

Loading transputer networks

INMOS Technical Note 34

INMOS Limited

72-TCH-034



You may not:

1. Modify the Materials or use them for any commercial purpose, or any public display, performance, sale or rental;
2. Remove any copyright or other proprietary notices from the Materials;

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

INMOS, IMS, OCCAM are trademarks of INMOS Limited.

INMOS Limited is a member of the SGS-THOMSON Microelectronics Group.

Contents

1	Introduction	4
1.1	Development	5
1.2	Characteristics	6
2	The TDS Extractor	7
3	Bootstrap and Loaders	11
3.1	The bootstrap	11
3.2	The bootloader	13
3.3	The loader	14
4	The loading message structure	15
4.1	Command structure	15
	Bits 7..6	16
	Bits 5..0	16
4.2	Loader action	16
	MESSAGE	17
	NUMBER	17
	FUNCTION	17
	Single transputer	18
	Multiple load	19
4.3	RS232	20
	Startup sequence	20
	Data encoding	21
5	Bootstrap code	21
6	Bootloader code	24
7	Loader occam	26
	References	29

1 Introduction

The Transputer Development System is a software package which is used for developing occam applications for execution on transputers. The TDS contains facilities for loading and running code on the host computer (which may be a transputer) or on a network of transputers connected to the host. This technical note describes the loading mechanism employed by the TDS to load code onto a network rather than onto the host.

Occam contains constructs which are used to specify the allocation of code to different processors in the network. The TDS compiler implements a subset of these allocation facilities which allows users to allocate occam compilation units to different processors. This specification is called the configuration. Two other utilities are used in the process of sending code to a network of transputers, these are the `EXTRACT` utility and the `LOAD NETWORK` utility, both of which are described below.

The following example configuration specifies a network of two processors which are connected by channel datalink placed on both processors at transputer link zero. The content of the compilation unit `root` is to be loaded onto the processor attached to the host computer and the content of the compilation unit `node` is to be loaded onto the other processor in the network. The textually first processor in a network is assumed to be connected to the host computer by a transputer link or serial line for loading and is referred to as the root processor.

```
{{{ PROGRAM using two processors
{{{F
... SC root (in)
... SC node (out)
CHAN OF BYTE datalink
PLACED PAR
PROCESSOR 1 T4
    PLACE datalink AT 4 -- Link 0 in
    root (datalink)
PROCESSOR 2 T4
    PLACE datalink AT 0 -- Link 0 out
    node (datalink)
}}}
```

The compiler checks that the configuration described by the user is valid and that every processor is loadable from the root processor. The compiler also checks that the code to be loaded to each processor is available and is compiled for the correct processor type. The compiler produces a fold containing a description of the configuration specified by the user. This

description is used by the extraction and loading utilities to control the distribution of code to the network. The extraction utility brings together all the different blocks of code to be sent to the network. At the same time bootstraps and routing and loading information is included with the code to initialise the processors and direct the code to the intended locations in the memory of the target processors. The loading utility sends the extracted code to the network, controlling any interaction with the root processor and reporting any failure to the user.

The TDS is designed to enable users to develop their network software easily and quickly. This environment calls for a network loading mechanism which is simple, reliable and reasonably efficient. It is expected that applications which require special performance from the loading software, such as loading every processor in a network with identical code but not, perhaps, knowing the topology of the network, would have a specific loading mechanism designed.

1.1 Development

The loading strategy used by the TDS was specifically developed to satisfy the requirements of the TDS, it is not the only way of distributing code to a network of transputers and may not be the best mechanism for many environments. The decisions behind the scheme can be more easily understood if the requirements are stated. These are:

1. Any code to be sent to the network should only be transmitted from the host to the network once, even if it is to be loaded at different addresses on different processors.
2. Blocks of code may be loaded in any order to any location on any processor.
3. The loading mechanism should not permanently occupy space in the target processor's memory.
4. The loading strategy should be reasonably efficient for the number of transputers likely to be used with the TDS - say 500.
5. The loader should be small enough to fit in internal memory so that a processor with large amounts of memory can be loaded via a processor with no external memory.
6. Each type of transputer must be supported.

To load code into every processor, it is necessary for a loader to be resident on each processor. This loader must be able to load code into the local memory and also pass code on for other processors. Requirement 1 and requirement 3 above are antagonistic for the design of the loader. Requirement 1 demands a loader which is capable of loading code to other processors when it has finished loading code into the local memory, while requirement 3 demands that space occupied by the loader code can be re-used for code being loaded into the local memory.

The sixth requirement, that all types of transputers be supported, had quite a different effect upon the loading scheme. The TDS had to support transputer types which did not boot into the same state and whose external links were at different addresses. This demanded that the bootstrap and loader for each processor in the network be directed to that processor alone.

1.2 Characteristics

The requirements placed upon the design of the loading scheme resulted in the characteristics described below.

Each processor is pre-loaded with a bootstrap and loaders which perform initialisation and loading tasks. The first program, the bootstrap, initialises the registers, the link and event process words and the queue pointers of the transputer and then loads the second program, the bootloader. The bootloader is a simple loader capable of loading code to contiguous blocks of memory, it is used to load the third program, the loader, and later in the load sequence, additional blocks of code not loaded by the loader. The loader performs the tasks of loading code into local memory as well as distributing code and information to other processors in the network. The bootstrap and loaders are grouped together as a set of message packets which are sent to each processor by the host before any other loading information.

