# NIRSPEC

**UCLA Astrophysics Program** | **U.C. Berkeley** | **W.M.Keck Observatory**


**George Brims** **Revised November 20, 1998**

## NIRSPEC Software Programming Note 22.00

# Introduction to transputers and occam programming


**1 Introduction** . . . . . . . . 2
**2 Transputer processor** . . . . . . . . 2
**3 Occam language** . . . . . . . . 4
- **3.1 Introduction** . . . . . . . . 4
- **3.2 Processes** . . . . . . . . 5
- **3.3 Channels** . . . . . . . . 6
- **3.4 Data types** . . . . . . . . 7
  - **3.4.1 Primitive data types** 7
  - **3.4.2 Scope of a declaration** 8
  - **3.4.3 Variables** 8
  - **3.4.4 Constants** 9
  - **3.4.5 Literals** 9
  - **3.4.6 Indices** 9
  - **3.4.7 Arrays** 10
- **3.5 SEQ construct** . . . . . . . . 11
- **3.6 PAR construct** . . . . . . . . 12
- **3.7 ALT construct** . . . . . . . . 13
- **3.8 Channels vs. PAR and ALT** . . . . . . . . 16
- **3.9 IF** . . . . . . . . 16
- **3.10 CASE** . . . . . . . . 17
- **3.11 Looping** . . . . . . . . 18
- **3.12 PLACE** . . . . . . . . 19
- **3.13 PRI PAR and PRI ALT** . . . . . . . . 19
- **3.14 TIMER** . . . . . . . . 20

**4 Inter-process communications** . . . . . . . . 23
- **4.1 Hardware links** . . . . . . . . 23
- **4.2 Software channels** . . . . . . . . 24
- **4.3 Protocols** . . . . . . . . 25

**5 Host-transputer communications** . . . . . . . . 27
**6 Interfacing to the outside world** . . . . . . . . 27
**7 Development tools and program building** . . . . . . . . 28
- **7.2 Source code and version control** . . . . . . . . 28
- **7.3 Compiling** . . . . . . . . 29
- **7.4 Linking** . . . . . . . . 29
- **7.5 Configuration** . . . . . . . . 29
- **7.6 Generating the executable file** . . . . . . . . 30
- **7.7 Loading and running** . . . . . . . . 30
- **7.8 Utilities** . . . . . . . . 31

## 1 Introduction

The data acquisition and control electronics in NIRSPEC (and NIRC2) are based on the SGS-Thomson transputer. The idea of this note is to give programmers an introduction to what a transputer is, what it does, how it is particularly suited to real-time control and data acquisition, and how we build programs. Obviously, it should be read in conjunction with the occam language manuals, since there’s no point in repeating everything they contain (and I don't). Since the transputer is a little different and the occam language a lot different from other things you might have encountered before, I thought it was worth putting this guide together to help support programmers get started. Some of the information is general, and some is specific to the hardware architecture of the DSP Systems boards and connection hardware we are using. Wherever possible I’ve used examples from our own code, since the whole idea is to give support to the more specific documents describing the code we run on the transputers and the communications between them.

There are a number of advantages to using transputers and occam from the point of view of instrument control or other real-time tasks. First of all, it is easy to implement parallelism of a number of processes so that multiple pieces of hardware can be controlled and monitored simultaneously. For instance, the spectrometer and slit-viewing camera subsystems in the NIRSPEC electronics can both take images at the same time. The only constraint is that the communications channel for issuing commands and passing back data is a single link between the transputers and the host, so one channel or the other might have to wait to pass back its data. Similarly, we could operate all the mechanisms in the system at once, again with passing of commands and acknowledgments over the same links being the only (slight) bottleneck. Being able to acquire data or move mechanisms simultaneously is essential for NIRSPEC to operate efficiently.

## 2 Transputer processor

First a minimal bit of history: the transputer was originally developed in the early 80's by the UK company Inmos. Inmos were taken over by a larger company (GEC) and later sold to SGS-Thomson. The Inmos name has essentially disappeared as far as SGS-Thomson are concerned, but still crops up all over the place in the documentation.

The name transputer was meant to indicate that it’s a com*puter* that can be used as a component, as discrete *trans*istors once were, to build up more complex systems. Although the transputer was very successful for a while (outselling all other processors worldwide at one time) the numerous changes of ownership have hampered development of faster processors. This failure to keep the line up to date has basically killed off the transputer, and SGS-Thomson have announced that all transputer products will be phased out at the end of 1998.

When it was introduced, the transputer had a unique combination of features, some of which were only present in much larger systems, such as the supercomputers of that era. Most now appear in other more modern microprocessors, particularly DSPs.

These features are:

• 32-bit processor (T400 series) with integrated FPU in the later versions (T800 series). There are also lower cost 16-bit processors (T200 series).

• Processor and fast memory (not cache but program RAM) combined in one device.

• Complete memory interface requiring no support chips (except drivers) to build a complete system.

• Inter-processor communications, in the form of four serial links, built into the chip. These links run at 10 or 20 Mbit/s.

• On-chip scheduling allowing each processor to run multiple concurrent processes.

• On-chip timers allowing for timing to microsecond resolution.

• Concurrent design of the processors and the high-level language occam, removing the need to resort to assembly language for performance gains.

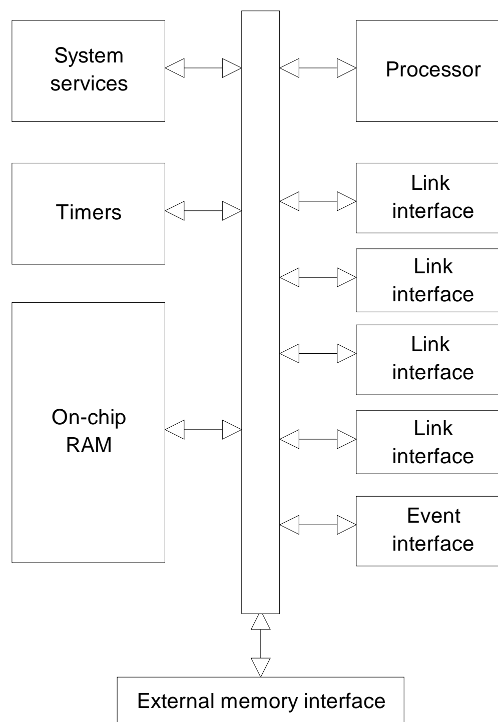



**Figure 1 Transputer architecture**

The power of this combination of features lies mainly in the ease with which parallel processing can be achieved. A transputer system can be built and the software prototyped and run on a single transputer, running multiple communicating parallel processes. As performance demands increase, more transputers can be added. The parallel processes will then run on different processors, communicating over the serial links, with minimal software changes. Unless the network becomes very complex or most of the work involves moving data around rather than operating on it, the performance of a system of transputers scales almost linearly with the number of processors, unlike a conventional system where transfer of data between processors isn't as efficient. For this reason very inexpensive mini-supercomputers based on transputers were marketed by companies such as Meiko.

