TDS EPROM Programming

INMOS Technical Note 4

David Cormie

October 1986
72-TCH-004

Contents

1 Introduction **4**

2 The EPROM Hex program **4**
2.1 Initialisation code 5

3 ROM contents for transputer networks **6**
3.1 Booting a network 6
3.2 Fold bundle contents 7

4 ROM contents for single-transputer systems **8**
4.1 Fold bundle contents 8

5 The Hex to Programmer program **8**
5.1 Output format 9

1 Introduction

A transputer-based system will be developed, typically, along the following lines. Before the hardware has been completed, the application software can be developed on the transputer development system (TDS). This can be done by building a harness of occam procedures which emulate the i/o hardware of the target system. The application software can run within this harness as if it was running on the target system. When development is sufficiently advanced, the application software can be downloaded from the TDS and tested on the target hardware. The TDS can be used to analyse and debug the system. When software and hardware development are complete, the software can be fixed in ROM.

This application note explains how to create PROMs suitable for booting a transputer or a network of transputers, using the tools supplied with the IBM PC version of the transputer development system.

The task of creating a PROM falls into two parts. First, a fold is built containing the PROM contents in ASCII hex. The PROM contents consist of: the user program, memory interface configuration data (if required), and appropriate bootstrap and initialisation code. This fold is created by the `EPROM Hex program`. Next, this ASCII hex must be converted to a format suitable for a PROM programmer and downloaded over a serial line from the PC. This is done by the `Hex to Programmer program`.

The `EPROM Hex program` and the `Hex to Programmer program` are supplied with the TDS as executable programs.

This document refers to the tds2.0 (30th June 1986) version of the `EPROM Hex program` and version beta.1b1 (19th September 1986) of the `Hex to Programmer program`.

2 The EPROM Hex program

The `EPROM Hex program` is applied to a fold bundle containing up to three folds. These are: a `CODE SC` fold, a `CODE PROGRAM` fold, and a memory configuration fold. The `CODE SC` fold should contain the application program when the target system is a single transputer: a network loader when the target system is a transputer network. The `CODE PROGRAM` fold should only be present when the target system is a transputer network and should then contain the application program. The memory configuration fold is only required when ROM contents are being generated for a processor with a configurable memory interface and none of the preset memory configurations is being used. It should have the same format as page 5 of the output

generated by the T4 External Memory Interface program which can be used unaltered for this purpose. The `CODE SC` fold must be present; the other two folds are optional. All the folds in the fold bundle must be filed.

The `EPROM Hex program` builds a new fold (labelled `EPROM hex`) within this fold bundle containing the ASCII hex of the intended ROM contents. The order of the `EPROM hex` fold contents is:

1. Contents of the `CODE PROGRAM` fold (if present).
2. Contents of the `CODE SC` fold.
3. Transputer initialisation code.
4. Memory configuration (if present).
5. Entry jump to the initialisation code.

These items are placed in contiguous areas of memory at the top of the transputer's address space. The entry jump to the initialisation code is always placed in the top two bytes of the address space and is generated automatically by the program. The first line of the `EPROM hex` fold holds the start address of the code in the processor's address space and identifies the intended processor type. For example:

```
.7FFFF22C    --    T4
```

2.1 Initialisation code

The transputer initialisation code is generated automatically by the `EPROM Hex program`. It is executed immediately after reset by the transputer which boots from ROM. The code produced depends on whether the ability to analyse the system is required and on whether a `CODE PROGRAM` fold is present in the fold bundle.

If the ability to analyse the system is required, the user should enter y (yes) when the `EPROM Hex program` asks

```
Insert copy for analyse (y/n)?
```

Answering yes causes the `EPROM Hex program` to produce initialisation code which copies the bottom 600 bytes of RAM into the top 600 bytes of RAM immediately after Reset. This allows an analyse program to overwrite the bottom 600 bytes but still be able to examine any data which was held

there (unfortunately the contents of the top 600 bytes are lost but, since processes fill the RAM from the bottom up, it is less likely that this space would be in use). To determine the address of the top 600 bytes, the `EPROM Hex program` asks

```
Ram size of board (in k-bytes)?
```

If it is not necessary to analyse the system the user should reply n (no) to the first question. In which case, the program will not ask for the RAM size and the initialisation code will not perform the 600 byte copy.

The initialisation code calls the code supplied in the `CODE SC` fold. If a `CODE PROGRAM` fold was present, its' contents are passed to the `CODE SC` code as a parameter. The `CODE SC` code may return to the initialisation code or it may be an infinite loop. If it does return, the initialisation code will call the code at the address returned in the parameter `entry.point`.

3 ROM contents for transputer networks

3.1 Booting a network

In a network of transputers one - the master transputer - should be set to BootFromROM: all others should be set to boot from link. The master transputer requires a set of ROMs containing the application software and a program (the network loader) which will export code to the network.

