IMS D703 DSP Development System

72 TDS 105 00

March 1987

1.1 Introduction

The IMS D703 DSP Development System provides a comprehensive environment for the development and analysis of IMS A100-based systems. It comprises an interactive simulator, example implementations of common algorithms, and support software for the IMS B009 Evaluation Board.

This manual describes in detail the internal operations of the **occam 2** software written for the IMS D703 DSP Development System, and how to customise it for specific user requirements.

For descriptions of how to use the IMS D703 as supplied, please refer to the **IMS D703 DSP Development System IBM PC User Manual.**

1.2 Scope

The IMS D703 is distributed in both source code and executable forms. This manual concentrates on describing the internal structure of the **occam 2** source code, to the extent that users can modify the existing code to meet specific requirements. Using the procedures and processes included in the IMS D703 source code, users can produce customised programs for modelling proposed IMS A100 based systems or for controlling dedicated IMS B009 based applications.

Throughout this manual, it is assumed that the reader is thoroughly familiar with the **occam 2** language, the transputer, and the Transputer Development System (TDS). Users not familiar with these topics should first read the appropriate Reference Manuals as listed below.

Many of the advanced features of the IMS B009 Evaluation Board are also described. For those users wishing to produce optimised code for IMS B009 based applications, they should be familiar with the facilities offered by the IMS B009, which are described in the IMS B009 Reference Manual. The applications themselves are described in the IMS D703 User Manual.

Users intending to develop applications based on modification of the IMS D703 must realise that **the source code is provided for information only, and is not supported in any way by INMOS**. The purpose of the source code is to provide experienced programmers with sufficient tested code to enable new applications to be rapidly and reliably written.

1.3 Conventions

Throughout this manual, the following conventions have been adopted:

bold	are names of occam processes, eg. **controller**. This name will be character for character identical with the actual process name in the source code.
`variable`	are names of variables, constants or channels, eg. `repeat.command`. These will be character for character identical with the variable names used within the source code. Where a choice of variables or constants is possible, the options will be expressed as `option1`|`option2`|`option3`... Only one of the options can be used at any time.
name	represents a value which is determined during execution.
$\boxed{\text{keyname}}$	indicates a single key which is labelled as shown, or whose function is that given.

For example:

"Within **address.decoder**, after the address translation takes place, `address` contains *translated address* once $\boxed{\text{return}}$ has been pressed..."

means that the variable `address` is loaded with the value of the *translated address* currently being calculated within the occam process **address.decoder**, once the key marked "Return" has been pressed.

1.4 Relevant Documents

The following documents are relevant to the devices and systems described in this manual. If you do not have any of the items listed below, and you wish to obtain a copy, please contact your local INMOS representative.

- IMS A100 Data Sheet
- IMS A100 Product Overview
- IMS A100 Application Notes
- IMS D703/D704 Product Overview
- IMS B009 Product Overview
- IMS D703 Users Manual
- IMS D703 Reference Manual
- IMS B009 Users Manual
- IMS B009 Reference Manual
- IMS D700 Transputer Development System User and Reference Manuals
- Transputer Reference Manual
- Occam Language Reference Manual
- Transputer Technical Notes
- Occam Technical Notes
- Occam Tutorial

1.5 System Description

The IMS B009 provides a complete environment for the development of IMS A100 based DSP applications. By providing both software emulations and access to physical hardware, the development process can proceed from initial feasibility to advanced system simulation, and for some applications right through to final hardware and software development of production systems.

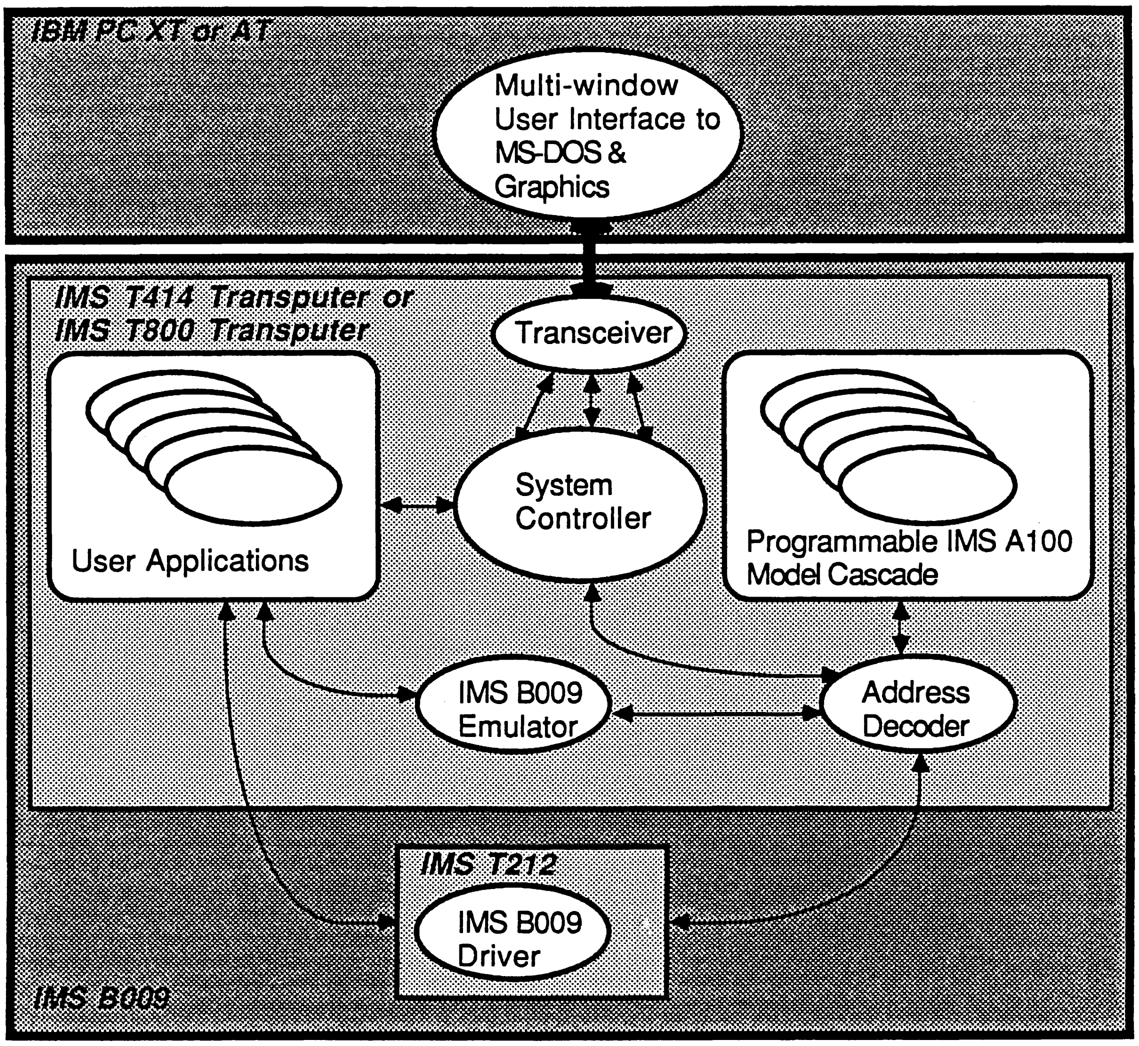
IBM PC XT or AT
Multi-window User Interface to MS-DOS & Graphics
IMS T414 Transputer or IMS T800 Transputer
Transceiver
System Controller
User Applications
Programmable IMS A100 Model Cascade
IMS B009 Emulator
Address Decoder
IMS T212
IMS B009 Driver
IMS B009

Figure 1: IMS D703 Structural Overview

One goal of the IMS D703 is to provide tools for all stages of the development process, so that once simulation is complete, the production software development can be greatly accelerated by reusing parts of the IMS D703 source code itself to produce reliable tested software. In order to make this as straightforward as possible, the source code has been written in such a way that it can be optimised and modified in a highly structured manner. Whilst this will clearly be the case for many IMS B009 applications, the approach taken in the construction of the IMS T212 driver and it's interaction with the host controller can also form a useful starting point for non-transputer based solutions.

The communications structure within the IMS D703 is fundamental to its ease of use and extendability. The extensive use of well defined protocols and controlled dictionaries is a useful model for constructing a wide range of occam applications, and as such provides a working example of how large parallel programs can be constructed. Also, the use of parallelism within the main program demonstrates how sensible localisation of processing and variables makes the program amenable to later distribution across a network of transputers.

The code for the IMS A100 model process is intended to illustrate how to define the boundaries between system and lower levels of simulation. By exploiting the synchronisation inherent in occam channels, localisation of state variables to reflect accessibility within the hardware, and using the overall synchronisation model of occam to eliminate the need for explicit clocks throughout the system, a powerful system simulator can be constructed which models both synchronous and asynchronous behaviour.

The occam processes of the IMS D703 interact with the host environment via an MS-DOS application task. This program is written in Turbo Pascal, and uses the "Turbo Pascal Graphix Toolbox" libraries to provide a simple window environment (Turbo Pascal and Turbo Pascal Graphix Toolbox are trademarks of Borland Internationl). The source code of the front end is provided, but users will need to obtain licenses for Turbo Pascal and Turbo Pascal Graphix Toolbox in order to modify this program. The server is provided simply as an example of how this function can be written; it is not described in detail here. However, from the descriptions of the communication protocols between the occam and pascal programs, the experienced user should have little difficulty in undertstanding the pascal program, and modifying it to suit specific needs.

1.6 Overview of execution flow

The overall function of each of the main processes of the IMS D703 is contained in the **IMS D703 IBM PC Users Manual**. To gain a basic understanding of what happens inside the IMS D703, the following is a summary of execution flow during normal startup, simple command decoding, execution of an application, and termination.

1.6.1 Startup

Each process performs its own initialisation of variables, control loops, etc. However, **command.interpreter** performs the major system initialisation sequence, which comprises:

1 Initialise all command intrepreter booleans, strings, and miscellaneous counters, including text graphics parameters

2 Create the two "plot" windows and single text-only "command" window on the IBM PC, by defining their size and initial text headings, and making requests of the server to create the actual windows in memory and draw them

3 Send a banner to the command window showing the IMS D703 is alive, and what the current version of the code is (this text is also shown in the command window header)

4 See if the IMS T212 link responds to being "peeked" - if so, assume an IMS B009 is present, and bootstrap it with the default driver contained in the file specified in `b009.bootstrap`

5 Request the descriptors of all the **application** processes, and store them for later use

6 Commence the main command input and decode loop

After initialisation, all processes are stalled, pending channel communications, as follows:

- The **A100.models** are waiting for an input on their Standard Microprocessor Interface (SMI) from **address.decoder**, or from their Data Input (DIN) channel (not normally used in the IMS D703 as supplied)
- The **address.decoder** is waiting for a memory access or command from the **controller** or **B009.emulator**
- The **controller** is waiting for a command from the keyboard, which arrives via the **transceiver**
- The **transceiver** is waiting for either data for the screen or graphics windows, or file I/O (from **controller**), or keyboard input from the **MS-DOS front end**

- The **applications** are waiting for the **controller** to accept their first display messages (this is important to understand, and is described later)
- The **B009.emulator** and **B009.driver** (if loaded) are waiting for commands from any **application** (note above)

Other simple processes are normally waiting for communications (eg. the message multiplex processes), but these are inherent to most occam programs, and are not discussed further here.

1.6.2 Command Decode

The **command.interpreter** expects its first command from the keyboard. Subsequent commands can arrive from the keyboard, **application** processes, or the previous command made available for edit if the "`Last`" command was entered. Commands from a keystroke file arrive via the `from.file` channel from the **front end**, where they are redirected by the **read.string** procedure so that the **controller** sees them arrive as if they came from the `from.keyboard` channel. If commands are being received from an **application**, they can be interrupted at the command line level (ie. not within a command line) by any keyboard activity due to a `PRI ALT` within this part of the **command.interpreter**.

The command is split into command and argument text strings by a common parser, then passed to the decoder. The decoder then decodes the first keyword of the command line, and invokes the appropriate command. This sequence comprises the main command decode loop.

During operation, **applications** can request to send a binary command, rather than an ascii command as described above. If so, the rest of the normal command decoding is bypassed, and control immediately passed to the binary decoder, which processes all arguments in a condensed binary form. Note the decode loop is such that binary and ascii commands are freely mixed, to simplify decoding, and allow binary sequences to be interrupted by keyboard activity. Binary and ascii commands are discussed further below.

1.6.3 Reads and Writes

When a user requests a read or write operation, the address and data are first validated to ensure they are 16-bit values by the decoder. These are then converted into `INT16` values, and sent to **address.decoder**. The **address.decoder** analyses the address according to the currently selected addressing regime, and if valid translates the address into the final address to be sent. If The IMS B009 hardware is used, no check is made; the address (and data if a write) are sent directly to the **B009 driver**. If the **A100.model** is active, the address is translated to one suitable for an individual device (ie. $0 \leq address < 128$), and this is sent (with data, if a write) to the appropriate **A100.model** directly via the selected model's channel. Note the **B009.emulator** sends requests to the **A100.model** in the same way.

If a read is requested, the result is returned as a 16-bit value via the address decoder in each case.

1.6.4 Application Execution

When the user requests an **application** to start, the **decoder** simply sets `external.data.active` to `TRUE`, and commences to process commands from the application. These will usually be a mix of interactive commands (ie. commands normally available from the keyboard, known as `ascii` commands) and non-interactive commands (ie. those involving non-keyboard commands such as high speed read/write, plot, file I/O, etc, known as `binary` commands).

An **application** process starts by sending its text descriptor to the **controller**, and performing any initialisation required. It then enters a loop, which sends text strings via the **controller** to the screen describing what the application does, gets parameters from the user, and performs the operation using either the **A100.model** (with `binary` read/write commands to the **controller**), the **B009.emulator** (with `driver` commands directly to the emulator itself), or the **B009.driver** if present (via a direct hardware channel).

The application completes by sending a "stop" command to the **controller**, which tells the decoder to resume accepting commands from the keyboard, rather than from the **application**. The application then simply restarts the execution loop, stalling on the first channel interaction with the **controller**.

1.1 Introduction

Most users wishing to modify the source code of the IMS D703 will commence by writing new **applications**. The source code has been structured to make this procedure as simple as possible. Users are encouraged to produce new applications, as this is the fastest way to exploit the full potential of the IMS D703 and IMS D704 systems. Having become familiar with producing new applications, users can then progress to writing application-specific optimised IMS T212 drivers, to obtain maximum performance from their system. Finally, experienced users may wish to extract useful procedures from the source code of the IMS D703 to create their own system models or application-specific versions of the IMS D703 environment.

Details of the **B009.driver**, and suggestions on how to optimise it, are described later in this manual. This section concentrates on how to write a new user application, and how to create the new binaries required to execute it.

1.2 The "Standard Harness"

User applications all function as described in the "System Overview". Their operation is intended to be as straightforward as possible so that users with little programming experience can successfully write new applications and execute them. In order to speed up this operation, a model of a typical application is provided, which includes all the necessary libraries, call formats, execution flows, and standard operations. This is known as the **standard.harness**, and is provided with the system in a COMMENT fold above the example **applications**. For details of what a COMMENT fold is, please refer to the **TDS Reference Manual.**

The **standard.harness** appears at the top level as shown in Figure 2.

```
PROC standard.harness( CHAN to.controller, from.controller,
                            to.b009.emulator, from.b009.emulator )

  ... STANDARD services

  ... APPLICATION_SPECIFIC variables and PROCs

  VAL descriptor IS "Standard Harness V01.00 (15-mar-87)."

  SEQ

    ... STANDARD initialisation

    WHILE TRUE
      SEQ
        ... describe application and input required parameters
        ... execute required tasks
        stop()
:
```

Figure 2: Standard Harness top level

The sections labelled STANDARD contain code which should not be modified. The STANDARD services provide all the procedure calls required to perform most operations. The STANDARD initialisation performs the appropriate startup communications with the **controller** during IMS D703 startup.

The APPLICATION_SPECIFIC fold is reserved for adding user defined constants, variables, procedures, etc. By localising these to a single fold, users can be sure of which areas are defined by themselves and which by INMOS.