The serial links are simple to use in terms of hardware, since they require no additional support chips when used over short distances (up to a few feet, perfectly adequate within a rack of electronics). They can be extended over longer distances using differential drivers or fiber-optics. They are also surprisingly simple to use from software, because the link hardware on the transputer chip is quite sophisticated.

Each transfer across a link is split up and sent as a series of packets, with a checksum tagged onto the end of each. The receiving transputer looks at the checksum and sends an acknowledgment message if it's correct. If not, a resend request is sent instead, and the sending transputer resends the packet. Once the information comes through cleanly the receiving code sees it, and the sending program can move on to the next line of code. Calculating and checking the checksums is invisible to the programmer; it is all handled transparently by the silicon. If you are working in a noisy environment you might notice a fall off in overall throughput if many packets are corrupted and need to be resent, but data will *always* transfer accurately. The only downside to this is that speed of light limitations come into play sooner if the links are extended (for example using optical fiber) over very long distances. Each successive packet is held up until the acknowledgment for the previous packet arrives, so there is an interval between the start of each packet, set by the round-trip time over the link.

## 3 Occam language

### 3.1 Introduction

The occam programming language was developed to take advantage of the transputer's architecture (and vice versa). In terms of writing sequences of procedural code, it's a pretty ordinary sort of language easily followed by anyone who has programmed in C, Pascal or FORTRAN. The fundamental concept of occam that's different from these other languages is that it has facilities for constructing code to operate in parallel pieces all communicating with each other, either on the same processor or distributed over many processors. (I should point out there are available C compilers with these facilities, but they produce executable code that runs 10 to 20% slower than matching occam.)

### 3.2 Processes

The fundamental entity in occam is the *process*. The term process is used interchangeably to mean a program, or a portion of a program, or even a single line of code. Processes communicate (and thereby also synchronize their operations) over *channels*.

The three most basic processes are:

```
variable := expression      -- assign a new value to a variable
channel ? variable          -- input a value from a channel to a variable
channel ! variable          -- output the value of a variable to a channel
```

Every occam program is built up from sequences of these three basic operations. A sequence of these primitive processes can form a more complex process, which can communicate with other complex processes. Conventional sequential code is built up from primitive processes combined using the constructs SEQ, IF, CASE and WHILE, and concurrent programs with the additional constructs PAR and ALT. These constructs are described in the following subsections.

An occam program is declared with the keyword PROC (meaning process rather than procedure) as in:

```
PROC myprog (linkin, linkout)
  INT x :                       -- variables & constants declared
  SEQ                           -- code
    linkin ? x                  -- ...
    x := x + 1                  -- ...
    linkout ! x                 —- ...
:                               -- end of process myprog
```

The parameters of the PROC declaration (`linkin` and `linkout` in our example) can be any combination of channels which the program will use to communicate with other processes.

This example illustrates an extremely important point about the mechanics of writing occam. *Indentation of lines of source code is significant.* Note that the colon signifying the end of the process is placed at the left margin like the PROC keyword, and everything else is indented at least 2 spaces. We will see more examples as we go through the use of the other keywords. The indentation rules make occam source code more readable, but can also be very annoying!

Also in the above example, you can see the use of the comment separator, which is two dashes in succession. These can delimit comments placed after the code on the same line, or on a line by themselves. In the latter case the comments must be indented to the current level just like the code.

Within a process we can list any other processes (equivalent to subroutines or functions in other languages) we need to use. These are declared with the same keyword, such as:

```
PROC myprog (linkin, linkout)
  ...
  PROC mathfunc (x,y)
    ...                          -- function code goes here
  :                              -- end of process mathfunc
  ...
  mathfunc (a,b)                 -- call mathfunc
  ...
:                                -- end of process myprog
```

Note again the indentation; the declaration of process `mathfunc` is indented by 2 spaces from the left, as is its terminating colon, and everything between the `mathfunc` declaration and the colon is indented at least 2 spaces further.

### 3.3 Channels

Each channel is a one-way connection between two processes. We can declare a pair of channels, one in each direction, to run over any hardware link between connected transputers, but we can also declare *virtual channels* to connect processes on the same transputer. These processes can be distinct programs or just different parallel code segments of the same process. Communication over either type of channel is handled in exactly the same way, so when two processes hosted on the same processor become too demanding of resources we can split them over two separate transputers and use almost exactly the same code. The only thing that will change is a configuration file which the software tools use to map the processes onto the hardware network. Setting up this configuration file is described later.

Note that communication over channels is *unbuffered* and *synchronous*. This means a sending process is stalled until the receiving process has accepted a message (and a receiving process is likewise stalled until the sending process sends all of it). This characteristic of channel communication can cause trouble (a deadlocked system) if the behaviors of senders and receivers aren't carefully matched. However, we can also exploit it in the situation where one process needs to wait until another one is ready before proceeding with some task. We can use a message to ensure synchronization. For instance to synchronize the clock generation and data acquisition transputers in the two camera sections of NIRSPEC, the clock generator sends out a message before it starts a frame readout. Each acquisition transputer readies itself (clearing buffers etc.) then passes the message to the next. The last acquisition transputer passes the message back to the clock generator (they're all connected in a loop). Only when it sees that message does the clock generator start clocking out the frame.

We will discuss and illustrate how channel declarations are coded in section 4.2.

## 3.4 Data types

Declaring variables etc. in occam is not always as simple as it might seem, since it includes a few facilities that are both useful and a bit strange.

Occam expressions can include four types of objects. These are literals, constants, variables, or indices. A literal is simply something you would type as part of an expression, such as 1, 2, 'H', 1.0E+6. A variable can be assigned a value through an assignment or a channel input, whereas a constant is defined when declared and is then read-only. An index is defined as part of a loop construct.

### 3.4.1 Primitive data types

Occam has the usual selection of data types. As in many languages, the meaning of the integer keyword varies with the type of processor. On the 32-bit processors (T800 & T400 series) it's 32 bits and on the T200 series it's 16 bits. To make an integer a required size the three other INT types let you be specific.

The data types available in occam are:

| Type | Description | Range of values |
|---|---|---|
| BOOL | boolean | True or false |
| BYTE | byte | 0 to 255 |
| INT | integer | Signed integer, number of bits depends on processor type |
| INT16 | 16-bit integer | -32768 to 32767 |
| INT32 | 32-bit integer | $-2^{31}$ to $(2^{31} - 1)$ |
| INT64 | 64-bit integer | $-2^{63}$ to $(2^{63} - 1)$ |
| REAL32 | 32-bit floating point | As IEEE standard 784[1] |
| REAL64 | 64-bit floating point | As IEEE standard 784 |

[1]As in any other language, there are rounding errors involved in using floating-point values.

You can define your own data types in occam, but they are formed from the above primitive data types. To create your own data types and declare variables of those types, you would use :

```
DATA TYPE LENGTH IS REAL32 :
DATA TYPE AREA IS REAL32 :
LENGTH x:
AREA y :
```

Note that LENGTH and AREA don't need to be written all in uppercase, but are written that way just to keep them consistent with the normal keywords like INT and make the code more readable. Defining your own variable types allows the type checking in the compiler to make sure you pass only the appropriate kind of data to a procedure. Although x and y in our example are both stored as REAL32 numbers, they are of different data types. So if a procedure is defined with a parameter of type LENGTH, passing it y as parameter is flagged as a type mismatch.

