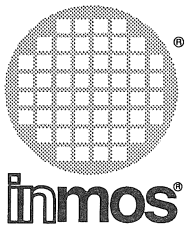


S708
USER GUIDE

72 OEK 227 00

April 1989

72 OEK 227 00

Contents

Contents overview

The user guide

The reference manual

The appendices

A	*Rebuilding the server*	Shows the user how to rebuild the INMOS server.
B	*INMOS standard link access routines.*	Describes the set of C routines for talking to a link from the host computer
C	*Softwire description language*	Syntax of the MMS softwire description language.
D	*Hardwire description language*	Syntax of the MMS hardwire description language.
E	*Edge mappings on the B008*	A Summary of the EDGE mappings on the B008.
F	*The IMS C004 programmable link switch*	A short description of the IMS C004.
G	*The stages of IMS C004 configuration*	Describes the method used to configure a system of motherboards.
H	*Distribution disk*	Lists contents and structure of the distribution disk.
I	*Bibliography*	Lists literature worth referring to.
J	*Glossary*	A glossary of terms used to describe the features of the toolset.

Preface

The S708 software supports the use of an IMS B008 board in an IBM PC AT or IBM PC XT.

The software includes the module motherboard software which can be used to set the programmable switches on the IMS B008 motherboard. These switches determine the topology of the transputers hosted on the motherboard. The module motherboard software also contains a network mapper (*worm*) program which is used to explore the inter-connections of these transputers and provide a means of checking the topology.

A DOS device driver is provided to interface the IMS B008 to the DOS operating system.

Programs are run on the B008 by using the server program provided. The server loads programs to transputer networks and provides file and terminal services to the executing program. Both the module motherboard software and the WORM are executed in this way.

1 How to use the guide

The S708 User Guide is broadly structured into four sections:

- Introduction
- User Guide
- Reference manual
- Appendices

1.1 Introduction

This section gives an overview of the components of the S708 product and its operating requirements.

1.2 User guide

This section provides information on how to use the components of the product

1.3 Reference manual

The reference manual gives the detailed technical information that was not appropriate to the user guide.

1.4 Appendices

The appendices are provided for rapid reference.

2 Introduction

This document relates to the **S708** device driver product for an IBM PC running MS-DOS.The **S708** is a software package consisting of a **MSDOS device driver** and a set of tools for use with the **IMS B008** board product. The device driver, once installed, provides a mechanism for loading transputers via a server.

2.1 Product components

The **S708** is supplied on a standard 360K MS-DOS floppy disk. The contents of the disk should be as follows:

- Device driver.
- INMOS server.
- Module Motherboard Software (MMS).

2.2 Operating requirements

The S708 device driver is intended for use with IBM PC and compatible machines running MSDOS V2.10 or greater.

3 Installation

3.1 Introduction

This section describes how to install the S708 Software on an IBM-PC Compatible.

3.2 Hardware Installation

Before it is possible to successfully install the device driver it is necessary that the **B008** card is installed, in accordance with the *B008 User Reference Guide* which is supplied with the board hardware. Whilst installing the board, take note of the following configuration information which will be required in order to install the device driver properly:

- Base IO Address of the B008 Card
- DMA Channel Number

3.2.1 Copying the Files

First move to the target disk drive and make a suitable directory using the DOS **mkdir** command. This directory can be anywhere on the disk.

Having made the directory, move in to it using the DOS **cd** command, insert the S708 distribution disk in drive **A:** and issue the following command line:

```
xcopy a:\*.* /s
```

If you have an earlier version of MS-DOS or PC-DOS which doesn't support the xcopy command, issue the following commands instead:

```
copy a:\*.*
mkdir iserver
copy a:iserver\*.* iserver
```

The exact contents of the release disk are given in an appendix. The contents of the directory after extraction will be as follows:

```
B008            ..... MMS Hardwire file for the B008
ISERVER.EXE     ..... Executable copy of the iserver
```

```
ISERVER          ..... Directory containing the sources
for the iserver
MMS2.B4          ..... The module motherboard software
PCMMS.ITM        ..... Iterm file for the MMS on a Sun
S708DRIV.SYS     ..... The DOS Device Driver
SOFTWIRE         ..... Example softwire configuration
```

3.2.2 Reconfiguring DOS to accept the driver

Having physically installed both the hardware and the software components in the PC it is necessary to tell DOS to recognise the new device.
This is done by altering a file called `CONFIG.SYS` in the root directory of the boot disk by adding a line describing the device driver.

The syntax of the CONFIG.SYS line for the B008 Driver is:

DEVICE=*pathname* [/A *address*] [/D *chan* | N] [/N *name*]

where: *pathname* is the file DOS pathname of the device driver file.

The pathname parameter is the pathname of the S708DRIV.SYS file on the disk as installed in the previous step.

address is the IO address of the B008 card, as set by the hardware switches on installation.

chan is the DMA channel number as set on the card (0, 1, or 3). If instead of a channel digit, the character 'N' is substituted then the driver will not attempt to use the DMA facilities of the B008. If DMA usage by the B008 has been disabled on the card switches then this parameter must be given as 'N'.

name is the DOS device name which the device will assume. Use this option with care if you intend to use inmos software products which may expect this name to be the default name '`LINK1`'. The name cannot be more than the DOS limit of eight characters.

The following are examples of typical CONFIG.SYS lines for the device driver:

```
DEVICE = C:\S708\S708DRIV.SYS /A 150 /NAME IMSB008 /D N
DEVICE = C:\S708\S708DRIV.SYS /A 200 /D 1
```

Having altered the config.sys file, reboot the machine and run the MMS as described in the relevant section of this document to confirm the correct installation of the device and software.

User guide

72 OEK 227 00

4 Module motherboard software

4.1 Introduction

The range of INMOS Module Motherboards[1] and Modules[2] allow many different configurations of modules and the connections between them to be specified without making physical changes to the boards. The configuration is performed by sending configuration data to the IMS C004 link switches[3] on the board. The MMS (Module Motherboard Software) is designed to make it easy to generate the data needed to configure a system of motherboards.

The MMS provides interactive control of a motherboard or a system of motherboards. It presents a menu-driven interface allowing the user to set up the motherboards and also to create configuration programs for use outside of the MMS.

This chapter describes how to use the hardware and software description languages needed to describe the hardware system and the desired connections within that system, together with a description of the MMS program itself.

The MMS uses a terminal description file called `PCMMS.ITM`. In order for the MMS to access this file it is necessary to set up an environment variable called `ITERM`. This can be done by including the following line in your `autoexec.bat` file:

```
set ITERM=PCMMS.ITM
```

4.2 Getting started

In the rest of this manual it is assumed that the motherboards in use have been set up, and that you are familiar with the user guides for them.

In order to be able to configure the links connecting the IMS C004s on the motherboards the MMS reads files, known as the 'softwire' and 'hardwire' files. The first of these contains a description of the connections that the user wants to make using the programmable link connections. The second contains a description of the hardware configuration of the boards being used.

The hardwire file is needed so that the MMS is able to determine what connections it is possible to make; it contains information on such things as the number of IMS C004s, number of module slots, and the connections between them. Once this description has been set up no changes will have to be made

unless physical changes are made to the motherboard system. If you are using a single IMS B008 or IMS B014 there should not be any need to understand the information in the hardwire file in great detail as the supplied hardwire description files for these boards can be used without modification.

The softwire file is needed to specify both the connections from module to module and from module to edge on a motherboard. Unlike the hardwire file the softwire file will be tailored for the application being run.

You should read section 4.4 on describing softwire connections and study the example files supplied with the MMS before attempting to run the MMS or trying to set up your own softwire description. To get going initially it is probably be easiest to modify a copy of one of the example filesets provided.

