TRANSPUTER BASED CONTROL OF MECHATRONIC SYSTEMS

Andrew P. Bakkers and Job van Amerongen
Mechatronics Research Centre Twente and
Control Laboratory, Department of Electrical Engineering, University of Twente,
P.O. Box 217, 7500 AE Enschede, Netherlands ,
e-mail: elbscbks @ henut5.bitnet

Abstract. The design of a control system is not finished with the derivation of the necessary control algorithms. The system designer has to schedule all control and calculation tasks within the sampling interval of the system. Higher sampling frequencies often improve the system performance. On the other hand, more sophisticated control algorithms require more computing time thus reducing the obtainable sampling frequencies. In this paper a systematic approach to obtain a design optimum is given. The design method is illustrated with the control of a flexible robot arm.

Keywords. *Parallel processing, Real time computer systems, Control engineering applications of computers, Robot control, Transputers, Occam.*

1 INTRODUCTION

Recently the systematic application of advanced control algorithms to mechanical systems has been given the name "mechatronics". This term originates from Japan, where in 1989 the first International Conference on Advanced Mechatronics was organized by JSME (1989). In a mechatronical design it is continuously considered whether the desired properties can be better realized by changing the mechanical construction or by adding electronic control, rather than adding the control system after finishing the design of the construction. This enables the design of systems with superior performance. Typical examples of mechatronic systems are, for instance, a compact disc player and an advanced photo camera with many electronic functions. Also robotics can be considered as part of mechatronics.
For the realization of a mechatronical system, advanced control algorithms and fast computer systems are needed. Because mechanical systems have very fast dynamics, a high sampling frequency is essential. Conventional, sequential, computer systems are too slow. By using parallel computing, the sampling frequency can be increased and more complex control algorithms can be realized. However, the use of parallel computing requires that the total concept of the realization of a real-time digital controller be reconsidered. This paper gives a systematical analysis of the problems which are met when a real-time parallel computer system is to be set up and gives solutions to various of these problems.
Although the ideas presented are more generally applicable, main emphasis will be given to the use of transputers. The transputer is a new type of processor which was designed to be used in parallel. Simultaneously with the development of the transputer a new parallel programming language has been developed: OCCAM. Together they have properties which make them especially attractive for the realization of a real-time parallel computer system. It may be expected that the transputer and other future parallel processors will have a great impact on the realization of advanced control algorithms with a very high sampling rate.
In Section 2 a description of the transputer and a short introduction to OCCAM will be given. Section 3 discusses the requirements for a scheduler in a real-time control system. It will be argued that at present there are no computers nor operating systems which are able to guarantee that the scheduling of the various tasks in a control system (sampling, computations, data logging etc.) are handled correctly. A suggestion for an optimal solution will be presented and it will be indicated how a sub-optimal solution can be realized.
One of the typical features of the transputer is that it has four external links which enable a fast data exchange to other transputers. This forms not only the basis for the parallel architecture of a transputer-based computer system, but it also enables a systematic design of a digital controller into various layers such as an interface layer, a protection layer a control layer etc.. This concept is worked out in Section 4. In Section 5 the concepts developed in this paper will be illustrated with an example of a typical mechatronical system, a flexible manipulator, where the inevitable vibrations are actively damped by a parallel control system realized with four transputers. A robust state-feedback controller, designed with a pole placement technique, is able to give the controlled system the appearance of a rigid construction. In order to exploit the parallel features of a transputer network, the controller has to be written in a parallel form. Various forms of parallelism are discussed in Sections 4, 5, and 6. They range from a general-purpose form of parallelism, where the program is split up into basic elements which can be calculated in parallel, to a much coarser form of parallelism, where ad hoc some large parallel parts are selected. The performance of the transputer-based control system with various types of parallelism will be compared with a controller realized in conventional, sequential hardware. The paper concludes with some conclusions and suggestions for future research in this area.

2 THE TRANSPUTER AND OCCAM

One way out of the calculation versus sampling time dilemma has for long been sought in the use of parallel computing. In the past, many attempts to realize parallel systems with traditional processors, have failed due to the software overhead burden. The theory of Communicating Sequential Processes by Hoare (1978) forms the foundation of the programming language Occam. Occam uses the concept of processes. Communication between processes is done by means of channels. Due to the synchronization properties of these channels the restriction to one processor no longer exists. The transputer hardware development was the next logical step. By using the formal specification language Z, the correct behavior of the implementation of the Occam constructs in hardware was formally proven. The hardware - software integration resulted in the transputer - Occam combination. The transputer may be considered a building block for parallel computers. Occam enables programming across these parallel transputers.

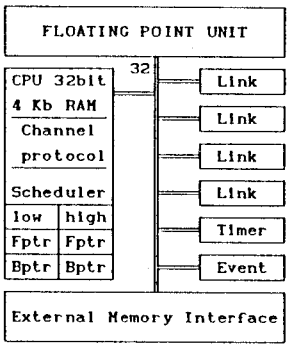


Fig. 1 Block diagram of the T800 Transputer

In the following paragraphs the operation of the transputer will be explained, based on the block diagram in fig. 1.
From this block diagram a number of significant elements of the transputer chip can be identified.

