Using transputers from EPROM

INMOS Technical Note 58

INMOS

June 1989
72-TCH-058-00

Contents

1 Introduction

The INMOS Transputer has a unique ability to start from cold without any EPROM or similar non-volatile storage. It is able to load its first program from its serial links. This allows large networks of transputers to be constructed without the need for an EPROM on each, or its associated glue logic.

Where networks of transputers are being used for compute intensive tasks, many users have developed systems that are hosted by an I/O processor that sends the first program to the transputer. This allows the myriad of transputer based accelerator boards on the market to use cheap high volume keyboard, screen and disc in the form of a PC or a workstation.

Few systems of this type require an EPROM. However there are two other areas where EPROMs are needed; the embedded system where there is no I/O processor or disc, and the workstation that is transputer based, i.e. has no other processor.

2 Requirements

The EPROM is required to boot the processor to which it is attached, and any other processors attached to that one by the links ... thus only one set of EPROMs for the entire network.

For the 'other' processors, no problem, as they are simply booting from link, as they were when attached to a development system. For the first processor, however, there are three possible requirements.

i) Transputers have a few kilobytes (2K on T212,T414, 4K on T425,T800 ...) of internal RAM that is extremely fast (50ns, 40ns). Thus it can be beneficial to use this rather than slow external EPROM for execution. The normal run from EPROM case would use this fast RAM as its dataspace.

ii) Transputer memory interfaces support RAM down to 100ns cycle time, (T801-25 80ns), which is faster than EPROM. EPROMs are widely available only down to 150ns access time, which means at least 200ns cycle time on a conventional bus. Thus, even if the program does not fit in the extremely fast internal memory, it can be beneficial to load it into external RAM where 100ns cycle time can be achieved.

iii) Especially on 16 bit machines, it may be desirable to run the code from EPROM, either to save providing RAM or to save space in the address map, but some critical piece or pieces of code may need to execute from internal RAM to achieve the necessary performance. When it is a single piece of

code, this must be downloaded from EPROM to RAM at start-up. When it is multiple pieces of code, a paging system may need to be implemented.

Another implementation possibility is to use EPROMs that are not in the transputer address space at all, but are controlled by a counter and some logic to drive a link adapter. At reset, the contents of the EPROM are sent down the link to the transputer or transputer network, which itself is set to boot from link. The target system then behaves exactly as if it were loaded from the development system. One may still need to use method three above if it were critical to execute a part of the code from internal RAM, when the dataspace required was more than the size of that RAM.

3 Methodology ... D700D TDS based

The INMOS Transputer Development System (TDS) (IMS D700D) provides support for the first two requirements outlined above, and these are described briefly below. The purpose of this note, however, is to demonstrate the two variants of requirement (iii).

3.1 Running from EPROM

To create an EPROM from the TDS, one simply takes a CODE SC fold and places it in a previously empty fold bundle. An additional fold that specifies the memory interface timing requirements may also be included for T414,T800,T425 processors which have an on chip DRAM controller. This fold is created by another development system utility. One then gets the EPROM HEX tool from the toolset fold, puts the cursor on the fold bundle and presses run.

```
{{{   fold bundle for eprom hex
...F  CODE SC           the compiled and linked code
...F  (configuration)   optional memory interface data
...F  EPROM HEX         output created by EPROM HEX program
}}}
```

This creates an additional member of the fold bundle, which is the EPROM HEX file, which simply contains the start address, the processor type, and the code in ascii hex.

Finally one runs the HEXTOPROGRAMMER utility on the hex fold, to drive an EPROM programmer, or an equivalent program to create a disc image for later use by a programmer.

The code sent out actually has been extended slightly to include a preamble whose task it is to initialise those parts of the transputer not initialised by reset. Only the absolute minimum is reset in hardware, so that the maximum state is available after a crash for debugging. Transputers boot by executing the instructions in the top two bytes of the address space. In these two bytes, a backward jump is placed, which jumps to the start of the preamble. Near the end of the preamble, it calls the user SC, and the final few bytes are a stop.process instruction in case the user program should return.

3.2 Running from RAM

If it is required to run from RAM, for reasons mentioned in 'Requirements' above, or other transputers in the network must be loaded from this one, then a slightly different method is adopted.

A fold bundle is created as before, but this time the CODE SC that is put in it is a loader. INMOS provides the source of such a loader in directory `\tds2\tools\eprom`, in a fold marked 'multi-board EPROM loader', or the user can provide their own.

Also in the fold bundle, one must provide a CODE PROGRAM fold, this being the complete program for the network to be loaded, including the host transputer. For single transputer systems, the program fold represents just that transputer.

```
{{{   fold bundle for EPROM HEX
...F  CODE SC  multi-board eprom loader
...F  CODE PROGRAM   the application
...F  (configuration) optional memory timing
}}}
```

Note that whilst the CODE SC will be placed in the eprom as executable binary, the CODE PROGRAM will be simply copied into the ROM in its existing message-packet form.