### 3.4.2 Scope of a declaration

When we declare a variable or constant in occam, we have to be aware what its *scope* is. Just because we declare something in line 10 of a program, it doesn't mean it's valid in line 100. Here's an example:

```
SEQ
  INT max :          -- specify max
  INT min :             -- scope of max    -- specify min
  SEQ                   --                    -- scope of min
    c ? max             --                    --
    c ? min             --                    --
    IF                  --                    --
      p < max           --                    --
        p := p + 1      --                    --
      p = max           --                    --
        p := min        --                    --
  SEQ
    ...
```

The variable max becomes valid from the colon at the end of the line declaring it, and min becomes valid at the end of the next line. Each one remains valid as long as the next line is indented at least the same amount, or indented further. Once the indentation level has increased further, then returns to the same level as the declaration (the second SEQ at the bottom of the example), the scope ends.

### 3.4.3 Variables

In occam variables must *always* be declared before you can use them in an expression or a channel input or output. Variables are declared as in the following example:

```
BYTE b :
INT j :
REAL32 x :
```

Each declaration ends with a colon, and becomes effective immediately following the colon. There is no restriction on the name except it can't be a reserved word like BYTE, SEQ etc. In the NIRSPEC code we tend to use names which are all lowercase or capitalized, to distinguish them from the occam keywords, combining words with dots (because they're easier to type than underlines). Case is significant in variable (and constant) names.

#### 3.4.4 Constants

You can also declare a constant, which can never be re-assigned a different value, as in:

```
VAL INT j IS 99 :
VAL BYTE character IS 123 :
```

#### 3.4.5 Literals

A literal in occam is anything you would write in a statement to use a particular value rather than the value of a declared variable or constant. There are default assignments of type, for instance:
`42` defaults to type `INT`, and `'h'` puts the ASCII value of the letter h in a byte. The default type can be overridden, so if for example we wanted use a specific numeric value as a byte (for instance to pass it as a parameter to an RS232 routine) we would force it to be a byte by adding the data type in parentheses, as in:

```
    RS232.out (42(BYTE))
```

Many times you don't have to be this precise if the context tells the compiler what type to assign. For instance if you are referencing an element of an array, such as `x[3]`, you don't have to put `(INT)` after the 3.

#### 3.4.6 Indices

An index in occam is declared implicitly whenever we create a loop using the SEQ keyword:

```
    SEQ i = 0 FOR 10
      out ! i
```

Here we never declare `i`, but since it can only be of type `INT`, the compiler just goes ahead and assigns it anyway. Note that within the scope of each pass through the loop, `i` is a constant and can't be overwritten.

### 3.4.7 Arrays

Just as in other language we can declare arrays. *Important: array elements always start at zero.* Arrays are declared using this syntax:

```
[5]INT x :
[3][4]BYTE y:
```

and array elements are referenced as, for instance:

```
x[4]
y[1][2]
[x FOR 4]
```

The last one means "the first 4 elements of array `x`". We could also have typed

```
 [x FROM 0 for 4]
```

to refer to the same thing. Say we have two arrays a and b, declared as:

```
[10]INT a :
[10]INT b :
```

Since the two arrays are of the same size and type, we can copy every element from one to the other more or less concisely as follows:

```
  a := b     OR      [a FROM 0 FOR 10] := [b FROM 0 FOR 10]
```

We can also use this syntax:

```
        [a FROM 0 FOR 5] := [b FROM 3 FOR 5]
```

to copy 5 elements of `b` into different consecutive locations in `a`.

Even more useful, we can *declare* an array to be a section of a larger one. Say we have a large data array, and some process which grabs successive small blocks of data. We can pass that process the name of a smaller array which we repeatedly define to be different sections of the larger one, as in the following example:

```
[65536] INT frame:
INT pointer :
SEQ
  pointer := 0
  SEQ i = 0 FOR 256
    input IS [frame FROM pointer for 256]:
    SEQ
      get.data (input)
    pointer := pointer + 256
```

The array `input` is re-defined each time we go through the loop, so each time we call the routine `get.data` it will put the data in a different section of the larger array `frame`. Now this is a really useful tool, but it's also potentially confusing to newcomers to occam. First, have to be careful with how occam assigns a *scope* to a variable declaration. After we declare the array `input` it might seem the SEQ and the indentation of the next line are unnecessary, but in fact they *are* for the following reason.

Recall that when we declare a variable, it becomes valid following the colon at the end of the line, and stays valid as long as lines beneath are indented to the same level or further (so `pointer` is valid from where it's declared through to the end of this code segment). But if the following lines are indented, once the indentation *returns* to the same level the scope ends. So because we add `SEQ` right after `input IS...`, we are forced to indent the line calling `get.data`, which then *also* means that the scope of `input` ends when that indentation ends. So by using the SEQ and the indent we *deliberately* limit the scope of the variable so it can be re-defined each time around the loop.

### 3.5 SEQ construct

The SEQ (sequence) construct simply defines a group of instructions that are to be executed in sequence, just as in a conventional program. Instructions grouped into a sequence by a SEQ keyword might be run in parallel with other sequences by a PAR or ALT (sections 3.6 & 3.7). A sequence construct is written as:

```
SEQ
  p1
  p2
  p3
```

Process p1 completes before p2 starts, and so on, just as we would see in any other language. For example:

```
SEQ
  channel1 ? x
  x := x + 1
  channel2 ! x
```

reads in a value on one channel, increments it, then sends it out on another channel.

Another note on indentation; the lines following a SEQ are always indented by two spaces from the SEQ itself. That tells the compiler that they are within the *scope* of that SEQ keyword.

### 3.6 PAR construct

The PAR (parallel) construct is used to execute instructions in parallel, and is written

```
PAR
  p1
  p2
  p3
```

Each process executes in parallel with the others. The whole set completes when all the component processes are complete. For example:

```
PAR
  chan1 ? x
  chan2 ! y
```

The processor will try to read in x and send out y at the same time over the two channels. When both transfers are complete, then the parallel construct is complete. For example if we have a following instruction, as here:

```
PAR
  chan1 ? x
  chan2 ! y
chan3 ! z
```

the third transfer (sending out variable z over channel 3) will not start until *both* the other messages have been sent/received. Here again it's the indentation that tells the compiler that the first two transfers are within the scope of the PAR keyword, and the third isn't.

Now each of these examples has single operations running in parallel. What if we want to perform groups of instructions in parallel? In that case we need to combine the SEQ and PAR constructs as in this example:

```
PAR
  SEQ
    x := x + 1
    chan1 ! x
  SEQ
    y := y - 1
    chan2 ! y
```

The two groups of operations following the two SEQ keywords are executed in parallel. The compiler knows which lines are within the scope of each SEQ by the indentation. It is good practice to head each parallel segment with a SEQ even if it is just a single line, as in the following example:

```
PAR
  SEQ
    x := x + 1
    chan1 ! x
  SEQ
    y := y - 1
    chan2 ! y
  SEQ
    chan3 ? z
```

This makes the code clearer. If we had written

```
  ...
  SEQ
    y := y - 1
    chan2 ! y
  chan3 ? z
```

the result would be just the same. However it might not be clear to someone looking over the code whether we really intended that last line to be a third parallel stream, or it was supposed to be the last line of the second one, and we forgot to indent it far enough.

