

# Appendix E

---

## Control system software

This Appendix lists the suspension control programs for the vehicle and the single electromagnet systems. The three development support programs are also described. The various system software components, all of which are written in occam, are listed below:

E.1	Occam	222
E.1.1	Primitive processes	222
E.1.2	Combining processes	222
E.1.3	Data types	223
E.1.4	Channels	223
E.1.5	Timers	224
E.1.6	Program configuration	225
E.2	Vehicle suspension controller	226
E.2.1	Top-level Program configuration	226
PROC	Head.Controller.Process	226
PROC	Tail.Controller.Process	227
PROC	ADC.Process	228
PROC	DAC.Process	228
PLACED	PAR	229
E.2.2	System Constants & Comms Protocols Library	230
E.2.3	ADC and DAC Driver processes Library	232
PROC	ADC.Driver	232
PROC	DAC.Driver	233
PROC	Switch.Monitor	234
E.2.4	Control processes Library	235
PROC	Reference.Signal.Generator	235

PROC Signal.Data.Manager . . . . .	237
PROC Vehicle.Mode.Controller . . . . .	239
PROC Magnet.Force.Controller . . . . .	242
PROC Command.Message.Manager . . . . .	244
PROC Exception.Report.Manager . . . . .	245
E.2.5 Executive processes Library . . . . .	246
PROC Monitor.Data.Buffer . . . . .	246
PROC User.Interface . . . . .	247
Procedure code for User.Interface . . . . .	252
E.2.6 Default parameters Library for Executive . . . . .	254
E.2.7 Data Monitor process Library for Executive . . . . .	255
PROC Data.Monitor . . . . .	255
E.3 Single magnet suspension controller . . . . .	258
PROC Maglev.Controller . . . . .	258
PROC Set.default.parameters . . . . .	258
PROC Interface.to.outside.world . . . . .	259
PROC Calculate.magnet.force . . . . .	260
PROC Calculate.magnet.current . . . . .	262
PROC Handle.exceptions . . . . .	264
PROC Data.Buffer . . . . .	264
PROC Data.Monitor . . . . .	265
PROC Controller . . . . .	267
PROC Levitate . . . . .	268
PROC Menu . . . . .	270
PROC Kernel . . . . .	272
Procedure code for Maglev.Controller . . . . .	272
PROC ADC.driver . . . . .	273
PROC DAC.driver . . . . .	273
Process and channel Configuration . . . . .	274
E.4 Single electromagnet real-time simulator . . . . .	275
E.5 Transputer network monitor . . . . .	279
E.6 Transputer network data logger . . . . .	280

## E.1 Occam

The features which differentiate occam from conventional languages are now outlined to assist the reader in understanding the control system software. The description of occam is comprised of six parts namely: primitive processes; combining processes; data types; channels; timers and program configuration.

### E.1.1 Primitive processes

Occam programs are built up using processes, the simplest of which is an action (primitive process). The primitive processes include assignment, input, output, skip and stop. Table E.1 lists examples of these occam primitive processes.

**Table E.1** Occam primitive processes

PRIMITIVE PROCESS	ACTION
assignment	$x := y + 4$
input	keyboard ? character
output	screen ! character
no action, terminate	SKIP
no action, never terminate	STOP

### E.1.2 Combining processes

The action of one or more processes can be combined by a construction. A construction builds a process of the kinds listed in Table E.2. The SEQ, IF, PAR and ALT constructions can be replicated by the addition of an index specifier similar to that found in a conventional FOR loop statement.

**Table E.2** Occam combining processes

COMBINING PROCESS	CONSTRUCTION
sequential	SEQ
conditional	IF
selected	CASE
repeated	WHILE
parallel	PAR
alternative	ALT

### E.1.3 Data types

The primitive data types supported by occam are boolean values, bytes, integers (16, 32 and 64 bits) and ANSI/IEEE standard floating point numbers (32 and 64 bits). Non-primitive data types are limited to arrays, including multi-dimensional arrays. There is no structured data support similar to conventional record structures. The occam handling of arrays of differing sizes when used as process parameters is more flexible than with many conventional languages.