4.3 Using the MMS

4.3.1 Running the MMS

To run the MMS, move to the directory in which the contents of the distribution disk were unpacked and type

```
iserver /sb mms2.b4 softwire hardwire
```

replacing *softwire* and *hardwire* by files containing the softwire description and hardwire description respectively. The MMS will display a menu screen and prompt **`key command`**. At this point the user can enter any of the command codes listed on the menu, including **`h`** for help and **`q`** for quit.

4.3.2 Menu options

The menu options available are as follows:

H — Help
Q — Quit
S — Set C004 links
C — Check source files
T — Toggle diagnostics
N — Network mapper
M — Manual command entry
L — Change link numbers
V — View source files
R — Reset subsystem
I — Initialise C004s
B — Create a bootable file
O — Create an occam table

The menu options are described below.

Help

The help option allows the user to call up a help screen for each of the menu options. The help screen for the help option displays some information about the MMS, including implementation limits of number of IMS C004s, IMS T212s, slots, etc. The MMS version number is also displayed.

Quit

Return to operating system.

Set C004 links

The set command performs the IMS C004 setting as specified in the softwire source file.

To carry out this command the MMS first reads the hardwire description, and builds up an internal representation of the motherboards. The MMS then attempts to boot the configuration pipeline with a special worm which allows commands to be sent to the IMS C004s. The MMS then reads the softwire file, and generates and sends the configuration commands to the configuration pipeline.

If errors are detected at any stage, they are reported and the command abandoned.

Check source files

The check source files command is essentially the same as the set command except that no attempt is made to perform the actual configuration of the boards. In this way it is possible to check a set of source files without having the corresponding hardware on-line.

Toggle diagnostics

This toggles the diagnostic mode. In this mode any command sequences that are generated are also displayed on the screen.

Network mapper

The network mapper command sends a worm into the transputer network using the currently set pipe-in link. The mapper is currently able to detect IMS T212s, IMS T414s, IMS T800s and IMS M212s, although 6K bytes of memory is required, and therefore it will not be able to find the IMS T212s in the configuration pipeline as they have no external memory.

Manual command entry

The manual command option allows the user to send IMS C004 command sequences to any IMS C004 specified in the hardwire file. These sequences are of the same form as those generated automatically:

- IMS C004 id
- IMS C004 command
- any parameters required by the command

It is not possible to send the enquire command (BYTE 2) as no facility is provided for returning information from the configuration pipeline.

Change link numbers

The link change options allows the user to change the links which the MMS uses to communicate with the configuration pipeline and the module pipeline. The default settings are:

Link 1 – configuration pipeline
Link 2 – module pipeline

It is not possible to specify the same link for both pipelines.

View source file

The view option allows the user to view the source of the softwire and hardwire files. It prompts for which file to view and which line number within that file to view. That line together with the preceding and following two are then displayed.

Reset subsystem

The reset option asserts the subsystem reset on the host transputer, causing the system of motherboards to be reset. This will not cause the IMS C004 configuration to be lost.

Initialise C004s

The initialise option causes a software reset to be sent to each IMS C004 in the motherboard system. In order to do this the hardwire file is read to determine the number and whereabouts of each of the IMS C004s within the system.

Create a bootable file

The bootable file option is similar to the set option except that the configuration commands generated are written to a file containing a program which will configure the network when it is booted from the server. The generated program expects the configuration pipeline to be connected to the root transputer via the configuration pipeline link set when the program is generated. This configuration program can be used without the MMS being present on the system. When run, the program will either print a message stating that the configuration was successful or unsuccessful.

Create an occam table

The occam table option is similar to the set command except that the configuration commands generated are written to a file in the form of an occam table together with a program which controls the configuration pipeline during the network configuration. This occam table can be sent to the configuration pipeline using the extraordinary link communication procedures[4] to output the table. The table output will fail if the network configuration is not successful.

For example the following piece of occam can be used configure a network, assuming that the table generated by the MMS is contained in the file `mmstable.occ`:

```
-- A procedure to configure module motherboards
-- at run time with a table produced by the mms2
```

```
-- software.

#INCLUDE "hostio.inc"
PROC config (CHAN OF SP fs, ts)

  #INCLUDE "linkaddr.inc"
  #INCLUDE "table.inc"    -- occam table
  #USE "hostio.lib"
  #USE "xlink.lib"

  CHAN OF ANY  from.t2, to.t2:
  PORT OF BYTE analyse, reset:

  PLACE to.t2   AT link1.out:
  PLACE from.t2 AT link1.in:
  PLACE reset   AT (0 >< (MOSTNEG INT)) >> 2:
  PLACE analyse AT (4 >< (MOSTNEG INT)) >> 2:

  VAL ONEms      IS 15:
  VAL fail.delay IS 8000:

  BOOL  failed, failed2:
  BYTE  result:
  INT   time:
  [4]BYTE num:
  TIMER timer:

  PROC Pause ()
    SEQ
      timer ? time
      timer ? AFTER time PLUS (5 * ONEms)
  :
```

```
  SEQ
    analyse ! 0 (BYTE)     -- Reset subsystem
    reset   ! 0 (BYTE)
    Pause ()
    reset   ! 1 (BYTE)
    Pause ()
    reset   ! 0 (BYTE)
    Pause ()

    timer ? time  -- send worm and configuration
    time := time  -- commands in the table.
    PAR
      OutputOrFail.t (to.t2, Table,
      timer, time, failed)
      InputOrFail.t (from.t2, num,
      timer, time, failed2)

    IF
      failed
SEQ
  Reinitialise (to.t2) -- clean up after failure
  so.write.string.nl (fs, ts,
   "Unable to configure T2 chain.")
     TRUE
VAL INT32 n RETYPES num:
  -- print no. of T2s found.
SEQ
  so.write.string (fs, ts,
       "Size of T2 chain: ")
  so.write.int32 (fs, ts, n + 1, 0)
  so.write.string.nl (fs, ts, ".")
:
```

4.4 Describing the software configuration

The following sections describe how to specify the soft connections required on a system of motherboards.

The syntax of both the softwire and hardwire descriptions are described in a modified Backus-Naur Form (BNF). For example,

edge.to.edge.line = **EDGE** *edge.id* **TO EDGE** *edge.id*

This means 'An *edge.to.edge.line* is the keyword **EDGE**, followed by an *edge.id*, followed by the keywords **TO EDGE**, followed by an *edge.id*'.

A vertical bar (|) means 'or', for example:

softwire.line = *slot.to.slot.line*
| *slot.to.edge.line*
| *edge.to.edge.line*

The written structure of the description is specified by the syntax. Each statement normally occupies a single line, and the indentation of each statement forms an intrinsic part of the syntax of the language. For example,

board.softwires.line = **PIPE** *board.id*
{ *softwire.line* }

This means 'A *board.softwires.line* is the keyword **PIPE** followed by a *board.id* followed by zero or more *softwire.lines*, each on a separate line, and indented two spaces further than **PIPE**'. Curly brackets { and } are used to indicate the number of times a syntactic object occurs. { *object* } means, 'zero or more *objects*, each on a separate line'. Similarly $\{_1$ *object* } means, 'one or more *objects*, each on a separate line.' [*object*] means that *object* is optional.

Comments are introduced by a double dash (--), and extend to the end of the line.

Summaries of the syntax of the description languages are given in appendices A and B.

4.4.1 Softwire definition

The softwire connections allow links on modules on a motherboard to be connected to other modules and edges, without requiring a direct hardwired route between the two. Instead the MMS routes the channels via the IMS C004s on the motherboard. It may not be possible to make every possible connection desired. This depends on how the IMS C004s and module slots are physically connected to each other.