1. Communication to other transputers is performed over self-synchronizing high speed (20 Mbit/sec) serial *links*.
2. A round robin process (task) *scheduler* is implemented in micro-code realizing a process switching time of 1 μsec.
3. A built-in *floating point co-processor* realizing 1.5 MFLOPS.
4. A built-in *timer* with an accuracy of 1 μsec. in high-priority mode.
5. Fast (50 nano-seconds) *internal memory* of 4 Kbyte.

2.1 The OCCAM process

The underlying idea of Occam is the notion of a process. A process is a part of a program that starts, performs an action and then terminates. The idea is that processes may be executed in parallel. The communication between processes is done via self-synchronizing channels. A channel is a one way point-to-point connection from one process to

another. The processes may be located on different transputers. In that case the communication over channels changes into communication over links. There is no need for the programmer to worry about the implementation of the channel or link protocol, because it is implemented in micro-code on the transputer chip.
Examples of so-called primitive processes in Occam are:

x := 3	assigns the value of three to the variable x
chan1 ? p	takes a value from an input channel and puts it in p
chan2 ! q	takes the value of q and outputs it over the channel

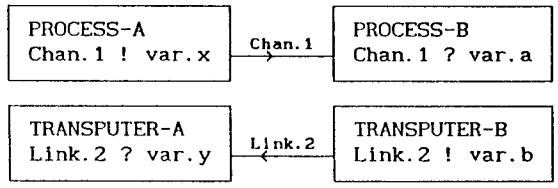
An Occam program consists of combinations of primitive processes, so that a number of these primitive processes may be combined into a larger construction. For this purpose Occam supports the following constructs:

SEQ	To indicate the start of
proc1	a series of sequential
proc2	processes.
PAR	To indicate the start of
proc1	a series of parallel
proc2	processes.

Note that the indentation under the SEQ and PAR construct are syntactic and are used to indicate the begin and end of the construct.

2.2 Channels and Links

In Occam the communication by means of channels or links looks like:



The input (?) and the output (!) processes may be considered as the synchronization between processes running on transputers A and B. If the process on transputer-A arrives at the input instruction on link.2 and transputer-B is not yet ready to provide this output, the process running on transputer-A will be de-scheduled until transputer-B signals that it is ready to send the data over link.2. At that point the inputting process on transputer-A is resumed and the data is transferred via link.2. This results in a synchronizing action between the processes running on different transputers. In this way the transputer channels provide the necessary system interconnection and synchronization. The only thing the programmer has to do, is to define the appropriate input and output actions.
Internally the channel administration is kept in one memory word. This memory word will contain the most negative value to indicate that the channel has not been called yet, or a value that is interpreted as a workspace pointer of a (de-scheduled) process. The transputer channels may therefore be considered as a standard system interface because they provide the necessary system interconnection and synchronization.
The link concept is, identical to the channel concept, used to communicate between processes located on different transputers. This unifying link/channel concept is very convenient during the system design and check-out phase. Different hardware and software layers can be developed independently as long as the software and hardware interface have been defined.

2.3 Interrupts

The transputer has an interrupt signal pin connected to the event channel. This event channel is read identically to a link although no data can be transferred. It can only be used to activate a process that is waiting on an input (?) from the event channel. As soon as the event channel is activated, the waiting process is scheduled by inserting it at the back of the *ready queue* in the low priority mode, or it may be scheduled at once as a high priority process. The use of the transputer interrupt signal is illustrated in the following Occam interrupt process that is activated every time the event pin is triggered. The program starts with the definition of the process name with the external channel definition. The internal channel event is declared as the type ANY, because there is no data transport over the event channel. This event channel is associated with the actual hardware pin.

```
PROC Interrupt (CHAN OF INT16 output)
  CHAN OF ANY event:
  PLACE event AT event.In:
  WHILE TRUE                        Do for ever loop!
    SEQ                             Start sequential process
      event ? any                   Wait for interrupt
      .....execute handler code :
```

The transputer has one high-priority and one low-priority operating mode. The interrupt process should preferably be executed in the high priority mode in order to reduce the interrupt response time. Theoretical response times of approximately 55 μsec. have been reported by Welch (1987)

2.4 The Timer

The timer channel should also be considered an input channel. Times of the internal running clock are obtained by reading the timer channel and operating on the time value. The Occam code to use the timer is illustrated with the following SEQuential process that causes a delay.

```
PROC delay (VAL INT interval)
    TIMER clock:
    INT time:
    SEQ
      clock ? time
      clock ? AFTER time PLUS interval :
```

Internally the *timer queue* differs from the *ready queue* in that it is kept sorted in the sequence of the wake up times of the different timed processes.