### E.1.4 Channels

Channels provide unbuffered, unidirectional, point-to-point communication of data between two concurrent processes. The format and type of data passed via a channel is specified by the channel protocol. The protocol declaration is similar to a record declaration in a conventional language. Variant protocols are supported in a similar way to conventional variant records. When a channel is declared, its type can be specified either by a data type or by a protocol. Channels are vitally important in concurrent systems because they facilitate synchronised communication between the concurrent program threads. Figure E.1 lists an example process which performs channel input and output using a replicated sequential construction.

```

PROTOCOL input.p IS INT, BYTE :

PROC data.expander( CHAN OF input.p  data.in,
                   CHAN OF BYTE  data.out )

    INT  length :
    BYTE character :

    SEQ
        data.in ? length, character
        SEQ index = 1 FOR length
            data.out ! character
    :

```

**Figure E.1** Occam channel example

### E.1.5 Timers

A timer provides a real-time clock facility. Timer is a primitive type, just like channels and data types. Only two operations are permitted on a timer: input and input-after. Input simply receives the present clock time, whilst input-after waits until the specified time before performing an input. Timers are very useful within control systems for generating signals at fixed or variable time intervals. Figure E.2 lists an example process which generates a periodic trigger signal. Modulo arithmetic must be used with timers since the hardware timer counters will restart periodically.

```

PROC Pulse.Generator(CHAN OF BOOL  trigger,
                    VAL INT  time.period )

TIMER  clock :
INT    time  :

SEQ
    clock ? time
    WHILE TRUE
        SEQ
            trigger ! TRUE
            time := time PLUS time.period
            clock ? AFTER time
    :

```

**Figure E.2** Occam timer example

### E.1.6 Program configuration

Having coded an algorithm, configuration information dependent on the physical resources being used may be needed. There are three areas that may require configuration. First, where a multi-processor system is used, the top hierarchical level of the program consists of a PLACED PAR construction which allocates processes to processors. Secondly, the channels linking processes which execute on different processors must use PLACE CHAN to allocate the channels to physical transputer links. The PLACE configuration can also be used to allocate variables to specific memory locations, eg. for memory mapped hardware devices. Finally, the processes within a parallel process construction can be prioritised using PRI PAR. However, this is limited to only two levels of priority on the T2, T4 and T8 series of processors. Figure E.3 lists a program configuration example.

```
CHAN OF REAL32  signal :  
  
PLACED PAR  
  
  PROCESSOR 1 T8  
    PLACE signal AT link0out :  
      signal.generator( signal )  
  
  PROCESSOR 2 T2  
    PLACE signal AT linklin :  
      convert.signal( signal )
```

**Figure E.3** Occam configuration example

## E.2 Vehicle suspension controller

### E.2.1 Top-level Program configuration

```

#USE "system.tsr"
#USE "controll.tsr"
#USE "executiv.tsr"
#USE userio

PROC Head.Controller.Process( CHAN OF BOOL data.conversion.request,
    CHAN OF converter.signal.p ADConverter.in,

    CHAN OF signal.p signal.exchange.out,
    CHAN OF signal.p signal.exchange.in,

    CHAN OF controller.p export.controller.params,
    CHAN OF exception.p import.exception.reports,

    CHAN OF ANY PC.screen,
    CHAN OF INT PC.keyboard )

-- Head.Controller() is the separately compiled process for all the processes
-- running on the First main controller processor (T800-25 Mhz, 1MB preferably).

CHAN OF converter.signal.p dummy.DACs :
CHAN OF signal.p reference.signals :

[2]CHAN OF signal.p accs.to.mode.cont, gaps.to.mode.cont :
[2]CHAN OF signal.p refs.to.mode.cont, force.from.mode.cont :
[2]CHAN OF signal.p gaps.forces.to.mag.cont, current.from.magnet.cont :

[2]CHAN OF signal.p mode.cont.data, magnet.cont.data :
CHAN OF monitor.p data.to.buffer, data.to.monitor, temp :

CHAN OF controller.p controller.params :
CHAN OF BOOL exception.manager.reset, exception.shutdown :

[2]CHAN OF controller.p mode.cont.params, magnet.cont.params :
[6]CHAN OF exception.p exception.reports :
CHAN OF BOOL dummy.switch :
CHAN OF exception.p exception.notification :

CHAN OF signal.gen.p ref.signal.params, ref.signal.status :
CHAN OF data.buffer.p data.buffer.params :
CHAN OF BOOL buffer.data.request :

PROC Buffer( CHAN OF monitor.p in, out)
    REAL32 time :
    [4][num.magnets]REAL32 magnet.data :
    [7][num.magnets]REAL32 mode.data :
    WHILE TRUE
        SEQ
            in ? time; magnet.data; mode.data
            out ! time; magnet.data; mode.data
    :

PRI PAR
    PAR
        Signal.Data.Manager( head.processor, ADConverter.in, reference.signals,
            accs.to.mode.cont, gaps.to.mode.cont,
            refs.to.mode.cont, force.from.mode.cont,
            gaps.forces.to.mag.cont, current.from.magnet.cont,
            dummy.DACs,
            signal.exchange.out, signal.exchange.in,
            mode.cont.data, magnet.cont.data, data.to.buffer )

        Command.Message.Manager( head.processor, controller.params,
            magnet.cont.params, mode.cont.params,
            export.controller.params, exception.manager.reset )

        Reference.Signal.Generator( data.conversion.request, reference.signals,
            ref.signal.params, ref.signal.status,
            exception.shutdown, exception.reports[0] )