### 3.7 ALT construct

The ALT (alternative) construct is an extremely useful facility of occam, particularly in a real-time system. It is written just like a PAR construct, listing a number of different code segments. However instead of telling the compiler the code segments are to be run in parallel, it provides a breakpoint where only one will be executed, depending on external events. Each code segment is "guarded" by a channel input, and the occam program will monitor the input channels and respond to whichever channel *actually receives something first*. So here we have an example where the program flow can't be determined by just examining the code — it is truly real-time in that it could behave differently each time it is run.

Where would we use ALT? Traditionally, we would have two choices as to how to implement a system that could check for and handle input from multiple sources. One way would be to set up a system of interrupts, including a mechanism for figuring out which input caused the interrupt. This can be a nasty thing to implement, and often means resorting to assembly language (it is worth noting here that there is *no* assembly language code in the NIRSPEC system). The other method is to use a polling loop which looks at the status of each source in sequence. This is generally a very compute-intensive method.

With the ALT construct we have a very simple way to code a sequence that makes the transputer wait until any one of the inputs is ready, then make the input and perform its following section of code. In essence the ALT construct is an interrupt facility built into the high-level language so that assembler coding is unnecessary, and we don't have to mess around with control and status registers and bit masks.

In the following example, the transputer waits for input on one of three channels, `c1`, `c2`, or `c3`. When any one of these channels is ready for input, the byte received is sent out on another channel, `c4`. Only the first channel to actually receive a message is serviced, and input from the other two is ignored.

```
ALT
  c1 ? byte1          -- read from channel c1
    c4 ! byte1        -- send out on channel c4

  c2 ? byte2          -- read from channel c2
    c4 ! byte2        -- send out on channel c4

  c3 ? byte3          -- read from channel c3
    c4 ! byte3        -- send out on channel c4
```

There are advantages to this construct over both of the traditional methods. We didn't have to implement anything in assembler or worry about servicing the wrong interrupt. Whichever input occurs first will *always* be the one serviced. Also this method is not compute intensive, for the following reason: if this code segment appeared in one of a group of parallel processes, the process would be de-scheduled while it waited for the inputs, and would consume *no* processor cycles until an input occurred.

The most important use of this construct in our application is where we nest such a piece of code in a loop, so that it continuously monitors multiple message streams and acts as a multiplexer for messages. The root transputer for instance has exactly this setup where it multiplexes replies from all the other transputers onto a single link back to the host.

What if we want to switch off one of the options in an ALT under certain circumstances, or in other circumstances always choose a particular one? To implement the former, we can combine a boolean expression with an input to guard one of the alternatives. For example:

```
ALT
  Monday & in1 ? data
    out ! data
  in2 ? data
    out ! data
```

Here `Monday` is a variable of type BOOL (boolean) and can only have the values TRUE or FALSE, and the ampersand character acts as a logical "and". If `Monday` is true, then the ALT will allow the

option of reading in data from channel `in1`, otherwise only an input on channel `in2` can cause the process to proceed.

There's also a way to always choose the same option if some controlling variable is true. To do this we combine a boolean with a SKIP in place of an input, thus:

```
ALT
  Monday & SKIP
    out ! new.data
  in2 ? data
    out ! data
```

Now if `Monday` is true, the SKIP ("do nothing" instruction) acts like an input that's always ready, and that option will immediately execute; otherwise the ALT will wait for input on in2 as normal.

The above examples deal only with input from channels. What about taking action to service a request from a piece of hardware? For example, say we have interfaced a FIFO buffer chip to the transputer to receive data from something external (this is how the NIRSPEC acquisition boards take in data from the A-D converters). The hardware will need to have a way to tell the transputer that it has incoming data. The facility that we can use here is called the event port. We can declare a special type of channel called a PORT and use the occam PLACE construct (see 3.12) to put it at a standard place in memory (memory location 8). We can then add to our ALT a line to read a dummy byte from it just as if it were a regular channel. This input won't complete until the transputer sees a transition on one of its external lines (called EventReq). Again this is not compute intensive — the processor isn't continually polling this dummy input.

For example we could declare the port and a dummy byte to read from it, thus:

```
BYTE event.byte
PORT OF BYTE event :
PLACE event AT 8 :
```

To trigger from a hardware event we would use:

```
ALT
  event ? event.byte
    chan1 ! got.data
  chan2 ? message
    chan1 ! got.message
```

The value read into `event.byte` will be irrelevant; what matters is that this read will trigger the ALT when some hardware event alters the state of the EventReq line.

### 3.8 Channels vs. PAR and ALT

There are restrictions on how we use channels with parallel processes. Imagine we have a piece of code as follows:

```
PAR
  SEQ
    ...
    out ! x,y,z
  SEQ
    ...
    out ! a,b,c
```

Recall that the two parallel streams are executing on the same processor, time-sliced by the on-chip scheduler. What would happen if the first stream started to write `a`, `b` and `c` to channel `out`, but was then de-scheduled to allow the second stream its share of the CPU time? The second stream could then start to write `x`, `y`, and `z` to `out` also. The result could be the receiving process getting a garbled sequence of data such as `a`, `b`, `x`, `c`, `y`, `z`. In order to avoid this possibility, two or more parallel streams are not allowed to use the same link in the same direction. In fact the compiler would flag the above code as an error.

However the following code *is* valid:

```
ALT
  in1 ? x,y,z
    SEQ
      ...
      out ! x,y,z
  in2 ? a,b,c
    SEQ
      ...
      out ! a,b,c
```

It is OK for each of the code sequences in the ALT to contain a write to channel out because *only one of them will be executed*, depending on which input occurs first.

### 3.9 IF

The conditional construct IF is much like any other language (with one nasty pitfall!). Usage is:

```
IF
  condition1
    p1
  condition2
    p2
  ...
    ...
```

If `condition1` is true, `p1` is executed, if `condition2` is true `p2` is executed, and so on. Only one option is executed, then the construct terminates. The pitfall is that we have to cover *all* eventualities, or the program will stall! For example:

```
IF
  x = 0
    y := y + 1
  x = 1
    y := y + 2
```

will get stuck if x is negative, or has a value of 2 or greater. If we change the second condition so we have:

```
IF
  x = 0
    y := y + 1
  x <> 0
    y := y + 2
```

("<>" means "not equal to") then *all* possible values of x are covered and one of the options will always be taken.

What if we really *did* only want to do something if x is 0 or 1, and take no action if it has other values? We can then add a third condition as follows:

```
IF
  x = 0
    y := y + 1
  x = 1
    y := y + 2
  TRUE
    SKIP
```

The meaning of the occam keyword TRUE is self-evident (there's also FALSE; either can be assigned to a variable). The SKIP keyword simply means "do nothing". This IF statement will always complete since if neither of the first two tests yields a true condition, the third will.