A **SOFTWIRE** description has the following basic structure:

```
SOFTWIRE
  PIPE 0
     .
     . soft connections for board 0
     .
  PIPE 1
     .
     . soft connections for board 1
     .
  .
  .
  .
  PIPE n
    .
    . soft connections for board n
    .
END
```

The syntax of a softwire description is:

softwire.description = **SOFTWIRE**
{ *board.softwires.line*}
END

board.softwires.line = **PIPE** *board.id*
{ *softwire.line*}

The softwire lines are specified in three ways:

- Edge to edge connections
- Slot to edge connections
- Slot to slot connections

The syntax for softwire lines is:

softwire.line = *edge.to.edge.line*
| *slot.to.edge.line*
| *slot.to.slot.line*

An edge to edge connection simply specifies that the two edges named are to be connected together. For example,

```
EDGE 4 TO EDGE 7
```

The syntax for an edge to edge line is:

edge.to.edge.line = **EDGE** *edge.id* **TO EDGE** *edge.id*

A slot to edge line specifies that the edge is to be connected to the specified link on the slot. For example,

```
SLOT 3, LINK 3 TO EDGE 6
```

The syntax for a slot to edge line is:

slot.to.edge.line = **SLOT** *slot.id, link.num* **TO EDGE** *edge.id*

A slot to slot line specifies a connection is to be made between a link on one module to a link on another module, for example:

```
SLOT 2, LINK 0 TO SLOT 1, LINK 0
```

specifies that link 0 of slot 2 will be softwired to link 0 of slot 1.

The slot to slot line has another form which includes a **VIA** statement. This form specifies that the connection is to be made via the two edges specifed. This form is really just a shorthand equivalent to two slot to edge lines. For example

```
SLOT 2, LINK 0 TO SLOT 12, LINK 3 VIA EDGE 3, 6
```

is equivalent to the longer form:

```
SLOT 2, LINK 0 TO EDGE 3
SLOT 12, LINK 3 TO EDGE 6
```

It is the user's responsibility to complete the connection by hardwiring the two edge connectors together. The purpose of this is to allow soft connections to be set up indirectly via edge links where the board architecture does not permit direct connection.

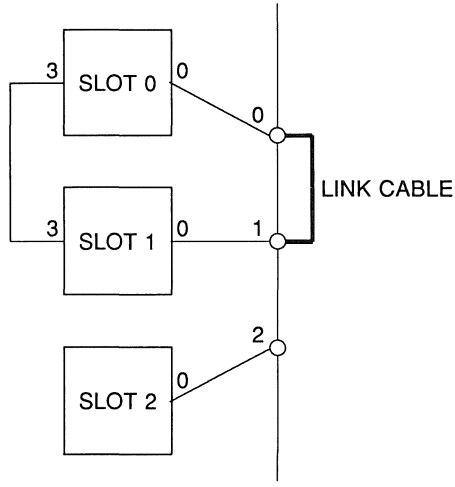
The syntax for slot to slot lines is:

slot.to.slot.line = **SLOT** *slot.id, link.num* **TO SLOT** *slot.id, link.num* [*via.section*]

via.section = **VIA EDGE** *edge.id, edge.id*

As an example of a complete file using these constructs, the following softwires file specifies all the connections in the diagram below:

```
SOFTWIRE
  PIPE 0
    SLOT 0, LINK 3 TO SLOT 1, LINK 3
    SLOT 0, LINK 0 TO SLOT 1, LINK 0 VIA EDGE 0, 1
    SLOT 2, LINK 0 TO EDGE 2
END
```



4.5 Describing the hardware configuration

This section describes how to define the hardware configuration of a motherboard system. The MMS needs to know how the slots, IMS C004s and edges are connected together on the board in order to be able to determine whether a particular set of softwire connections is possible or not.

The following sections will describe what is required in each section of a board definition, including some examples.

4.5.1 Hardwire definition

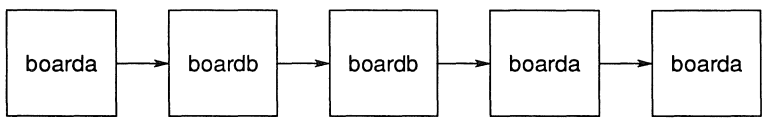
A typical hardwire definition would look something like the following:

```
.
. define boarda
.
DEF boardb
  -- sizes section
  -- t2chain section
  -- hardwires section
PIPE boarda, boardb, boardb, boarda, boarda END
```

The definition consists of two separate parts

- The definition of board types
- The definition of the pipeline

The pipeline definition tells the MMS how the boards in the system are arranged. In the example above we have the following system:



The board definition, on the other hand, specifies the connections within a particular board type. Each section of the board definition will now be described in more detail.

The syntax for a hardwire description is:

hardwire.description = $\{_1$ *board.definition* $\}$
pipeline.description

board.definition = **DEF** *board.name*
sizes
t2.chain
hardwires

pipeline.description = **PIPE** { *board.name* }

Sizes section

The sizes section is used to tell the MMS how many IMS T212s, IMS C004s, slots and edges are present on the board for example:

```
SIZES
  T2    1
  C4    1
  SLOT  3
  EDGE  2
END
```

describes a board with one IMS T212, one IMS C004, three module slots, and two edge connections.

The syntax of the sizes section is:

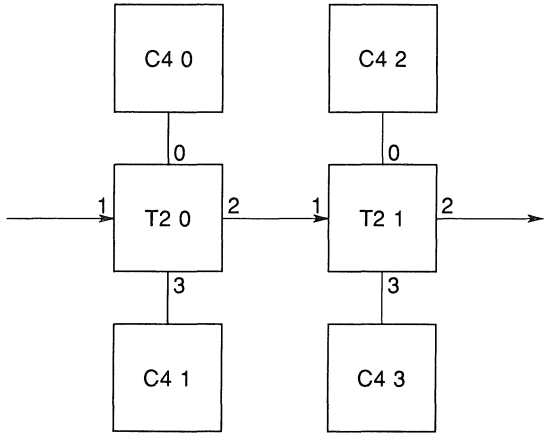
sizes = **SIZES**
  T2 *positive.integer*
  C4 *positive.integer*
  SLOT *positive.integer*
  EDGE *positive.integer*
END

T2 chain section

The T2 chain section tells the MMS how the T2 chain is connected to the IMS C004s. It specifies which links of the IMS T212s are connected to the configuration links of the IMS C004s. For example,

```
T2CHAIN
  T2 0, LINK 0 C4 0
  T2 0, LINK 3 C4 1
  T2 1, LINK 0 C4 2
  T2 1, LINK 3 C4 3
END
```

describes the following system:



The syntax of the T2 chain section is:

t2.chain = **T2CHAIN**
{ *t2.c4.line*}
END

t2.c4.line = **T2** *t2.id, chain.link.num* **C4** *c4.id*

t2.id = *positive.integer*

chain.link.num = 0
| 3

c4.id = 0..31

Hardwire section

The hardwire section describes how the slots, edges and IMS C004s are connected together. A typical structure is as follows:

```
HARDWIRE
  --  pipeline
  --  slots to IMS C004s
  --  edges to IMS C004s
  --  slots to edges
END
```

The sections may appear in any order and lines from each may be freely intermixed, although organising it as above will aid understanding.