```

```

PAR index = 0 FOR 2
  Vehicle.Mode.Controller( index, refs.to.mode.cont[index],
                           accs.to.mode.cont[index], gaps.to.mode.cont[index],
                           force.from.mode.cont[index], mode.cont.data[index],
                           mode.cont.params[index], exception.reports[index+1] )

PAR index = 0 FOR 2
  Magnet.Force.Controller( index, gaps.forces.to.mag.cont[index],
                           current.from.magnet.cont[index],
                           magnet.cont.data[index],
                           magnet.cont.params[index],
                           exception.reports[index+3] )

Monitor.Data.Buffer( data.to.buffer, buffer.data.request,
                    data.to.monitor, data.buffer.params,
                    exception.reports[5] )

Buffer( data.to.monitor, temp )

Exception.Report.Manager( head.processor, exception.manager.reset,
                          import.exception.reports, exception.reports,
                          dummy.switch, exception.notification,
                          exception.shutdown )

-- LOW priority process --
User.Interface( ref.signal.params, ref.signal.status,
               controller.params, data.buffer.params,
               PC.keyboard, PC.screen,
               buffer.data.request, temp,
               exception.notification )
:

#USE "system.tsr"
#USE "controll.tsr"

PROC Tail.Controller.Process( CHAN OF converter.signal.p currents.to.DACs,
                             CHAN OF BOOL DAC.switch,

                             CHAN OF signal.p signal.exchange.out,
                             CHAN OF signal.p signal.exchange.in,

                             CHAN OF exception.p export.exception.reports,
                             CHAN OF controller.p import.controller.params )

-- Tail.Controller() is the separately compiled process for all the processes
-- running on the Second main controller processor (T800-20 Mhz 32KB, ideally).

CHAN OF converter.signal.p dummy.ADC :
CHAN OF signal.p          dummy.sig.gen :

[2]CHAN OF signal.p accs.to.mode.cont, gaps.to.mode.cont :
[2]CHAN OF signal.p refs.to.mode.cont, force.from.mode.cont :
[2]CHAN OF signal.p gaps.forces.to.mag.cont, current.from.magnet.cont :

[2]CHAN OF signal.p mode.cont.data, magnet.cont.data :
CHAN OF monitor.p dummy.buffer.data :

CHAN OF controller.p dummy.params.out :
CHAN OF exception.p dummy.exception.report :
CHAN OF BOOL exception.manager.reset, dummy.shutdown :

[2]CHAN OF controller.p mode.cont.params, magnet.cont.params :
[4]CHAN OF exception.p exception.reports :

PRI PAR
  PAR
    Signal.Data.Manager( tail.processor, dummy.ADC, dummy.sig.gen,
                        accs.to.mode.cont, gaps.to.mode.cont,
                        refs.to.mode.cont, force.from.mode.cont,
                        gaps.forces.to.mag.cont, current.from.magnet.cont,
                        currents.to.DACs,
                        signal.exchange.out, signal.exchange.in,
                        mode.cont.data, magnet.cont.data, dummy.buffer.data )

    Command.Message.Manager( tail.processor, import.controller.params,
                             magnet.cont.params, mode.cont.params,
                             dummy.params.out, exception.manager.reset )

  PAR index = 0 FOR 2