The development system on the host computer, the TDS, maintains all knowledge of the structure of the network. This allows the loader on each processor in the network to be simple. At each stage it is told exactly what to do by the communications received from the host.

The bootstrap and loaders for each processor in the network are transmitted from the host to the processor being booted, they do not propagate from one processor to the next. To all processors, apart from the processor being booted, the bootstrap and loader code is indistinguishable from any other code.

Loading code to the network proceeds in distinct phases. Firstly, the bootstrap and loaders for each processor are transmitted from the host in a man-

ner which ensures that a processor which lie on the route to the recipient processor has itself already received its own bootstrap and loader. Secondly, the code to be loaded is transmitted from the host and propagated to all recipient processors. Thirdly, code to call the loaded code is transmitted from the host in a sequence which ensures that a processor which has received its calling sequence will not receive any more loading information from the host and may therefore run this call code. The bootstrap and loaders are loaded onto a processor in the lowest available addresses (nearest to MOSTNEGINT). The code to be run on a processor is loaded so that the most negative addresses will be workspace. Normally, therefore, the loader resides in memory which will become the workspace of the application being loaded. If, however, there is a requirement to load code into the space occupied by the loader, then the loader can be overwritten by blocks of code loaded by the bootloader after the loader has terminated.

The loading messages are collections of single bytes and packets of bytes. The single bytes are commands which control the routing and loading of information. The packets of bytes contain transputer code to be loaded into the memory of a transputer. The packets of bytes are 60 bytes or less. The value 60 was chosen for a variety of reasons. Firstly, it is necessary to provide a buffer in the loader for passing code on to other processors and the larger this is the more space the loader uses. Secondly, a message protocol could be devised which simplified the loader if the message length was never greater than 63. Thirdly, the buffer had to be large enough to contain the bootstrap, which is 53 bytes in length, as a single packet.

2 The TDS Extractor

The extraction and loading utilities, provided as part of the TDS, control the loading mechanism. The extract utility determines the order in which processors are loaded and the location of code loaded on every processor and selects the specific bootstrap and loader for each processor in the network. The loading utility sends the code to the network, controlling any interaction with the root processor and reporting any load failures. The functions of the extraction and loading utilities can be performed as one action within the TDS; the descriptions given in this section will be phrased as if this is the mode of operation being described and the term 'extractor' will be used for the combined function. This section gives a brief overview of the extractor and the order in which code is transmitted to the network with particular reference to an example. The bootstrap and loaders are described in more detail in later sections.

From link connection information and processor load data provided by the

compiler, the extractor builds a graph representing the network to be loaded. From this data structure the order in which the processors in the network receive the bootstrap and loader code is determined.

To determine the order, the graph of the network is first pruned to a strict tree structure with only the shortest paths from the host to all the processors remaining. The order is then determined from the tree by the following algorithm.

Boot the root processor (the processor connected to the host). Then for links 0,1,2,3 in turn of the root processor, boot the network attached to the link. If the link is connected to a processor, boot the processor connected to the link, and boot the networks connected to links 0,1,2,3 of the newly booted processor. Note that the links are not necessarily used in the direction defined within the occam configuration.

This can be illustrated with reference to the following example configuration.

```
... SC process.1
... SC process.2
... SC process.3
... definitions and declarations
PLACED PAR
  PROCESSOR 0 T4
    PLACE L1 AT link3.in :
    PLACE L0 AT link1.out :
    PLACE L6 AT link2.in :
    process.1 (L1, L0, L6)
  PROCESSOR 1 T4
    PLACE L2 AT link1.in :
    PLACE L6 AT link0.out :
    PLACE L7 AT link2.in :
    process.2 (L2, L6, L7)
  PROCESSOR 2 T4
    PLACE L0 AT link0.in :
    PLACE L2 AT link3.out :
    PLACE L4 AT link2.in :
    process.3 (L0, L2, L4)
  PROCESSOR 3 T4
    PLACE L3 AT link3.in :
    PLACE L1 AT link0.out :
    PLACE L5 AT link2.out :
    process.1 (L3, L1, L5)
  PROCESSOR 4 T4
    PLACE L5 AT link3.in :
    PLACE L7 AT link0.out :
    PLACE L4 AT link1.out :
    process.3 (L5, L4, L7)
```

The above occam configuration can be represented by the following diagram:

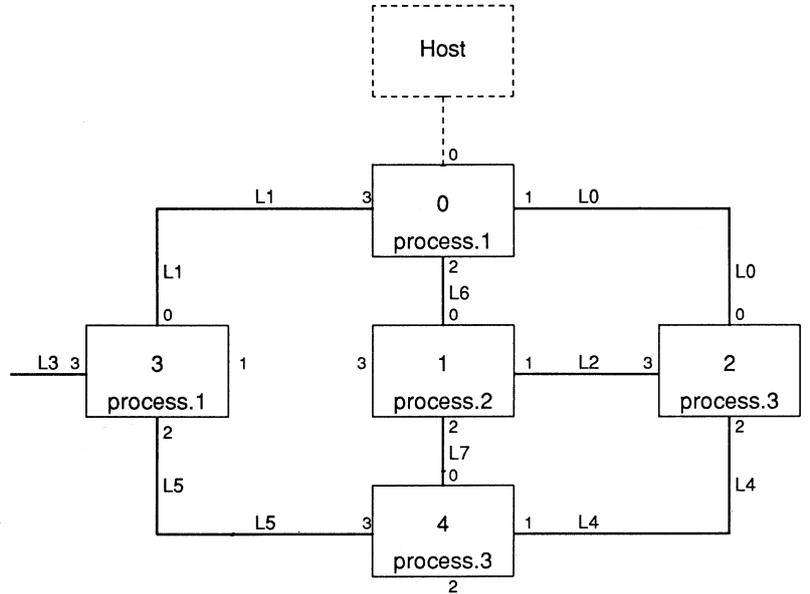


Figure 1: Example network

This example configuration generates the following boot path:

```
processor 0 from host
processor 2 from processor 0 link 1
processor 4 from processor 2 link 2
processor 1 from processor 0 link 2
processor 3 from processor 0 link 3
```

After all of the processors in a network have been booted (loaded with the bootstrap and loaders), the compiled code is transmitted to the network. The code of the procedures to be transmitted to the network is sent in the order in which the procedures are declared in the PROGRAM fold. The loading order is the same as the boot order, each processor taking a copy or not of a code packet, then passing it to zero or more output links.

The SC code loaded to the network shown in figure 1 will be sent in the following order:

```
process.1
  0 load 3 load
process.2
  0 pass 1 load
process.3
  0 pass 2 load 4 load
```

The compiler generates a small amount of code to call the procedure which has been loaded onto each processor, this is referred to as the main program. The main program contains code which initialises the parameters to the application code, the call of that code and, following the code, an instruction which will stop the processor if the application program terminates and returns to the main program. The main program code is loaded so that it is contiguous with the previously loaded application code and is at more negative addresses. The layout of the loaded code and workspace on a transputer is shown in the following diagram:

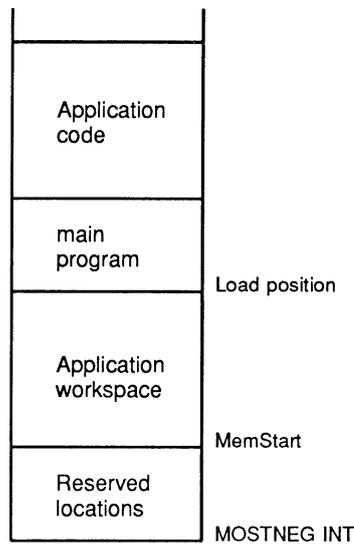


Figure 2: Application code and workspace

The main program code is sent to the network by traversing the pruned tree representing the network in the following 'depth first' manner: For links 0,1,2,3 in turn of the root processor, load the network attached to the link. If the link is connected to a 'new' processor, load the networks connected to links 0,1,2,3 of the new processor, followed by the new processor. Finally load the root processor. A new processor is one which has not previously been encountered during this phase of the loading.

The main body code loaded to the network described above will be sent in the following order:

```
processor 4
processor 2
processor 1
processor 3
processor 0
```

The loading position of the code in any processor is determined by the workspace requirement of the code to be loaded to that processor. The load address is calculated by adding the size of the workspace and a base workspace address. If this load address is less than a minimum value, then the minimum value is used as the load address. The minimum value is the lowest address to which code can be loaded onto a processor without overwriting the workspace of the code doing the loading (the bootloader).

The workspace requirement on a processor may be small and consequently the calculated load address may overlap the space occupied by the loader program, which resides in low memory addresses (nearest to MOSTNEG INT) as described in the next sections. Rather than adjust the loading address to avoid the loader, the code which overlaps the loader is held back in an internal buffer within the extractor. When the distributing phase of the network load has finished, the saved code is sent to the network with the main body code for each processor. The main bodies are loaded remote processor first, so that a processor receiving a main body will not receive any further load path information. The loader can, therefore, return to the bootloader, which can load contiguous code packets which do not require any load directives. This allows the saved code to be loaded to the space previously occupied by the loader.

The bootstrap, bootloader and loader for each processor type are contained within the extractor occam as a table of bytes organised as a sequence of length bytes followed by the specified number of bytes. The table is generated by a program provided with the TDS. This program contains within it a mechanism for inserting transputer instructions directly into the table, and for reading the code of a compiled occam program and adding the intents to the table. The bootstrap and the bootloader are coded directly into the table, the loader is written in occam. The extractor transmits the contents of the table to the network as length byte, code packet pairs.

3 Bootstrap and Loaders

3.1 The bootstrap

After power-on or reset, a transputer waits until it receives a communication on any one of its links. If the value of the first byte of this communication is 2 or greater, then that number of bytes is input from the link into the memory starting at MemStart and the processor starts executing at MemStart. The TDS extractor sends the bootstrap to each processor as a length byte followed by the bootstrap code.

The bootstrap, the first packet of the bootstrap and loader sequence, is a short program which initialises the processor and memory. Section 5 'Bootstrap code' gives the full listing of the bootstrap which is written in transputer assembler instructions. The sequence of actions performed by the bootstrap is as follows:

1. Allocate workspace for bootstrap and loader variables.
2. Reset high and low priority process queues.
3. Clear or set the halt on error flag.
4. Clear error.
5. Initialise all link and event process words to NotProcess.
6. Initialise some of the loader parameters.
7. Load the bootloader.