The syntax of the hardwire section is:

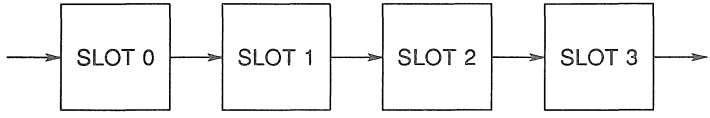
hardwires = HARDWIRE
{ *hardwire.line* }
END

hardwire.line = *slot.to.slot*
| *c4.to.slot*
| *c4.to.edge*
| *slot.to.edge*

The pipeline section describes how the module slots on the motherboard are connected together to form the module pipeline. In general, link 2 of a slot is connected to link 1 of the following slot so that it conforms with the module motherboard architecture[1]. It is not possible to separate the input and output channels of the links. For example,

```
SLOT 0, LINK 2 TO SLOT 1, LINK 1
SLOT 1, LINK 2 TO SLOT 2, LINK 1
SLOT 2, LINK 2 TO SLOT 3, LINK 1
```

describes the following four module pipeline



The syntax of slot to slot lines is:

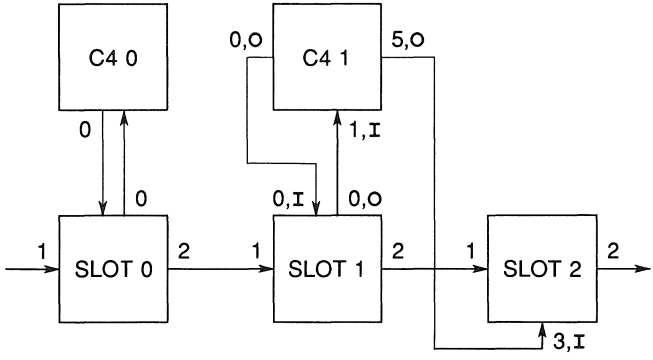
slot.to.slot = **SLOT** *slot.id, link.num* **TO SLOT** *slot.id, link.num*

slot.id = *positive.integer*

The slots to IMS C004s section describes how the non-pipeline links of the slots are connected to the IMS C004 link switches. In general both links 0 and 3 will be taken to an IMS C004. It is possible to specify that the input and output channels of a link are taken to different IMS C004s by including an I or O in the definition. For example,

```
C4 0, LINK 0 TO SLOT 0, LINK 0
C4 1, LINK 0, O TO SLOT 1, LINK 0, I
C4 1, LINK 1, I TO SLOT 1, LINK 0, O
C4 1, LINK 5, O TO SLOT 2, LINK 3, I
```

specifies the following connections



The syntax of IMS C004 to slot lines is:

c4.to.slot = **C4** *c4.id, c4.link.no* [, *i/o*] **TO SLOT** *slot.no, link.num* [, i/o]

i/o = **I**
| **O**

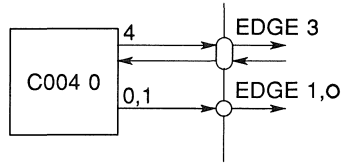
link.num = 0
| 1
| 2
| 3

c4.link.no = *positive.integer*

The edges to IMS C004s section specifies which edges, if any, are connected to the IMS C004s on the board. As with slots to IMS C004s, the input and output channels can be handled separately. For example,

```
C4 0, LINK 0, I TO EDGE 1, O
C4 0, LINK 4 TO EDGE 3
```

describes the following connections



The syntax of IMS C004 to edge lines is:

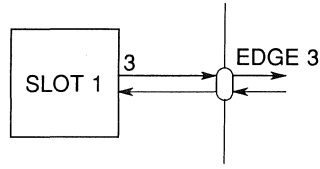
c4.to.edge = **C4** *c4.id, c4.link.no* [, *i/o*] **TO EDGE** *edge.id* [, *i/o*]

edge.id = *positive.integer*

The slots to edges section specifies which edges are connected to slots. It is not possible to separate the input and output channels for slot to edge connections. For example

```
SLOT 1, LINK 3 TO EDGE 3
```

describes the following connection:



The syntax of a slot to edge connection is:

slot.to.edge = **SLOT** *slot.id, link.num* **TO EDGE** *edge.id*

4.6 Error reporting

4.6.1 Errors in the hardwire description

There are a number of different types of error that may be detected by the MMS when reading the hardwire file:

- File Reading Errors
- Syntax Errors
- Range Checking Errors
- Duplication Errors

Most error messages should be self-explanatory.

File reading errors

If the MMS is not able to read the source files an error will be reported and explained. In some cases errors of this type will be detected first as a syntax error and reported as such.

Syntax errors

Any syntax errors in the hardwire file will be reported, producing one of the following types of error message:

'... unexpected symbol found ...'
'... unexpected number found ...'
'... unexpected word found ...'

The symbol that was expected at that point is usually displayed as well, together with the source line number that the error was found on. This line is also displayed in full below the error message.

For example, if the SIZES section of the hardwire file looked like this:

```
SIZES
  T2      2
  C4      4
  SLOT    32
END
```

The MMS would produce the following error message:

```
Error detected in HL1 file at line 4 :
- Unexpected symbol found ('END'). 'EDGE' was expected

Line 4 : END
```

Range checking errors

Numbers outside the following ranges will cause out of range error messages:

- implementation limit restrictions
- values defined in the SIZES section
- link values outside range 0-3

Duplication errors

If any link from a slot, IMS C004 or edge is mentioned more than once in the HARDWIRE section, a duplication error will occur and an error message will be displayed. Similarly, duplicated IMS T212 links or IMS C004 IDs in the T2CHAIN section will give rise to errors.

For example,

```
.
.
C4 0, LINK 4, O TO SLOT 4, LINK 3, I
.
.
C4 0, LINK 4, O TO SLOT 7, LINK 0, I
.
.
```

will produce an error message similar to:

```
Error detected in HL1 file at line x :
- The C004 link in this connection is already involved
in a C004 to slot connection

Line x : C4 0, LINK 4, O TO SLOT 7, LINK 0, I
```

Links may not be checked for duplication in the same order as they appear in the line.

4.6.2 Errors in the softwire description

Many errors in the softwire definition are handled in the same way as the hardwire description. In addition to these errors, however, the MMS will also report soft connections which it is unable to establish.

This can be for one of two reasons:

- A 'hard link' mentioned in a soft connection is not defined as connected anywhere in the hardwire description
- Two hard channels are required to have a soft connection between them, but are connected to different IMS C004s making their connection impossible.

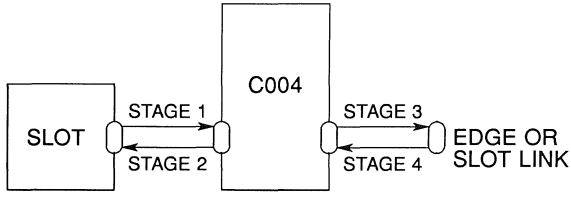
To make it easier to report and correct such errors the MMS error messages break the process of establishing a soft link down into four stages. An error may be detected and reported at any of these stages:

1 From 'from link' output to IMS C004 input

2 From IMS C004 output to 'to link' input

3 From 'to link' output to IMS C004 input

4 From IMS C004 output to 'from link' input



For example, in the following line:

```
SLOT 0, LINK 3 TO SLOT 1, LINK 0
```

the stages are as follows:

1 Check slot 0, link 3 output is connected to a IMS C004 input

2 Check IMS C004 output is connected to slot 1, link 0 input

3 Check slot 1, link 0 is connected to a IMS C004 input

4 Check IMS C004 output is connected to slot 0, link 3 input.

Reference manual

72 OEK 227 00

5 Device driver call definitions

5.1 The system calls available

The interface between the user program and the drivers is via a set of **system calls**. Because of the way a device driver integrates the device handling into the operating system, these system calls are the normal MS-DOS calls for doing input and output to a file or device.

The device driver interfaces the B008 to the operating system as a *character* device, this means that the device can be accessed by **READ** or **WRITE** calls with variable length buffers.

- **OPEN** — Open device for reading or writing.
- **READ** — Read a number of bytes from the link.
- **WRITE** — Write a number of bytes to the link.
- **IOCTL** — Ioctl calls to perform the following:

  1 **ReadFlags** – Read the value of the transputer subsystem error flag and the driver timeout flag, as well as the status of the read and write channels of the link.

  2 **SetFlags** – Sets up the following actions:

    (a) **Reset** – Reset the transputer network.

    (b) **Analyse** – Analyse the transputer network.

    (c) **Settimeout** – Sets the timeout value and enables time-outs.

- **CLOSE** — Close device.