```



```

    Vehicle.Mode.Controller( (index + 2), refs.to.mode.cont[index],
                             accs.to.mode.cont[index], gaps.to.mode.cont[index],
                             force.from.mode.cont[index], mode.cont.data[index],
                             mode.cont.params[index], exception.reports[index] )

    PAR index = 0 FOR 2
        Magnet.Force.Controller( (index + 2), gaps.forces.to.mag.cont[index],
                                   current.from.magnet.cont[index],
                                   magnet.cont.data[index],
                                   magnet.cont.params[index],
                                   exception.reports[index+2] )

        Exception.Report.Manager( tail.processor, exception.manager.reset,
                                   dummy.exception.report, exception.reports,
                                   DAC.switch, export.exception.reports,
                                   dummy.shutdown )

    PAR
        SKIP
:

#USE "system.tsr"
#USE "converte.tsr"

PROC ADC.Process( CHAN OF BOOL start.conversion.in,
                 CHAN OF converter.signal.p ADC.signals.out )

    -- ADC.Process() is the separately compiled process for all the processes
    -- running on the ADC card processor (T212 - 16 bit integer processor).

    -- Define PORTs for the ADC hardware --
    PORT OF INT multiplexor :
    PLACE multiplexor AT #6100 :

    PORT OF INT adc :
    PLACE adc AT #6000 :

    CHAN OF BOOL end.of.conversion :
    VAL event IS 8 : -- Transputer external hardware interrupt channel
    PLACE end.of.conversion AT event :

    -- Program code for ADC.Process() --
    PRI PAR
        ADC.Driver( start.conversion.in, ADC.signals.out,
                   multiplexor, adc, end.of.conversion )
        SKIP
:

#USE "system.tsr"
#USE "converte.tsr"

PROC DAC.Process( CHAN OF converter.signal.p DAC.signals.in,
                 CHAN OF BOOL switch.pressed.out )

    -- DAC.Process() is the separately compiled process for all the processes
    -- running on the DAC card processor (T212 - 16 bit integer processor).

    -- Define PORTs for the DAC hardware --
    PORT OF INT dac1, dac2, dac3, dac4 :
    PLACE dac1 AT #6100 :
    PLACE dac2 AT #6200 :
    PLACE dac3 AT #6300 :
    PLACE dac4 AT #6400 :

    -- Define a PORT for the watchdog timer --
    PORT OF INT watchdog :
    PLACE watchdog AT #6700 :

    -- Define a PORT for the front panel push switch --
    PORT OF INT switch :
    PLACE switch AT #6000 :

    -- Program code for ADC.Process() --
    PRI PAR
        DAC.Driver( DAC.signals.in, dac1, dac2, dac3, dac4, watchdog )
        Switch.Monitor( switch, switch.pressed.out )