2.5 The Scheduler

In addition to the channel protocol concept, there is another important difference between transputers and other microprocessors. That is the built-in scheduler. The transputer should be considered as handling tasks or processes rather than machine instructions. These processes are controlled by a built-in scheduler. The process administration is the main activity of the scheduler. The scheduler keeps track of the different processes by administering a list of processes in an area of memory that is called the workspace. The workspace of the current process is pointed to by the transputer register called workspace pointer. The workspaces are linked together to form a queue. The scheduler keeps track of four queues i.e. for each priority a ready queue and a timer queue. The beginning and end pointer to a queue are maintained in the transputer registers called Front pointer and Back pointer. Workspaces are added to the ready queue by reference to the back pointer and workspaces are extracted by reference to the front pointer.

2.5.1 High-priority mode

The process (or task) scheduler can operate in one of two modes i.e. the high-priority or the low-priority mode. In the high-priority mode the executing process can not be interrupted by another process (non-preemptive scheduling). De-scheduling can only occur when:

- a process is completed
- communication with a channel / link is not (yet) possible
- the process waits for a timer to elapse

2.5.2 Low-priority mode

In the low-priority mode the processes are, by means of time-slicing, scheduled in a round-robin manner. Here processes will be de-scheduled by:

- completion of the process
- awaiting link or channel communication
- expiration of a time slice

The de-scheduled process will be added to the back of the ready queue and the process from the front of the ready queue will be executed. The time slice in a transputer is typically 1 msec. and the process switching time is of the order of 1 μsec.
In Occam the priority may be assigned using a PRIority addition to the PAR construct.

No priority	With priority
PAR	**PRI PAR**
process.1	**process.1**
process.2	**process.2**

The processes 1 and 2 are executed in parallel, either both at the same priority, or with the PRI PAR: process 1 will be executed in high priority and process 2 in low priority. The PAR process is only finished after both processes have been executed.

3 A SCHEDULER FOR REAL-TIME CONTROL

3.1 The Sampling Process

The sampling process of a control system may, according to Bakkers (1987), be divided into:

1. Time-bounded processes, such as the sampling or actuation actions.
2. Time-limited processes such as the calculation of the control action.
3. Background and alarm processes.

The time-bounded processes have to be scheduled at specific instants. The time-limited processes should be scheduled to meet their deadline. Synchronization between these processes is required in order to guarantee that the calculation is performed before the actual control action takes place. If a control system can not inherently give this guarantee the resulting misses of the correct synchronization should:

1. result in a non-detrimental control action.
2. result in error messages that are meaningful to the control engineer.
3. never cause a deadlock of the system.

3.2 Real-Time Scheduling

From the theory of real-time scheduling, Liu (1973) has defined the optimum schedule for the execution of control processes: it is the deadline driven schedule. A sub set of it is the monotonic rate schedule, which may be used in the case of fixed priorities. An example of the deadline driven scheduling rule is illustrated in fig. 2.

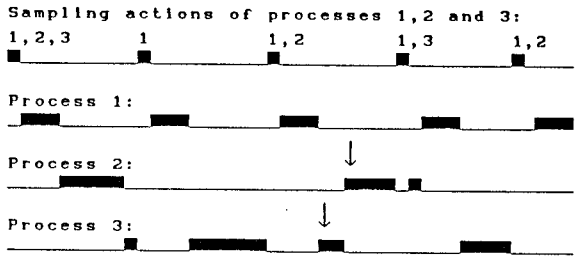


Fig. 2 Deadline driven scheduling

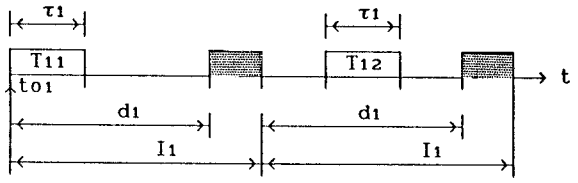
Note the position indicated where a fixed-priority monotonic rate algorithm would have resulted in a selection of process 2 instead of process 3. In general a schedule of a number of different processes, may be defined as:

$$Schedule = \{T_i\}, \{\tau_i\}, \{l_i\}, \{d_i\}, \{to_i\} \qquad (1)$$

where:

T_i = one of n processes
τ_i = the time to execute process T_i
l_i = the period of process T_i
d_i = the time when proces d_i is due
to_i = start of scheduling process i

This may be illustrated as follows:



The deadline-driven algorithm assigns the processor, at each decision-time, to the process whose deadline is closest. A necessary condition is caused by the fact that a processor at most may be kept busy all the time or:

$$\sum_{i=1}^{n} \frac{\tau_i}{l_i} \leq 1 \qquad (2)$$

Although this is a necessary condition, it has not been proven that this condition is also sufficient. Only for the sub case in which $d_i = l_i$ was proven that the necessary condition is also sufficient. If we also make the assumption that in a control system all sampling actions start at the same time, we have the additional requirement of:

$$to_i = to_j \qquad (3)$$

The optimum scheduling algorithm has not been implemented yet in industrial real-time operating systems (Stankovic, 1988). The main reasons is the computing effort necessary to keep the priority list of the processes sorted at all times. This rule however plays an important role in the manual scheduling of processes in the traditional control system design. It must be stressed however that this optimum rule is defined only for a single processor case.