The bootstrap is loaded by the transputer at MemStart. When the initialisation is complete, the bootstrap loads the bootloader at MemStart and then jumps to MemStart to enter the bootloader. Because the bootstrap loads the bootloader at the same location as itself, the bootstrap is at least two bytes longer than the bootloader (so that the instruction by which control is passed to the bootloader is not overwritten by the bootloader code being loaded). The bootstrap for the T4 transputer is 53 bytes in length and the corresponding bootloader is 51 bytes.

The memory layout for a T4 transputer while the bootstrap is running is given in the following diagram.

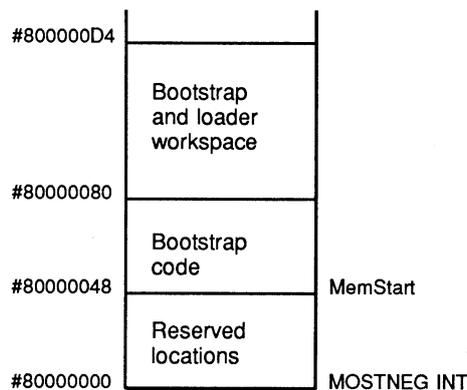


Figure 3: T4 Bootstrap memory usage

Addresses for the T2 and T8 which correspond with those given in the above diagram for the T4 are given in the following table.

Transputer	T2	T4	T8
MOSTNEG INT	#8000	#80000000	#80000000
MemStart	#8024	#80000048	#80000070
Bootstrap top	#8050	#80000080	#800000A8
Workspace top	#808C	#800000D4	#800000FC

3.2 The bootloader

The bootloader, which is the second packet of the bootstrap and loader sequence, is a short program capable of loading contiguous blocks of code into memory. The code of the bootloader, which is written in transputer assembler instructions, is listed in section 6 'Bootloader code'. It loads two different sets of code packets. Firstly, it is used to load the loader and secondly, after the loader has finished, the bootloader loads the main program code packets prior to starting the loaded code. The bootloader performs the following functions:

1. Initialise remaining parameters for loader.
2. Load code from boot link until terminator.
3. Initialise workspace pointer and call code just loaded.
4. Start clock.
5. Prepare to load more code.
6. Go to step 2. The main program code loaded does not return, so this loop is only obeyed twice.

The bootloader is loaded by the bootstrap at MemStart. The bootloader creates the loader buffer starting at the address of the variable with the greatest offset in the workspace reserved by the bootstrap. The loader is then loaded at the first free location after the buffer. The bootloader loads the second set of code packets at an address returned by the loader. The messages input by the bootloader are a sequence of length byte and data packet pairs.

The code position and workspace layout while the bootloader is loading the loader is given in part (a) of figure 4 and the memory layout while the bootloader is loading the final code packets is given in part (b) of figure 4.

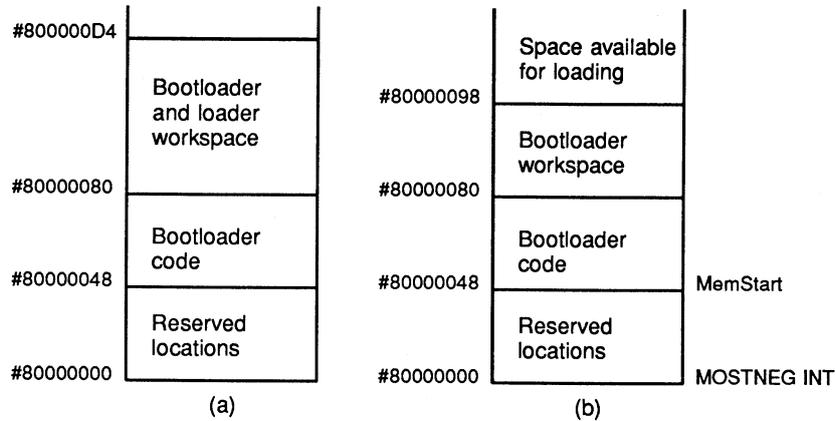


Figure 4: Bootloader memory usage

3.3 The loader

The third component of the bootstrap and loader sequence loaded onto each processor is the loader. The loader is a short occam program which loads and distributes code. It obeys a sequence of commands received from the host which direct it to perform the following functions:

- Load a code packet to the current load address and increment the current load address.
- Output a code packet to a link.
- Set a new current load address.
- Pass commands to a link.

The command structure is described in detail in the next section. The information received by the loader from the host is a stream of single byte commands and packets of code. The commands are nested within bracketing command bytes so that each processor can interpret commands for itself, remove one level of bracketing and pass on commands intended for other processors later in the load path. The commands received change the value of variables within the loader. When packets of code are received by the loader, the value of the variables previously affected by the commands determines the destination of the code. The occam source text of the loader is listed in section 7 'Loader occam'.

The memory layout while the loader is running is as follows.