The contents of the main execution loop are entirely up to the user, except that the `stop()` call must appear as the last statement in the loop. The **standard.harness** includes framework code which is similar to that used by all of the example **applications**, so that users can quickly produce applications which are consistent with those supplied with the IMS D703 system.

Applications are written by using procedure calls from the STANDARD services, and combining them with standard occam code. These services are described below. Users planning a new application are advised to first look at some of the example applications, then refer to the following sections to understand the usage of the standard calls.

1.3 How to Recompile the IMS D703 Binaries

Once the necessary modifications have been made within the TDS, or a new application created, the following steps will generate a new binary image ready for execution:

1. Compile the SC fold containing the new application
2. Create an instantiation of the new process in the USER_DEFINED instantiations fold, by taking a copy of one of the other instantiations and modifying it (make sure the channels numbers are unique). Ensure that if you have added any additional channels (eg. peripheral specific I/O channel), they are declared here
3. Compile, configure, and extract the new code for the complete system, with the T4 compiler (or T8 if you are using an IMS T800), with USAGE CHECKING OFF
4. Generate a bootable binary with the "makeboot" utility, calling it a suitable name *newfilename*`.T4` (ie. NOT IMS D703.T4!). The "makeboot" utility is a separate user application supplied with the TDS. To use it, $\boxed{\text{Get}}$ the utility (it is a user process), then position the cursor on the CODE fold of the extracted code. Press $\boxed{\text{Run}}$, and enter the name of the file you wish to be created - this is the name of the final binary image. When it is complete, it will ask you to type $\boxed{\text{space}}$ to return to the TDS. Having done this, you can now leave the TDS.
5. execute the new code using `runsim` *newfilename*`.T4`

Note that this procedure also applies to creating a new IMS B009 driver program, except that the T2 compiler should be used, with EXTENDED TYPES OFF.

The IMS D703 binaries comprise one file for the code executing on the IMS T414, and one executing on the IMS T212 of the IMS B009. Thus, each can be regarded as a totally separate program. To see how to execute alternative drivers within an application, refer to the section on "booting the T2" below.

Note that if several new user applications are created, you may run out of space. To verify this, use the "Config Info" compiler function of the TDS, and check the total size of the extracted code. If it exceeds #100000 (hex), you are in trouble for an IMS B009-2, and must remove something to reduce the code size. Remember that since the TDS cannot know what amount of memory you will be executing with, it will not flag if you exceed the size of available memory on the IMS T414.

1.4 Standard Services

The most important fold of the **standard.harness** is the STANDARD services.These provide access to the full services of the controller, plus access to the **A100.models**, **B009.emulator**, and the IMS B009 itself, either with the standard **B009.driver** or a user written driver.

The protocols used for communication between the **applications** and the **controller** are described in the "Controller" section of this manual. However, most users will not need to worry about this, since the procedures described below provide the services in the form of standard calls.

Within the services fold are three main sections: services for all applications, services specific to address mode 0, and services specific to address modes 1 & 2. Within each fold are the necessary constants and variables required for all services, so the user need not declare them again. Indeed, users are advised to use the existing variables wherever appropriate to minimise code size. In particular, variables such as `str.arg`, `byte.arg` and `real.arg` are useful for temporary variables holding the results of an `enquiry` call.

The following sections describe the procedure calls provided by the STANDARD services.

1.4.1 execute keyboard command

Command: `command( VAL []BYTE` *string* `)`

Description: Send *string* to the **command.interpreter** to execute as if it were entered from the keyboard. All `ascii` commands are executed from an **application** using this procedure.

Example: `command("Set Address 1")`

Use the interactive "`set`" command to set the address decoder to mode 1, ie. use the **B009.emulator**.

1.4.2 bootstrap IMS T212

Command: `boot.t2( VAL []BYTE` *string*, `BOOL` *error.flag* `)`

Description: Reset the IMS T212, and boot it with the contents of *string*. If the file cannot be found, or the IMS T212 is not present, *error.flag* will be `TRUE`.

Example: `boot.t2( "OPTPNT.T2", error.flag )`

bootstrap the IMS T212 with the binary image contained in the file "OPTPNT.T2" (which was created with the "makeboot" utility - see above).

1.4.3 stop application

Command: `stop()`

Description: Tell the **command.interpreter** to resume getting commands from the keyboard. This is always the last statement in the main execution loop of an **application**. After executing this call, the **application** itself continues to execute until it reaches a point where it needs to communicate with the **controller**, at which point it waits until the **controller** is requested to execute the application again.

Example: `stop()`

1.4.4 perform initialisation with controller

Command: `initialisation.with.controller()`

Description: This call passes the `descriptor` to the **controller** when requested during startup of the IMS D703. It is always the first statement executed in any **application**, and is contained in the STANDARD initialisation fold.

Example: `initialisation.with.controller()`

1.4.5 get current status

Command:
```
get.current.status( INT current.no.of.A100s,
INT decoder.mode, BOOL b009.available,
[]BYTE t2.driver.id )
```

Description: Gets the current status of the IMS D703 from the **controller**. Note that the call uses all standard variables, which are predefined in the STANDARD services fold. Users are advised to use the call in this manner, to ensure consistent behaviour is maintained. The variables `b009.available`, `no.of.A100s`, and `decoder.mode` are used in some calls to determine the appropriate action. In particular, the services for address modes 1 & 2 use `decoder.mode` to determine whether to use the **B009.emulator** or the hardware **B009.driver**.

Example: `get.current.status( current.no.of.A100s, decoder.mode, b009.available, t2.driver.id`

1.4.6 display string

Command: **display.str(VAL []BYTE** *string* **)**

Description: Displays *string* in the command window. Note that end of line characters must be explicitly specified - this enables lines composed of several strings to be constructed.

Example: `display.str("This is a message.*c*n")`

displays the message in the command window, and leaves the cursor at the beginning of the next line.

1.4.7 display integer

Command: **display.int(VAL []BYTE** *string*, **VAL INT** *integer* **)**

Description: Displays *string*, followed by the **INT** variable *integer*, after conversion to text format, in the command window.

Example:

```
display.int("The number of A100s is: ",
current.no.of.A100s)
```

displays on the screen the following text (if the value of **current.no.of.A100s** is 5):

```
The number of A100s is: 5
```

Note the cursor is left immediately following the "5".

1.4.8 display real

Command: **display.real(VAL []BYTE** *string*, **VAL REAL64** *real64* **)**

Description: Displays *string*, followed by the **REAL64** variable *real64*, after conversion to text format, in the command window.

Example: `display.real("The size of x is: ", real64.arg)`

displays on the screen the following text (if the value of **real64.arg** is 5.00E03):

```
The size of x is: 5.00E03
```

Note the cursor is left immediately following the "5.00E03".

1.4.9 enquire string

Command: **enquire.str(VAL []BYTE** *string*, **[]BYTE** *str.arg* **)**

Description: Displays *string* in the command window, then waits for the user to enter a response at the keyboard, terminated by $\boxed{\text{return}}$. The string is returned in *str.arg*.

Example: `enquire.str("What is your name? ", str.arg)`

displays the text "What is your name? " in the command window. The response entered by the user is returned in **str.arg**.

1.4.10 enquire integer

Command: **enquire.int(VAL []BYTE** *string*, **INT** *int.arg* **)**

Description: Displays *string* in the command window, then waits for the users to enter an integer value at the keyboard. The number entered will be converted into internal **INT** format, and returned in *int.arg*. Note that the first non-valid character entered by the user will be taken as the delimiter for the end of the number, however the routine will always wait until $\boxed{\text{return}}$ is typed, to allow the user to edit the entered number prior to "sending" it to the application.

Example:

```
enquire.int("How many A100s? ", int.arg)
```

displays the text "How many A100s? " in the command window. The value of the number entered is returned in **INT** format in **int.arg**.

1.4.11 enquire real

Command: **enquire.real(VAL []BYTE** *string*, **REAL64** *real64.arg* **)**

Description: Displays *string* in the command window, then waits for the users to enter a real number at the keyboard. The number entered will be converted into internal **REAL64** format, and returned in *real64.arg*. Note that the first non-valid character entered by the user will be taken as the delimiter for the end of the number, however the routine will always wait until $\boxed{\text{return}}$ is typed, to allow the user to edit the entered number prior to "sending" it to the application.

Example:

```
enquire.real("Maximum value? ", real.arg)
```

displays the text "Maximum value? " in the command window. The value of the number entered is returned in **REAL64** format in **real.arg**.

1.4.12 enquire key

Command: **enquire.key(VAL []BYTE** *string*, **BYTE** *key.arg* **)**

Description: Displays *string* in the command window, then waits for the user to enter a single character. This character is immediately returned in *key.arg*. This is useful for interactive applications where either any character is used to continue the application, or characters may be used to manipulate cursors etc. in more advanced programs.

Example:

```
enquire.key("Hit any key to continue...*c*n", key.arg)
```

displays the text "Hit any key to continue..." in the command window, leaves the cursor at the beginning of the next line. When the user types any key, the value of that key is returned in **key.arg** and the application can continue.

1.4.13 plot textual

Command: **plot.text(VAL INT** *data.column0*, *data.column1* **)**

Description: Display the data items as a single line of text, plotted according to the current text plot parameters, with *data.column0* plotted in column 0, and *data.column1* plotted in column 1. See "**set graphics**" command in the **IMS D703 Users Manual** for further details of the text plotting facility.

Example:

```
SEQ i = 0 FOR number.of.data.points
plot.text( input[i], output[i] )
```

plots the contents of **[]INT** vectors **input** and **output** in the command window in textual form, one line per data pair.

1.4.14 plot graphics

Command:

```
plot.graphics( VAL INT window.id;
VAL []BYTE title.string;
VAL BOOL clear.window;
VAL INT plot.type;
VAL INT line.style;
VAL REAL32 x.start; REAL32 x.finish;
VAL []REAL32 y.args )
```

where:

window.id	is 2 for the upper plot window, 3 for the lower plot window
title.string	is the text to be displayed in the banner at the top of the selected window
clear.window	is `TRUE` if the window is to be cleared prior to displaying the plot
plot.type	is 1 for polygon with linestyle *line.style* is 2 for histogram otherwise polygon with vertical lines from x-axis
line.style	is 0 for `**************` is 1 for `*    *    *    *` is 2 for `*****     *****` is 3 for `***   *   ***   *` is 4 for `***  ***  ***  ***` otherwise bit pattern of lowest 8 bits.
x.start	is the lowest value of data on the x-axis
x.finish	is the highest value of data on the x-axis
y.args	is the array of y values to be plotted against the y-axis. Note their corresponding x-axis value is computed by equal increments between *x.start* and *x.finish*

Description: Plot the real data contained in `[]REAL32 y.args` in the selected plot window, according to the above parameters. For further information on the plot facility, refer to the **Turbo Pascal Graphics Manual.**

Example:

```
plot.graphics( 2, "The Plot Title", TRUE,
1, 4, 0.0, 1.0, [y.data FROM 0 FOR 256] )
```

plots the first 256 data points in `y.data` in the upper plot window. The title "The Plot Title" is displayed in the banner above the window, and the window is cleared before the plot is drawn. The plot will be drawn as a polygon with a line style of the form "`***  ***  ***`".

1.5 Filer Services

The following procedures provide a very simple filing system for **applications**. Note that using the "block.read" and "block.write" procedures will be faster than their corresponding single byte calls, however neither is particularly fast since single character I/O is used at the Pascal level throughout. This enables users to mix single character and block I/O.

1.5.1 open file for read

Command: `open.read.file( []BYTE` *filename*, `BYTE` *file.id*, `BOOL` *error.flag* `)`

Description: Open the MS-DOS *filename* read only. The call returns an identifier *file.id* which must be used in all future calls referring to this file. The filename follows MS-DOS conventions, and thus any pathname can be specified. If no pathname is given, the system assumes the file is in the current working directory. If *error.flag* is `TRUE`, the open was not successful.

Example:

```
open.read.file( "C:\FIRDATA\COEFFTS.DAT",
coefft.file.id, error.flag )
```

opens the MS-DOS file "c:\firdata\coeffts.dat" for read. The identifier of this file is returned in `coefft.file.id`.

1.5.2 open file for write

Command: `open.write.file( []BYTE` *filename*, `BYTE` *file.id*, `BOOL` *error.flag* `)`

Description: Open the MS-DOS *filename* for write. The call returns an identifier *file.id* which must be used in all future calls referring to this file. The filename follows MS-DOS conventions, and thus any pathname can be specified. If no pathname is given, the system assumes the file is in the current working directory. If *error.flag* is `TRUE`, the open was not successful. Note that if the file already exists, it will be overwritten.

Example:

```
open.write.file( "C:\FIRDATA\COEFFTS.DAT",
coefft.file.id, error.flag )
```

opens the MS-DOS file "c:\firdata\coeffts.dat" for write. The identifier of this file is returned in `coefft.file.id`.

1.5.3 write byte to file

Command: `write.byte.to.file( BYTE` *byte*, `VAL BYTE` *file.id* `)`

Description: Writes a single byte *byte* to the file pointed to by *file.id*.

Example:

```
write.byte.to.file( char, coefft.file.id )
```

writes `char` to the file referred to by `coefft.file.id`.

1.5.4 block write bytes to file

Command: `file.block.write( []BYTE` *data*, `BYTE` *file.id*, `BOOL` *error.flag* `)`

Description: Writes a block of bytes *data* to the file pointed to by *file.id*. If any error occurs, *error.flag* will be `TRUE`. The number of bytes sent is determined by the `SIZE` of the *data* vector.

Example:

```
file.block.write( [data FROM 0 FOR 128], text.file.id,
error.flag )
```

writes the first 128 bytes in `[]BYTE data` to the file pointed to by `text.file.id`. If any errors occur, `error.flag` will be `TRUE`.

1.5.5 read byte from file

Command: **`read.byte.from.file(`** **`BYTE`** *byte*, **`VAL BYTE`** *file.id*, **`BOOL`** *error.flag* **`)`**

Description: Reads a single byte *byte* from the file pointed to by *file.id*. If the *file.id* is invalid, or the end of file has been reached, *error.flag* will be **`TRUE`**.

Example:

```
read.byte.from.file( char, coefft.file.id, error.flag )
```

reads a byte from the file referred to by **`coefft.file.id`** into **`char`**. If **`error.flag`** is **`TRUE`**, the value of **`char`** is not defined.

1.5.6 block read bytes from file

Command: **`file.block.read(`** **`[]BYTE`** *data*, **`VAL BYTE`** *file.id*, **`VAL BYTE`** *file.id*, **`BYTE`** *byte.count*, **`BOOL`** *error.flag* **`)`**

Description: Reads a block of bytes *data* from the file pointed to by *file.id*. If the *file.id* is invalid, or the end of file has been reached, *error.flag* will be **`TRUE`**. The value *byte.count* is returned, and contains the number of bytes actually read. Note that the *data* vector should be <=255 bytes long.

Example:

```
file.block.read( text, text.file.id, byte.count, error.flag)
```

reads a block of bytes from the file referred to by **`text.file.id`** into **`text`**. If **`error.flag`** is **`TRUE`**, the operation is invalid, or end of file was reached.

1.5.7 close file

Command: **`close.file(`** **`VAL BYTE`** *file.id* **`)`**

Description: Close the file pointed to by *file.id*.

Example:

```
close.file( coefft.file.id )
```

closes the MS-DOS file referred to by **`coefft.file.id`**.

1.6 Specific services for IMS A100 models

These calls are only relevant when using address mode 0, ie. the IMS A100 model cascade. The model cascade uses a simple linear addressing scheme, whereby each IMS A100 in the cascade uses 128 consecutive addresses. The device closest to the final output, usually known as the "last" device in the cascade, is referred to as device 0, and uses addresses 0-127. Likewise device 1 uses addresses 128-255, etc. The address map for each device is that defined in the IMS A100 data sheet. An additional "magic" address, `-1`, is defined to enable users to write to DIRs of all devices in the cascade.

The size of the model cascade is programmable using the **set cascade** command (see the IMS D703 User Manual). To make this possible, a "phantom" register, the Model Control Register (MCR) was added to the IMS A100 model to allow control of the cascade size. When MCR bit 0 is set to one, the cascade input of that model is ignored, and the model itself sees only zeros coming through the cascade shift regisetr. This enables us to describe a configurable cascade.

A special constant **`model.common.dir`** is defined, which is the address used to write to all devices' DIR register at the same time.