With the transputer hardware it is possible to construct and program a parallel machine. The question arises how the control system design should be performed with such hardware. One should realize that the process scheduler on the transputer chip is a round robin scheduler which in low-priority mode performs a process switch about every msec. There is also a high-priority mode in which the time slicing is disabled so that the processes run till completion or are de-scheduled due to a wait for input or output. In order to realize deadline scheduling on a transputer a scheduler that accomodates priorities will have to be implemented.

In the ideal case, the realization of the priority ordering should be be performed by a hardware ordering mechanism on the microprocessor chip. A large number of priorities will have to be implemented. This will result in a fine granularity of the priorities. Because in a deadline driven priority system the priority axis is basically the time axis, this large number of priorities results in a fine resolution of the time axis, together with a reasonable cycle time for the total process cycle: of a few days. In order to restrict the length of the priority sort list to an hardware-technical implementable value, the processes that are part of a set of processes or tasks should get the same priority as their main process. This allows the use of one process queue per priority class.

The scheduling for a deadline driven scheduler should be a preemptive scheduler. This requires far more overhead than the present low-priority scheduler, that schedules only on certain instructions, like a jump. Thus ensuring that the data structure (stack) that has to be saved is minimal. If this way of thinking is adopted for the priority scheduler a so called semi-preemptive scheduler will result that performs well within the specification of the real-time requirements and also maintains a fast process switch time. The proposed priority system should accommodate:

Time bounded processes	at Hi-Priority preemptive
Alarm processes	at High priority
Time limited processes	at Medium priority
User process	at Low priority
Background processes	at Round-robin scheduler

Future implementation of a deadline driven scheduler for the time bounded control tasks should be considered by the hardware manufacturers.

The assignment of the priorities is the task of the control engineer. He will have to analyze the time requirements of the system processes beforehand. If this extra work is not performed, the scheduler should fall back on the lowest, or round-robin, scheduler without loss of performance.

A sub-optimal solution is the following. Allow enough time for all the time-limited tasks to be completed on time. Add a safety margin. Because the completion of the tasks can never be *guaranteed*, timer guards should be installed to detect processes exceeding the time limit. A transputer timer process is well suited for this purpose.

4 SYSTEM ARCHITECTURE

The use of transputers in a control system design, necessitates the design of a transputer network that reflects the parallelism of the particular control system. There is no standard procedure to convert a sequential problem into a transputer topology with a corresponding parallel program. Therefore, the system architecture should be tailored to the (control) problem. There are basically three methods to obtain a parallel architecture i.e.

1. Parallelism by recognizing various *layers*. This is further worked out below.

2. *Ad-hoc* based on the properties of the problem. In this situation it is necessary to recognize dedicated parallelism of the control system. An example of this technique is used with the controller for the flexible robot arm that is described in Section 5.

3. By means of a mechanism that creates *massive parallelism* from repetetive use of basic building blocks. For distribution of these blocks over a number of processors a method developed by Hilhorst (1987), should be used to obtain an optimal distribution of the tasks over a given network. This is described in more detail in Section 6.

As a result of experience gained in the realization of several transputer-based control systems by Bakkers (1986) and Stavenuiter (1989), a layered architecture is proposed. The basic idea is that there are several layers in a practical control system as indicated in fig. 3 for a robot control system. A more detailed description of the different layers is given below.

4.1 First Layer

The first layer consists of the hardware and software directly connected to the sensors of the control system. It can be realized as the first transputer layer. This interface layer includes, for example: Analog-to-Digital, Digital-to-Analog and Resolver-to-Digital converters.

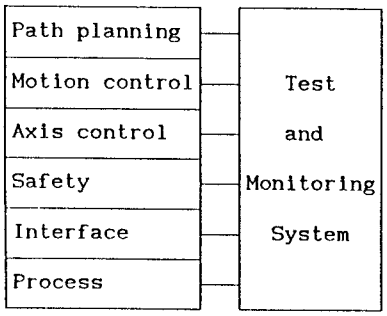


Fig. 3 Layer structure for a robot system

The software of this layer performs primarily the sampling and control action. This layer is also the right place to perform the necessary filtering of the measured data. The processed measurements are available at the link interface input and outputs. Schematically the first layer and its interconnections to the second layer and the monitor and display may be represented as in fig 4.