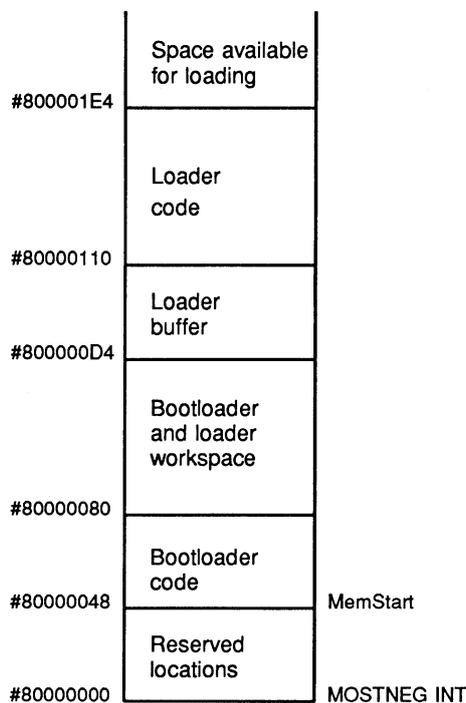


Figure 5: Loader memory usage

4 The loading message structure

4.1 Command structure

Load commands and data transmitted to and through a transputer consist of a word length independent mixture of single bytes and packets of bytes. The single bytes are commands to be interpreted by the loader to control the routing and loading of information, the packets of bytes contain transputer code to be loaded into the memory of a transputer. The bootstrap packets conform to the protocol and thus a processor, which is passing a bootstrap to another processor, cannot detect that bootstrap packets are being transferred.

The commands are applied using an operand word as a parameter to the command. The value in the operand word is created by OR'ing in the bottom six bits of information from the command byte into the bottom six bits of the operand word. One of the four command values allows this to be repeated by shifting the value in the operand word six places ready to receive another six bits. The command bytes are thus encoded from two components:

Bits 7..6

These two bits define the command which should be applied to the current value contained in the operand word after the data part of the command byte has been OR'd into it. The operand word is always cleared after obeying a command other than PREFIX.

- 0 : MESSAGE** The operand word contains the size of the message which follows this command byte. The next 'operand' bytes is the message. The protocol is implemented so that all messages will not exceed 60 bytes in length and thus, not require PREFIXES.
- 1 : NUMBER** The operand word contains a single number.
- 2 : FUNCTION** The operand word contains a value that is to be obeyed as an independent command which is not applied to the operand word.
- 3 : PREFIX** The current operand word is shifted left by six places. This allows arbitrary length values to be built.

Bits 5..0

These six bits provide the data (operand) part of the received character. This data is always OR'd into the bottom of the operand word which is used according to the command ode in the top two bits of the received byte.

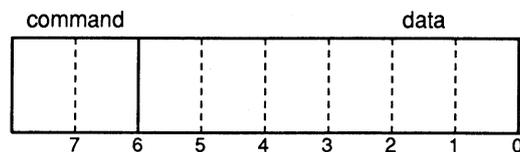


Figure 6: Command byte format

The packets of bytes always follow a MESSAGE command. By making the value of MESSAGE 0 (zero), a MESSAGE command will be interpreted by an unbooted transputer as a length byte and, consequently, bootstrap sequences conform to the command structure. All message packet transfers are sent and received on transputer links as single communications.

4.2 Loader action

The loader is an occam program which responds to input commands by altering the value of one or more local variables. These local variables maintain a current load address, a current output link, the set of active output

links and whether or not any code received is to be loaded at the current load address. The variable which controls whether code is loaded into memory is initialised to FALSE (FALSE means don't load, TRUE means load).

The loader actions in response to input commands are described in more detail in the following sections.

MESSAGE

After receiving a message command the message packet is input from the boot link. If the transputer is currently loading, the message is input to the current load address and the current load address is incremented by the size of the message. If the transputer is not currently loading, the message is input into a buffer.

The message command and message packet are copied in turn to all the links which are in the set of active output links.

NUMBER

The current output link is set to the value of the data part of the number command. The value is also remembered as one of the set of active output links to which code should be copied. The number will not contain prefixes. NUMBERS can also occur following an address function, where they are interpreted as a new loading address as described below.

FUNCTION

There are six functions as follows:

- 0 : LOAD** sets the state of the variable which controls whether code is loaded into memory to TRUE. Any future code packets received will be input at the current load address as described for MESSAGE above. The set of active links is reset to none.
- 1 : PASS** sets the state of the variable which controls whether code is loaded into memory to FALSE. Any future code packets received will be input into a buffer as described for MESSAGE above. The set of active links is reset to none.
- 2 : OPEN** indicates that all command bytes received up to but not including a matching CLOSE function should be copied without interpretation to the current link. All commands other than MESSAGE

can occur between an OPEN and the matching CLOSE command, including paired OPEN and CLOSE commands.

- 3 : CLOSE** brackets a nested command sequence, matching a previous OPEN function.
- 4 : ADDRESS** indicates that the NUMBER which follows should be used as the current load address for future code packets. The address used for loading is an offset in bytes from MOSTNEG INT, rather than the transputer byte address, because access to the memory of the transputer is to an occam array parameter of the loader placed at MOSTNEG INT. ADDRESS is always followed by a NUMBER, the NUMBER may have prefixes. The value of the last address received by the loader is returned to the bootloader and is used as the entry point/initial workspace address of the loaded code.
- 5 : TERMINATE** indicates that the distributed phase of the load is finished and the loader returns to the bootloader. TERMINATE will always be preceded by the final load address.