```

```

:

-- Define physical link addresses --
VAL link0out IS 0 :
VAL link1out IS 1 :
VAL link2out IS 2 :
VAL link3out IS 3 :
VAL link0in IS 4 :
VAL link1in IS 5 :
VAL link2in IS 6 :
VAL link3in IS 7 :

CHAN OF BOOL          conversion.request :
CHAN OF converter.signal.p sensor.signals :

CHAN OF signal.p signals.to.tail :
CHAN OF signal.p signals.to.head :

CHAN OF controller.p tail.controller.params :
CHAN OF exception.p tail.exception.reports :

CHAN OF converter.signal.p current.demands :
CHAN OF BOOL          DAC.switch.pressed :

CHAN OF ANY PC.screen :
CHAN OF INT PC.keyboard :

PLACED PAR

PROCESSOR 1 T8 -- Head.Controllers
  PLACE conversion.request AT link1out :
  PLACE sensor.signals AT link1in :

  PLACE signals.to.tail AT link3out :
  PLACE signals.to.head AT link3in :

  PLACE tail.controller.params AT link2out :
  PLACE tail.exception.reports AT link2in :

  PLACE PC.screen AT link0out :
  PLACE PC.keyboard AT link0in :

  Head.Controller.Process( conversion.request, sensor.signals,
                           signals.to.tail, signals.to.head,
                           tail.controller.params, tail.exception.reports,
                           PC.screen, PC.keyboard )

PROCESSOR 2 T8 -- Tail.Controllers
  PLACE current.demands AT link2out :
  PLACE DAC.switch.pressed AT link2in :

  PLACE signals.to.head AT link0out :
  PLACE signals.to.tail AT link0in :

  PLACE tail.exception.reports AT link1out :
  PLACE tail.controller.params AT link1in :

  Tail.Controller.Process( current.demands, DAC.switch.pressed,
                           signals.to.head, signals.to.tail,
                           tail.exception.reports, tail.controller.params )

PROCESSOR 3 T2 -- ADC
  PLACE conversion.request AT link2in :
  PLACE sensor.signals AT link2out :

  ADC.Process( conversion.request, sensor.signals )

PROCESSOR 4 T2 -- DAC
  PLACE current.demands AT link0in :
  PLACE DAC.switch.pressed AT link0out :

  DAC.Process( current.demands, DAC.switch.pressed )

```

## E.2.2 System Constants &amp; Comms Protocols Library

```

-- CONSTANTS --

VAL num.magnets IS 4 : -- Number of vehicle magnets to be controlled
VAL num.modes IS 4 : -- Number of vehicle modes to be controlled

VAL head.processor IS TRUE :
VAL tail.processor IS NOT head.processor :

-- Controller modes --
VAL local IS 0 :
VAL vehicle IS local + 1 :
VAL user IS vehicle + 1 :

-- PROTOCOLS --

PROTOCOL exception.p IS -- Used to convey exception reports --
  INT:[]BYTE; -- Exception message text
  BYTE; -- Magnet or mode index
  REAL32; -- Exceptional or errant signal value
  BOOL: -- Severity flag (de-levitation requirement)

PROTOCOL controller.p -- Used to convey commands to the controllers --
CASE
  magnet.control.params; -- Conveys parameters to magnet force controllers
  REAL32; -- Control system sampling interval
  [num.magnets][num.modes]REAL32 -- Modes-magnets transformation matrix
  mode.control.params; -- Conveys parameters to suspension mode controllers
  REAL32; -- Control system sampling interval
  [num.modes][num.magnets]REAL32; -- Magnets-modes transformation matrix
  REAL32; -- State estimator filter frequency
  REAL32; -- Track trajectory filter frequency
  [num.modes]REAL32; -- Mode controller integral action time constant
  [num.modes]REAL32; -- Mode controller position feedback gain
  [num.modes]REAL32; -- Mode controller velocity feedback gain
  [num.modes]REAL32; -- Mode controller acceleration feedback gain
  INT; -- Control mode
  BOOL -- Reference signal injection select
  initialise.system -- Requests relevant system initialisation
:

PROTOCOL data.buffer.p -- Used to convey commands to the data buffer --
CASE
  data.buffer.params; -- Conveys parameters for storage of logged data
  REAL32; -- Data logging sample interval
  REAL32 -- Data logging duration
  data.buffer.log -- Commands storage of data at next ref signal cycle start
  data.buffer.select; -- Select source of data output from data buffer
  BOOL -- Output stored data (NOT on-line data stream)
:

PROTOCOL signal.gen.p -- Used to convey commands to the signal generator --
CASE
  signal.gen.params; -- Conveys parameters to the reference signal generator
  REAL32; -- Control system sampling interval
  REAL32; -- Mean value of air gap reference
  [num.modes]REAL32; -- Mean air gap mode factors
  BOOL; -- Dynamic reference signal select
  BOOL; -- Select sine wave reference signal (NOT square wave)
  REAL32; -- Reference signal frequency
  REAL32; -- Reference signal amplitude
  [num.modes]REAL32 -- Position reference mode factors
  signal.gen.levitate; -- Command vehicle levitation up/down
  BOOL -- Levitate vehicle (NOT de-levitate)
  signal.gen.levitated; -- Status report from signal generator
  BOOL -- Vehicle levitated (NOT de-levitated)
:

PROTOCOL converter.signal.p -- Used to convey converter signals --
CASE
  ADC.acc.raw.t; INT16 -- Raw ADC acc measurement
  ADC.gap.raw.t; INT16 -- Raw ADC gap measurement
  DAC.I.mA.t; BYTE; INT16 -- Magnet index; current in mA