1.6.1 read model

Command: **`model.read.a100( VAL INT16`** *address*, **`INT16`** *data* **`)`**

Description: Reads the location *address* of the model cascade, and returns the contents in *data*.

Example: **`model.read.a100( INT16 74, last.DOL )`**

reads the contents of the DOL register of device 0 in the cascade, and returns the contents in **`last.DOL`**.

1.6.2 write model

Command: **`model.write.a100( VAL INT16`** *address*, **`VAL INT16`** *data* **`)`**

Description: Loads the location *address* with the value *data*.

Example: **`model.write.a100( INT16 model.common.dir, 0(INT16) )`**

writes the value 0(INT16) to the DIRs of all devices in the model cascade.

1.6.3 set cascade size

Command: **`set.cascade.size( VAL INT`** *size* **`)`**

Description: Sets the size of the model cascade to *size*. This command is a more convenient way of using the "**`set cascade`**" **`ascii`** command.

Example: **`set.cascade.size( size.entered )`**

sets the model cascade size to the number of devices specified in **`size.entered`**.

1.7 Specific services for IMS B009 emulator/hardware

These calls are only relevant when using address modes 1 or 2, ie. the IMS B009 emulator or hardware. These calls all use the value **decoder.mode** to decide whether the command is sent to the hardware driver or the emulator. By using common calls, users can write applications which execute either using the simulator, or the hardware itself. This is very useful during debugging, as the simulator can log all commands to the driver, and all memory accesses if necessary, whilst this is not possible when using the hardware itself.

1.7.1 Protocols and constants

A copy of the driver protocol is kept in the STANDARD services, for use by the procedures described below. The MASTER of this protocol is kept in the **B009.driver**.

1.7.2 Channel usage with IMS B009 hardware

Each application includes a pair of channels which are connected directly to the **B009.emulator**. This enables them to avoid the **controller**, which gives a considerable speed up when executing simulations using the **A100.models**.

To access the IMS B009 hardware, all **applications PLACE** a pair of channels **to.b009.hardware** and **from.b009.hardware** at the same link addresses. This is done to avoid the performance degradation of using a link multiplex process to connect all applications to the driver. Users must therefore ensure that two applications cannot execute at the same time, otherwise data corruption will clearly occur. The **controller** ensures this for all **applications** executing under the standard IMS D703.

Wherever arrays of data are declared below, it is assumed that all elements of the array are passed. If the **SIZE** parameter is less than the actual **SIZE** of the array, only the first **SIZE** elements of that array will be used.

1.7.3 Read A100s via link

Command: **b009.read.a100.to.link(INT16** *base.address*, **SIZE** *data***(INT16), []INT16** *data*, **BOOL** *error.flag* **)**

Description: Reads *size* locations from the **a100.space** into *data*, starting at an offset *base.address* to **a100.base.address**. If the parameters are invalid, *error.flag* will be **TRUE**.

Example: **b009.read.a100.to.link(ucr.0, 128(INT16), ucr.values, error.flag)**

reads the contents of the UCRs of all four IMS A100s (ie. 128 coefficients) into **ucr.values**.

1.7.4 Write A100s via link

Command: **b009.write.a100.from.link(INT16** *base.address*, **SIZE** *data***(INT16), []INT16** *data*, **BOOL** *error.flag* **)**

Description: Loads *size* locations of the **a100.space** with *data*, starting at offset *base.address* to **a100.base.address**. If the parameters are invalid, **error.flag** will be **TRUE**.

Example: **b009.write.a100.from.link(ucr.0, 128(INT16), ucr.values, error.flag)**

loads the contents of the UCRs of all four IMS A100s (ie. 128 coefficients) with the data stored in **[128]INT16 ucr.values**.

1.7.5 Write DIRs via link

Command: **b009.write.DIR.from.link(INT16** *size*, **[]INT16** *data*, **BOOL** *error.flag* **)**

Description: Writes the first *size* values in the array *data* to the DIR of all devices. If the parameters are invalid, `error.flag` will be `TRUE`.

Example: `b009.write.DIR.from.link( 128(INT16), first.data, error.flag )`

writes the first 128 values in `first.data` to the DIR registers of all four IMS A100s.

1.7.6 Read data.buffer via link

Command: `b009.read.buffer.to.link( INT16` *base.address*, `INT16` *size*, `[]INT16` *data*, `BOOL` *error.flag* `)`

Description: Reads *size* locations from `buffer.space` in the driver into *data*, starting at offset *base.address* to `data.buffer.base.address`. The address mapper can be used to reorder data if required during this operation.

Example: `b009.read.buffer.to.link( 0(INT16), 1024(INT16), buffer.data, error.flag )`

reads the first 1024 locations of `buffer.space` into `buffer.data`.

1.7.7 Write data.buffer via link

Command: `b009.write.buffer.from.link( INT16` *base.address*, `SIZE` *data*`(INT16), []INT16` *data*, `BOOL` *error.flag* `)`

Description: Loads `buffer.space` with the contents of the array *data*, starting at offset *base.address* to `data.buffer.base.address`. If the parameters are invalid, `error.flag` is `TRUE`. The address mapper can be used to reorder data during this operation.

Example: `b009.write.buffer.from.link( 2048(INT16), 128(INT16), buffer.data, error.flag )`

loads locations 2048-2175 of `buffer.space` with `buffer.data`.

1.7.8 Write DIRs from data.buffer

Command: `b009.write.DIR.from.buffer( INT16` *base.address*, `INT16` *size*, `BOOL` *error.flag* `)`

Description: Writes *size* locations in `buffer.space` starting at offset *base.address* into DIR. If the parameters are invalid, `error.flag` is `TRUE`. The address mapper can be used to reworder data during this operation.

Example: `b009.write.DIR.from.buffer( 2048(INT16), 128(INT16), error.flag )`

sequentially writes the contents of locations 2048-2175 of `buffer.space` into all DIRs.

1.7.9 Process data with A100s using 16-bit output

Command: `b009.process.buffer.data.out16( INT16` *base.address.source*, `INT16` *base.address.destination*, `INT16` *size*, `BOOL` *error.flag* `)`

Description: This command uses DIRDOLmode of the IMS B009 for maximum speed, and as such achieves the highest performance for processing data (see the section on "The IMS B009 Driver, DIRDOLmode" later in this manual). Since DIRDOLmode is used, the input data must appear in every even location starting at offset **`base.address.source`** to **`data.buffer.base.address`**. The output will appear in every odd location starting at offset **`base.address.destination`** to **`data.buffer.base.address`**. The input data as defined above is written to the DIR of all devices, after which the DOL of the last device is read and stored as defined above. If the parameters are invalid, *error.flag* will be **`TRUE`**.

In order to avoid the data pre/post ordering problems with DIRDOLmode, the address mapper should be used with this function such that input and output data can be stored as contiguous arrays. See example 1 in Appendix A of the IMS D703 User Manual for further detail of how to achieve this.

Example:

```
b009.process.buffer.data.out16( 0(INT16), 2048(INT16),
1024(INT16), error.flag )
```

write locations 0-1023 of **`buffer.space`** into the common DIR; after each write, read **`last.DOL`** and store the results in locations 2048-3071 of **`buffer.space`**. In other words, pass 1024 data samples through the IMS A100 cascade and store the 16-bit results. This assumes the input and output data is as described above, and the address mapper is not enabled.

1.7.10 Process data with A100s using 24-bit output

Command: **`b009.process.buffer.data.out24( INT16`** *base.address.source*, **`INT16`** *base.address.destination*, **`INT16`** *size*, **`BOOL`** *error.flag* **`)`**

Description: With the address mapper disabled, write *size* locations in **`buffer.space`**, starting at relative address *base.address.source* into the common DIR of the IMS A100s. After each write, read both the DOL and DOH registers of device 0 (ie. the last device in the cascade) and store the result in **`buffer.space`** starting at relative address *base.address.destination.* If the parameters are invalid, *error.flag* will be **`TRUE`**.

Note that since both DOL and DOH are read, the maximum performance of the IMS A100 cascade cannot be achieved with this command. However the address mapper can be used to minimise data ordering overheads before and after processing.

Example:

```
b009.process.buffer.data.out24( 0(INT16), 2048(INT16),
1024(INT16), error.flag )
```

write locations 0-1023 of **`buffer.space`** into the common DIR; after each write, read **`last.DOL`** and store the results in locations 2048-3071 of **`buffer.space`**. In other words, pass 1024 data samples through the IMS A100 cascade and store the full 24-bit results. Note that to read the results, use the **`b009.read.buffer.to.link`** command, and **`RETYPE`** the resulting data as an **`[]INT32`**. If for example 1000 data points were processed, read 2000 locations from **`buffer.space`**, and **`RETYPE`** them into **`[1000]INT32`**. The full results can then be directly used as **`INT32s`**.

1.7.11 Read address.mapper via link

Command: **`b009.read.mapper.to.link( INT16`** *base.address*, **`INT16`** *size*, **`[]INT16`** *data*, **`BOOL`** *error.flag* **`)`**

Description: Read *size* locations of the **`mapper.space`** into *data*, starting at offset *base.address* to **`mapper.base.address`**. If any parameters are invalid, **`error.flag`** will be **`TRUE`**.

Example:

```
b009.read.mapper.to.link( 0(INT16), 1024(INT16), map.data,
error.flag )
```

read the contents of the first 1024 locations in the address mapper into **`map.data`**.

1.7.12 Write address.mapper via link

Command: **b009.write.mapper.from.link(** **INT16** *base.address*, **SIZE** *data*(**INT16**), **[]INT16** *data*, **BOOL** *error.flag* **)**

Description: Load the **mapper.space** with *data*, starting at offset *base.address* to **mapper.base.address**. If any parameters are invalid, **error.flag** will be **TRUE**.

Example: **b009.write.mapper.from.link(0(INT16), 1024(INT16), map.data, error.flag)**

load the contents of the first 1024 locations in the address mapper with **map.data**.

1.7.13 Enquire system status

Command: **b009.enquiry(** **INT16** *register.name*, **INT16** *contents*, **BOOL** *error.flag* **)**

where

register.name is: **b009.status.address |**
b009.error.address |
b009.ext.event.address

Description: Read register *register.name* of the IMS B009 and return its contents into *contents*. For further descriptions of the registers and their bit allocations, refer to the source code of the **B009.driver** or the **IMS B009 Reference Manual**. If the register is invalid, *error.flag* will be **TRUE**.

Example: **b009.enquiry(status.address, status.reg, error.flag)**

reads the status register of the IMS B009 and returns its current contents into **status.reg**.

1.7.14 Get driver descriptor

Command: **b009.driver.id(** **[]BYTE** *string* **)**

Description: Returns the descriptor of the currently loaded device driver in the IMS B009. If the emulator is enabled, it's identifier is returned.

Example: **b009.driver.id(t2.driver.id)**

gets the descriptor of the current driver (hardware or emulator depending on the current address decoder setting), and returns the string in **t2.driver.id**.

1.7.15 Modify current status

Command: **b009.status.modifier(** **INT16** *required.state*, **BOOL** *error.flag* **)**

where

required.state is:	**MAP.disable**	Disable the address mapper
	MAP.enable	Invoke the address mapper
	PROTECT.disable	Disable PROTECT mode
	PROTECT.enable	Invoke PROTECT mode
	EXT.clock.disable	Disable the external clock
	EXT.clock.enable	Use the external clock
	disable.all.modes	Disable all above modes

Description: Set the appropriate operating mode for the IMS B009. These commands are used to switch the address mapper in and out, and for using the PROTECT mode if required. The clock from the external connector is enabled or disabled here, which determines if the IMS A100s are clocked using the internal sources or external. If the nominated mode is invalid, *error.flag* will be **TRUE**.

Example:

```
b009.status.modifier( MAP.enable, error.flag )
```

invokes the address mapper so that all addresses in positive address space pass through the address translation mechanism.

3.1 Introduction

This section describes in detail the operation of the **controller** process. This process handles all keyboard commands, commands from applications, and performs most primary command decoding functions. It can interact directly with **A100.model** processes via **address.decoder**, however commands for **B009.emulator** and **B009.driver** are directly generated by **applications**.

3.2 ASCII and Binary Commands

The **command.interpreter** understands two forms of command: ascii and binary. These will be subsequently denoted by **binary** *command* or **ascii** *command.*

An **ascii** command is any command normally entered from the keyboard, which are documented in the **IMS D703 Users Manual**. These commands are:

- Application
- Continue
- Dump
- Execute
- Help
- Keep
- Last
- Quit
- Read
- Set
- Write

They will be denoted throughout the rest of this manual in the form "**command string**", eg. "**Read** *address*". For information regarding their function and arguments, please refer to the **IMS D703 Users Manual**.

A **binary** command is a command used only within the IMS D703 software environment, and can be regarded as an "internal" command. They are used by **application** processes, and comprise:

- Bootstrap T2
- Display (INT/REAL/string)
- Enquire (INT/REAL/string/char)
- File I/O (Open/Close/Read/Write)
- Get status (of controller)
- Plot (textual)
- Read A100
- Stop (application)
- TPlot (Turbo Plot - high resolution)
- Write A100

They will be denoted throughout the rest of this manual as **[binary command** *args***]**, eg. **[Display Integer** *string integer***]**. Note that for all commands, only the first letter of each keyword of commands and qualifiers to commands is sent by the **application**. For example, the above command sequence will actually be sent as:

[]BYTE "DI"; **SIZE** *string*; **[]BYTE** *string*; **INT** *integer*

3.3 Command Sources, Parsing, and Decoding

At the start of the command interpreter loop, **command.interpreter** clears out all the argument buffers, and if the next comand is **ascii** from the keyboard, displays the **Command:** prompt. The decision of whether the following command is **ascii** or **binary** is determined by **command.type**, which is set once the start of the command line is received. The command can come from one of two channels:

from.keyboard (handled via the **read.string** procedure), or

from.application.process[].

The decision as to which is determined by the value of **external.data.active**, which is **FALSE** for keyboard commands, and **TRUE** for commands from an **application**. If the command comes from an application, the interpreter will first receive a **command.type.int16**, which determines if it is an **ascii** or **binary** command. Note that if a character is typed at the keyboard, the interpreter should take this in preference to an input from **from.application.process[]**, thus allowing the user to halt execution of commands from an **application**, and return to entering commands from the keyboard. Note, however that if the application is executing commands via the **B009.driver** or **B009.emulator**, the only point at which these can be interrupted is when they request a new command from the **controller**.

3.3.1 ASCII commands

All **ascii** commands take the form:

Command *arg1 arg2 arg3 arg4 ...*

The command string will initially appear in **this.instruction**, and **ascii.cmd** will be **TRUE**. The **command.interpreter** accepts a text string in this form, and using one or more spaces as delimiters, splits up the command string. It places the characters of the first word of the command line into **[]BYTE command**, and the characters for all subsequent arguments into a two dimensional array [*arg.number*] **[]BYTE argument**. The value of **command** is then used by **decode** to invoke the appropriate command.

Note that before parsing commences, the parser first skips over leading spaces in the command line, in order to find the start of the actual command. If the first character is "**-**", it ignores the rest of the command as a comment line. Comments are very useful in keystroke files, most importantly for annotating what is happening at the command level, and for removing potentially troublesome commands within the file. Keystroke file processing is discussed below.

If the command is not "**last**", or a return (ie. no command entered), **this.instruction** is copied into **last.instruction** prior to parsing. This is so that if the command is required many times (eg. "**write -1 128**", the user can then execute it with the "**last**" command, rather than reenter it each time.

If **repeat.command** is TRUE, the contents of **last.instruction** are displayed, and the cursor placed at the end of the line for simple editing by adding characters or using delete.

3.3.2 Binary commands

Binary commands are received by the **controller** through the **from.application.process[]** channels from an **application**. They consist of one or more single **BYTE**s specifying the command, followed by arguments in the appropriate internal format, usually **INT**s or **REAL**s. They are decoded in **binary.command**, which receives the first **BYTE** of the command, and receives the rest of the command directly within the command decoder itself. Note this is in contrast to **ascii** commands, where the string is received and decoded prior to processing by **ascii.command**.

The response to **binary** commands is sent to the **application** via the **to.application.process[]** channel.

The format and function of these commands is described below.