The examples which follow show how simple and more complex loading information is encoded and directed, to the recipient transputers for the configuration described in section 2 'The TDS Extractor'. The symbols used in the examples have the following meaning.

```
{bootstrap} -- a message containing bootstrap code
{code}      -- a message containing some code
{}         -- a message of length 0 used as a terminator
0          -- a number used as to set up the current link
#300       -- a number used as the current load address
L          -- the function Load
P          -- the function Pass
(          -- the function Open
)          -- the function Close
A          -- the function Address
T          -- the function Terminate
...        -- sequence of preceding item
```

Single transputer

The sequence to load only processor 0 is given in the following lines.

```
{bootstrap} ... {}
L A #300 {code} {code} ...
L A #500 {code} {code} ...
L A #230 T {code} {}
```

This load sequence begins with the bootstrap and loaders, these are followed by the first set of code packets which are loaded starting at offset #300 from the most negative address, the next set of code packets are loaded starting at offset #500 from the most negative address and the final set of code packets is loaded starting at offset #230 from the most negative address. The first group of messages and the last group of messages are loaded by the bootloader which terminates on receipt of a message length of 0. The other two groups of messages are loaded by the loader which examines each command to determine the next action and thus does not require a message sequence terminator.

After the receipt of the terminate operation, the loader is exited and control is returned to the bootloader which has the ability to load sequences of code packets at consecutive addresses. The final parts of the loaded program can overwrite the loader program if necessary. The entry point of the loaded code is always the last address received by the loader. This is also the initial value of the work space pointer.

Multiple load

Load instructions for transputers not directly connected to the host are bracketed between an Open and a Close function. Each transputer removes the first and last brackets and passes the contents byte by byte to the current output link. If the load items for processor 0 are not included, the sequence to load processor 2 is given in the following lines.

```
P 1 {bootstrap} ... {}  
P 1 (L A #300) {code} {code} ...  
P 1 (L A #230 T) {code} {}
```

The first line loads processor 2's bootstrap and bootloader. The Pass command resets the set of active output links and indicates that any future code received should be copied to the set of active output links via the buffer. The next command, the number 1, adds link 1 to the set of active output links and sets link 1 as the current output link. This is followed by the command Open (the open bracket) which causes all items up to but not including the matching Close to be copied to the current output link.

Copying the same piece of code to more than one processor is achieved by having a load path for each recipient of the code. This is demonstrated with the following sequence to load processor 4 and processor 3 with the same piece of code, at address #400 on processor 4 and at #500 on processor 3 (note that the example configuration does not allocate the same code to processors 1 and 3).

```
P 1 (P 2 (L A #400)) 2 (L A #500)) {Code} ...
```

Taking a copy of a code packet and passing it to another processor is achieved by using the load rather than the pass function as is shown by the following sequence to load processor 2 and processor 4 with the same piece of code, at address #900 on both processors.

```
P 1 (L A #900 2 (L A #900)) {code} ...
```

4.3 RS232

A transputer connected to a host computer by means other than a transputer link must be set to boot from ROM. The ROM code must then receive bootstrap and loading information from the communication medium and perform the load accordingly. Inmos transputer evaluation boards are designed so that a board which is booted from ROM will receive its load commands from an RS232 serial port. Normally only the root processor (i.e. the processor connected to the host) is set to boot from ROM.

The Inmos evaluation boards communicate with the host using a standard protocol which is described below.

Startup sequence

The first three bytes received from the host are used to determine the baud rate of the transmission, the communication mode and the operating function required. Each correct wakeup character read is acknowledged by transmitting an acknowledge (ACK) code to the host computer, an incorrect character is acknowledged with a not acknowledge (NAK) code. The three wakeup sequence bytes are described in more detail below.

'??' An initial wake up code (which can be used by the receiving processor to determine the transmission speed of the serial line).

'H' or 'B' If 'B' is received then all subsequent data is transmitted as full eight bit binary data. If the 'H' character is received then all subsequent data from the host is to be read in encoded form.

'L' or 'A' This command is used to determine the operating function that the ROM is to perform. 'L' indicates that a load sequence will follow, 'A' indicates that an analyse sequence will follow. The analyse sequence is used when the host is interrogating the network to retrieve details of the previous program loaded. Analysing is described

in more detail in an accompanying technical note. This function will be received as two ASCII chars if the previous command was an 'H'.

Data encoding

In order to avoid transmitting 8-bit binary values to a host computer all values transmitted to the host are printable ASCII characters. The following standard definitions are used:

```
VAL ACK IS '0' :  
VAL NAK IS '3' :  
VAL HEX IS "569ABDGHKMNPSVYZ" :
```

The 16 values of the HEX table above are used instead of the hexadecimal digits 0,1...E,F. The values are used to encode all binary numbers that have to be transmitted to the host as well as to encode all input from the host if the startup sequence include the 'H' code to indicate encoded transmission. Encoded binary data is thus transmitted as two ASCII characters that can be used to create a single byte value. For example:

```
#00 is received as '5' followed by '5'  
#42 is received as '9' followed by 'B'  
#FC is received as 'S' followed by 'Z'
```

The ASCII characters have been chosen so that they are all at least two bits different from each other, and each one has an even number of bits set (even parity with a zero parity bit).

Every message packet is followed by another byte value; i.e. messages from the host have one more byte than the number given in the operand word. This extra byte is a checksum value: the checksum is correct if the exclusive or of all the bytes in the message and the checksum itself yields a zero value. If the checksum is correct then the board responds with an ACK to the host; otherwise the board responds with NAK to the host. Checksums and handshaking are not used when communication is via transputer links.