### 3.10 CASE

The CASE construct is similar to that used in other languages. An example would be:

```
CASE x
  y
    out1 ! x
  z,2
    out2 ! x
```

If x has the same value as y then it is sent via channel out1, but if it's 2 or equal to z then it goes via out2. This is exactly how command messages from the host are routed through our transputer network. The "root" transputer connected to the host passes on messages to the spectrometer camera, the slit-viewing camera, and the RS232 and motion control transputers. Each message consists of a byte command id and an integer parameter value; a very long CASE statement examines the command id and passes the message on via the appropriate channel.

### 3.11 Looping

There are a couple of different ways to implement a loop in occam. One uses the SEQ keyword, and the other uses the WHILE keyword. To use the SEQ keyword to loop, we combine it with an index variable, as in this example:

```
SEQ i = 0 FOR 10
  SEQ
    j := i * 2
    out1 ! j
```

There are a few interesting things to note here. First, we don't have to declare the variable i — in fact this is the *only* case where we can use a variable in occam without declaring it first, and it's *not* valid outside the loop. The value of i will start at 0 and end up as 9. We didn't have to start at 0; we can start at any integer. There is no way to make i increment by more than one each time around, so to get a sequence 0, 2, 4 etc. we had to multiply it by 2 to yield j. Note the second SEQ keyword. This lets us use a sequence of lines in the loop; we could have omitted it if we only wanted to execute one line repeatedly.

The other way to loop, using the WHILE keyword, is implemented as follows:

```
running := TRUE
WHILE running
  SEQ
    in ? command
    IF
      command = quit
        running := FALSE
      TRUE
        execute.command (command)
```

The WHILE keyword is used with a controlling boolean variable. In this example the loop will be performed as long as the boolean running remains true. The only way for the loop to end is for a message to arrive containing a quit command, so that running is changed to false. It's considered bad form in occam to use the WHILE keyword with the keyword TRUE in place of a boolean variable; the program will loop indefinitely with no way to exit, short of a complete crash of the processor. Nonetheless we do exactly that in the majority of our programs! Each program has an

initialization section followed by an endless loop where it sits and waits for host commands, executing each in turn then going back to the head of the loop to wait for the next.

### 3.12 PLACE

The occam PLACE construct does just that — it *places* a variable at a particular location in the memory space of the processor. Generally the actual location will be application-specific.

There are a couple of ways we use this facility. The first is the one mentioned in section 3.7, where the transputer processor has a number of specific addresses that are reserved for things like the event port, allowing the high-level language access to hardware-level interrupt services.

The other use is to allow easy interfacing to memory-mapped hardware. For example:

```
[80] INT inbuffer
[80] INT newdata
PLACE inbuffer AT #400000
```

This tells occam that the zeroth element of array `inbuffer` is at location $400000_{16}$ in memory (meanwhile `newdata` is an array in normal memory and we don't actually care where it's located). There is no real memory at address $400000_{16}$, but instead a piece of hardware providing a memory-mapped interface to some data source. Array `inbuffer` is not a real variable but a dummy pointing at that location. If we need to read data from the hardware we simply assign the values in the dummy array to another array, thus:

```
[newdata FROM 0 FOR 80] := [inbuffer FROM 0 FOR 80]
```

The FROM...FOR syntax effects a DMA transfer of a block of values from one array to the other (faster than an element by element copy in a loop). Like the ALT, PLACE is a very powerful construct because it removes the need for assembly language coding and keeps control of hardware-level interfacing in the high-level language.

### 3.13 PRI PAR and PRI ALT

One of the difficulties with using parallel processes on the same processor is that sometimes we would prefer one of the processes to have a bigger share of resources than another. The occam scheduler will normally give equal shares of time to all processes, except that processes that are stalled waiting for input from channels are de-scheduled. By replacing PAR with PRI PAR we can have the processor prioritize execution of parallel streams. For instance if we have:

```
PRI PAR
  SEQ
    x := x + 1
    y := y + x
    chan1 ! y
```

```
SEQ
  z := z + y
```

the second parallel stream will only be executed once the write of `y` to channel `chan1` is either complete, so ending the first stream, or the first stream hangs up for a while because the receiving process at the other end of `chan1` isn't ready.

The transputer's on-chip scheduler has only two levels of priority, high and low, but the compiler allows us to list multiple streams in the desired priority order and has a mechanism for handling the multiple priorities in software (shades of VMS for those who remember!).

In section 3.7, where we described the ALT construct, we said that the transputer accepts input from whichever of the sources being monitored first receives a message. What happens when two sources just happen to get a message coming in on *exactly* the same clock cycle of the processor? The result is described as "indeterminate", which isn't exactly desirable. In some cases we would have a preference as to which input would be serviced if this were to occur, and the PRI ALT construct exists to deal with this situation. If we use PRI ALT in place of ALT, then the order in which we list the input options becomes significant. The first one listed will have higher priority than the second, and so on. That means there is never any doubt as to which input will be read first should they happen to coincide.

### 3.14 TIMER

One of the most useful features of the transputer-occam combination is the TIMER. The transputer has hardware timers which are accessible from the high-level language. We can declare a timer, then read a value from it as if it was a channel. The value returned is the current value in the hardware timer, which increments at fixed intervals. The data type of the value returned is INT, which is either 16 or 32 bits depending which type of processor we are using. Typical usage would be:

```
TIMER clock :
INT t :
SEQ
  ...
  clock ? t
  ...
```

Integer `t` now contains the instantaneous value of the timer `clock`. The value of the hardware clock is cyclic, so when it reaches the largest positive integer value it rolls over to the most negative integer then increments though zero to the most positive integer again. Obviously when comparing times we need to take this into account, just as we distinguish times on a regular clock. We know that 11 am is earlier than 1 pm, even though 1 is less than 11.

It doesn't do us much good just to know the value in the clock. However we can use the clock to actually control the execution of our program, in a number of ways. First of all, we can make the

program wait until a specific timer value comes up. To do this we add the keyword AFTER, as follows:

```
TIMER clock :
INT t :
SEQ
  ...
  t := x
  clock ? AFTER t
  ...
```

This is still not very useful. The value we assigned to `t` isn't referenced to anything in the outside world or to any other event. However if AFTER is used in conjunction with the modulo operator PLUS, then it can provide a time delay, as in the following example:

```
TIMER clock :
INT t :
INT delay :
SEQ
  ...
  delay := 100
  clock ? t
  clock ? AFTER t PLUS delay
  ...
```

The first read of the clock assigns the current clock value to variable `t`. The second clock read hangs up until the clock count has reached the value `t PLUS delay`. We use PLUS to add delay to `t` because the PLUS operator takes care of the clock rolling over. An analogy would be setting our kitchen timer at 11 o'clock so it will go off at 1 o'clock. We know that requires a 2 hour delay. The limitation is that `delay` must be less than a clock cycle.