3.4 Binary Command Descriptions

The purpose of this section is to illustrate how the protocols are used to interact with the **controller**, and thus how to extend these services if additional **controller** functions are required. For most users, these commands are normally accessed via the STANDARD Services procedures defined for User Applications. Please refer to the "User Applications" chapter for details of the relevant calls and their usage. Where a constant is predefined in STANDARD services, it will be shown as **`constant.value`**, to indicate that this is the actual name of the constant to be used in this command.

Throughout the descriptions below, the "command" is sent by the **application** to the **controller** via the **`from.application[]`** channels, and the "response" sent by the **controller** to the **applications** via the **`to.application[]`** channels.

3.4.1 B - Bootstrap the IMS T212

Format: Command: **`BYTE`** 'B'; SIZE *filename*; **`[]BYTE`** *filename*

Response: **`BOOL error`**

Function: Open the file *filename*, reset the IMS T212, and boot it with the contents of the file specified. If the IMS T212 is not available, or the file cannot be opened, an error is generated. The **controller** always returns **`error`** as a success/fail argument to the requesting **application**.

3.4.2 D - display string and typed argument

Format: Command: **`BYTE`** 'D'; **`BYTE`** *arg.type*; **`SIZE`** *string*; **`[]BYTE`** *string*; **`INT|REAL64`** *arg*

where:

arg.type is 'I' for **`INT`** (integer)
is 'R' for **`REAL64`** (64-bit real number)
any other character - only *string* is displayed.

arg is **`INT`** or **`REAL64`** as specified; not present if *arg.type* is not 'I' or 'R'.

Function: Display *string* in the command window, followed by the value of the argument *arg*, converted to text form.

3.4.3 E - enquiry displaying string, return typed argument

Format: Command: **`BYTE`** 'E'; **`BYTE`** *arg.type*; **`SIZE`** *string*; **`[]BYTE`** *string*

Response: **`INT|BYTE|REAL64|[]BYTE`** *arg*

where:

arg.type is 'I' for **`INT`** (integer)
is 'K' for **`BYTE`** (single character)
is 'R' for **`REAL64`** (64-bit real number)
is 'S' for **`[]BYTE`** (string)

arg is **`INT`** or **`BYTE`** or **`REAL64`** or **`[]BYTE`** as specified

Function: Display *string* in the command window, then wait for the user to enter a valid value of the type specified by *arg.type*. This value is converted to the appropriate internal format and returned to the **application**.

3.4.4 F - perform file I/O

Format: Command: **BYTE** 'F'; **BYTE** **open.read.tag** ; **SIZE** *filename* **(BYTE)** ; **[]BYTE** *filename*

Response: **BYTE** *file.id*

Notes: *file.id* = 255(BYTE) means bad file open

Command: **BYTE** 'F'; **BYTE** **open.write.tag** ; **SIZE** *filename* **(BYTE)** ; **[]BYTE** *filename*

Response: **BYTE** *file.id*

Notes: *file.id* = 255(BYTE) means bad file open

Command: **BYTE** 'F'; **BYTE** **read.byte.tag** ; **BYTE** *file.id*

Response: **BYTE** *success.flag* [; **BYTE** *byte*]

where *success.flag* will be:

filing.error.tag means an error has occurred; second byte is not sent, or
file.data.input.tag; *byte* means valid data has been received

Command: **BYTE** 'F'; **BYTE** **read.block.tag** ; **BYTE** *file.id*; **SIZE** *max.data* **(BYTE)**

Response: **SIZE** *data* **(BYTE)**; **[]BYTE** *data*]

Note: if **SIZE** *data* **(BYTE)** = 0, an error has occurred. Also, the size returned may not equal the size sent - *max.data* is the maximum size of data block to be read. If less, this probably means that end of file has been reached, and this is all the available remaining data.

Command: **BYTE** 'F'; **BYTE** **write.byte.tag** ; **BYTE** *file.id*; **BYTE** *byte*

Response: **BYTE** *success.flag*

where *success.flag* will be:

filing.error.tag means an error has occurred, or
file.data.input.tag; *byte*, which means that a valid byte is being received.

Command: **BYTE** 'F'; **BYTE** **write.block.tag** ; **BYTE** *file.id*; **SIZE** *data* **(BYTE)**; **[]BYTE** *data*

Response: **BYTE** *success.flag*

where *success.flag* will be:

filing.error.tag means an error has occurred, or
file.data.input.tag; *data* was written successfully

Command: **BYTE** 'F'; **BYTE** **close.file.tag** ; **BYTE** *file.id*

Function: Perform the appropriate operation as defined by the command tag given. These tags are included in the standard protocols section of the **application**. After opening the file, the *file.id* is used to identify which file is being addressed. Up to ten files can be open at any one time in the system as supplied.

3.4.5 G - get current status

Format: Command: **BYTE** 'G'

Response: **INT number.of.A100s**; **INT active.state**; **BOOL b009.available**; **SIZE** *hardware.id.string*; **[]BYTE** *hardware.id.string*

Function: Return to the application the current status of the **controller**, where **number.of.A100s** is the current number of **A100.models** active in the model cascade, **active.state** is the current **address.decoder** decode mode, **b009.available** indicates whether the IMS B009 hardware is available or not, and *hardware.id.string* is the descriptor of the IMS B009 driver, if it has been loaded.

3.4.6 P - plot data in command window

Format: Command: **BYTE** 'P'; **INT** *arg.0*; **INT** *arg.1*

Function: Plot the integers *arg.0* and *arg.1* using the text plotting function in the command window. The parameters controlling this plot function are set using the "**set graphics**" command.

3.4.7 R - send read request to address.decoder

Format: Command: **BYTE** 'R'; **INT16** *address*

Response: **INT16** *data*

Function: Read the location *address* in **A100.space** by sending it to the **address.decoder**. The value *data* returned depends on the current mode of the decoder, set with the "**set decoder**" command.

3.4.8 S - stop execution of application commands

Format: Command: **BYTE** 'S'

Function: A message is displayed informing the user that the current application has now terminated, and the **command.interpreter** proceeds to accept new commands from the keyboard.

3.4.9 T - Turboplot data in plot windows (ie. high resolution Turbo Graphics)

Format:

Command: **BYTE** 'T'; **INT** *window.id*; **SIZE** *title.string*;
[]BYTE *title.string*; **BOOL** *clear.window*; **INT** *plot.type*;
INT *line.style*; **REAL32** *x.start*; **REAL32** *x.finish*;
SIZE *y.args*; **[]REAL32** *y.args*

Function: Display the **REAL32** data in a plot window according to the parameters specified using "Turbo Graphix". For details of the parameters, refer to the "User Application" section of this manual, under the procedure **plot.graphics**.

3.4.10 W - send write request to address decoder

Format: Command: **BYTE** 'W'; **INT16** *address*; **INT16** *data*

Response: **INT16** *success.flag*

Function: Load the location *address* with *data* by sending both to the **address.decoder**. The location loaded depends on the current mode of the decoder, set with the "**set decoder**" command. If *success.flag* <> 0, an error has occurred.

3.5 Displaying messages on the screen - message multiplexer

Throughout the IMS D703, a hierarchy of message multiplexers are placed so that messages from any part of the system can be displayed on the screen in a controllable manner, ie. they don't scramble each other, wherever possible. Since messages can be generated by several processes at the same time, the message multiplexers must route messages by packets rather than byte by byte. Thus, all messages are sent in the form of strings, so that once the size of the string is detected, the rest of the string is read by a block move. The message is then forwarded up the tree by a similar block move. The tree structure is as shown in Figure 2.

```
 Filing        Screen    Keyboard     Printer    (Matrox)
 System          |          |            |           |
   |_____________|__________|____________|___________|
                            |
                   MS-DOS Front End
                            |
                            v                            IBM PC
====================(Link Adaptor)========================
                            ^                        transputer
                            |
                       transceiver
                            |
                            |
                    message.processor
                            |
          __________________|______________________
          |                  |                      |
command.interpreter  from.proc.messages  A100.error.handler
          |               /        \      \___________  |
          |              /          \                 \ |
applications    B009.emulator  address.decoder  A100.models
```

Figure 2: Message channel hierarchy in the IMS D703

The **message.processor** multiplexes all messages from the three processes below it, and forwards them to the transceiver for display in the command window. The **command.interpreter** generates all its messages this way, and likewise all messages from the **applications** pass via the **command.interpreter**, usually using **`binary`** commands.

The **from.proc.messages** multiplexer collects messages from many processes concerned with the simulator, ie. all **A100.models**, the **address.decoder**, and the **B009.emulator**. In order to identify these processes' messages uniquely in the command window, the **message.multiplexer** automatically appends an appropriate identifier of the message, if the first character of that message is a "%". For example, if one of the **A100.models** is generating the message, the identifier "%%A100*x*-" is appended to the front of the message. In this case, *x* represents which device in the model cascade is generating the message, and is generated by adding ASCII "a" to the device number. Thus, if **A100.model** number 3 sent the message:

"%E-AsyncWr: Invalid write address*c*n"

to **message.multiplexer** the message would be displayed in the command window as:

"%%A100d-E-AsyncWr: Invalid write address"

The **A100.error.handler** receives all errors generated by all **A100.models**, and also consumes the output of the final **A100.model**. It's output can be enabled if required with the "**`set message 22 on`**" or "**`set message 23 on`**" commands (see the IMS D703 User Manual).

3.6 Communications with MS-DOS Front End - the transceiver

All communications between the occam processes and MS-DOS pass via the **transceiver**. This process multiplexes several logical channels from the **controller**:

- `to.screen`, `from.keyboard`
- `to.graphic`, `from.graphic`
- `to.file`, `from.file`
- `to.matrox`, `from.matrox`

Note: The `matrox` channels are included to show how the system can be extended to interface to user-specific peripherals. In this case, these channels were used during internal development of the IMS D704 system for high speed transfer of data to and from a frame grabber board resident in the IBM PC. These channels have been left, together with their associated calling procedures, in both the **controller** and **front end** as an illustration of how to provide peripheral I/O to the IMS D703. However, the services provided in the `matrox` interface are not described further in this manual.

The `to.screen` and `from.keyboard` channels are the conventional ones used by most occam programs interfacing to a screen and keyboard. The protocol is very simple, with the `to.screen` channel regarding the output as a dumb scrolling alphanumeric terminal, and the `from.keyboard` channel receiving characters directly, ie. untagged.

The `to.graphic` and `from.graphic` channels provide a logically separate interface to the Turbo Graphics routines. The command protocol supports communication with potentially all services of this library, although only a subset of them have been enabled by the **front end**. Due to the wide range of parameters passed to the various routines, a general purpose protocol is used to enable bidirectional communication of a wide range of parameters of different types, so that any procedure in the **front end** can be potentially invoked by the occam program. This approach is particularly interesting to users wishing to interface to their own MS-DOS procedures, as it provides the basis for communications between occam programs and any procedure call in the language of the MS-DOS server.

The `to.file` and `from.file` channels provide the interface to the MS-DOS filing system, providing for up to ten files to be open at any one time. Both single byte and simple block I/O are supported.

3.6.1 Protocols

The following protocols are used in the communications described above. They are defined in the **transceiver** process. The communications described below are as seen by the **controller**. Note the protocols between the **transceiver** and the **front end** are not described here, however the information given here should be sufficient for the interested user to study the source code of these two processes to see exactly what happens. The general purpose protocol used in the `graphics` channels is described as a method for constructing general purpose host-transputer interfaces.

File I/O

`to.file ! open.read.tag; SIZE` *filename* `(BYTE); []BYTE` *filename*

`from.file ? file.open.return.tag; BYTE` *file.id*

`to.file ! open.write.tag; SIZE` *filename* `(BYTE); []BYTE` *filename*

`from.file ? file.open.return.tag; BYTE` *file.id*

`to.file ! close.file.tag; BYTE` *file.id*

`to.file ! read.byte.tag; BYTE` *file.id*

`from.file ? file.data.input.tag; BYTE` *byte* or `filing.error.tag`

`to.file ! write.byte.tag; BYTE` *file.id*; `BYTE` *byte*

`to.file ! read.block.tag`; **`BYTE`** *file.id*; **`BYTE`** *block.size*

`from.file ? file.block.transfer.tag`; **`SIZE`** *byte.array*`(BYTE)`; `[]`**`BYTE`** *byte.array* or **`filing.error.tag`**

`to.file ! write.block.tag`; **`BYTE`** *file.id*; **`SIZE`** *byte.array*`(BYTE)`; `[]`**`BYTE`** *byte.array*

`from.file ? filing.error.tag`

Screen Output

`to.screen !` **`SIZE`** *string*; `[]`**`BYTE`** *string*

Note: **`string[0] = shutdown.command.tag`** - terminates execution of the IMS D703.

Keyboard Input

`from.keyboard ?` **`BYTE`** *byte*

Graphics I/O

`to.graphics ! graphics.command.tag`;
`BYTE` *parameter.type.tag*; `[]`**`BYTE RETYPES`** *parameter*,
[**`BYTE`** *parameter.type.tag*; `[]`**`BYTE RETYPES`** *parameter*,
`BYTE` *parameter.type.tag*; `[]`**`BYTE RETYPES`** *parameter*,]

`from.graphics ? function.reply.tag`; **`BYTE`** *graphics.command.tag*;
[**`BYTE`** *parameter.type.tag*; `[]`**`BYTE RETYPES`** *parameter*,
`BYTE` *parameter.type.tag*; `[]`**`BYTE RETYPES`** *parameter*,
`BYTE` *parameter.type.tag*; `[]`**`BYTE RETYPES`** *parameter*,]

Notes on Graphics I/O protocol

When calling any of the graphics routines, the general sequence is to first send the relevant command tag **`graphics.command.tag`**, followed by one or more parameters. Each parameter is sent by first sending its type tag **`parameter.type.tag`**, followed by the parameter itself, **`RETYPE`**d into a string of bytes. For all character and integer I/O, this is fine; however the Turbo Pascal used in the server supplied with the IMS D703 does not use standard IEEE P754 floating point representations. The appropriate conversions must therefore be made to reconstruct these numbers in the appropriate format. Other language implementations should not give this sort of problem; however users must always take care of byte ordering when using such techniques.

When the **`graphics.command.tag`** is received by the **front end**, it uses this as a vector into a **`case`** statement which then expects the appropriate number of parameters, of the correct type, to be sent. Thus, an immediate check is made to ensure that all arguments are sent, of the right type. Clearly there are many methods of validation at this stage. Once all arguments are received, they are used to call the appropriate procedure.

Once the command completes, it may wish to send some results back to the occam program. This is accomplished by a similar protocol. The procedure first sends it's command tag as verification, then sends the results back using the same format as for supplying the arguments to the call. When all the results are sent, an **`end.tag`** is sent to indicate no more results will be sent. The occam program can then validate the quality of the returned data.

For example, if the pascal procedure to be called has the form:

```
procedure user.cmd( byte.arg:char,
number.arg:integer,
str.arg:[0..40] of byte)
```

which accepts **byte.arg** and **number.arg** as inputs, and returns **str.arg** as the result, the command sequence to call this from occam would be:

to.user.io ! user.cmd.tag; byte.tag; BYTE *'byte.arg'*; **int.tag; INT** *number.arg*

from.user.io ? user.cmd.tag; string.tag; SIZE *str.arg* **(BYTE); []BYTE** *str.arg*

This provides a general purpose method of calling any form of procedure executing on the IBM PC host from any occam program executing via a user server program.

Note, however, that the argument return part of the protocol is not used on any of the communications between the **controller** and the **front end** of the IMS D703 as supplied.

Note that for the purposes of the IMS D703, several modifications were made to the Turbo Graphics libraries. These comprised:

1 rewriting the screen dump routines to provide a better aspect ratio, at higher speed

2 modification of the draw axis procedure to enable it to label axes that are not very tall

3 increasing the size of the plot array vector

3.7 Communications with IMS A100 models and IMS B009 hardware

All communications between the **controller** and either the software model or hardware IMS A100s is done via the **to.decoder** and **from.decoder** channels and the **address.decoder**. Since the protocol across this channel directly reflects the actual transactions on the IMS A100's SMI, and should emulate the behaviour of the IMS T212 memory interface for some modes of operation, a strict protocol is imposed:

to.decoder ! INT16 memory.read; INT16 *address*
from.decoder ? INT16 *data*

to.decoder ! INT16 memory.write; INT16 *address*; **INT16** *data*

This protocol is observed throughout the IMS D703 whenever a memory access is being performed. Thus, when the **B009.emulator** wishes to read a location, it sends the request in the above form to the **address.decoder**. The **address.decoder** translates the address, and if valid sends the translated memory request, in the same protocol form, to the appropriate **A100.model**. The data resulting from the read request will be sent back via the **address.decoder** to the **B009.emulator** observing the last part of the read protocol.