5 Bootstrap code

This section lists the local workspace used by the bootstrap and the boot-loader, which should be read with reference to this workspace layout. The workspace used by the bootstrap is organised so that the 6 words used by the bootstrap and bootloader for directing the loading are at the lowest off-

sets. These six words are overwritten by the loader and then repositioned to the lowest available addresses for the second call of the bootloader

```

VAL base          IS 1 :  -- loop index
VAL count         IS 2 :  -- loop count

VAL load.start    IS 0 :  -- start of loader
VAL load.length   IS 1 :  -- loader block length
VAL next.address  IS 2 :  -- start of next block to load
VAL bootlink      IS 3 :  -- link booted from
VAL next.wptr     IS 4 :  -- work space of loaded code
VAL return.address IS 5 :  -- return address from loader
VAL temp.workspace IS return.address -- workspace used by both
                                -- preamble and loader
VAL NotProcess    IS 6 :  -- copy of MinInt
VAL links         IS NotProcess : -- 1st param to loader (MinInt)
VAL bootlink.param IS 7 :  -- 2nd parameter to loader
VAL memory        IS 8 :  -- 3rd parameter to loader
VAL buffer.start  IS 9 :  -- 4th parameter to loader
VAL entry.point   IS 10 : -- 5th parameter to loader
VAL entry.address IS 11 : -- referenced from entry point
VAL MemStart      IS 12 : -- start of boot part 2

```

The initial workspace requirement is found by reading the workspace requirement from the loader occam and subtracting the size of the workspace used by both the loader and the bootstrap (temp.workspace). This value is incremented by 4 to accommodate the workspace adjustment by the call instruction used to preserve the processor registers.

```

initial.adjustment := (loader.workspace + 4) - temp.workspace
-- occam work space, + 4 for call to save registers, - adjustment
-- made when entering occam. Must be at least 4
IF
  initial.adjustment < 4
    initial.adjustment := 4
TRUE
SKIP

```

The bootstrap is listed in a transputer assembler format. It was, however, actually developed by using an occam program to encode defined values into a table ready for insertion into the TDS extractor.

```

-- set up work space, save registers,
-- save MemStart and NotProcess
start:
  ajw  initial.adjustment -- see above

```

```

    call    0          -- save registers

    ldc    start - addr0 -- distance to start byte
    ldpi                   -- address of start
addr0:
    stl    MemStart     -- save for later use

    mint
    stl    NotProcess   -- save for later use

-- initialise process queues and clear error
    ldl    NotProcess
    stlf                   -- reset low priority queue

    ldl    NotProcess
    sthf                   -- reset high priority queue

-- use clrhalterr here to create bootstrap for REDUCED application
    sethalterr           -- set halt on error
    testerr              -- read and clear error bit

-- initialise T8 error and rounding
    fpu.clearerr         -- floating clear error instruction

-- initialise link and event words
    ldc    0
    stl    base          -- index to words to initialise
    ldc    11           -- no. words to initialise
    stl    count         -- count of words left
startloop:
    ldl    NotProcess
    ldl    base          -- index
    ldl    NotProcess
    wsub                   -- point to next address
    stnl  0              -- put NotProcess into addressed word
    ldlp  base          -- address of loop control info
    ldc    endloop - startloop -- return jump
    lend                   -- go back if more
endloop :

-- set up some loader parameters. See the parameter
-- structure of the loader
    ldlp  entry.address -- address of entry word
    stl   entry .point  -- store in param 5

    ldlp  MemStart     -- address start of buffer
    ldl   NotProcess   -- bottom of memory
    diff                   -- convert address to memory offset
    stl   buffer.start -- buffer offset in param 4

```

```

    ldl    NotProcess    -- bottom of memory
    stl    memory        -- store in param 3

    ldl    bootlink      -- copy of bootlink
    stl    bootlink.param -- store in param 2

-- load bootloader over bootstrap
-- code must be 2 bytes shorter than bootstrap
    ldlp   load.length   -- packet size word
    ldl    bootlink      -- address of link
    ldc    1             -- bytes to load
    in     in             -- input length byte

    ldl    MemStart      -- area to load bootloader
    ldl    bootlink      -- address of link
    ldl    load.length   -- message length
    in     in             -- input bootloader

-- enter code just loaded
    ldl    MemStart      -- start of loaded code
    gcall  gcall         -- enter bootloader

```

6 Bootloader code

The bootloader is produced by the same mechanism which produces the bootstrap. Both programs become single message packets preceded by a length byte (which is also a loader MESSAGE command) and are transmitted from the TDS extractor through the network as MESSAGE communications.

```

-- initialise bootloader workspace
    ldc    packet.length -- buffer size
    ldlp   MemStart      -- buffer start address
    bsub   bsub          -- end of buffer address
    stl    next.address  -- start of area to load loader

    ldlp   temp.workspace -- pointer to loader's work space zero
    stl    next.wptr     -- work space pointer of loaded code

restart:
    ldl    next.address  -- address to load loader
    stl    load.start   -- current load point

-- load code until terminator
startload:

```

```

    ldlp  load.length  -- packet length
    ldl   bootlink    -- address of link
    ldc   1           -- bytes to load
    in                 -- input length byte

    ldl   load.length  -- message length
    cj    endload     -- quit if 0 bytes

    ldl   next.address -- start of area to load loader
    ldl   bootlink     -- address of link
    ldl   load.length  -- message length
    in                 -- input code block
    ldl   load.length  -- message length
    ldl   next.address -- area to load
    bsub  next.address -- new area to load
    stl   next.address -- save area to load

    j     startload   -- go back for next block
endload:

-- initialise return address and enter loaded code
    ldc   return - addr1 -- offset to return address
    ldpi  return      -- return address
addr1:
    stl   return.address -- save in W0
    ldl   next.wptr      -- wspace of loaded code
    gajw  next.wptr     -- set up his work space
    ldnl  load.start    -- address of first load packet
    gcall load.start    -- enter loaded code

return:
    ajw   -(temp.workspace + 4) -- reset work space after return

-- start clock
    ldc   0
    sttimer

-- initialise reduced workspace for loading main body code
    ldl   bootlink.param -- new copy of bootlink
    ldl   entry.address  -- loaded code entry offset
    ldl   NotProcess     -- convert to entry address
    bsub  entry.address  -- address of work space/entry point
    ldc   0              -- reset load length byte
    ajw   4 - (initial.adjustment - 4)
                                -- reset workspace to start + 4 for call
-- this means that while the last few blocks are being loaded
-- the below work space requirement overlaps these last few
-- instructions which are never used again.
    call  0              -- store in new workspace

```

```

ldl    next.address    -- loaded code work space pointer
stl    next.wptr       -- work space pointer for entry

j      restart         -- go back for remaining blocks

```

7 Loader occam

This section lists the occam source of the loader. It is included as part of the extractor table by the program which 'assembles' the bootstrap and bootloader, as a sequence of MESSAGE command message packet pairs.

The overall layout of the procedure is:

```

PROC loader ([4]CHAN OF ANY links,
            CHAN OF ANY    bootlink,
            [4]BYTE       memory,
            VAL INT       buffer.address,
            INT           entry.point )

... constants
BYTE  command :
INT   links.to.load, output.link :
INT   last.address :
BOOL  loading :

SEQ
bootlink ? command
WHILE command <> function.terminate
  INT tag, operand :
  SEQ
    tag := (INT command) >> data.field.bits
    operand := (INT command) /\ data.field
  IF
    ... tag = message
    ... tag = function
    ... tag = number
  bootlink ? command
:

```

The command and function constant definitions are

```

VAL data.field      IS #3F :
VAL data.field.bits IS 6 :
VAL tag.field       IS #C0 :
VAL tag.field.bits  IS 2 :

```

```

VAL message          IS 0 :
VAL number           IS 1 :
VAL function         IS 2 :
VAL tag.function     IS function << data.field.bits :
VAL prefix           IS 3 :
VAL tag.prefix       IS prefix << data.field.bits :

VAL load             IS 0 :
VAL pass             IS 1 :
VAL open             IS 2 :
VAL function.open    IS BYTE (tag.function \ / open) :
VAL close            IS 3 :
VAL function.close   IS BYTE (tag.function \ / close) :
VAL address          IS 4 :
VAL terminate        IS 5 :
VAL function.terminate IS BYTE (tag.function \ / execute) :

```

The component processes of the outer level IF are expanded in the following sections.

If the command was message

```

tag = message
INT load.address :
SEQ
  IF
    loading
    SEQ
      load.address := last.address
      last.address := load.address PLUS operand
    TRUE
      load.address := buffer.address

  IF
    operand <> 0
      bootlink ? [memory FROM load.address FOR operand]
    TRUE
      SKIP

  SEQ i = 0 FOR 4
    IF
      (links.to.load /\ (1 << i )) <> 0
      SEQ
        links[i] ! command
      IF
        operand <> 0
          links[i] ! [memory FROM load.address FOR operand]
      TRUE
        SKIP

```

```
TRUE
SKIP
```

If the command was number

```
TRUE -- tag = number (last component of IF)
SEQ
  output.link := operand
  links.to.load := links.to.load \/ (1 << output.link)
```

If the command was function

```
tag = function
IF
  operand = load
  SEQ
    loading := TRUE
    links.to.load := 0
  operand = pass
  SEQ
    loading := FALSE
    links.to.load := 0
  operand = open
  INT depth :
  SEQ
    depth := 1
    WHILE depth <> 0
      SEQ
        bootlink ? command
        IF
          command = function.open
          depth := depth + 1
          command = function.close
          depth := depth - 1
        TRUE
        SKIP
      IF
        depth <> 0
        links[output.link] ! command
      TRUE
      SKIP
  operand = address
  SEQ
    BOOL more :
    SEQ
      last.address := 0
      more := TRUE
```

```
WHILE more
  SEQ
  last.address := last.address << data.field.bits
  bootlink ? command
  last.address := last.address PLUS
    ((INT command) /\ data.field)

  more := (INT command) >= tag.prefix
entry.point := last.address
```

References

- [1] "Transputer Development System"
Prentice Hall, London 1988
- [2] "Transputer Reference Manual"
Prentice Hall, London 1988
- [3] "The Transputer Instruction Set - A Compiler Writers' Guide"
INMOS Ltd, Bristol
- [4] "Exploring multiple transputer arrays", Technical note 24
INMOS Ltd, Bristol
- [5] "Analysing transputer networks", Technical note 33
INMOS Ltd, Bristol