If we put this construct in a loop we can have a program that does something at regular intervals:

```
  delay := 100
  clock ? t
  WHILE looping
    SEQ
      t := t PLUS delay
      clock ? AFTER t
      do.something()
```

This code will execute the process `do.something` every 100 ticks of the hardware clock. Once we have taken the first clock value, we increment `t` each time around the loop. Because we use PLUS to do that, the value will constantly roll over as long as the loop continues.

We keep talking about the clock value without saying what the increment ("tick") is. There are two possible values, 1 microsecond or 64 microseconds. If a process is running at high priority it has access to the faster tick, otherwise a tick is 64 μs. To see how to place a process at high priority, see section 7.5.

The most common use of the time delay facility in our programs is to time integrations in the two camera sections, as in the following code section:

```
clock ? time.now              -- store current time
global.reset ()               -- reset the array
loadwave (read)               -- get ready to read frame
clock ? AFTER time.now PLUS itime.spec.ticks  -- wait itime
clockwave (read)              -- read out the array
coadd.num := coadd.num + 1    -- keep a count of coadds
```

Note that during the delay (between the two clock reads) we can perform whatever operations we need to do. We will only interfere with the accuracy of the timing if those operations aren't done by the time the timer is due to expire. In this case we have calibrated the time taken for the `global.reset` and `loadwave` operations, and set a minimum allowable value for the integration time, `itime.spec.ticks`.

Another way to use a timer is in measuring the time taken for a series of operations, as follows:

```
TIMER clock :
INT time1 :
INT time2 :
INT delay :
SEQ
  clock ? time1
  ...                         -- operations to be timed
  ...
  clock ? time2
  delay := time2 MINUS time1
  to.host ! delay
```

Here we take two time readings, before and after the operations we want to calibrate. We difference them using the MINUS operator, which like PLUS can deal with the clock rolling over between the two reads (and like PLUS is limited to a total delay of one clock cycle). Once we have the delay time we send it back to the host computer. This facility is extremely useful in measuring real-world timings.

Another use of a timer is as one of the inputs to an ALT construct. Recall that reading a clock looks just like reading a message from a channel, and in fact it can be inserted in place of a channel as a guard to one of the options. We can use this to prevent a program from becoming completely stalled when some piece of hardware doesn't respond, or to combine some operation we want performed at fixed intervals with an interactive section of code.

The following example shows how to implement a timeout on an input or output operation, thus:

```
clock ? t
ALT
  from.host ? command
    parse.command ()
  clock ? AFTER t PLUS timeout
    SKIP
```

The ALT will wait until either some input arrives on the channel `from.host` (in which case it executes the process `parse.command`) or the delay set by the value of `timeout` expires. In the latter case nothing happens thanks to the SKIP keyword, which is there just because we have to have *some* code in each option of the ALT, but the ALT comes to an end.

Both our clock generator transputers use a TIMER in an ALT to implement a periodic erase of the IR detectors. The ALT is nested in a loop, so the program is always waiting for host commands, executing each as it arrives. If no command comes in for 5 seconds, the timeout expires and the program sends out an erase frame to the detector.

## 4 Inter-process communications

### 4.1 Hardware links

The hardware links are simple TTL point-to-point connections that can only run over a few feet of twisted-pair cable (they are not RS232 or any similar convention). Various commercial solutions exist to solve the problem of talking over longer distances. Assembling a system of transputers is like doing organic chemistry: the four links of the transputers correspond to the four covalent bonds of a carbon atom!

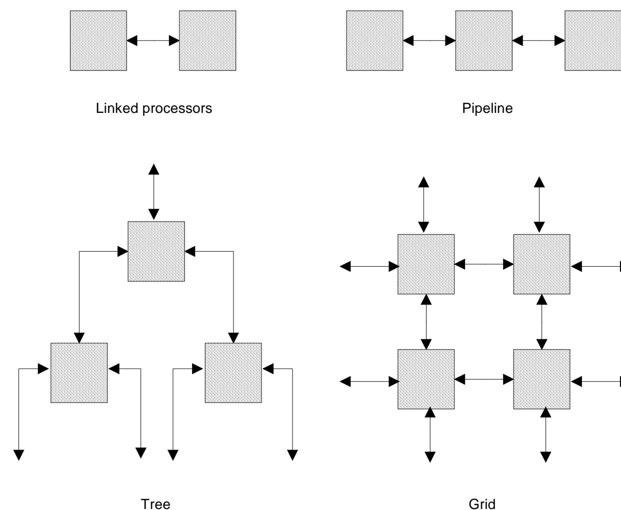



**Figure 2 Transputer connections**

### 4.2 Software channels

We can declare a channel over each of the hardware links, as well as between processes on the same processor. The channels which run over hard links are declared in the configuration file, in the form:

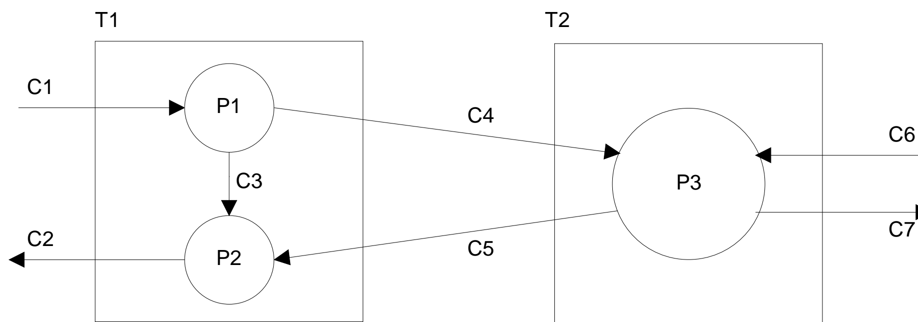
```
CHAN OF prot chan1 :
CHAN OF prot chan2 :
```

where `chan1` and `chan2` are the names of channels, and `prot` is a protocol which has already been declared. For very simple message passing we can also use

```
CHAN OF INT chan :
```

to declare a channel which will always send or receive messages consisting of a single integer (we could also use BYTE).

If we have two separate programs running on the same transputer, the channels are declared in the same way in the configuration file. If we have a channel that links two parallel code streams in the same program, we simply declare it (in the same format) along with the variables of the program.




This diagram illustrates the different connections channels can make between processes. Transputer T1 has 2 processes, P1 and P2, while T2 runs a single process, P3. Channel C3 allows P1 to send messages to P2. P1 can also talk to P3 via C4, and P2 can receive from P3 via C5. C4 and C5 are defined to run across a physical link whereas C3 is a virtual link. C1 and C2 connect P1 and P2 to some other process off-page, while C6 and C7 also connect C3 to some other transputer.

It's important to remember that each *channel* runs in one direction, while each *link* (4 per processor) is bidirectional and so can carry two channels. That means a process that runs alone on a transputer could have as many as 8 channels all connecting it to processes on other chips. Of course if we have multiple processes on a chip, there is no limit to how many *virtual* links we can have between them. Also there is no need to declare channels on every link of a transputer, or even both directions of a particular link. In fact we have a couple of links in our system that only carry messages in one direction.