By adhering to this rule, many advantages are gained. For example, a system using a single **A100.model** can dispense with the **address.decoder** altogether, if required, by simply joining the **to.decoder** and **from.decoder** channels of the **controller** directly to the appropriate SMI channels of the **A100.model**. Conversely, several levels of address decoding could be used if required, with the same protocol linking each. Also, memory requests to the IMS B009 can be relayed directly to the **B009.driver** without translation. Most importantly, the protocol directly models the memory interface of a 16-bit microprocessor, so that the move from software to hardware is trivial. Indeed, if suitable procedure calls are used, the code which sends such commmunications down channels could equally read or write the specified locations directly, by simply replacing the procedure.

In addition to the memory accesses, the **controller** needs to be able to control operation of **address.decoder**. These commands are communicated over the same channels as the memory access, but with different command tags. These are:

to.decoder ! t2.b009.getID
from.decoder ? SIZE *string*; **[]BYTE** *string*

Requests the current descriptor from the device driver. The hardware driver or emulator descriptor is returned, depending on the current decoder mode.

to.decoder ! decoder.control; **enquiry.modes**
from.decoder ? INT16 *active.state*; **INT16** *cascade.size*;
SIZE *B009.emulator.id*; **[]BYTE** *B009.emulator.id*;
SIZE *B009.hardware.id*; **[]BYTE** *B009.hardware.id*

Requests the current status of the **address.decoder**. Returns the current active decode mode *active.state*, the current size of the model cascade *cascade.size*, and the descriptors of the **B009.emulator** and **B009.driver** if available.

to.decoder ! decoder.control; **set.cascade.size**; **INT16** *cascade.size*
from.decoder ? INT16 *error*

Sets the size of the model cascade to *cascade.size*. If the size is invalid, **bad.size** is returned, otherwise **size.ok** is sent.

to.decoder ! decoder.control; **set.address.mode**; **INT16** *address.mode*

Sets the address mode to *address.mode*, which can take the following values: **A100.model.active**, **B009.emulator.active**, or **B009.hardware.active**.

to.decoder ! t2.b009.check
from.decoder ? BOOL *error*

This is used prior to bootstrap to see if the IMS T212 exists at the end of the channel **to.b009.hardware**. If it does, **error** is **FALSE**, otherwise **TRUE** indicates that no IMS B009 is available.

to.decoder ! t2.b009.send.byte; **BYTE** *byte*

This is used during bootstrap of the IMS T212, and passes the byte raw down the bootstrap channel **to.b009.hardware**.

3.8 Communications with Applications

The communications between the **controller** and the **applications** can be considered to be an extension to the normal keyboard input to the **command.interpreter**. This is because the **application** can supply **ascii** commands as if they were coming from the keyboard, or **binary** commands, which are only sensibly executed from within a program.

When the **command.interpreter** is ready to accept a new command, the source of that command is determined by the boolean **application.active**. This boolean is **FALSE** for keyboard input, however the command "**application** *n*" sets the **current.application** to *n* and sets **application.active = TRUE**.

When **application.active** is **TRUE**, the **command.interpreter** reads the next byte from the channel **from.application[current.application]**. This byte determines if the command is **ascii** or **binary**. Depending on the result, the rest of the input from that channel is directed to the appropriate decoder. Commands continue to be read from that channel until either the **application** sends a **[stop]** command, or the user used the keyboard interrupt to force the **command.interpreter** to switch back to taking commands from the keyboard.

The protocols used for the **binary** commands are as defined above. For **ascii** commands, the protocol is simply that of passing the string normally entered at the keyboard, as defined in the **IMS D703 Users Manual**.

3.9 Other communications

All data communications with the **A100.models** apart from those involving the SMI use **`INT64`** throughout. This is explained below in the "A100 Model" section. The **`from.A100.error`** channels send a dummy **`INT16`** value, since they only signify the fact that an event has occurred.

3.10 Parameters and Constants

The "Constants" declaration area in the **controller** comprises the definitions of protocols of **controller - address.decoder**, and **controller - transceiver**. Note that the protocol fold header shows if the declarations are the MASTER set or a COPY of a protocol which is MASTERed elsewhere. The location of the MASTER declarations is identified in brackets after the description. It is worthwhile following this convention, until your TDS implementation supports libraries!

Note that a COPY of the device specifics for the IMS A100B is included here, as some of the commands need to know such parameters for decoding, etc. The default file names for the "**`help`**" facility and the initial bootstrap binary image are also held here. The characters used for text plotting are defined here, so that the user can alter the characters used for high, low, or centre of the column where the data point lies, the zero axis, delimiter between hexadecimal data and plot, etc.

3.11 Useful Procedures

A wide range of standard procedures are used within the **controller** for string and numeric I/O, graphics, files, and communications. These are described below.

3.11.1 Character and String I/O

The following routines are provided:

`read.byte( BYTE ch )`

Reads a single byte from **`from.keyboard`**, or **`from.file`** if **`demo.running.flag`** is **`TRUE`**. This routine also handles a variety of services, including escape sequences, window resizing, printer logging, and file handling for keystroke files. If **`kepp.flag`** is **`TRUE`**, all input is saved to file.

`write.byte( CHAN messages, VAL BYTE ch)`

Sends a single byte on the channel **`messages`**. Normally used by **`write.string`**.

`write.string( CHAN messages, VAL []BYTE string )`

Sends the string on the channel **`messages`**, preceded by its **`SIZE`**. Normally used to display messages on the command window.

`read.string( INT length, []BYTE string, CHAN messages )`

Reads **`string`** terminated by $\boxed{\text{return}}$, using **`read.byte`**. The length of the string read is returned in **`length`**, and the characters are echoed using **`write.byte`** to channel **`messages`**. Note that $\boxed{\text{delete}}$ is supported.

`edit.string( INT length, []BYTE string, CHAN messages)`

Assumes **`string`** has already been displayed with the cursor left after the last character, and **`length`** is set to the length of **`string`**. Allows the user to edit a **`string`** already displayed, and returns the edited version. Used in the **command.interpreter** within the "**`last`**" command.

A wide range of numeric conversion routines are also provided, for converting strings to hexadecimal or decimal numbers, both **INT** and **REAL**. Their usage is clear from the procedure call itself:

```
string.to.INT( []BYTE string, INT number )
string.to.INT32( []BYTE string, INT32 number )

INTread( CHAN messages, INT number )
REAL64read( CHAN messages, REAL64 number )

INT16.to.dec.string( VAL INT16 number, [6]BYTE result )
INT16.to.hex.string( VAL INT16 number, [4]BYTE result )
INT16.display.dec( CHAN messages, VAL INT16 number, VAL INT size )
INT16.display.hex( CHAN messages, VAL INT16 number )

INT16.display.dec.and.hex(CHAN messages, VAL INT16 number, VAL INT size)
-- used in "read" and "write" commands

INT32.to.dec.string( VAL INT32 number, [6]BYTE result )
INT32.to.hex.string( VAL INT32 number, [4]BYTE result )
INT32.display.dec( CHAN messages, VAL INT32 number, VAL INT size )
INT32.display.hex( CHAN messages, VAL INT32 number )

INT32.display.dec.and.hex(CHAN messages, VAL INT32 number, VAL INT size)

REAL64.to.string( VAL REAL64 number, INT before.point,
                  INT after.point, []BYTE string )
```

3.11.2 Graphics I/O

In order to easily generate the main graphics commands, a number of procedures are provided. These simply implement the Turbo Pascal Graphics Functions, and are thus documented in the manual supplied with that software. Users wishing to use the graphics facilities to recreate new versions of RUNSIM must procure a copy of Turbo Graphixs Toolbox, together with Turbo Pascal.

3.11.3 File I/O

The following procedures are provided for file I/O with the standard MS-DOS filing system. These procedures are invoked by the **binary** command described above. Note that these procedures are similar to those used in the **applications**, but communicate directly with the **transceiver**. They use the same protocols as the **applications** filer calls, so that the file commands from **applications** are simply relayed by the **controller** directly to the **transceiver**.

```
open.read.file( []BYTE filename, BYTE file.id, BOOL error )
```

Opens the file **filename** for read if possible; **error** indicates if it was successful or not. The procedure returns **file.id**, which must be used in all future references to that file until it is closed. Up to ten files can be open at any one time.

```
open.write.file( []BYTE filename, BYTE file.id, BOOL error )
```

Opens the file **filename** for write if possible; **error** indicates if it was successful or not. The procedure returns **file.id**, which must be used in all future references to that file until it is closed.

```
file.read( BYTE char, BYTE file.id, BOOL error )
```

Read a byte from the file pointed to by **file.id** into **char**. If an error occurs (eg. end of file), **error = TRUE**.

```
file.write( BYTE char, BYTE file.id, BOOL error.flag )
```

Write a byte **`char`** to the file pointed to by **`file.id`**.

`file.block.read( []BYTE block.of.char, VAL BYTE file.id, BYTE block.size, BOOL error )`

Read a block of **`SIZE`** *block.of.char* bytes of data from the file pointed to by **`file.id`** into **`block.of.char`**. If end of file is reached before the block is full, **`error`** is **`TRUE`**, and **`block.size`** contains the number of bytes returned in **`block.char`**.

`file.block.write( VAL []BYTE block.char, VAL BYTE file.id, BOOL error.flag )`

Write **`block.char`** to the file pointed to by **`file.id`**.

`close.file( VAL BYTE file.id )`

Close the file pointed to by **`file.id`**.

3.11.4 Bootstrapping the IMS T212 at runtime

This facility enables users to reboot the driver executing on the IMS T212 if required. This is normally accessed via the `[BOOT` *filename*`]` command from an **application**. The bootstrap routine is defined within **command.interpreter**, and is the same routine as that used during startup.

`boot.t2( []BYTE` *filename*, `BOOL` *error* `)`

resets the subsystem connected to the host transputer, checks to ensure that a transputer is connected to the **`to.b009.hardware`** channel, and if so boots it with the contents of *filename*. Note that this routine expects the driver to be able to respond to a request for its descriptor, so writers of new drivers should ensure this facility is available.

3.12 Adding new commands

Adding commands to the IMS D703 **command.interpreter** is straightforward, once the parsing operation is understood. For simple **`ascii`** commands, it is a matter of finding the **`ascii.command()`** procedure, which is located in the **`decode()`** procedure within **command.interpreter**. The safest method is to copy an existing command fold, and modify it to your own needs. Take care that the new command is unique, or that similar commands' decode expressions are extended to ensure unambiguous decoding.

Adding **`binary`** commands is similar, except they reside in **`binary.command()`**. Note that binary commands are available to all applications, so if new ones are added, the following procedure is recommended to remain consistent with the conventions adopted during development of the IMS D703.

When adding these commands to the appropriate decoder procedure, ensure they are uniquely identified by adding the prefix SITE-SPECIFIC to a fold containing all your own additions. Then, create a suitable set of PROCs for calling your new **`binary`** commands, and keep these in a separate fold called SITE-SPECIFIC Services. Place this fold in the "standard harness" area, then copy that fold into all applications requiring the new service. This enables you to keep track of exactly which routines you added. This will prove important if future releases of the IMS D703 are made, since you will probably want to integrate your new commands with the new releases, and this ensure you can quickly identify your own extensions.

4.1 Introduction

The **address.decoder** provides the same service as it's hardware equivalent: it translates logical addresses into physical addresses, and routes them, together with the data, to the correct destination. In the IMS D703, we are using occam to model the complete interface between host and IMS A100s. This is demonstrated in the IMS D703 by modelling two addressing regimes:

1. A simple linear address model, where each IMS A100 occupies 128 contiguous 16-bit word locations, starting at location 0
2. A complex address decoder, which emulates the IMS B009 address decoder

4.2 Principles of Operation

An **INT16** command word is received, which determines if the next words are for a memory read, memory write, or a command local to the **address.decoder** itself. The next **INT16** word contains the address to be translated. This address is validated, and translated as required. A decision is also made as to which channel the output should be sent.

If the command was for a read, a **memory.read** word is sent to the output channel, followed by the translated address. The **INT16** data reply is passed back to the process requesting the operation. For a write, the data is passed after the translated address to the decoded destination.

The decoder has a message channel which enables it to display every transaction it performs, if the appropriate "**set**" command has been performed.

The **address.decoder** can respond to requests from both the **controller** and the **B009.emulator**, and these two decoders execute in **PAR**allel. Since the **B009.emulator** is normally only accessed by **applications** directly, in normal operation only one of the two sources of addresses will be active at any time. However, users should note that if **applications** are interrupted which are executing with the **B009.emulator**, there is the possibility of conflict between the two sources of requests.

Note also that the **address.decoder** is connected to the **B009.emulator** as another **application**. This enables the **controller** to interrogate the **B009.emulator** in exactly the same way as it can interact with the **B009.driver**, for example when requesting descriptors.

The **address.decoder** has in effect three addressing regimes, which are referred to as addressing modes. These are:

4.3 Address mode 0: IMS A100 Model

The memory map is simply allocated so that the normal IMS A100 device memory map is seen by the user, translated by *n*128, where *n* is the number of the device in the model cascade. For example, the DIR for device 0 is 72, for device 1 is 200, for device 2 is 328, etc.

A special address, −1, is also provided. If a "**write -1** *data*" command is made in this mode, all DIRs are loaded with the same data at the same time.

4.4 Address mode 1: IMS B009 Emulator

This memory map accurately represents the decoding of the IMS A100s in the memory map of the IMS B009, such that the addresses received by the decoder are identical to those received by the IMS B009 decoder from the IMS T212 during normal operation. Note that these addresses are after the address mapper, which is modelled in the **B009.emulator** itself.

For further details of the IMS B009 address map, please refer to Table 1, and the IMS B009 Reference Manual.

4.5 Address mode 2: IMS B009 Hardware

This mode enables the user to read and write the IMS A100 devices on the IMS B009 itself. Note that the IMS T212 memory itself is not accessible, nor are the contents of the address mapper. This is because the **B009.driver** deliberately isolates access to these from accesses to the IMS A100s. However, if users wish to gain access via this mechanism to the complete IMS B009 memory map, the extension is fairly straightforward.

The main use of this mode is to enable users to "peek" and "poke" the actual IMS A100 hardware, and validate that the hardware does what is expected. This can be useful during early debugging, if the user is not convinced that an application has set up the IMS A100s correctly.

It is also useful for setting up a static configuration for use with external data. A keystroke file can be used to directly set up and load the IMS A100 devices, and start operation.

4.6 Communications with other processes

Most communications between the **address.decoder** and processes connected to it is via the standard "memory access" protocol described above. This includes communications from the **controller**, and the **B009.emulator**, to the **A100.models**. Note that "from" means the decoder receives the memory requests; "to" ,means that the decoder issues the memory requests.

An additional channel is provided to the **controller**, for communicating messages for display via the message channel. Channels are also provided to the **B009.driver** and the **B009.emulator**, which use the protocol defined for communication with those drivers. In the case of the **B009.emulator**, the channels `to.b009.emulator` and `from.b009.emulator` are used for memory requests from the emulator to the decoder; the channels `to.b009.emulator.com` and `from.b009.emulator.com` are used by the decoder to send driver commands to the emulator.

4.7 Parameters and Constants

A COPY of the protocol for driver communications is contained in the decoder, together with a COPY of the controller - decoder protocol. The device parameters for the IMS A100 are also held here as they are required in order to generate the correct decode for the devices.

4.8 Special Link Services

The decoder performs the actual task of determining whether the IMS T212 is present or not. If the link is not connected, a normal link communication would result in the process "hanging" forever. To avoid this, a special link function `OutputOrFail` is used within the `t2.b009.check` command in the `controller` decoder. This procedure will time out after the nominated period if the link does not receive an "acknowledge" to its transmission. For further details of this and related special link functions, please refer to INMOS Technical Note 1: Extraordinary use of Links".