The bootstrap and load process is complex but a brief description follows. Full details can be found in section 5 of the Transputer System Design Tools manual: "Loading Transputer Networks".

Following reset, the master transputer exports (via the links) a small bootstrap program to all other transputers in the network. This piece of code loads and enters the distributing loader (also exported by the master transputer). Packets of the user's application code are then exported together with routing information. The distributing loader in each transputer interprets the routing information and either passes on or accepts code packets according to whether or not they are to run in that transputer. At the end of the load process every transputer has only the application code which it is to execute. The application program then begins to run.

3.2 Fold bundle contents

The application software must be a PROGRAM (complete with configuration statements) and should be placed in the fold bundle for the `EPROM Hex program` as a `CODE PROGRAM` fold. When a `CODE PROGRAM` fold is created, the configurer automatically includes the bootstrap code and the distributing loader along with the application code. The contents of a `CODE PROGRAM` fold will, therefore, boot a transputer network if they are sent, byte by byte, through a link into that network.

The network loader should be included in the fold bundle for the `EPROM Hex program` as a `CODE SC` fold.

The network loader runs on the master transputer and interprets the contents of the `CODE PROGRAM` fold. Any code to be executed by the master transputer must be retained and the rest of the `CODE PROGRAM` fold must be exported, byte by byte, to the rest of the network. Therefore, the network loader must be able to understand the routing information embedded in the `CODE PROGRAM` fold.

As an example, the loader used on the B001 is supplied with the TDS. It can be found in the fold:

```
[EPROM.TOP \ EPROM interface program \ Sample EPROM monitors
                                    \ SC multiboard eprom loader]
```

Note that this loader is only suitable for running on a 32 bit transputer (e.g. a T414) though the network it loads may contain 16 bit transputers. This is because some of the integer constants it contains are too large for a 16 bit processor.

It will, therefore, be necessary to modify this loader for running on a 16 bit transputer. Refer to the sections: "Creating a Transputer Loader" and "Loading Transputer Networks" in the Transputer System Design Tools manual.

A memory configuration fold can be included if required.

The bootstrap process for a transputer with this type of ROM is:

1. The initialisation code performs the 600 byte memory copy, if required.
2. The network loader (contents of the `CODE SC` fold) is called with the contents of the `CODE PROGRAM` fold passed to it as a parameter.
3. The network loader exports the contents of the `CODE PROGRAM` fold to the network (thus booting the network). Any code to be executed by the master transputer is retained.

4. The network loader returns the parameter `entry.point` containing the start address of the code to be executed by the master transputer.
5. The initialisation code calls the code at `entry.point`.

Thus, when generating ROM contents for transputer networks, the fold bundle for the `EPROM Hex program` should contain: a `CODE SC` fold (the network loader), a `CODE PROGRAM` fold (the application program), possibly a memory configuration fold, and no other folds.

4 ROM contents for single-transputer systems

4.1 Fold bundle contents

When a transputer boots from ROM, and is not required to boot any other transputers, the initialisation code produced by the `EPROM Hex program` performs all the necessary bootstrapping and it is only necessary to call the application program. The contents of the fold bundle for the `EPROM Hex program` should, therefore, be as follows.

The application program, which must be a single `PROC`, should be included as a `CODE SC` fold.

A `CODE PROGRAM` fold should not be included in the fold bundle.

A memory configuration fold can be included if required.

The bootstrap process for a transputer with this type of ROM is:

1. The initialisation code performs the 600 byte memory copy, if required.
2. The application program (contents of the `CODE SC` fold) is called.

The application program must not return to the initialisation code (it should be an infinite loop). If it does return the system will crash.

Thus, when generating ROM contents for single-transputer systems, the fold bundle should contain: at least a `CODE SC` fold (the application program), possibly a memory configuration fold and no other folds.

5 The Hex to Programmer program

The `Hex to Programmer program` is applied to the `EPROM hex` fold produced by the `EPROM Hex program` and produces output in a form suitable

for dumping to a PROM programmer. The output is sent to the file COM1 on the IBM PC. The file COM1 is treated by DOS as a communications port. Therefore, to connect a PROM blower to your PC requires a serial card installed as COM1. It may be necessary to use the DOS MODE command to configure the serial card to the correct baud rate, parity etc. for your PROM programmer.

The programming procedure depends on the width of the memory interface on the processor for which the PROMs are intended. The `Hex to Programmer program` reads the first line of the `EPROM hex` fold to determine the processor type and hence the number of ROMs required. The T414 has a 4 byte wide memory interface and therefore requires 4 byte-wide ROMs. The T212 has a memory interface which can be configured dynamically to be 1 or 2 bytes wide. If the code is intended for a T212, the program will ask whether the ROM is being accessed in byte mode (1 ROM required) or word mode (2 ROMs required). If more than one ROM is required they must be programmed separately and the user must identify which is being programmed. The `Hex to Programmer program` will select the appropriate bytes from the `EPROM hex` fold.