### 4.3 Protocols

We have to define how information passed over each channel is to be formatted by declaring a *protocol*. Every message that is sent over a link has to be sent in a recognized protocol, which has been named in the declaration of that link. That may sound restrictive, but in fact a protocol can consist of a list of different protocols for different types of messages. There's also a standard protocol called ANY, which means no protocol at all. ANY works fine as long as both sending and receiving processes expect to send and receive the same set sequence of bytes, integers etc. This is fraught with danger, so the ANY protocol has now been made obsolete and generates compiler warnings! A simple protocol might look like:

```
PROTOCOL CHAR IS BYTE :
```

This means each message will consist of a single item, a byte. We can also have a protocol which is a sequence of different items, such as:

```
PROTOCOL MSG IS BYTE; INT :
```

There is also a way to declare a protocol which will pass variable length messages:

```
PROTOCOL DATA IS INT::[]INT :
```

The syntax with the two colons and the empty square brackets denote an integer, followed by an integer array, with the length of the array defined by the first integer. Obviously this is most useful for transferring data packets.

To send using the three example protocols above, we would use:

```
out ! x
out ! x; y
out ! y; z::array
```

where `out` is a channel, `x` is a byte, `y` an integer, `z` an integer and `array` an integer array, of which z elements will be sent.

What if we want to be able to send messages of different formats over the same channel? Each channel is specified to have a single protocol, but fortunately a protocol can be specified with multiple formats, using the CASE keyword. If we combine the two previous examples, we get the following protocol declaration taken directly from the NIRSPEC code:

```
PROTOCOL TRANS2TRANS
  CASE
    msg; BYTE; INT :
    data; INT; INT::[]INT :
```

As the name implies, this is the protocol used between all the transputers in the NIRSPEC network. When we combine protocols like this, we have to give the two formats separate names; here we have `msg` for messages, and `data` for data transfer. Our example has only two formats but you can specify as many as you like.

When sending a message over a link with a multiple protocol, the lines of code performing the message send and receive must include the name of the optional format used. For instance in the acquisition transputer code you can find the following line:

```
out ! data; tnum; 65536::frame      -- send out frame data
```

This sends information over channel `out` using the `data` format. The first element is just the word "`data`", the label denoting which of the two formats we are using, followed by `tnum`, the transputer number (so the host knows which part of the frame this is). The number 65536 defines how many elements of the array `frame` will follow. Obviously the message must be formatted properly for the format named, or we will generate a compiler error. For instance `tnum` can not be of type BYTE.

What about reading in via a multiple-format protocol? If the receiving process is sure which type of message is coming next, then it can read the message in that format by likewise including the appropriate protocol name:

```
in ? data; tnum; framesize::frame      -- read in frame data
```

However a receiving process might have to deal with messages in either form coming in. For instance the root transputer of the network takes replies in msg format and data packets in the data format. This is dealt with, again using the CASE keyword, as follows:

```
    in ? CASE
      msg; cid; param
        SEQ
          out ! msg; cid; param
      data; tnum; frame.size::frame
        SEQ
          out ! data; tnum; frame.size::frame
```

So instead of reading in with a specific protocol name after the "?", we have CASE. Following each possible input format is a sequence of code that's only performed if the message coming in satisfies that format. The meaning of this construct is "read a message, and if it's in one format do this, and if it's in the other format do this". This piece of example code simply reads messages then passes them along on another channel, whichever form they take.

More details of our transputer network implementation are given in NSPN21, and also in the documents on each specific transputer (root, clocking, motor control etc.).

## 5 Host-transputer communications

The hardware connection to the transputers from the host Sparcstation is through a device called a Matchbox, made by Transtech. This device contains a transputer running code stored in ROM, and provides a bridge between the Sparcstation's external SCSI port on one side and the transputer network on the other. (Fiber-optic extenders stretch the SCSI to bridge the distance between the host and the instrument.) There is a library of code for the host side provided with the Matchbox, so the Sparcstation can download and run the bootable occam code, then send and receive messages and data across the link provided by the Matchbox.

As far as the transputers are concerned, this combination looks just like another transputer connected to one link of the root transputer. The protocol running over that link however is different from that used between all the other transputers (examples in section 4.3). The protocol declaration is

```
PROTOCOL TRANS2HOST IS INT16::[]BYTE :
```

This means a 16-bit integer arrives first, informing the receiver of the size of an array of bytes to follow. This is not our choice but mandated by the requirements of the host software package. The root transputer has to do some unpacking and re-packing of messages and data between this format and the protocol we use everywhere else, but that is the only overhead.

For a more comprehensive description of host-transputer communication, see NSPN08.

## 6 Interfacing to the outside world

There are a couple of ways of interfacing a transputer system to the outside world in order to do real-time control and monitoring. The first is to use the CO11 support chip, which provides an interface between a transputer serial link and 8 bits of parallel I/O. The CO11 chip has handshake lines for coordinating the I/O at the parallel side. On the serial side it looks to a transputer just like another transputer, so in the code we would just use channel communication to talk to or receive from it. However this method is limited to 20 Mbit/s throughput, and it's also awkward to coordinate reading in data that comes as more than 8 bits. You either have to serialize successive bytes of your data into one CO11 or use several side by side, with attending synchronization problems.

If we want higher throughput we need to forget that it's a transputer we're using and memory-map the I/O just as we would with any other processor. This is the method used in the DAQ15 and DAQ17 boards built for us by DSP systems. Using programmable gate arrays the various I/O ports and their control and status registers are mapped in hardware to specific memory locations. The occam PLACE construct (section 3.12) is particularly useful when we do this. By declaring an integer and then using PLACE to tell the compiler where it is located in memory space, we can read or write a register by simply copying its value or assigning it a new value. We can also make an I/O port look like a block of memory locations by declaring an array and using PLACE to place it at the location of the port. That allows us to use DMA transfers to grab data from the port and transfer it

straight to another array in real memory. It's not an exaggeration to say that the use of memory-mapped I/O is almost as important as the parallelism/link setup in making the transputer systems we have used a success.

One part of the transputer standards we haven't mentioned before is the TRAM, or TRansputer Application Module. This allows vendors to package transputers with some kind of standard I/O such as RS232 or SCSI in a standard daughtercard format. Vendors also supply motherboards for PC or VME bus to hold a number of these modules. Usually the transputers on a TRAM have ROM code and/or drivers that allow the purchaser to program the particular interface easily.

The RS232 communication between the transputers and the controllable power strip and the LakeShore temperature controllers/readouts is performed by a couple of TRAMs, each giving us 2 channels of RS232 I/O.

## 7 Development tools and program building

### 7.1 Tools overview

The tools we use to generate the code that runs on the transputers are supplied by SGS-Thomson, and are collectively known as the occam toolset. There are a whole set of manuals for this toolset, giving the details of the various options, and another document, NSPN26, giving particulars of our setup, so I will just give an outline here. The main thing to know is that the occam source code is stored on the host computer, and from it the executable code is generated using the facilities of the toolset. There is no operating system or even disk storage on the transputers. When they power up they have no code to run, until the host downloads the executable file over the link.