4.9 Adding new address decoding modes

Adding new decoding modes involves two stages:

1. Defining the new address translation process
2. Establishing the new decoder mode in both the **address.decoder** and the **controller**

Since only two commands are possible for any decoding process, ie. `memory.read` or `memory.write`, the decoder combines the decision with that of which mode is required. Thus, the decoder is not nested, the decode expression being simply:

4.9 Adding new address decoding modes

```
(address.mode = new.mode) AND (command = memory.read)
(address.mode = new.mode) AND (command = memory.write)
```

If this is **`TRUE`**, the address (and data if a write) is received from the appropriate channel, decoded as required, and the result sent out channel **`to.A100.SMI`**`[`*device.number*`]`.

In order to add a new mode, the protocol must first be updated to define the new mode, after the three already present. The number must be the next available **`INT`**, ie. 3 for the IMS D703 as supplied. This is updated in both the **controller** and **address.decoder**. The value of **`number.of.address.modes`** must also be incremented.

Next, the **`set.cascade.mode`** command in the decoder must be updated to accept the new mode. Finally, the **`set.command`** procedure within the **`ascii.command`** decoder must be updated, including the enquiry mode.

5.1 Introduction

The occam model of the IMS A100 contained in the IMS D703 is a complete system level model of the device. It has been specifically developed to emulate all modes of operation, with the exception of certain test modes, and as such forms a concise specification of the device. However, by using occam channels for modelling the communication of all data flow through the IMS A100 data ports, the **A100.model** process also provides an excellent starting point for constructing models of complete systems. To demonstrate this capability, the IMS D703 provides an emulation of the IMS B009 evaluation board, including address mapper, optimised address decoder, and status registers. By understanding the relationship between the occam model and the actual hardware, systems involving complex synchronous behaviour (eg. interleaved devices, recirculating buffers, etc) can be reliably modelled with the tools provided in the source code of the IMS D703. For those interested in Systems Description Languages (SDLs), the IMS D703 demonstrates one way in which both synchronous and asynchronous communication can be described using a simple unbuffered handshaken model of communication.

Figure 4 shows the users model of the IMS A100, as used by hardware designers. Figure 5 shows how the main occam processes within **A100.model** are structured, and the communications between processes. Note that in Figure 5, the thick lines represent channels which model data ports of the IMS A100 device, whilst the thin lines are channels internal to the model. The **message.processor** provides a means of the model displaying messages via the **controller** to the command window. Note that the model can warn users of situations the device itself cannot - these are explained below.

5.2 Principles of operation

The model is divided into two key processes: **asynchronous.functions** and **synchronous.functions**. These processes emulate the split of functions of the actual device, where the former contains all the coefficient and status registers, and provides the SMI interface to the controlling processor, whilst the latter performs the actual calculations, including selection, cascading, rounding, etc. The channels providing data to the model connect to the appropriate part of the model. For example, data supplied on the DIN pins of the device is fed to the **synchronous.functions** process on the **`to.A100.external.data`** channel, whilst data for DIR is input via the **`to.A100.SMI`** channel to the **asynchronous.functions** process.

Comparison of Figures 4 and 5 will reveal that several pins are missing from the model: GO, RESET, OUTRDY, BUSY, and CLOCK. This is because their functions are to provide synchronisation between the IMS A100 and hardware connected to it. For example, when data is valid on the DIN pins, the GO signal is taken high, and sampled by the CLOCK. The DIN pins are sampled on the next rising clock edge and the result latched. The functionality of this process is modelled by the synchronisation behaviour of occam channels, whereby the act of sending the data on the **`to.A100.external.data`** channel implies that the sending hardware is synchronised to the IMS A100 receiving that data. Likewise, when data output is valid, for hardware the OUTRDY signal synchronises external hardware to the valid data on the DOUT pins. In the occam model, the act of sending the result out on the **`from.A100.external.data`** achieves the same goal by synchronising the transfer to the receiving process. Thus, the action of sending and receiving data via occam channels can be considered to model a synchronous transfer, since both sender and receiver must be synchronised. Therefore, we don't need a clock. The clock is also not required for internal operation, since this is a system level model, so we use the multiplication instructions of the transputer directly rather than try to emulate the precise behaviour of the IMS A100 itself at the bit level.

The SMI, however, is an asynchronous interface. Here, we model the communication as a master - slave operation, ie. whenever we request a read of a location, the device will return the valid data when it is ready to do so. This is equivalent to qualifying the output data with the OUTRDY pin of the IMS A100. It is important to understand that the model of the SMI is a strict subset of the behaviour of the real hardware, since in reality the hardware supplying data to the SMI does not attempt to synchronise to the IMS A100 in any way. Therefore, certain operations will occur correctly with the model which may not in the hardware, if for example the hold times are wrong for read requests. We could build an elaborate model to attempt to model this behaviour, however it would be far too detailed for what this model tries to achieve.

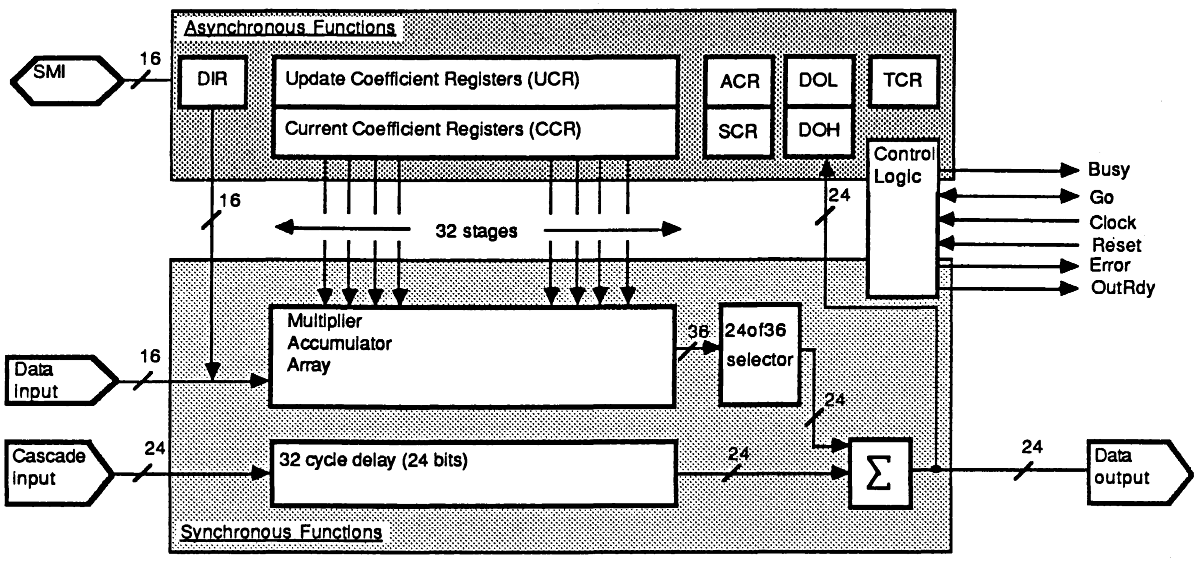

Figure 4: IMS A100 user model of physical device

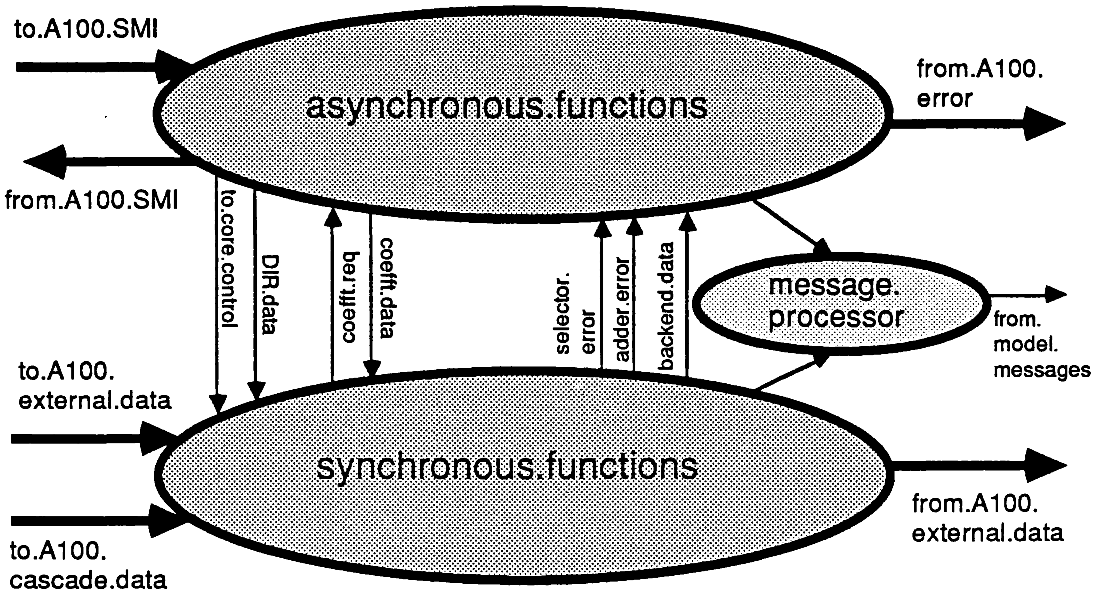

Figure 5: A100.model occam process structure

The overriding assumption made by the model is that all data transfers, both asynchronous and synchronous, are correctly implemented in hardware. Thus, when the processor issues a read request to the IMS A100, the timing is correct, so that the address will be set up in time, and the data will be latched when valid. If this is the case, the model will then show exactly what result will appear at the output. It will NOT tell you when it will appear, nor will it be able to accurately model the pipeline effect of the back end of the IMS A100 device - these are again hardware design issues.

Once these issues are understood, the power of the model can be appreciated. By making all communications via channels, we can simply join a collection of parallel **A100.model** processes via their channels to emulate a cascade. We can then supply data to that cascade either via the `to.A100.external.data` channel, or via the SMI with a write request. We can observe any errors that would normally be generated by the device by watching the `from.A100.error` channel, and see if anything strange is happening by observing the `from.model.messages` channel. We can model any decoded system by modifying the **address.decoder**, and can even model systems using FIFOs, external adders, multiprocessors, or other external devices by writing high level models for those devices, and observing the above constraints of the communication model. Indeed, we can also model interleaved devices, or any other configuration, by virtue of the synchronisation inherent in the occam model.

5.3 Asynchronous process

The asynchronous functions include storage of coefficients, control via the SCR, ACR, and TCR, and data I/O via DIR and DOL/DOH. Note that the main processing area of the IMS A100, ie. the multiplier accumulator array, is often referred to as the "core". The cascade adder, field selector, and output multiplexer are known together as the "backend".

5.3.1 Initialisation

The initialisation phase consists of setting the SCR, ACR, and TCR to their normal power-on default values, and setting several booleans. Note in this version of the model the partial products in the multiplier accumulator array, the coefficients and the cascade shift register are all set to zero. This simplifies matters for users who forget to flush the core, as it avoids transient errors. However, it is important for users to realise that the real device could have anything in these locations.

5.3.2 Main execution loop

The main control loop of the **asynchronous.functions** is an `ALT` between the SMI, requests for coefficients from the core, errors from the backend, and data from the backend. The normal operation would be that data would be written to DIR via the SMI, which is passed to the core. The core will immediately request the Current Coefficient Registers (CCRs) prior to performing the calculation for that cycle. Data is passed from the core to the backend, forming a simple pipeline. If any errors occur in the backend, they will then appear, causing an update of ACR. Finally, the data appears from the back end, which is loaded into DOL and DOH.

Note that an interlock has been included, `DIR.serviced`. Since the **synchronous.functions** model not only the arithmetic, but also pipeline behaviour of the IMS A100, it is possible to deadlock the processes surrounding the model if too many writes to DIR occur at once. To avoid this, the model "holds off" any SMI requests once a major cycle has commenced with a guard on the `ALT`, until the results for that major cycle are received. This does not compromise the model's accuracy in functional terms, since we are deliberately working in a timing regime which is only accurate to the nearest major cycle of the IMS A100, so speed of the SMI cannot be accounted for. Users of the IMS A100 concerned with overrunning the device with excessive DIR writes should use the GO and OUTRDY pins, which indicate when the DIR is sampled, and when the result is stable in DOL/DOH. Thus, by using this to generate wait states if required, the danger is eliminated, although this does compromise maximum throughput with the SMI. The IMS B009 includes a "PROTECT" mode, which enables users to ensure major cycles complete before reading DOL/DOH. The IMS B009 also uses GO to ensure that at no time can too many writes to DIR be made, which is important when using the DIRmode of the IMS B009.

All communication on the SMI is via the `to.A100.SMI` and `from.A100.SMI` channels, and follows the protocol defined above for memory accesses. A simple decoder is used to emulate the internal memory map of the IMS A100. Note however that undefined memory locations have not been implemented - in the real device some of the undefined locations in fact decode to the registers several times. If the SCR is updated, this information is relayed to the core via the **`to.core.control`** channel. This traffic can be observed by enabling the **A100.model** message channel. Bank swaps are implemented by a pointer switch, and errors received generate an **`INT16`** token to be sent on the **`from.A100.error`** channel.

If data is written to DIR, that data is only loaded into the core if the appropriate bit in the SCR is set. The data is sent via the **`to.core.DIR.data`** channel.

A Model Control Register (MCR) is included which is unique to the software model. It is used to switch the cascade input to always supply zero data if required. It is only used by the "**`set cascade`**" command to allow a programmable cascade size without having intermediate "switch" processes between each **A100.model** in the cascade. Users should take care when using the MCR in system models, since it does not exist on the actual device.

When the core requests the coefficients via the **`from.core.coefft.request`** channel, all of the CCRs are sent down **`to.core.coefft.data`**. During this operation, each coefficient is checked to ensure all significant bits are used. If not, a warning message is generated, and the model uses only those bits that are specified by the SCR. For example, if the user loads the value #0011 (hex) into a coefficient, and the SCR is set to only use 4-bit coefficients, the core will receive the value #0001 (hex), and a warning message will be generated. This gets complex when negative numbers are

5.4 Synchronous functions

The **synchronous.functions** receive input data, perform the multiply-accumulate function, apply the field selector, add this with the appropriate cascade data, and output the result to the **asynchronous.functions** and to the outside world via the **`from.A100.external.data`** channel.

The main calculation is performed in **core.evaluation**, and is fully accurate. Input data is accepted from **`to.core.input.data`**, which contains either data from DIR or the DIN port, depending on the setting of the SCR. The **async.sync.interface** sorts out which data stream is used. Note that input data on **`to.A100.external.data`** is **`INT64`**, and that all communications and calculations except those using the SMI are performed with **`INT64`** data. This is to guarantee full accuracy, and allow for users trying to use the model with larger width data than the device itself accepts (see "parameterising the model"). Once calculated, the results are output via a channel to the "backend", so that the core is ready to accept another data sample.

The **backend.and.cascade** receives the core output and applies the field selector, then the cascade adder. The selector will generate errors for overflow or underflow, and will indicate which with an error message. The selector also behaves like the silicon in selecting the correct bits for the output, even if an error occurs, as does the back end adder. Both also set the appropriate bits in the ACR when errors occur. Once these have been performed, the result is passed as an **`INT64`** via **`from.A100.external.data`** to the outside world, and to DOL/DOH via **`from.backend.data`**.

5.5 Parameters and Constants

The IMS A100 model is parameterised to allow knowledgable users to explore the potential for variants of IMS A100 silicon designs. These parameters are MASTERed in the **A100.model**, and comprise the memory map, bit decodes of control registers, and major silicon implementation parameters such as **`number.of.stages`**, **`input.data.width`**, and **`output.data.width`**. These enable users to explore, for example, the benefits of a 64-stage A100, or one with 32-bit outputs, or perhaps 8-bit inputs. These constants are used by the model to constrain its behaviour to the current silicon implementation. For this reason, the **A100.model** is referred to as a "generic" IMS A100 model, since we can model any basic variant of the device by simply altering these parameters.

6.1 Introduction

The **B009.driver** is a stand-alone occam program which is bootstrapped into the IMS T212 of an IMS B009 during startup of the IMS D703. It provides access to all the facilities of the IMS B009, without having to learn the intricacies of the block move, address mapper, or protection modes of the board. The driver has a straightforward set of commands which are defined below, which enable users to use the IMS B009 with very little development effort. For users wishing to develop optimised drivers, the **B009.driver** supplied provides an excellent framework, since all of the modes of operation of the board, together with all special memory locations, are already defined and their use demonstrated. The basic structure of the IMS B009-2 is shown in Figure 6.