The start address of the code within the processor's address space is also read from the first line of the `EPROM hex` fold. This, and the size of the PROM being programmed, is used to calculate the start address of the code within the PROM. The PROM size is entered by the user.

5.1 Output format

As supplied, the `Hex to Programmer program` produces output in the Intel Hex format. This can be understood by a wide variety of PROM programmers but, if necessary, can be changed quite easily. The format-specific parts of the program are in `PROC send.buffer` which can be found in the fold

```
[F HEXTOP00.TSR to programmer \ XP640 for IBM/B004 using
            IBM.serial.convertor \ procedures \ PROC send.buffer]
```

The following pages list the parts of the `Hex to Programmer program` which format and output the hex data. In particular, the format of a data frame is specified in the fold `else data record` and the end-of-file frame is specified in the fold `sent all of data, so send end of file record`. These are the only parts of the program which should be changed if a different output format is required.

`PROC send.buffer` defines some useful constants and procedures before entering `output data`. `PROC send.hex.byte` sends a byte and adds it

to the checksum; `PROC send.hex.word` sends a two byte word by calling `send.hex.byte` twice.

```
PROC send.buffer (CHAN from.line, to.line,
                  VAL []BYTE buffer,
                  VAL INT address, count, byte.no)
  INT named.result :
  VAL colon IS INT ':' :
  VAL data.record IS #00, end.of.file.record IS #01 :
  INT sum, bytes.sent :
  BOOL going:
  ...  send.hex.byte
  ...  send.hex.word
  INT record, reply :
  SEQ
    ...  init
    ...  output data
    ...  terminate
:
```

`output data` formats the data into records of no more than `record.size` bytes and outputs them to the programmer, terminating with an end-of-file record.

```
{{{  output data
WHILE going
  VAL bytes.left IS count - bytes.sent :
  VAL record.size IS 16 :
  INT bytes.this.record :
  SEQ
    IF
      bytes.left >= record.size
        bytes.this.record := record.size
      TRUE
        bytes.this.record := bytes.left
    IF
      ...  sent all data, so send end of file record
      ...  else data record
    ...  delay to allow programmer to deal with record
}}}
```

The format of the Intel Hex end-of-file record is specified by

```
{{{  sent all data, so send end of file record
bytes.this.record = 0
  SEQ
```

```
    string.to.screen ("*cEnd of file record")
    to.line ! colon       -- delimiter
    sum := 0
    send.hex.byte (0)
    send.hex.word (0)     -- dummy address
    send.hex.byte (end.of.file.record)
    VAL checksum IS - (sum \ 256) :
    send.hex.byte (checksum)
    -- n.b. serial convertor ignores *c but sends *c*n on receiving *n
    write.string (to.line, "*c*n")
    going := FALSE
}}}
```

and the format of the data record by

```
{{{  else data record
TRUE
  VAL address.this.record IS address + bytes.sent :
  SEQ
    string.to.screen ("*cRecord number ")
    int.to.screen (record)
    record := record + 1
    to.line ! colon     -- delimeter
    sum := 0
    send.hex.byte (bytes.this.record)
    send.hex.word (address.this.record)
    send.hex.byte (data.record)
    ...  send data bytes
    bytes.sent := bytes.sent + bytes.this.record
    VAL checksum IS - (sum \ 256) :
    send.hex.byte (checksum)
    -- n.b. serial convertor ignores *c but sends *c*n on receiving *n
    write.string (to.line, "*c*n")
}}}
```

The conditions at the top of these folds must not be changed as they are the tests in the `IF` construct which selects the record type to be sent. The channel `to.line` is the output to COM1.

`else data record` uses the following code to send the next `bytes.this.record` data bytes to the programmer.

```
{{{  send data bytes
SEQ i = bytes.sent FOR bytes.this.record
  INT databyte :
  SEQ
    IF
```

```
      byte.no >= 0
        databyte := INT buffer[(i * bpwd) + byte.no]
      TRUE
        databyte := INT buffer[i]
    send.hex.byte (databyte)
}}}
```

The program waits for 3000 timer ticks between sending frames to allow the programmer to deal with each record. This may need to be changed for different PROM programmers and should be changed to 125 000 if the `Hex to Programmer program` is being run on a pre-production (rev A) T414.

```
{{{  delay to allow programmer to deal with record
TIMER time :
INT now :
VAL pause IS 3000 :      -- about 0.2 sec on rev. B low priority timer
SEQ
  time ? now
  time ? AFTER (now PLUS pause)
}}}
```