5.1.1 OPEN

The open call to MSDOS, generates a file descriptor to be used to access the device. The open call is passed a file name which MSDOS first checks against its list of character device names. If a match is found then the MSDOS sends all subsequent IO requests associated with the descriptor to the device driver rather than to a file. The IBM Disk Operating System Technical Reference gives a full description of how to call the operating system, the following code segment

shows a typical open call.

```
    name        db      'LINK1', 0

    open_file:  mov     dx, offset name
mov     al, MODE
mov     ah, 3Dh
int     21h

    ;File handle returned in AX
```

or from C

```
    fd = open("LINK1", O_RDWR);
```

There is a problem however with the DOS open call to open a device. If there is no device which has the same name as the parameter then DOS will decide that the open call was to a real file, thus a file descriptor to an open file will be returned rather than to a device. In this way it is not possible by means of the open call to determine whether the device driver is installed or not.

There is however a DOS IOCTL call which returns information about the type of an open file descriptor. One of the bits in the returned status word corresponds to whether the descriptor points to a device or to a plain file. The procedure for opening a descriptor to the link should be to perform an open call on the name of the device, then perform an ioctl call on the new descriptor to check that it does in fact reference the device rather than a file of the same name.

Assuming the open file descriptor in bx, the following code will test whether the desriptor references a device:

```
    mov     ax,4400h    ;IOCTL call, function 0
    mov     bx,<file handle>
    int     21h
    test    dl,80h
    jz      not_a_device
```

The DOS IOCTL call is not usually implemented directly in C runtime libraries.

5.1.2 READ

Having opened the device, the read call allows data to be read from the link. The DOS Read call takes a file descriptor (which should be the descriptor obtained from open) a buffer pointer and a length. The buffer pointer and length are then passed to the device driver for service.

The device driver attempts to read the requisite number of bytes into the buffer, returning control to DOS and hence the user program when:

- All the bytes have been read
- A Timeout occurs see the timeout ioctl call
- The user presses ctrl-break

If all the bytes were read, then read will return the passed length. If only part of the request was read due to either timeout or break detection then the number of bytes read is returned. If a timeout or break occured before any bytes were read then −1 is returned by the read call.

5.1.3 WRITE

The write call allows data to be written to the link adapter. The DOS Write call takes a file descriptor (which should be the descriptor obtained from open) a buffer pointer and a length. The buffer pointer and length are then passed to the device driver for service.

In a similar way to the Read call, the device driver attempts to write the correct number of bytes from the buffer to the link, returning control to DOS and hence the user program when:

- All the bytes have been written
- A Timeout occurs see the timeout ioctl call
- The user presses ctrl-break

If all the bytes were written, then write will return the passed length. If only part of the request was written due to either timeout or break detection then the number of bytes written is returned. If a timeout or break occured before any bytes were written then −1 is returned by the write call.

5.1.4 IOCTL

In DOS, the ioctl call is a way of passing and retrieving control information to the device or the device driver. This information is passed or retrieved using sub-function 02 and 03 of the ioctl (ah=44h) DOS INT 21 system call. The ioctl call is not usually supported directly by C compilers, it must be executed through either asssembly language or a generic operating system call interface of the language.

The call to INT 21h, AH=44h, AL=03h is used to write certain control information to the the device driver. In all cases a 32-bit (4-byte) value is passed into the call. The top 16-bits specify the kind of operation to be done, the bottom sixteen bits contain the parameter to the operation (if any)

Top 16 bits	Bottom 16 bits	meaning
0000h	don't care	Reset the root transputer
0001h	don't care	Assert analyse and reset
0002h	Timeout period (x 0.1 seconds) (0 means no timeout)	Set the timeout period

Example code to reset the root transputer:

```
reset_word  dw  00
    dw  00
reset_link:
    mov     ah, 44h          ;Function 44 - 03
    mov     al, 03h
    mov     bx, <file handle> ; Handle returned by open
    mov     cx, 4       ;Size of status string
    mov     dx, offset cs:reset_word
    push    cs
    pop     ds               ;Address in ds:dx
    int     21h              ;Make DOS call
```

The call to INT 21h, AH=44h, AL=02h is used to read the status information from the device driver. The call should be made, specifying a 32-bit transfer area in CX. On return the transfer area will be as follows:

Bit Number	Meaning
0	The transputer error flag is set
1	A Timeout occured on the last IO operation
2	The transputer link is able to accept at least one byte
3	The transputer link has a least one byte available

Example code to retrieve the error flag:

```
status      dw  00
    dw  00
reset_link:
    mov     ah,44h          ;Function 44 - 02
    mov     al,02h
    mov     bx, <file handle>
    mov     cx,4        ;Size of status string
    mov     dx,offset cs:status_word
    push    cs
    pop     ds              ;Address in ds:dx
    int     21h             ;Make DOS call
    mov     ax,[status]     ;AX=returned status
    and     al,1          ;Only leave error bit
```

6 INMOS server

6.1 Introduction

This section describes the structure and function of the INMOS **server**. The server provides the route by which user programs can be loaded onto a transputer and access host services. The server is written in the C programming language and is supplied in both binary and source form.

6.2 Running the server

To run the host file server use the following command line:

iserver {*option*}

where: *option* is any file server option, given in table 6.1

All options are two letters long and start with a 'S'. None of these options may be used for program parameters. Any other text on the command line is supplied to programs.

If **iserver** alone is typed then the server provides brief help information.

Option	Description
SA	Analyse root transputer and peek 8K of memory from it.
SB *filename*	Boot program contained in named file.
SC *filename*	Copy named file to link.
SE	Terminate if the error flag becomes set.
SI	Produce information messages.
SL *name*	Specify link address or device name.
SR	Reset the root transputer.
SS	Serve link (i.e. provide host support to program communicating on link)
Options must be preceded by '-' for UNIX based toolsets. Options must be preceded by '/' for non-UNIX based toolsets. Spaces between options and the case of letters in the parameters are not significant. Options may be in any order Note: **-SB** *filename* is equivalent to **-SR-SS-SI-SC** *filename*	

Table 6.1 File server options

6.2.1 Loading programs

The name of the file containing the program to be loaded is specified by the SB option. If the file cannot be found an error is reported. When this option is used the board is reset prior to loading the program. When the program has been loaded the server then provides host services to the program.

Using the SB option is equivalent to using the SR, SS, SI and SC options together.

If you want to load a program onto a board without resetting, or if the board is already reset, use the SC option. This should only be done if the transputer being loaded is reset or has a resident program that can understand the file being sent by the server.

If you want the server to terminate after loading the program do not use the SB option. Instead use SC and SR to reset and load the board. The server will then terminate after loading the program.

To reset a transputer run the server with only the SR option.

6.2.2 Specifying link address

The server contains a default address for communicating with boot from link boards. This address may be changed by the SL option. The option is followed by the link address. The link address must be given in hexadecimal.

7 Server protocol definitions

This section provides a technical description of the server's functionality and the protocol used to implement requests to and replies from the server. This information is intended to help those porting the server to a new host machine, or are extending server functionality.

7.1 `iserver` protocol

Every communication, both to and from the server, is a counted array of bytes. The first two bytes are a (little endian) count of the following message length. In the to–server direction, there is a minimum packet length of 8 bytes (i.e. a minimum message length of 6 bytes). In both to and from directions there is a maximum packet length of 512 bytes. A further restriction is that the message must always be an even number of bytes.

s1	s2	message of length s1+(256*s2)

In occam these messages can be routed as `INT16::[]BYTE` protocol.

The server code on the host can take advantage of the fact that it will always be able to read 8 bytes from the link at the start of a transaction.

7.2 Server functionality

This section describes the basic set of server functions. All versions of the `iserver` will support these functions, enabling programs to be ported between any version of the toolset.