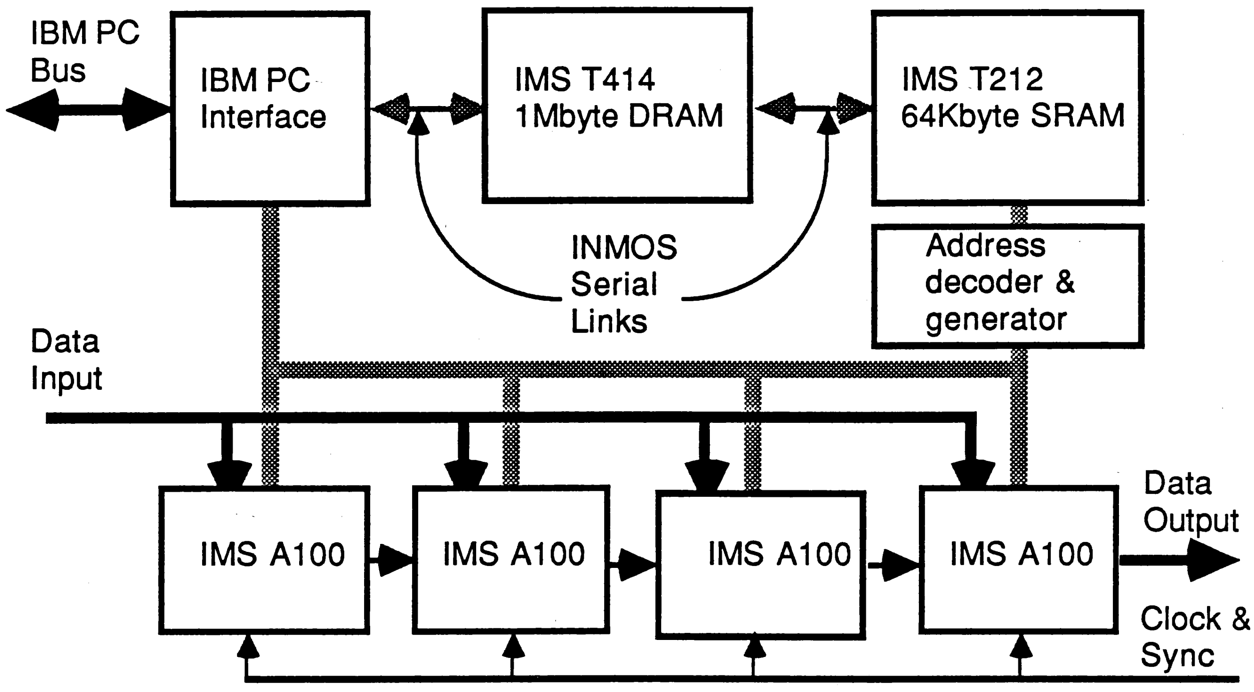

Figure 6: Overview of the IMS B009-2

The buffers within the driver enable the user to perform true concurrent processing with an IMS B009-2 or an IMS B009-1 with another transputer supplying data. A task can load data into the buffer, and with a single command request the driver to pass up to 16k words of data through the IMS A100 cascade without intervention from the host transputer.

Since the protocol for communicating with the driver is defined below, and a link adaptor interface to the IBM PC is provided on the IMS B009, users can bootstrap the IMS T212 with this driver and communicate with it via any MS-DOS application. This means that an application written in C, Pascal, etc on the IBM PC can use the IMS B009 with very simple code. To assist users in developing suitable procedures for talking to the link adaptor, examples are provided in Appendix A written in Pascal.

6.2 Principles of Operation

The driver is based on a simple command decoder, which accepts a command token, and passes control to the appropriate procedure which receives the remaining parameters and performs the appropriate function. All communication uses `INT16` throughout, except when the descriptor is returned as a `[]BYTE` array. Note that the driver is a purely sequential program, ie. no `PAR`s are present in the code. This is because during most operations the full memory bandwidth of the IMS T212 is required, and any interruption to a data transfer would corrupt some modes of operation. Ambitious users may, however, be able to use **PRI PAR** if a parallel implementation appears suitable.

During initialisation, the driver performs an **ALT** to determine which link is to be used for communication. This link is then assigned a pointer (with an abbreviation), and that link used for all future communications. This is a useful means of avoiding the problems of different link configurations, since any of the IMS T212's links can be used to boot and talk to the driver. Note that after booting, the **controller** expects the driver to respond to a request for a descriptor. Thus, if users are creating their own drivers, this facility must be allowed for, or the relevant code in the **controller** modified.

The memory map of the driver is shown in Figure 7. A full description of the facilities of the IMS B009 is given in the **IMS B009 Reference Manual**, however a brief review of the main operational modes is given below. Examples of the uses of this driver are given in the examples contained in Appendix A of the IMS D703 User Manual.

6.3 Operational modes of the IMS B009

In normal operation, the memory map of the IMS B009 behaves as a conventional memory-mapped machine, as shown in Figure 6. However, the key to the IMS B009's performance depends on making best use of the full bandwidth of the External Memory Interface (EMI). To do this, several special modes of operation are provided in hardware, and are selected by bits in the B009 Status Register (B9SR). Note that all special modes apply only to *positive* address space, ie. all physical addresses from #0000 to #7FFE. These correspond to logical addresses #4000 to #7FFF (as used in occam **PLACE** statements to ensure consistency across different wordlength machines).

When passing data through the cascade of IMS A100s, all input data must be written to a single memory location, and all output is read from a single memory location. In order to move data at maximum speed, a block move must be performed with the IMS T212, otherwise **SEQ** loop code is executed which degrades the effective data rate substantially. However, a block move generates a sequence of incremented addresses for both source and destination, which is not consistent with what is required for passing data through the IMS A100 cascade. Therefore, the special modes all perform some form of address modification, so that the functional needs are met at full IMS T212 EMI speed.

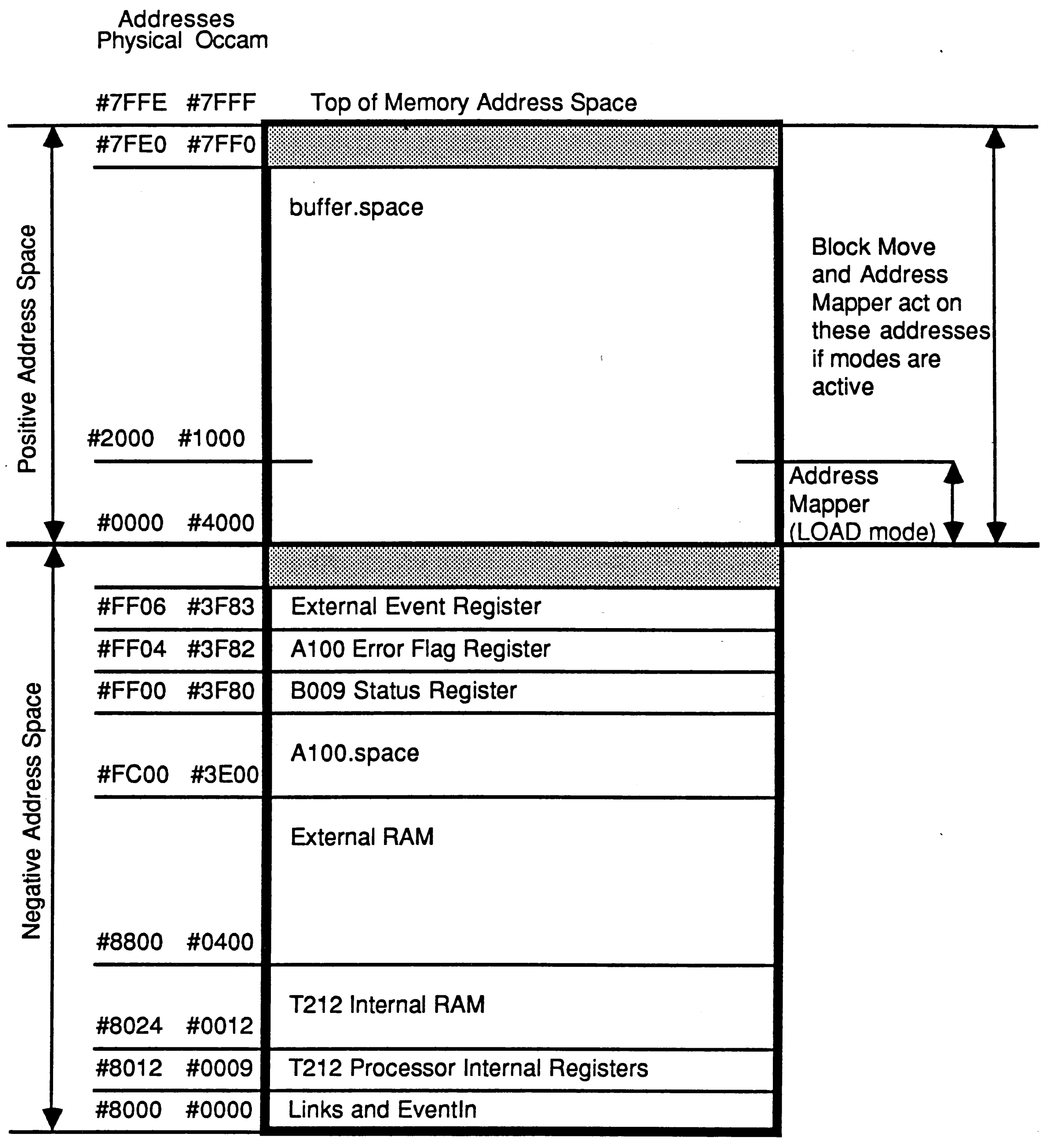

Figure 7: Memory Map of the B009 Driver

Relative Address	Contents	Comments
0-31	UCR[0..31] Device 0	Organised so that a single Block Move can load all 128 coefficients
32-63	UCR[0..31] Device 1	
64-95	UCR[0..31] Device 2	
96-127	UCR[0..31] Device 3	
128-159	CCR[0..31] Device 0	
160-191	CCR[0..31] Device 1	
192-223	CCR[0..31] Device 2	
224-255	CCR[0..31] Device 3	
256	SCR Device 0	For many applications, can predefine all these registers' contents, and thus initialise the complete cascade with a single Block Move.
257	SCR Device 1	
258	SCR Device 2	
259	SCR Device 3	
260	ACR Device 0	
261	ACR Device 1	
262	ACR Device 2	
263	ACR Device 3	
264	TCR Device 0	
265	TCR Device 1	
266	TCR Device 2	
267	TCR Device 3	
268	DIR Device 0	
269	DIR Device 1	
270	DIR Device 2	
271	DIR Device 3	
272	DOL Device 0	DOL/DOH pairs so they can be read by an IMS T212 as an **INT32**
273	DOH Device 0	
274	DOL Device 1	
275	DOH Device 1	
276	DOL Device 2	
277	DOH Device 2	
278	DOL Device 3	
279	DOH Device 3	
320	DIR Devices 0-3	Common DIR for all devices
384	**b009.status.register**	Controls the main addressing modes of the IMS B009
386	**a100.error.register**	Shows if an IMS A100 error has occurred
387	**ext.event.register**	Shows if external event has occurred from the IMS T212 EMI connector.

Base address is: **A100.base.address**.

Memory vector is **A100.space**. Device 0 is closest to the final output of the cascade.

Table 1: IMS A100 Address Space on the IMS B009.

Register	Bit Number	Read/Write	Description
EXT.event.register	0	Read Only	If set, a memory request has been made on the IMS T212 EMI connector. This is used to distinguish between an event generated by the external event signal on the top edge connector, and an error from one of the IMS A100s, when an EventIn occurs on the IMS T212.
a100.error.register	0	Read Only	If set, an error has been generated by one of the IMS A100s. This is used to distinguish between an event generated by the EMI, and an error from one of the IMS A100s, when an EventIn occurs on the IMS T212.
b009.status.register	0	Read/Write	Spare
b009.status.register	1	Read/Write	Spare
b009.status.register	2	Read/Write	**EXT.clock**: If 1, use clock from the external 96-way connector. If 0, use the clock source selected by the switch SW3-1.
b009.status.register	3	Read/Write	**PROTECT.mode**: If 1, hold off reads until data is guaranteed valid in DOL/DOH registers. If 0, reads can occur immediately following a write to DIR.
b009.status.register	4	Read/Write	**LOAD.mode**: If 1, the address mapper is available in positive address space. If 0, normal RAM is available.
b009.status.register	5	Read/Write	**MAP.mode**: If 1, the address mapper is used to translate all positive addresses. If 0, addresses are used without translation.
b009.status.register	6	Read/Write	**DIR.mode**: If 1, all writes in positive address space are translated into writes to the common DIR. If 0, addresses are used unmodified.
b009.status.register	7	Read/Write	**BLOCK.mode**: If 1, addresses in positive address space are decoded so as to provide high speed I/O through the IMS A100s during calculations. See DIRmode and DIRDOLmode on the next page.

Table 2: Bit Definitions for the IMS B009 control registers

6.3 Operational modes of the IMS B009

6.3.1 DIRDOLmode

In this mode, one major cycle of the IMS A100 cascade is achieved for every two read/write operations (ie. four memory cycles). This means that for the IMS B009-1 and IMS B009-2 as supplied, the maximum continuous data rate using data through the SMI is 600ns per sample, or 1.66Msamples/s. (The 600ns EMI cycle is due to delays through the address decoder, the speed of the SRAMs used, and the requirement to provide the address mapper "on the fly", thus necessitating one wait state. Faster SRAMs and decoder would enable the wait state to be removed, thus providing a 2.5Msample/s maximum data rate.)

This mode operates by decoding the addresses as follows:

Cycle	**Address**	**Description**
Read	Even (A1=0)	Normal Read cycle
Write	Even (A1=0)	External RAM is disabled. Data is written to DIR of all four IMS A100s
Read	Odd (A1=1)	External RAM is disabled. Data is read from DOL of the last device of the cascade, ie. device 0
Write	Odd (A1=1)	Normal Write cycle

The normal use of the DIRDOLmode is to commence the block move of data from an even word address to an even word address. The input data is expected to have been stored in every alternate location, ie. **source**, **source+2**, **source+4**, etc. The result will be written into **dest+1**, **dest+3**, **dest+5**, etc. Thus, once the data is loaded in the correct manner into the buffer, a single block move of the form:

```
[dest FROM 0 FOR size] := [source FROM 0 FOR size]
```

will pass all the data through the IMS A100s, and collect the results. To avoid the problems associated with preordering the data, the address mapper can be used (MAPmode) in conjunction with DIRDOLmode to ensure data stored in a contiguous array can be fed directly into the IMS A100s, and the outpu data is stored in a contiguous array. see example 1 in Appendix A of the IMS D703 User Manual for further details.

6.3.2 DIRmode

When performing the first *n* major cycles of the IMS A100 cascade, where *n* is the number of stages of the cascade, many situations do not require the program to gather the first *n* outputs, since they are partial products that are meaningless. In this situation, a simpler decode is provided:

Cycle	**Address**	**Description**
Read	All	Normal Read cycle
Write	All	Write data to DIR of all IMS A100s

This mode is useful for performing functions such as flushing the cascade. Note, however, that in this mode the IMS T212 can supply data faster than the IMS A100s can accept it; however the IMS B009 uses the GO pin to ensure that this does not occur.

6.3.3 MAPmode

For some DSP algorithms, the data must be reordered before and/or after processing. This normally requires the host processor to perform this operation explicitly, which is very expensive computationally. To avoid this problem, a 4k word lookup table is provided, which can be used to convert a block of sequential addresses to any arbitrary sequence, with no processing overhead. The address mapper takes as input address pins A1-A12, and uses these as the address into the lookup table. The output is then used as the address to be decoded. Note this only occurs for positive addresses, ie. A15 is 0. A13 and A14 are passed unmodified.

For descriptions of examples of using the address mapper for algorithms, refer to the **IMS D703 User Manual** Appendix A. Note that this facility is very useful when combined with the DIRDOLmode to avoid data pre/post organisation.