The EPROM-HEX program is run as before, and it produces hex as shown below, where the first line gives the address to load the first byte, then the rest of the fold is a stream of bytes expressed as two digit ascii hex, separated by spaces and/or newlines.

```
{{{
.7FFFEA20  T4
69 67 54 23 45 65 76 87 90 01 9A 77 AE 7C 34 87
```

```
23 45 65 76 87 90 01 9A 77 AE 7C 34 87 69 67 54
65 76 87 90 01 9A 77 AE 7C 34 87 69 67 54 23 45
76 87 90 01 9A 77 AE 7C 34 87 69 67 54 23 45 65
}}}
```

The loader reads the packets of program from the EPROM, and obeys the embedded commands exactly as if it was part of a system booting from link passing on code for elsewhere, loading those for itself into local RAM. Finally it returns control to the preamble, which then tails an artificial main program embedded in the program to keep the entry point consistent. This main program then calls the loaded code, and supports the endprocess should the user program complete.

3.3 Running from EPROM, with critical code in RAM (statically)

When it is necessary to run certain time critical sections of code from internal RAM, leaving the rest of the program running from EPROM, the task becomes far more complex.

Using the INMOS occam compiles, there is a predefined procedure kernel.run that allows code previously loaded into a data array to be executed. There are also other various predefines to allow the parameters to be loaded for that code.

Thus the call of

```
signal.process(x,y,z)
```

where the formal parameter associated with x is a VAL INT, with y is []BYTE, with z is [10] BYTE, becomes:

```
VAL nparams IS 4:    -- x,y,z,SIZE z

params IS  workspace FROM ((SIZE workspace) - (nparams + 2))
                     FOR (nparams + 2) :

SEQ
  params[1] := x
  LOAD.BYTE.VECTOR(params[2],y)
  params[3] := SIZE y
  LOAD.BYTE.VECTOR(params[4],z)
  KERNEL.RUN(code.space,code.entry,workspace,nparams)
```

Note the assumption here that the code has already been loaded into the vector code.space, that code.entry is known, and that the vector workspace is large enough for both the real workspace and the parameters.

To analyse the code above ... note that the parameter space is at the top of the workspace, with one spare word above and below it, hence the nparams + 2.

The two extra spaces are for the return address, and for the old workspace pointer. These are put in by the compiler/kernel.run. If the code loaded used separate vector space, one would put nparams + 3.

Note that vectors in the parameter list become two parameters if the formal parameter of the procedure to be called was unsized, the first being the address of the vector, the second its size. Note that all items requiring a pointer to be passed must be retyped as byte vectors. Thus had the formal parameter associated with x been an INT rather than a VAL INT, then the code would have been

```
[]BYTE x.v RETYPES x:
LOAD.BYTE.VECTOR(params[1],x.v)
```

Also if y had been a vector of integers, its code would be

```
[]BYTE y.bv RETYPES y:
LOAD.BYTE.VECTOR(params[2],y.bv)
params[3] := SIZE y
```

Note that the size passed with the parameter is the number of elements in the array, not the actual number of bytes used, so as the called procedure is expecting an integer array, it is given SIZE y rather than SIZE y.bv.

3.4 Loading the code

The above section assumed that the code had been loaded into the vector code.space. This area requires some elucidation.

For the static case covered here, this need only be done at start-up, so the solution is to insert some code at the top of the program, sequentially before the application proper, to copy the code into the internal RAM. This assumes that the space has been allocated, and we know where to find the code in the first place.

The best way of finding the code is to use the disassembler provided with the D700D TDS. This can be found in directory `\TDS2\TOOLS\SRC`, and must

be compiled before use as it is shipped as source only. One of its options is to create an occam hex table of the code, so that this table can be buried in your main application source code.

Thus the procedure that must be put in internal RAM is compiled and extracted as a separately compiled foldset SC. The disassembler is applied to this foldset and adds another fold to it that contains the code in an occam table of the form

```
VAL code.table IS "*#67,*#24,*#55,....etc":
```

This fold is taken to the main application and embedded there.

There are two system infelicities to note in this operation. Firstly, the disassembler will not write to a foldset that is marked as compiled, so one has to 'break' it by going in to the source of the SC with the editor and typing, then deleting, a space, before running the disassembler. Secondly, the output fold, the table, is marked as type COMMENT, although the word 'comment' does not appear, so when compiled, the table is ignored. This is overcome by making another fold around the table fold, which will be of type occam source, and then removing the inner fold.