The functions implemented by the server are separated into three groups:

1 Interacting with files

2 Interacting with the host environment

3 Interacting with the server itself

In the descriptions that follow, the arguments and results of server calls are listed in the order that they appear in the data part of the protocol packet. The length of a packet is the length of all the items concatenated together, rounded up to an even number of bytes.

All server calls return a result byte as the first item in the return packet. If the operation succeeds the result byte will be zero. If the operation fails the result byte will be non-zero. The result will be one (1) in the special case where the operation failed because it was not implemented[1]. If the result is not zero, some or all of the return values may not be present, resulting in a smaller return packet than if the call was successful.

In the descriptions below, occam types are used to define the format of data items in the packet. All integer types are communicated least significant byte first. Negative integers are represented in 2s complement. Strings and other variable length blocks are introduced by a 16 bit signed count.

7.2.1 File commands

Open files are identified with 32 bit descriptors. There are three predefined open files:

```
0  standard input
1  standard output
2  standard error
```

If one of these is closed it may not be reopened.

[1]Result values between 2 and 127 are defined to have particular meanings by occam server libraries, Result values of 128 or above are specific to the implementation of a server.

Fopen – Open a file

```
Synopsis:    StreamId = Fopen( Name, Type, Mode )

To server:    BYTE            Tag = 10
              INT16::[]BYTE   Name
              BYTE            Type = 1 or 2
              BYTE            Mode = 1...6

From server: BYTE             Result
              INT32           StreamId
```

Fopen opens the file **Name** and, if successful, returns a stream identifier **StreamId**.

Type can take one of two possible values:

1 Binary. The file will contain raw binary bytes

2 Text. The file will be stored as text records. Text files are host–specified.

Mode can have 6 possible values:

1 Open an existing file for input

2 Create a new file, or truncate an existing one, for output

3 Create a new file, or append to an existing one, for output

4 Open an existing file for update (both reading and writing), starting at the beginning of the file

5 Create a new file, or truncate an existing one, for update

6 Create a new file, of append to an existing one, for update

When a file is opened for update (one of the last three modes above) then the resulting stream may be used for input or output. There are restrictions, however. An output operation may not follow an input operation without an intervening **Fseek**, **Ftell** or **Fflush** operation.

The number of streams that may be open at one time is host–specified, but will not be less than eight (including the three predefines).

Fclose – Close a file

```
Synopsis:      Fclose( StreamId )

To server:     BYTE            Tag = 11
               INT32           StreamId

From server:   BYTE            Result
```

Fclose closes a stream `StreamId` which should be open for input or output. Fclose flushes any unwritten data and discards any unread buffered input before closing the stream.

Fread – Read a block of data

```
Synopsis:      Data = Fread( StreamId, Count )

To server:     BYTE            Tag = 12
               INT32           StreamId
               INT16           Count

From server:   BYTE            Result
               INT16::[]BYTE   Data
```

Fread reads `Count` bytes of binary data from the specified stream. Input stops when the specified number of bytes are read, or the end of file is reached, or an error occurs. If `Count` is less than one then no input is done. The stream is left positioned immediately after the data read. If an error occurs the stream position is undefined.

`Result` is always zero. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

Fwrite – Write a block of data

```
Synopsis:      Written = Fwrite( StreamId, Data )

To server:     BYTE            Tag = 13
               INT32           StreamId
               INT16::[]BYTE   Data

From server:   BYTE            Result
               INT16           Written
```

Fwrite writes a given number of bytes of binary data to the specified stream, which should be open for output. If the length of Data is less than zero then no output is done. The position of the stream is advanced by the number of bytes actually written. If an error occurs then the resulting position if undefined.

Fwrite returns the number of bytes actually output in `Written`. `Result` is always zero. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

If the StreamId is `1` (standard output) then the write is automatically flushed.

Fgets – Read a line

```
Synopsis:      Data = Fgets( StreamId, Count )

To server:     BYTE            Tag = 14
               INT32           StreamId
               INT16           Count

From server:   BYTE            Result
               INT16::[]BYTE   Data
```

Fgets reads a line from a stream which must be open for input. Characters are read until end of file is reached, a newline character is seen or the number of characters read is not less than `Count`.

If the input is terminated because a newline is seen then the newline sequence is *not* included in the returned array.

If end of file is encountered and nothing has been read from the stream then Fgets fails.

Fputs – Write a line

```
Synopsis:      Fputs( StreamId, String )

To server:     BYTE            Tag = 15
               INT32           StreamId
               INT16::[]BYTE   String

From server:   BYTE            Result
```

Fputs writes a line of text to a stream which must be open for output. The host-specified convention for newline will be appended to the line and output to the file. The maximum line length is host–specified.

Fflush – Flush a stream

```
Synopsis:      Fflush( StreamId )

To server:     BYTE            Tag = 16
               INT32           StreamId

From server:   BYTE            Result
```

Fflush flushes the specified stream, which should be open for output. Any internally buffered data is written to the destination device. The stream remains open.

Fseek – Set position in a file

```
Synopsis:      Fseek( StreamId, Offset, Origin )

To server:     BYTE            Tag = 17
               INT32           StreamId
               INT32           Offset
               INT32           Origin

From server:   BYTE            Result
```

Fseek sets the file position for the specified stream. A subsequent read or write will access data at the new position.

For a binary file the new position will be `Offset` characters from `Origin` which may take one of three values:

1 Set, the beginning of the file

2 Current, the current position in the file

3 End, the end of the file

For a text stream, Offset must be zero or a value returned by Ftell. If the latter is used then Origin must be set to 1.

Ftell – Find out position in a file

```
Synopsis:      Position = Ftell( StreamId )

To server:     BYTE            Tag = 18
               INT32           StreamId

From server:   BYTE            Result
               INT32           Position
```

Ftell returns the current file position for `StreamId`.

Feof – Test for end of file

```
Synopsis:      Feof( StreamId )

To server:     BYTE            Tag = 19
               INT32           StreamId

From server:   BYTE            Result
```

Feof succeeds if the end of file indicator for `StreamId` is set.

Ferror – Get file error status

```
Synopsis:      ErrorNo, Message = Ferror(StreamId)

To server:     BYTE            Tag = 20
               INT32           StreamId

From server:   BYTE            Result
               INT32           ErrorNo
               INT16::[]BYTE   Message
```

Ferror succeeds if the error indicator for `StreamId` is set. If it is, Ferror returns a host–defined error number and a (possibly null) message corresponding to the last file error on the specified stream.

Remove – Delete a file

```
Synopsis:      Remove( Name )

To server:     BYTE            Tag = 21
               INT16::[]BYTE   Name

From server:   BYTE            Result
```

Remove deletes the named file.

Rename – Rename a file

```
Synopsis:      Rename( OldName, NewName )

To server:     BYTE            Tag = 22
               INT16::[]BYTE   OldName
               INT16::[]BYTE   NewName

From server:   BYTE            Result
```

Rename changes the name of an existing file **OldName** to **NewName**.

7.2.2 Host commands

Getkey – Get a keystroke

```
Synopsis:      Key = GetKey()

To server:     BYTE            Tag = 30

From server:   BYTE            Result
               BYTE            Key
```

GetKey gets a single character from the keyboard. The keystroke is waited on indefinitely and will not be echoed. The effect on any buffered data in the standard input stream is host–defined.

Pollkey – Test for a key

```
Synopsis:      Key = PollKey()

To server:     BYTE            Tag = 31

From server:   BYTE            Result
               BYTE            Key
```

PollKey gets a single character from the keyboard. If a keystroke is not available then PollKey returns immediately with a non-zero result. If a keystroke is available it will not be echoed.The effect on any buffered data in the standard input stream is host–defined.

Getenv – Get environment variable

```
Synopsis:     Value = Getenv( Name )

To server:    BYTE           Tag = 32
              INT16::[]BYTE  Name

From server:  BYTE           Result
              INT16::[]BYTE  Value
```

Getenv returns a host–defined environment string for **Name**. If Name is undefined then Result will be non-zero.