With DIRDOLmode as described above, data input could not be simply loaded as a vector with a block move, since every other location was ignored by the hardware. Thus, the data would need to be reorganised by the IMS T212 to ensure consecutive data items are read correctly. This is a very expensive operation computationally. If the address mapper is used, it can convert the addresses in such a manner that the input data can be passed from the host to the IMS T212 via a block move to the **data.buffer**, and the results stored in consecutive locations ready for block move transfer up the link to the host. For a description of this, refer to the **IMS D703 User Manual** Appendix A.

6.3.4 LOADmode

The address mapper needs to be loaded with data, yet the IMS T212 address space is already occupied with external RAM. To avoid wasting 4k words of address space, a special load mode provided, which replaces the normal RAM in positive address space with the address mapper RAM. Since the address mapper is only 4k words, it replicates across the entire positive address space. Note that the hardware ensures that the external RAM cannot corrupt the address mapper RAM, and vice versa, in any of the addressing modes.

6.4 Driver Commands

The following sections describe the commands of the driver, including the protocol required. Since this protocol is that seen from the link by the IMS T212, the commands can be sent by any host via a link adaptor, or by any transputer. The **data.buffer** occupies almost all of the positive address space, with the exception of the top few locations. This is to avoid possible address calculation overflows that have been observed in some compilers.

All arguments in all commands are **INT16**, except for the descriptor which is returned as **[]BYTE**s. When the arguments are read, they are validated, and if correct the *success.flag* will be **TRUE**. If any data is to be returned, the *success.flag* will be followed by the data to be returned. If data is being supplied to the driver, it will only accept it if the arguments defining the destination of the data are valid. Therefore, these protocols include the return of a *success.flag* prior to accepting the input data.

If *success.flag* is **FALSE**, it will always be followed by a reason code. These are:

tag	Reason
bad.address	The address is out of range
bad.size	The address + size will generate at least one address out of range
bad.command	The comand tag could not be decoded

Note that the verification performed on commands by the driver clearly makes it less than optimal for performance. Some suggestions are given later as to ways in which the driver can be optimised for specific needs.

Users should also be careful that if the address mapper is used, the start addresses will be prior to mapping. Therefore, the specific calls must take into account any address translation scheme currently active.

6.4.1 Read A100 to link

from.application ? read.A100.to.link; *start.address*; *size*
to.application ! *success.flag*;

to.application ! *size*; [*data* **FROM (A100.base.address+***start.address***) FOR** *size*], or

to.application ! *failure.code*

Reads *size* locations from the IMS A100s, starting at *start.address* relative to the base address of the A100 address space.

6.4.2 Write A100 from link

from.application ? write.A100.from.link; *start.address*; *size*
to.application ! *success.flag*;

from.application ! [*data* **FROM (A100.base.address+***start.address***) FOR** *size*], or

to.application ! *failure.code*

Writes *size* locations in the IMS A100s, starting at *start.address* relative to the base address of the A100 address space.

6.4.3 Write DIR from link

from.application ? write.DIR.from.link; *size*
to.application ! *success.flag*;

from.application ! [*data* **FROM (A100.base.address+***start.address***) FOR** *size*], or

to.application ! *failure.code*

Writes *size* data samples into all the IMS A100s' DIR registers. For this operation, DIRmode is used.

6.4.4 Read buffer to link

from.application ? read.buffer.to.link; *start.address*; *size*
to.application ! *success.flag*;

to.application ! *size*; [*data* **FROM (data.buffer.base+***start.address***) FOR** *size*], or

to.application ! *failure.code*

Reads *size* locations from **data.buffer**, starting at *start.address* relative to the base address of the data buffer.

6.4.5 Write buffer from link

from.application ? write.buffer.from.link; *size*
to.application ! *success.flag*;

from.application ! [*data* **FROM (data.buffer.base+***start.address***) FOR** *size*], or

to.application ! *failure.code*

Writes *size* data samples into **data.buffer**, starting at *start.address* relative to the base address of the data buffer.

6.4.6 Write DIR from buffer

from.application ? write.DIR.from.buffer; *start.address*; *size*
to.application ! *success.flag*; then if error,

to.application ! *failure.code*

Writes *size* data samples from **data.buffer** into all the IMS A100s' DIR registers, starting at *start.address* relative to the base address of the data buffer. For this operation, DIRmode is used.

6.4.7 Process buffer data out (16 bit mode)

from.application ? process.buffer.data.out16; *source.start.address*; *dest.start.address*; *size*
to.application ! *success.flag*; then if error,

to.application ! *failure.code*

Write data from **data.buffer** into DIR, and read the DOL of the last device, *size* times. For this operation, DIRDOLmode is used.

6.4.8 Process buffer data out (24 bit mode)

from.application ? process.buffer.data.out24; *source.start.address*; *dest.start.address*; *size*
to.application ! *success.flag*; then if error,

to.application ! *failure.code*

Write data from **data.buffer** into DIR, and read both the DOL and DOH of the last device, *size* times. Since two reads are required for every write operation, block move cannot be used; this operation is thus comparatively slow compared with **process.buffer.data.out16**.

6.4.9 Read mapper to link

from.application ? read.mapper.to.link; *start.address*; *size*
to.application ! *success.flag*;

to.application ! *size*; [*data* **FROM (address.mapper.base+***start.address***) FOR** *size*], or

to.application ! *failure.code*

Reads *size* locations from **address.mapper**, starting at *start.address* relative to the base address of the address.mapper. For this operation, LOADmode is used.

6.4.10 Write mapper from link

from.application ? write.mapper.from.link; *size*
to.application ! *success.flag*;

from.application ! [*data* **FROM (address.mapper.base+***start.address***) FOR** *size*], or

to.application ! *failure.code*

Writes *size* data samples into **address.mapper**, starting at *start.address* relative to the base address of the address mapper. For this operation, LOADmode is used.

6.4.11 Enquiry

from.application ! enquiry; *identity* where:

identity is	**driver.id**	descriptor of the driver; returns: **SIZE** *id*; **[]BYTE** *id*
	status.register	Contents of status register; returns: *status.register*
	error.register	Contents of A100 error register; returns: *error.register*
	ext.event.register	Contents of external event register; returns: *ext.event.register*

6.4.12 Status modifier

from.application ! status.modifier; *qualifier*

where:

from.application ! enquiry; *identity*

where:

qualifier is	**MAP.mode.enable**	Enable address mapper
	MAP.mode.disable	Disable address mapper
	PROTECT.mode.disable	Enable protected DOL/DOH access
	PROTECT.mode.disable	Disable protected DOL/DOH access
	EXT.clock.enable	Use clock from edge connector for A100s
	EXT.clock.disable	Use on-board 20MHz clock for A100s
	disable.all.modes	Disable all above modes

6.5 Adding new commands

Adding new commands to the driver is very straightforward. To add the command itself, the easiest way is to copy an existing command, and modify it. The main area to be careful of is that the driver protocol is defined in all user applications, therefore it is wise to update the protocols in the "standard harness" as well as all the applications.

6.6 The B009 Emulator

For those users who do not have an IMS B009, or who wish to debug **applications** written using the **b009.driver**, the **b009.emulator** is also provided. This provides exactly the same facilities as the driver described above, but uses the occam **A100.models** rather than the hardware. Thus, it provides an accurate emulation of the complete facilities of the IMS B009 as seen by the **B009.driver**.

Additional facilities have been provided to make the emulator a useful debugging tool Most useful is the command "**`set message 21 on`**", which displays the parameters passed to the driver for each call made to it. If these messages do not reveal the problem, "**`set messages 20`**" can be used to display every memory access, thus showing possible illegal access, or invalid data.

The provision of the **B009.emulator** enables users of the IMS D703 to develop **applications** using any transputer development system with an IMS T414 or IMS T800, and at least 1Mbyte of RAM. If the standard procedures provided are used, this software will be able to execute on the IMS B009 (if present) unmodified.

6.7 Optimising the driver

The driver checks all operands, and tells the user if the parameters are correct before data transfer takes place, to avoid unnecessary block moves. Unfortunately, this incurs an overhead in both the driver and the applications, due to the generation, communication, and checking of the **`success.flag`** and **`failure.flags`**. Thus an immediate improvement can be made by removing the "safety" of these checks.

The I/O bandwidth between the host and the IMS T212 can be increased by the use of multiple links in parallel. the transputer link DMA engines operate concurrently, so that two links will deliver almost twice the throughput of a single link. This could be useful, for example, when transferring complex data, using one link for real data, and another for imaginary data. More application-specific drivers can consider doing more of the application tasks local to the IMS T212. For example, by storing more of the data in the available RAM of the IMS T212, and providing instructions for shuffling data between temporary buffers in positive and negative address space, link traffic can be minimised. This also applies to multiple address maps, which could again be stored local to the IMS T212.

The address mapper offers a wide range of possibilities, particularly if multiple maps are stored in the mapper, and different base addresses are used to switch between different translations. This may be useful if both straight DIRDOLmode style data ordering and scrambling of the form used in the Prime Number Transform is required, eg. correlating a DFT result.

Finally, fine tuning of an application can be achieved with the IMS B009-2 or similar configuration by careful balancing of the processing of both the IMS T414/IMS T800 host, and the IMS T212. A simple example of this is given in the convolution example. Other examples of optimising transputer occam programs are described in a separate transputer application note from INMOS.

```
{Setups for the globals used}

const addrOffset = $150 ;  {Port offset for link adaptor}
      timeOut    = 1000 ;  {No. of tries sending data}


      writeA100FromLink=1; {Write into the IMS A100s memory map}

      enquiry=10;  { Protocol to enquire T2 driver's ID }
      driverId=100;

      successfulCommand = -1;

var errorFlag : boolean;

type WrkString = string[80] ;
     coeffArray= array[0..63] of integer;

{ Procedure that resets the IBM's subsystem,
  resetting T4,T2 and link adaptor}

procedure ResetLink;
begin
  port[addrOffset+16]:=1;
  port[addrOffset+17]:=0;    {analyse}
  delay(100);
  port[addrOffset+16]:=0;
  delay(100);
end;


{Function giving the state of the IBM's subsystem,
        TRUE if OK, FALSE if error set}

{Check transputer error flag}
function transputerOK:boolean;
begin
  if (port[addrOffset+16] and 1)=0 then transputerOK:=false else
  transputerOK:=true;
end;

{Send a byte to transputer, waiting 10ms between trys,
 trying timeOut times}

procedure Send(data:byte);
var counter :integer;
begin
  counter:=0;
  while (((port[addrOffset+3] and 1)=0) and (counter<timeOut))
         and not(errorFlag)
  begin
    counter:=succ(counter);
    if not(transputerOK) then
    begin
      writeln('%%FRNT-F-ErrFlgSet: Transputer error flag set');
      errorFlag:=true;
    end;
    Delay(10);
  end;
  if counter<timeOut then port[addrOffset+1]:=data
  else writeln('%%FRNT-E-TimeOut: Time out when sending byte');
end;

{Get byte from the transputer, checking error flag}

function Receive: byte;
var gotByte : byte ;
begin
  if not errorFlag then
  begin
    repeat
      if not(transputerOK) then
      begin
        writeln('%%FRNT-F-ErrFlgSet: Transputer error flag set');
        errorFlag:=true;
      end;
    until ((port[addrOffset+2] and 1)=1) or errorFlag;
    gotByte:=port[addrOffset];
  end;
  Receive:=gotByte;
end;

procedure BootNetwork(filename:WrkString);

var bootFile : file of byte;
    data     : byte;
begin
  ResetLink; { Reset the transputers in question ready for booting }
  assign(bootFile,filename);
  reset(bootFile);
  while not(EOF(bootFile)) do
  begin
    read(bootFile,data);
    Send(data);
  end;
  close(bootFile);
end;

{ Send INT16 to link }

procedure Send16( data:integer);

begin
  Send(data and $FF);   {Least significant byte}
  Send(data shr 8);     {Most significant byte}
end;

{ Receive INT16 from link }

function Receive16 : integer;

var data:byte;

begin
  data:=Receive;
  Receive16:=(Receive*256)+data;
end;


{ get & display the driver's ID }
```

```
procedure displayID;
var i,status:integer;

begin
  Send16(enquiry);
  Send16(driverId);
  status:=Receive16;
  if status<>successfulCommand then
    writeln ('Enquiry failed! Error code produced was ',Receive16)
  else
  begin
    for i:=0 to Receive16 do
      write(chr(Receive));
    writeln;
  end;
end;

procedure loadCoeff(data: coeffArray);
var i,status:integer;

begin
  Send16(writeA100FromLink);
  Send16(0); {Start address}
  Send16(64); {Length}
  status:=Receive16;
  if status<>successfulCommand then
    writeln ('Loading failed - error code was ',Receive16)
  else
  begin
    for i:=0 to 63 do
      Send16(data[i]);
  end;
end;

var data  : coeffArray;
    i     : integer;

begin
  BootNetwork('imsd703.t2');
  Send(255); { The driver used requires a byte sent to find
               which link the IMS~T212 uses for communications}
  displayID;
  for i:=0 to 63 do
    data[i]:=i;
  loadCoeff(data);
end.
```

The following files in the Turbo Graphix Toolbox library require modification to function as used in the IMS D703 **front end**:

typedef.sys
kernel.sys
newaxis.hgh

The modifications are as follows:

typedef.sys	The constant `MaxPlotGlb` should be changed to 2200. The **front end** will not produce graphs successfully without this modification
kernel.sys	the screen dump procedure should be replaced by the one listed below, to give correct aspect ratios and reasonable speed. Note this is not fundamental to the operation of the **front end**
newaxis.hgh	the statement (at line 167 in our version):

```
  if (abs(Yk0 - Yk1) >= 35) and
(abs(Xk2 - Xk1) >= 150) then
```

should be changed to

```
  if true then
```

to allow axis numbering on the graphs produced by the package.

The headers in `runsim.pas` should then be modified to point to the new versions of these files.

```
procedure HardCopy(Inverse : boolean; Mode : byte); { For Epson MX }

var
  I, J, Top : integer;
  ColorLoc, PrintByte : byte;

procedure DoLine(Top:integer);

function ConstructByte(J, I : integer) : byte;
const
  Bits : array[0..7] of byte = (128,64,32,16,8,4,2,1);
var
  CByte, K : byte;
begin
  I := I shl 2;
  CByte := 0;
  for K := 0 to Top do
    if PD(J, I + (K div 2)) then
      CByte := CByte or Bits[K];
  ConstructByte := CByte;
end; { ConstructByte }

begin { DoLine }
  Write(Lst,Chr(27),'L',Chr(Lo(2*(XScreenMaxGlb+1))),
                        Chr(Hi(2*(XScreenMaxGlb+1))));
  for J := 0 to XScreenMaxGlb do
  begin
    PrintByte := ConstructByte(J, I);
    if Inverse then
      PrintByte := not PrintByte;
    Write(Lst, Chr(PrintByte));
    Write(Lst, Chr(PrintByte));
  end;
  Writeln(Lst);
end; { DoLine }

begin { HardCopy }
  Top := 7;
  ColorLoc := ColorGlb;
  ColorGlb := 255;
  Writeln(Lst);
  Write(Lst,chr(27),chr(65),chr(8));
  for I := 0 to ((YMaxGlb + 1) shr 2) - 1 do
    DoLine(7);
  I := ((YMaxGlb + 1) shr 2);
  if (YMaxGlb + 1) and 7 <> 0 then
    DoLine((YMaxGlb + 1) and 7);
  WriteLn(Lst,chr(27),chr(50));
  ColorGlb := ColorLoc;
end; { HardCopy }
```

INMOS Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
UK
Telephone (0454) 616616
Telex 444723

INMOS Corporation
PO Box 16000
Colorado Springs
CO 80935
USA
Telephone (303) 630 4000
TWX 910 920 4904

INMOS GmbH
Danziger Strasse 2
8057 Eching
Munich
West Germany
Telephone (089) 319 10 28
Telex 522645

INMOS SARL
Immeuble Monaco
7 rue Le Corbusier
SILIC 219
94518 Rungis Cedex
France
Telephone (1) 46.87.22.01
Telex 201222

INMOS International
Room 308 Kowa No. 16 Annex
9-20 Akasaka 1-chome
Minato-ku
Tokyo 107
Japan
Telephone 03-505-2840
Telex J29507 TEI JPN

March 1987

72 TDS 105 00