To load the code at run-time, the code looks this:

```
VAL code.table IS "*#67,*#24,*#55,....etc":
VAL enough IS SIZE code.table:
[enough]BYTE code.space:
VAL code.entry IS 0:
...   application declarations
SEQ
  [code.space FROM 0 FOR SIZE code.table] := code.table
  ...
  ... rest of application, using kernel.run to call it
  ...
```

Note that the disassembler puts a jump on the front of the code, so that it can always be entered from address offset 0. Also code.space has been sized from the code table, so no space is wasted.

Note that if separate vector space is being used, the line:

```
PLACE code.space IN WORKSPACE:
```

is required, immediately after the declaration of code.space, if there are more declarations than fit in internal memory, in order that the code space is on-chip. The declaration is put before the application declarations in order that it gets first call on internal memory.

3.5 Running from EPROM, with critical code paged into RAM (dynamically)

Extending the above method for dynamic paging where all the code originates in EPROM is extremely simple. Where code may be coming in either on a link or from a disk or active compiler/linker, it is more complex, but this is unlikely to occur on an embedded system.

Assuming all the code exists in the EPROM, and some part of it needs to be paged into fast memory and executed, and the process repeated on demand for other functions adds little complexity. If there were ten functions to be performed, these would be disassembled to occam tables as before and incorporated in the source. A CASE statement would then receive a command, with the correct parameters forced by the protocol, and run the appropriate function:

```
VAL code.table0 IS "...........":
.
.
VAL code.table9 IS "...........":
... decls

PROTOCOL commands
  CASE
    iob0 ; INT ; INT
    .
    .
    .
    job9 ; BYTE ; BOOL
    end --tag meaning stop, no data needed
:
CHAN OF commands command.chan:

SEQ
  running := TRUE
  WHILE running
    command.chan ? CASE
      job0 ; int.param1 ; int.param2
        SEQ
          [code.sgace FROM 0 FOR SIZE code.table0] := code.table0
          ... load parameters
          kernel.run(code.space,0,work.space,nparams)
      .
      .
      .
      job9 ; byte.param1; bool.param2
        SEQ
          .
```

```
      .
    end
      running := FALSE

    ELSE
      ...  error.message
```

Note that in this version, each job has a parameter list defined in the channel protocol, and the case input then automatically selects the correct load-code, load-parameters and run sequence, but we have had to write the code out ten times and use appropriate memory space.

A much simpler approach, albeit more restrictive, is to use a two dimensional table for the code tables. This is only possible if the rows are the same length, so one needs to pad all the compiled code tables to the length of the longest ... not always feasible.

One also needs to ensure that all the commands take the same parameter list. This is usually not a problem, particularly if they are channels for input and output.

The code then becomes

```
VAL code.table IS
  [ [contents of code.table0],
    [contents of code.table1],
    .
    .
    [contents of code.tabie9] ] : --each padded to longest
...  decls

CHAN OF msq user.in,user.out:

PROTOCOL commands IS INT ; INT ; INT:
CHAN OF commands command.chan:

SEQ
  running := TRUE
  WHILE running
    SEQ
      command.chan ? job ; param1 ; param2
      IF
       job <> stop.code

         SEQ
           [code.space FROM 0 FOR
             SIZE code.tabie[0]] := code.table[job]
           --warning, not all compilers accept that linebreak
```

```
          ... load params, including chans user.in,user.out

          kernel.run(code.space,0,workspace,nparms)

      TRUE
        running := FALSE
```

The final development is to allow the code of a job to migrate from transputer to transputer. This cannot be done after execution of the code has started, but can between jobs. Thus a full dataflow style system can be built. Assuming variable parameter lists, packed at source into a byte array, so that the formal parameters are a single array, but not assuming constant code or data sizes, the code becomes:

```
PROTOCOL commands IS INT::[]BYTE ; INT::BYTE :

CHAN OF commands command.chan:

...  decls

SEQ
  running := TRUE
  WHILE running
    SEQ
      command.chan ? code.length::code.space ;
                     data.length::work.space
      IF
        code.length <> 0
          SEQ
            ...  load parameters
            kernel.run(code.space,0,work.space,nparms)
        TRUE
          running := FALSE
```

Such jobs would of course create output to be sent elsewhere. This is easily achieved by passing user.in, user.out channels as in the previous example, or by passing out a results array.

An extension of this example would be in a transputer network where all the code was held in the EPROMs on one transputer, and the other transputers sent it a message asking for a block of code as required, then running it. This is a combination of the last two examples above.

4 Conclusions

Despite user reservations caused by the revolutionary nature of the transputer, many functions are more easily performed on a transputer than a more conventional machine. Whilst code and data are cleanly separated by the occam language and the development system, each can be treated as the other, so that dynamic systems to make best use of the 50 nanosecond transputer internal RAM are easily achieved, without the need to resort to assembly language.