### 7.2 Source code and version control

Source code is stored in the nirspec computer in directory /kroot/kss/nirspec/keyword/transputer and its subdirectories. The code for each transputer (the file extension is **.occ**) resides, with its include files (extension **.inc**), in a separate subdirectory. There are a number of old versions of each module in its directory. The base module has no number while the

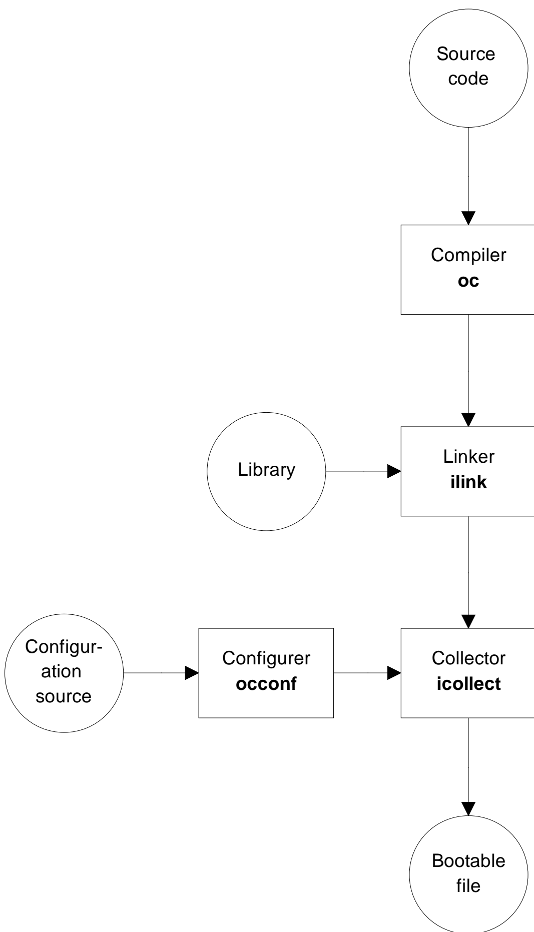



**Figure 4** Program build model

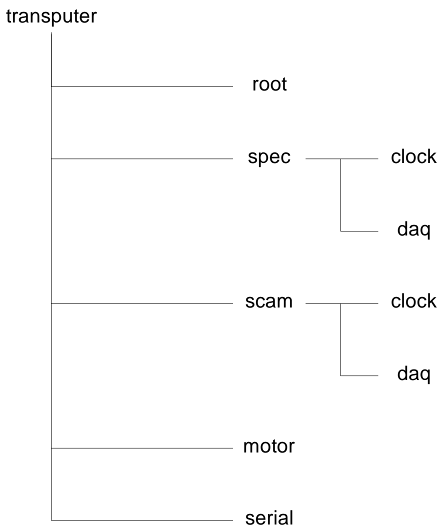



**Figure 5** Development directories

consecutive sersions are numbered So for instance the root transputer code is **root.occ**, while successive versions are **root01.occ**, **root02.occ** etc. To compile with a new version one copies the new module to **root.occ**, overwriting it but keeping the previous version in its own file. At a later date this all needs to be merged with CVS.

### 7.3 Compiling

The occam compiler program is called **oc**. If you are logged in as nirspec, you can run this (and any of the other tools) from any directory. To compile a module one would type

**oc progname.occ**

There are a number of occam compiler flags; **-h** gives a short list and **-help** gives a full list. The most useful one is **-c** (check) which checks the code syntax but doesn't produce any output. In the latest version of the compiler, it is compulsory to use **-t805** or **-t225**, which specify which type of processor the code is to run on (there are other types but not in our system).

Also useful are a number of flags which turn off various warning-level compiler messages, so you only get screen output for actual errors. There's even a **-wqual** option, which gives you a bunch of criticisms of your coding style (mostly omission of TRUE...SKIP at the end of an IF — see section 3.9).

### 7.4 Linking

The linker is called **ilink**, and combines the compiler output files with extension **.tco** with any necessary library code (**.lib** files).

### 7.5 Configuration

The configuration process is performed by the program **occonf**, and takes as input a file with extension **.pgm**. This file contains:

- a list of all the transputers in the network (including processor type and available RAM)
- a list of how they are all linked together physically
- a listing of the linked software modules to be used
- declarations of all the link protocols
- a list of which programs are to be placed on each transputer in the network.

All these assorted pieces of information are checked for consistency. For instance if the **.pgm** file places a program on a processor and list it as having two I/O channels, then the occam code also ought to declare two channels. In particular the rules about processes in parallel both using the same channel are enforced (section 3.6).

Here we also take care of specifying which processes are to run at high priority. We do this using the PRI PAR construct, as in this example:

```
PROCESSOR spec.clock
  PRI PAR
    SEQ
      specclock (...channel declarations...)
    SEQ
      SKIP
```

The PRI PAR construct puts parallel processes at different priorities. Here we put the `specclock` process in parallel with nothing at all (signified by SKIP) just so it will have access to the high priority (1 microsecond per tick) clock.

The output file from this process (extension **.cfb**) is used in the next stage by **icollect**, so it knows how to combine all the codes for the different transputers into one bootable file.

### 7.6 Generating the executable file

The final stage in program building is running the **icollect** program, which collects together the linked code and combines it according to instructions in the output from the **occonf** program. The final output of the whole build process is a file with extension **.btl**. This is the bootable file which can be sent down the link to the transputer network.

### 7.7 Loading and running

There are two different ways to run an occam program. We can use a program called **iserver**, which runs on the host, downloads the bootable file to the transputers, and talks to the transputers over the link, or use a library of supplied routines to download and communicate from within our own C program (the server).

In order to use **iserver** the occam code needs to use routines from a host I/O library, since **iserver** treats the host computer as a set of peripherals — keyboard, screen and filestore. However that restricts you to a very simple interface on the host end (basically an ANSI terminal), and you can't then combine the transputers with other facilities such as IDL or DataViews, so we abandoned use of iserver many years ago. The only thing we kept was the **iserver** command format, consisting of a byte command id and an integer parameter. This makes the whole occam software system keyword based, just like the host code.

The routines used to connect the C code in the server to the occam code are actually supplied with the Matchbox, replacing those with the toolset. The toolset routines only work with transputers plugged into the VME backplane on the older Suns (or the AT bus in the PC version). Transtech wrote a bunch of corresponding routines which can operate through the SCSI bus and the Matchbox. Use of these routines is described more fully in NSPN08.

### 7.8 Utilities

There is a useful set of utilities supplied with the Matchbox to allow you to troubleshoot hardware in the transputer network. These are **check**, **ftest** and **mtest**. **check** sends a piece of probe code down the link and finds whatever processors are there and how they're linked together (obviously you can then see whether any processors or links are missing, compared to what you thought you built!) . To check that each processor is healthy, you can pipe the output of **check** to **ftest**, which then sends a test program to each processor. You get a listing similar to **check**, but with "OK" opposite each processor if it passes the tests. If you use **check | mtest**, you get another similar listing but this time the memory of each transputer is listed.