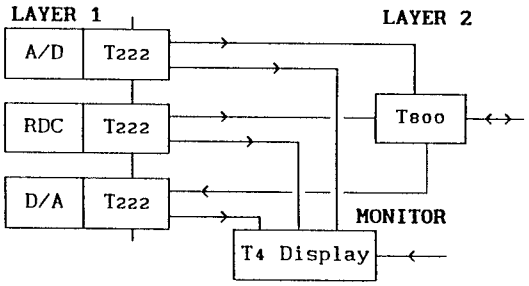


Fig. 4 Layer 1 interconnections

4.2 Second layer

The second layer consists of transputers that execute the safety software. This layer completely isolates the sensors from the rest of the control system. Actual sensor data arrive here and may be compared with the control signals to check whether the combination of the two may lead to a dangerous situation. If so, the safety software has priority and will set the control signals at a safe value. The topology of this layer could include one or more transputers, as illustrated in fig. 5. where T800 transputers are used.

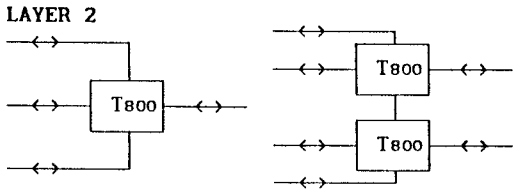


Fig. 5 Layer 2 the safety layer

4.3 Third layer

The third layer typically contains the first calculation layer, in robot terms this could be the calculation of the robot dynamics. It is not very likely that this calculation can be performed in one transputer. Therefore, as an example, a transputer network of eight transputers is given in fig. 6. For the distribution of the code over this topology the method of Hilhorst (1987) may be used. This method, based on the list scheduling analysis, also takes communication times via the links into account. By using a critical path analysis, an (almost) optimum distribution of the various processes over the different transputers can be obtained.

4.4 Test layer

This layer collects the properly buffered test data and sends it to a graphical display. It acts as a software test rig. The test data may also include warnings for synchronization errors and sensor data. The synchronization error messages enable the control system designer not only to spot errors in the system design, but also associates these errors with the exact location of the erred sampling or control process. Schematically this layer may be represented as in fig. 7.

One has to realize that parallel programs can not be tested for correct operation by the insertion of a number of write statements in the program code. The channel concept is very powerful. On the other hand it contains a possible danger. If the test data are not properly de-coupled by a-synchronous buffers, the monitor and test unit may set the pace, or worse, it may cause a deadlock of the control system.

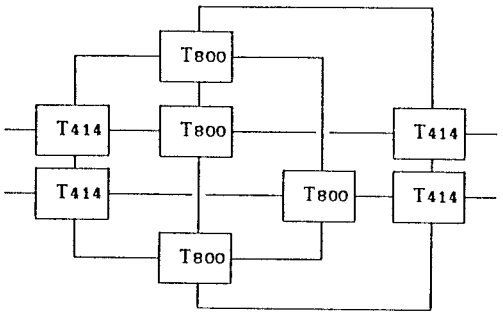


Fig. 6 Robot dynamics layer

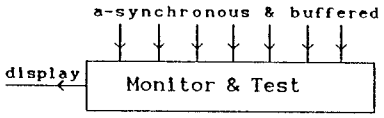


Fig. 7
The software test rig

5 CONTROL OF A FLEXIBLE ROBOT LINK

The ad-hoc parallelism will be illustrated with the example of a flexible robot link as described by Stavenuiter (1989) and Kruise (1988). It is constructed as a long aluminum strip (190 x 6 x 0.4 cm) driven by a DC motor. When controlled as a rigid system, the flexible robot link will heavily vibrate. Therefore, the vibrations and bending have to be taken into account in the controller design. The bending of an accelerating link has been described by Fukuda (1983) as:

$$W(r,t) = \sum_{n=1}^{\infty} Y_n(r)\, M_n(t) \qquad (4)$$

where:
$Y_n(r)$, called the 'mode shape' function, describes the shape of the arm at its n^{th} resonance frequency.

$M_n(t)$, the modal function, is a time dependent function which may be described by:

$$\omega_i^2 M_i + 2z_i \omega_i \dot{M}_i + \ddot{M}_i = -A_i \ddot{\Phi} \qquad (5)$$

where:

ω_i = the i^{th} resonance frequency
$\ddot{\Phi}$ = is the acceleration of the arm
z_i and A_i are arm dependent parameters

The combination $Y_i(r)M_i(t)$ is called a mode. Clearly the mode shape function cannot be controlled. However, the flexible arm can be made to rotate like a rigid one by controlling the acceleration in such a way that the modal function decays rapidly. In practice, it is not possible to control an infinite number of modes, but the model of the arm can be simplified by assuming that the motor bandwidth is less than ω_{mo}, which implies that excitation of frequencies higher than ω_{mo} may be disregarded. Furthermore, higher modes tend to have smaller amplitudes and therefore, they may be disregarded as well.
Simulations by Kruise (1988) indicate that control of the first three modes yields a satisfactory response. Furthermore, as these modes are independent, the control signal for the entire system can be computed by adding the control signals for the three modes. Taking the Laplace transform of (5) yields:

$$s^2 M_i + 2z_i\omega_i s M_i + \omega^2 M_i = -A_i s^2 \Phi \qquad (6)$$

which omitting subscripts, can be written as

$$M + \frac{2z\omega M}{s} + \frac{\omega^2 M}{s^2} = -A\Phi \qquad (7)$$

This equation can be translated into the block diagram of fig. 8. The modal function M is controlled by using state feedback, where the states are defined to be Φ, $\int M$ and $\int\int M$, where $\int M$ stands for:

$$\int_0^t M\, d\tau$$

From these states only Φ can be measured, $\int M$ and $\int\int M$ have to be estimated. The estimation is updated by measuring the vibrations of the arm by means of the strain gauges. In the case that three modes are considered to contribute to the vibrations of the arm, three pairs of strain gauges are used. A similar controller and estimator is implemented for each mode. The controller and estimator for one mode is depicted in fig. 8.

In this case a quite natural parallelism is present that may be translated into a four transputer network configuration. The calculations of each mode will be assigned to one slave transputer. The master transputer collects the data and performs preliminary data analysis. This configuration may be represented as illustrated in fig. 9.
The use of transputers makes it necessary to use parallel programming techniques. The transputer environment may be programmed in Occam

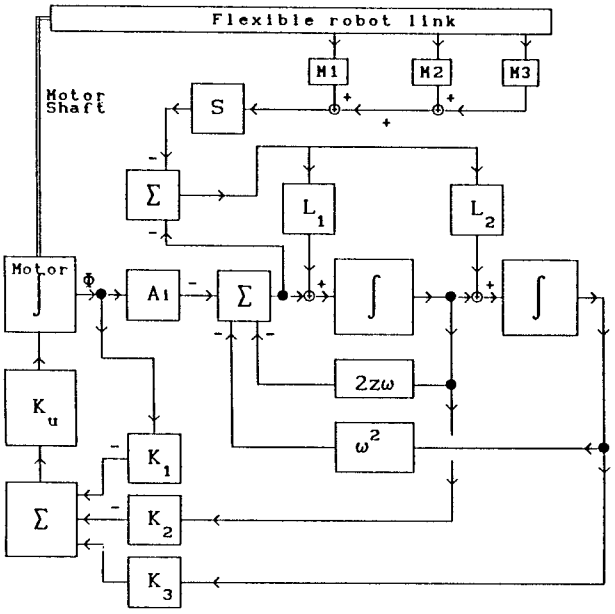


Fig. 8 One mode estimator and controller

(the preferred parallel transputer language) or in parallel C. Methods of parallel programming are given in Bakkers (1988, 1989). Code from different programming languages may be cast into an Occam harness to exploit the parallelism between processes.

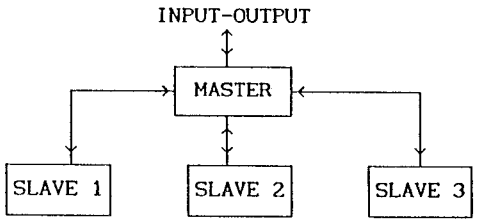


Fig. 9 Transputer network configuration

5.1 Natural system parallelism.

In the case of the flexible robot parallelism is naturally obtained by distrubuting the tasks of calculating the esitmator and the controller of the three vibration modes over three transputer. This way the natural parallelism of the different modes is reflected in the architecture of the transputer system as illustrated in fig. 9. Because this is not a rule that can be applied to any system, this technique should be considered an ad-hoc technique. The advantage is that the processing speed remains the same whether one or ten modes are calculated, of course requiring one or ten transputers. The combination of this ad-hoc system allocation together with the basic building block method, to be discussed in the next section, has been applied to the flexible robot arm.

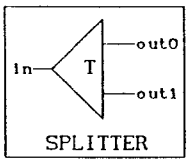
6 THE BASIC BUILDING BLOCK METHOD

A systematic way to introduce parallelism, is to start with very small processes and build a system with these elementary processes. This approach is similar to methods used im simulation programs. The internal scheduler on the transputer will realize parallelism while running these processes. If this philosophy is applied to the controller design of the flexible arm we can realize the controller of fig. 9 with the aid of only five basic elements i.e.

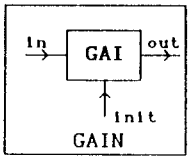
an integrator block:	int
a summer block:	sum
a minus block:	min
a gain block:	gai
a splitter block	tee

Lets first define these basic elements and then write the controller program using these basic elements.

