Not Available		fprintf	par_fprintf
ProcStop	thread_stop			

Table 2: Comparison of icc vs 3L concurrency library functions

Upgrading code to IMS Dx214 from IMS D711

If you are recompiling code from IMS D711 with the IMS Dx214 C toolset you should include the `conndx11.h` file. This contains macros to convert the IMS D711 channel and thread calls to equivalent IMS Dx214 calls.

B Useful hints when writing C toolset programs

Here are a few simple rules that you may find useful when writing parallel programs using the IMS Dx214 C toolset.

Program design methodology

Writing parallel programs can be a confusing task! Good program design is essential when writing software that may execute on tens or even hundreds of separate processors. Further details on writing concurrent software are available in [3].

Let us assume that you have broken the problem down into parallel blocks, an easy way to test the system is to write a software simulation model. This means running many parallel processes on a single transputer. We have done this already with our upper case example (see section 4), where we used the ProcPar command to run the Master and Worker processes on a single transputer. Once we have this system working we simply write a configuration script and use that to map the processes onto processors. Always have your processes defined in separate files! This means that you can reuse the code in both a simulation and configured modes. For each of your processes add the following:

```
/*
-- Master process
*/

#ifdef CONFIG
int main( void ) {

  Channel *ToWorker;
  Channel *FromWorker;

  /*
  -- Access the external configuration channels.
  */

  FromWorker = (Channel *)get_param( 3 );
  ToWorker   = (Channel *)get_param( 4 );
#else
void Master( Process *Mptr, Channel *FromWorker, Channel *ToWorker ) {
#endif

  ... Rest of the process
}
```

This code segment makes use of a conditional compilation flag called CONFIG, when this is set it will turn the process into one which can be configured. If the flag is not set then the process is assumed to be running on one processor.

B.1 What if the program will not run?

Now that you have compiled, linked, collected, configured and submitted your program to a transputer (using iserver), you discover that It will not function correctly. This usually means that the program runs for a time then simply grinds to a halt, the following is a checklist you may find useful:

- Make sure that any external transputers, such as in an INMOS ITEM rack, have power!
- Use the checkout tools to see if your transputer network is what you thought it was!
- If the program requires setting of any IMS C004s, has this been done?
- Compile with the debugging option (/g) on and use the debugger!
- Run the iserver with the /se option on to test if the error flag has been set.
- Ensure that all ProcPar calls are NULL terminated.
- Ensure that all ProcAlt calls are NULL terminated.
- When you specify that there are n parameters passed to a process make sure all n parameters are passed.
- Make sure you have used `exit_terminate` for configured programs. The server will not terminate until this has been executed!
- Make sure you have allocated some space for a channel.
- Check that the `get_params` are connected to the correct channels in the configuration script.
- Check that the transputers, in the network, have enough memory. Do not specify 1 MByte in the configuration script when only 32K is available on the TRAM.
- Does the processor type in the network match what you have compiled for?
- Make sure that there is enough stack and heap space.

References

[1] IMS D711 3L Parallel C user manual, INMOS Limited, February 1989 (INMOS document number 72-TDS-179-00)

[2] IMS Dx214 ANSI C toolset user manual, INMOS Limited, August 1990 (INMOS document number 72-TDS-224-00)

[3] Program design for concurrent systems, INMOS Technical Note 5, Philip Mattos
INMOS Limited, Bristol.