Time – Get the time of day

```
Synopsis:     LocalTime, UTCTime = Time()

To server:    BYTE           Tag = 33

From server:  BYTE           Result
              INT32          LocalTime
              INT32          UTCTime
```

Time returns the local time and *Coordinated Universal Time* if it is available. Both times are expressed as the number of seconds that have elapsed since midnight on 1st January, 1970. If UTC time is unavailable then it will have a value of zero.

System – Run a command

```
Synopsis:     Status = System( Command )

To server:    BYTE           Tag = 34
              INT16::[]BYTE  Command

From server:  BYTE           Result
              INT32          Status
```

System passes the string **Command** to the host command processor for execution. If Command is zero length then System will succeed if there is a command processor. If Command is not null then **Status** is the return value of the command, which is host–defined.

7.2.3 Server commands

Exit – Terminate the server

```
Synopsis:       Exit( Status )

To server:      BYTE            Tag = 35
                INT32           Status

From server:    BYTE            Result
```

Exit terminates the server, which exits returning `Status` to its caller.

If Status has the special value 999999999 then the server will terminate with a host–specific 'success' result.

If Status has the special value –999999999 then the server will terminate with a host–specific 'failure' result.

CommandLine – Retrieve the server command line

```
Synopsis:       String = CommandLine( All )

To server:      BYTE            Tag = 40
                BYTE            All

From server:    BYTE            Result
                INT16::[]BYTE   String
```

CommandLine returns the command line passed to the server on invocation.

If `All` is zero the returned string is the command line, with arguments that the server recognised at startup removed.

If All is non–zero then the string returned is the entire command vector as passed to the server on startup, including the name of the server command itself.

Core – Read peeked memory

```
Synopsis        Data = Core( Offset, Length )

To server:      BYTE            Tag = 41
                INT32           Offset
                INT16           Length

From server:    BYTE            Result
                INT16::[]BYTE   Core
```

Core returns the contents of the root transputer's memory, as peeked from the transputer when the server was invoked with the analyse option.

Core fails if Offset is larger than the amount of memory peeked from the transputer or if the transputer was not analysed.

If (Offset + Length) is larger than the total amount of memory that was peeked then as many bytes as are available from the given offset are returned.

Version – Find out about the server

```
Synopsis:       Id = Version()

To server:      BYTE            Tag = 42

From server:    BYTE            Result
                BYTE            Version
                BYTE            Host
                BYTE            OS
                BYTE            Board
```

Version returns four bytes containing identification information about the server and the host it is running on.

If any of the bytes has the value 0 then that information is not available.

`Version` identifies the server version. The byte value should be divided by ten to yield a version number.

Host identifies the host box. Currently 5 are defined:

1 PC

2 NEC-PC

3 VAX

4 Sun-3

5 Sun-4

OS identifies the host environment. Currently 4 are defined:

1 DOS

2 Helios

3 VMS

4 SunOS

Board identifies the interface board. Currently 7 are defined:

1 B004

2 B008

3 B010

4 B011

5 B014

6 DRX-11

7 QT0

INMOS reserves numbers up to and including 127 for these three fields.

Appendices

A Rebuilding the server

The INMOS server consists of a set of C modules and header files. The source can be altered to be used with various applications. When this is done the server needs to be recompiled and linked using the hosts C compiler, to make this process easier a *Makefile* is supplied.

B INMOS standard link access routines

This appendix describes a standard set of 'C' bindings for talking to transputer links from a host computer. These routines are independent of the host specific software that drives the hardware (e.g. a device driver, or an assembly language routine). INMOS has implemented versions of these routines for all its development boards across several hosts and uses this scheme in its server.

If you wish to create a version of the server for your own board it should only be necessary to replace these functions in the server provided.

B.1 Link initialisation

```
/*  OpenLink
 *
 *  Ready the link associated with 'Name'.
 *  If 'Name' is NULL or "" then
 *     any free link can be used.
 *  Returns any positive integer as a link id or
 *  a negative value if the open fails.
 */

int OpenLink ( Name )
    char *Name;
{
}

/*  CloseLink
 *
 *  Close the active link 'LinkId'.
 *  Returns 1 on success or
 *    negative if the close failed.
 */

int CloseLink (LinkId )
   int LinkId;
{
}
```

B.2 Data operations

```
/*  ReadLink
 *
 *  Read 'Count' chars into 'Buffer'
 *    from the specified link.
 *  LinkId is a valid link identifier,
 *    opened with OpenLink.
 *  'Timeout' is a non negative integer representing
 *    tenths of a second.
 *  A 'Timeout' of zero is an infinite timeout.
 *  The timeout is for the complete operation.
 *  If 'Timeout' is positive then ReadLink may return
 *  having read less than the number of chars asked for.
 *  Returns the number of chars placed in 'Buffer'
 *  (which may be zero) or negative to indicate an error.
 */

int ReadLink ( LinkId, Buffer, Count, Timeout )
  int LinkId;
   char *Buffer;
   unsigned int Count;
   int Timeout;
{
}

/*  WriteLink
 *
 *  Write 'Count' chars from 'Buffer'
 *    to the specified link.
 *  LinkId is a valid link identifier,
 *    opened with OpenLink.
 *  'Timeout' is a non negative integer representing
 *    tenths of a second.
 *  A 'Timeout' of zero is an infinite timeout.
 *  The timeout is for the complete operation.
 *  If 'Timeout' is positive then WriteLink may return
 *  having written less than the number of chars asked for.
 *  Returns the number of chars actually written
 *  (which may be zero) or negative to indicate an error.
 */

int WriteLink ( LinkId, Buffer, Count, Timeout )
  int LinkId;
   char *Buffer;
   unsigned int Count;
   int Timeout;
{
}
```

B.3 Subsystem control

```
/*  ResetLink
 *
 *  Reset the subsystem associated
 *    with the specified link.
 *  Returns 1 if the reset is successful,
 *    negative otherwise.
 */

int ResetLink ( LinkId )
   int LinkId;
{
}

/*  AnalyseLink
 *
 *  Analyse the subsystem associated
 *    with the specified link.
 *  Returns 1 if the analyse is successful,
 *    negative otherwise.
 */

int AnalyseLink ( LinkId )
   int LinkId;
{
}
```

B.4 Error testing

```
/*  TestError
 *
 *  Test the error status associated
 *    with the specified link.
 *  Returns 1 if error is set, 0 if it is not and
 *  negative to indicate an error.
 */

int TestError ( LinkId )
   int LinkId;
{
}
```

B.5 Data ready tests

```
/*  TestRead
 *
 *  Test input status of the link.
 *  Returns 1 if ReadLink will return one byte
 *    without timeout,
 *  0 if it may not and negative to indicate an error.
 */

int TestRead ( LinkId )
   int LinkId;
{
}

/*  TestWrite
 *
 *  Test output status of the link.
 *  Returns 1 if WriteLink can write one byte
 *    without timeout,
 *  0 if it may not and negative to indicate an error.
 */

int TestWrite ( LinkId )
   int LinkId;
{
}
```