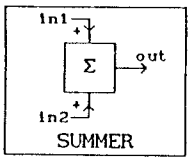
```
PROC tee (CHAN OF REAL32 in, out0, out1)
  REAL32 x :
  WHILE TRUE
    SEQ
      in ? x
      PAR
        out0 ! x
        out1 ! x :
```



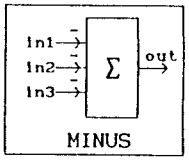

```
PROC gai (CHAN OF REAL32 in, out, init)
  REAL32 x,k :
  SEQ
    init ? k
    WHILE TRUE
    PRI ALT
      init ? k
          SKIP
      in ? x
          out ! k * x :
```



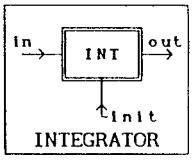

```
PROC sum (CHAN OF REAL32 in1, in2, out )
  REAL32 x1, x2 :
  WHILE TRUE
  SEQ
    PAR
      in1 ? x1
      in2 ? x2
    out !     x1 + x2 :
```




```
PROC min (CHAN OF REAL32 in1, in2, in3, out)
  REAL32 x1, x2, x3 :
  WHILE TRUE
    SEQ
      PAR
        in1 ? x1
        in2 ? x2
        in3 ? x3
      out ! -((x1 + x2) + x3 ) :
```




```
PROC int(CHAN OF REAL32 in, out, init)
  REAL32 x,t :       -- STATE
    SEQ
    init ? x;t  -- INITIALIZE STATE AND TIME
    WHILE TRUE
      PRI ALT
        init ? x;t :
          SKIP
        TRUE & SKIP
          REAL32 dx :
          SEQ
            PAR
              in ? dx      -- Input in parallel
              out ! x      -- with output
            x := x + (dx * t) :
```



The one mode estimator and controller for the flexible robot arm as illustrated in fig. 8 may be converted to consist of the basic building blocks mentioned above. The resulting block diagram is illustrated in fig. 10. The sign of some of the GAIN blocks has been changed to standardize on one SUMMER block. Note that the insertion of the SPLITTER blocks is necessary because the Occam channels are point-to-point connections. Installation of a SPLITTER block looks like a solder joint.

After the definition of the basic building blocks, the complete calculation process for one mode looks like the following Occam process. The external channels I[13] and the internal channels D[27] are not mentioned in the block diagram to avoid cluttering.

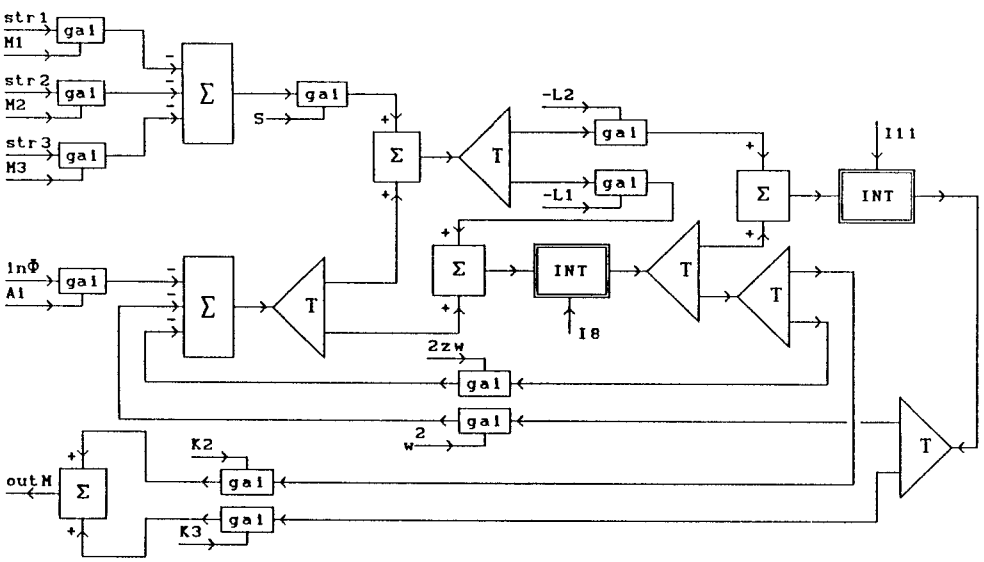


Fig. 10 Basic building-block diagram for one mode

```
PROC calc ([4]CHAN OF REAL32 in, CHAN OF REAL32 outM,
[13]CHAN OF REAL32 I)
  [27]CHAN OF REAL32 D :
  PAR
    gai ( in[0], D[0], I[0] )
    gai ( in[1], D[1], I[1] )
    gai ( in[2], D[2], I[2] )
    gai ( in[3], D[3], I[3] )
    min ( D[0], D[1], D[2], D[8] )
    min ( D[3], D[4], D[5], D[9] )
    sum ( D[6], D[7], outM )
    gai ( D[8], D[12], I[6] )
    gai ( D[10], D[6], I[4] )
    gai ( D[11], D[7], I[5] )
    tee ( D[9], D[13], D[14] )
    sum ( D[12], D[13], D[15] )
    sum ( D[16], D[14], D[17] )
    gai ( D[18], D[5], I[10] )
    gai ( D[19], D[4], I[11] )
    tee ( D[15], D[20], D[21] )
    int ( D[17], D[23], I[9] )
    sum ( D[22], D[24], D[26] )
    gai ( D[20], D[22], I[7] )
    gai ( D[21], D[16], I[8] )
    tee ( D[23], D[24], D[25] )
    tee ( D[25], D[10], D[18] )
    int ( D[26], D[27], I[11] )
    tee ( D[27], D[19], D[11] ) :
```