C Softwire description language

softwire.description = **SOFTWIRE**
{ *board.softwires*}
END

board.softwires = **PIPE** *board.id*
{ *softwire.line*}

softwire.line = *slot.to.slot.line*
| *slot.to.edge.line*
| *edge.to.edge.line*

slot.to.slot.line = **SLOT** *slot.id, link.num* **TO SLOT** *slot.id, link.num* [*via.section*]

via.section = **VIA EDGE** *edge.id, edge.id*

slot.to.edge.line = **SLOT** *slot.id, link.num* **TO EDGE** *edge.id*

edge.to.edge.line = **EDGE** *edge.id* **TO EDGE** it edges.id

link.num = 0
| 1
| 2
| 3

slot.id = *positive.integer*

edge.id = *positive.integer*

D Hardwire description language

hardwire.description = $\{_1$ *board.definition* $\}$
pipeline.description

positive.integer = a positive integer varying between implementations

pipeline.description = **PIPE** { *board.name* }

board.definition = **DEF** *board.name*
sizes
t2.chain
hardwires

sizes = **SIZES**
T2 *positive.integer*
C4 *positive.integer*
SLOT *positive.integer*
EDGE *positive.integer*
END

t2.chain = **T2CHAIN**
{ *t2.c4.line*}
END

t2.c4.line = **T2** *t2.id, chain.link.num* **C4** *c4.id*

t2.id = *positive.integer*

chain.link.num = 0
| 3

link.num = 0
| 1
| 2
| 3

c4.id = 0..31

hardwires = **HARDWIRE**
{ *hardwire.line* }
END

hardwire.line = *slot.to.slot*
| *c4.to.slot*
| *c4.to.edge*
| *slot.to.edge*

slot.id = *positive.integer*

c4.link.no = *positive.integer*

edge.id = *positive.integer*

slot.to.slot = **SLOT** *slot.id, link.num* **TO SLOT** *slot.id, link.num*

c4.to.slot = **C4** *c4.id, c4.link.no* [, *i/o*] **TO SLOT** *slot.no, link.num* [, *i/o*]

i/o = **I**
| **O**

c4.to.edge = **C4** *c4.id, c4.link.no* [, *i/o*] **TO EDGE** *edge.id* [, *i/o*]

slot.to.edge = **SLOT** *slot.id, link.num* **TO EDGE** *edge.id*

E Edge mappings for the B008

The MMS2 hardwire file for the B008 as supplied in the file 'b008' contains *EDGE* definitions for the output connectors of the B008. The edge definitions map onto the connectors as follows (The connector label on the link break-out board is also given)

Hardwire Mnemonic	*Signal Name*	*Breakout Label*
EDGE 0	EdgeLink0	L0
EDGE 1	EdgeLink1	L1
EDGE 2	EdgeLink2	L2
EDGE 3	EdgeLink3	L3
EDGE 4	EdgeLink4	L4
EDGE 5	EdgeLink5	L5
EDGE 6	EdgeLink6	L6
EDGE 7	EdgeLink7	L7
EDGE 8	PatchLink0 (via patch header)	L8
EDGE 9	PatchLink0 (via patch header)	L9

F The IMS C004 programmable link switch

The IMS C004 programmable link switch provides a full crossbar switch between 32 link inputs and 32 link outputs. It will switch links running at standard transputer speeds (10 and 20 Mbits/sec). The IMS C004 is programmed via a separate serial link called the configuration link.

Each input and output is identified by a number in the range 0 to 31. A configuration message consisting of one, two or three bytes is transmitted on the configuration link. The configuration messages sent to the switch are shown below.

Configuration Message	Function
[0][input][output]	Connects **input** to **output**
[1] [link1] [link2]	Connects **link1** to **link2** by connecting the input of **link1** to the output of **link2** and the input of **link2** to the output of **link1**.
[2] [output]	Enquires which input the **output** is connected to. The IMS C004 responds with the input. The most significant bit of this byte indicates whether the output is connected (bit set high) or disconnected (bit set low). Early versions do not respond to the command.
[3]	This command byte must be sent at the end of every configuration sequence which sets up a connection. The IMS C004 is then ready to accept data on the connected inputs.
[4]	Resets the switch. All outputs are disconnected and held low. This also happens when **Reset** is applied to the IMS C004.
[5] [output]	Output **output** is disconnected and held low.
[6] [link1] [link2]	Disconnects the output of **link1** and the output of **link2**

For more detailed information on the IMS C004 see [3] and [5].

G The stages of IMS C004 configuration

This appendix is designed to give some extra information about the method used to configure the system of motherboards. The configuration takes place in a number of stages, as described below.

A special IMS T212 worm is sent down the configuration pipeline which looks for IMS T212s attached to link 2. The total number of IMS T212s found is passed back up the pipeline to the host transputer. If the number found is different from the number described in the hardwire file then an error is reported and the configuration abandoned.

At this stage a hardware reset of the IMS C004s is performed (if available on the motherboard in use), by writing to the external memory interface of the IMS T212.

The host transputer now sends the identification numbers of the IMS C004s in the system down the pipeline. This enables the worm on any particular IMS T212 to intercept commands intended for the IMS C004s it controls and pass on commands for others.

The configuration pipeline is now in a state where it is able send configuration data to the IMS C004s.

Before sending any configuration data to the pipeline, the host transputer sends a software reset command to each IMS C004 in the system to ensure that the IMS C004s are in a known state.

The configuration data for the network is then send down the configuration pipeline, each command preceded by the identification number of the IMS C004 it is meant for.

The configuration will now be complete and it is possible to reset the system of motherboards without destroying the soft configuration.

H Distribution disk

H.1 Contents of the release disk

The release disk is a standard 360K (40 Tracks, 9 Sectors, Double Sided) MS-DOS floppy disk.

The disk contains the following files:

```
b008
ibmpc.itm
iserver.exe
mms2.b4
softwire
s708driv.sys
```

The directory '`iserver`' on the release disk, contains the following files:

```
b004asm.asm
b004link.c
b008link.c
b011link.c
b014link.c
change.log
filec.c
helios.c
hostc.c
inmos.h
iserver.c
iserver.h
link.c
makefile
manifest
msdosc.c
pack.h
qt0link.c
serverc.c
```

I Bibliography

1 *Module motherboard architecture*, Trevor Watson, Technical note 49, INMOS Limited, Bristol. 1988

2 *Dual inline transputer modules (TRAMs)*, Paul Walker, Technical note 29, INMOS Limited, Bristol. 1988

3 *IMS C004 programmable link switch*, Data sheet, INMOS Limited, Bristol.

4 *Extraordinary use of transputer links*, Roger Shepherd, Technical note 1, INMOS Limited, Bristol. 1987

5 *Design and applications for the IMS C004*, Glenn Hill, Technical note 19, INMOS Limited, Bristol. 1987

6 *Exploring multiple transputer arrays*, Neil Miller, Technical note 24, INMOS Limited, Bristol. 1987

7 *Transputer instruction set: a compiler writer's guide*, Prentice Hall. 1988

J Glossary

Analyse Assert a signal to a transputer to tell it to halt at the next de-scheduling point, and allow the state of the processor to be read. In the context of 'analysing a network', analyse all processors in the network. One of the system control functions on transputer boards.

Bootstrap A transputer program, loaded from ROM or over a link after the transputer has been reset or analysed, which initialises the processor and loads a program for execution (which may be another loader).

Hard channels Channels which are mapped onto links between processors in a transputer network (used in contrast to *Soft channels*).

Module Motherboard Motherboards are provided to interface to various buses. They have slots for inserting INMOS TRAMs and switch chips which can be programmed to connect the transputers in different topologies.

Network A set of transputers connected together using links, as a connected graph (i.e. in such a way that there is a path, via links, and other transputers, from one transputer to every other transputer in the set).

Reset Transputer system initialisation control signal.

Root transputer The transputer connected directly to the host machine.

Server A program running in a host computer attached to a transputer network which provides access to the filing system and terminal I/O of the host computer. The server is normally used to boot up the network as well.

Soft channels Channels declared and used within a process running on a single transputer (used in contrast to *Hard channels*).

TRAM Standard hardware module which can be used to quickly construct systems for a particular application or for a prototype. TRAMs consist of transputers, memory and sometimes application specific circuitry. They conform to a published specification.

Worm A program that will distribute itself through a network of transputers (perhaps with an unknown topology) and allow all the processors in the network to be loaded, tested or analysed.