With these instructions the scheduler of the transputer can go to work and schedule all these parallel processes for execution. The advantage of this method is the speed of the development of the control program. The above program is for the control of one mode, therefore this program is assigned to one dedicated transputer for every mode that needs to be controlled. There are, however, some remarks to be made about this design method. Because a large number of building blocks are interconnected, there is a danger of deadlock. It is a property of so-called I/O-parallel blocks, that if combined into networks, they will exhibit deadlock free properties. An I/O parallel block executes the input *and* output in parallel and not sequentially. The careful use of a few I/O-parallel blocks in a network will result in a deadlock free behavior of the network. For that reason the integrators in fig. 10 are I/O parallel as indicated in the corresponding Occam program, resulting in a deadlock free controller.

7 RESULTS CONCLUSIONS AND SUGGESTIONS

The controller for the flexible robot arm has been implemented on a number of different processors. The results varied as follows:

Z80 + co-processor:	16 Hz
10 MHz AT + co-processor	116 Hz
4 T414 transputers	1000 Hz
4 T800 transputers	4500 Hz

For the architecture of the system was as illustrated in fig. 9. The actual controller was implemented using the basic building blocks technique. The overall conclusion of the experience gained over a number of projects is that the application of transputers allows the design of high-performance control systems. For complex systems the layered structure is the best approach to realize the parallel architecture. However, ad-hoc parallelism, as used in the control of the flexible robot arm, will give the best performance, because it minimizes the inter-transputer communication.

The basic building block technique together with the recognition of dedicated parallelism quickly results in an error-free design. For the design of more "general purpose" software for higher levels of the controller, the automatic task allocation program still has to be refined further.

The optimum scheduler for real-time systems is the deadline driven scheduler, it is recommended that this scheduler be implemented in future processors. A sub-optimal solution has been presented, where the timer process of the transputer is used to guard the timing sequence. The transputer with its built-in scheduler may at present be considered the best state-of-the-art implementation of a parallel building block for the realization of control systems.

8. REFERENCES

Bakkers, A.W.P. (1989) editor of: *Applying transputer based parallel machines*. Proceedings of the 10th Occam User Group Technical Meeting, 3-5 April 1989, Enschede, Netherlands.

Bakkers, A.W.P. (1988). *Application of parallel processing to robot control*. Proceedings of the First International Conference on Robot Technologies, TECHRO'88, September 19-26, 1988, Djuni, Burgas, Bulgaria.

Bakkers, A.W.P., R.M.A. van Rooij and L. James, (1987). *Design of a real-time operating system (RTOS) for robot control*. Proceedings of the 7th OCCAM User Group, Grenoble France, pp 318-327.

Bakkers, A.W.P. and W.L.A. Verhoeven, (1986). A Robot Control Algorithm Implementation in Transputers. Intelligent Autonomous Systems Conference, Amsterdam, pp 118-122.

Fukuda, T and Y. Kuribayashi, (1983), *Precise control of flexible arms with reliable systems*. Proc. Int. Conf. on advanced robotics, Japan Ind. Robot Assoc., Tokyo, Japan Vol 1 pp 237-244.

Hilhorst, R. (1987). *Parallelisation of Computational Algorithms for a Transputer Network*. Proceedings of the 7th OCCAM User Group Technical Meeting, Grenoble France, pp 420-424

Hoare, C.A.R. (1978). *Communicating sequential processes "CSP"*. CACM, vol. 21, No.8, 666-677.

JSME, (1989). *The international Conference on Advanced Mechatronics*. May 21-24 Tokyo, Japan, The Japan Society of Mechanical Engineers.

Kruise, L, J. van Amerongen, P. Löhnberg and M.J.L. Tiernego, (1988). *Modelling and control of a flexible robot link*. Proc. IUTAM/IFAC Symp. dynamics of controlled mechanical systems.

Liu, C.L. and J.W. Layland, (1973). Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. JACM, vol 20 (1), 46-61.

Stankovic, John A., *Misconceptions About Real-Time Computing A Serious Problem for the Next-Generation Systems*, IEEE Computer, October 1988, 10-19.

Stavenuiter, A.C.J., G. ter Reehorst, and A.W.P. Bakkers, (1989). *Transputer control of a flexible robot link*. Microprocessors and Microsystems Vol 13 No 3, April, 227-232.

Welch, P. (1987). *Managing hard real-time demands on transputers*. Proceedings of the 7th OCCAM User Group, Grenoble France, pp 135-145.

Wijbrans, K.C.J., J. Meijer, and A.W.P. Bakkers, (1989). *Real-time Sampling Sub System for the PC/AT*, to be published in Journal A, vol 30, nr 3.