```

```
:  
  
PROTOCOL signal.p -- Used to convey controller signals (all high priority) --  
CASE  
  
    time.stamp.t;      REAL32          -- Test signal time reference  
    mode.pos.refs.t;  [num.modes]REAL32 -- Mode position references  
  
    magnet.gap.t;     BYTE; REAL32     -- Magnet index; magnet air gap  
    magnet.gaps.t;   [num.magnets]REAL32 -- Magnet air gaps  
  
    magnet.acc.t;     BYTE; REAL32     -- Magnet index; magnet acceleration  
    magnet.accs.t;   [num.magnets]REAL32 -- Magnet accelerations  
  
    mode.force.t;     BYTE; REAL32     -- Mode index; mode force demand  
    mode.forces.t;   [num.modes]REAL32 -- Mode force demands  
  
    magnet.current.t; BYTE; REAL32     -- Magnet index; magnet current demand  
    magnet.currents.t; [num.magnets]REAL32 -- Magnet index; DAC current value  
  
    magnet.force.t;  BYTE; REAL32     -- Magnet index; magnet force demand  
  
    mode.block.t;    -- Transfer block for internal mode controller signals --  
    BYTE;           -- Mode index  
    REAL32;         -- Mode air gap  
    REAL32;         -- Mode acceleration  
    REAL32;         -- Mode velocity  
    REAL32;         -- Mode position  
    REAL32         -- Mode track position  
  
:  
  
PROTOCOL monitor.p IS -- Used to convey signals to the data buffer & monitor --  
REAL32;              -- Time reference (see above)  
[4][num.magnets]REAL32; -- Magnets gap, acceleration, force & current  
[7][num.modes]REAL32:  -- Modes pos.ref, gap, acc, vel, pos, trk.pos & force
```

## E.2.3 ADC and DAC Driver processes Library

```

#USE "system.tsr"

PROC ADC.Driver( CHAN OF BOOL data.request.in,
                CHAN OF converter.signal.p ADC.signals.out,
                PORT OF INT multiplexor, adc,
                CHAN OF BOOL end.of.conversion )

-- ADC.Driver waits for a start conversion trigger signal and then requests the
-- A/D conversion of each acceleration and air gap signal before scaling them and
-- outputting them to the data signal pool manager.
--
-- This process uses no external variables and must be configured for
-- high priority operation on the ADC card processor (T212).

-- Gap and Acc sensor multiplexor addresses :
VAL gap1.chan IS 1 :
VAL gap2.chan IS 2 :
VAL gap3.chan IS 4 :
VAL gap4.chan IS 8 :
VAL acc1.chan IS 16 :
VAL acc2.chan IS 32 :
VAL acc3.chan IS 64 :
VAL acc4.chan IS 128 :

-- Channel sequence definition (accs & gaps must be ordered 1-4).
-- Accelerations are read first since they can be processed before the air
-- gap signals are needed.
VAL INT num.adc.reads IS 2 * num.magnets :
VAL [num.adc.reads]INT sensor.sequence IS
    [ acc1.chan, acc2.chan, acc3.chan, acc4.chan,
      gap1.chan, gap2.chan, gap3.chan, gap4.chan ] :

BOOL go, done :
INT adc.reading, value, time :
TIMER clock :

SEQ

adc ? adc.reading -- Start the converter rolling

WHILE TRUE
  SEQ
  -- ADC conversions are achieved by setting the multiplexor channel address,
  -- then reading the ADC (which gives the previous conversion value and starts
  -- the next conversion. End of conversion is notified by a channel.

  multiplexor ! sensor.sequence[ 0 ] -- Select first multiplexor channel
  clock ? time
  clock ? AFTER (time PLUS 3) -- 3 us settling time for multiplexor

  data.request.in ? go -- Wait for request for data

  end.of.conversion ? done -- Clear last end of conversion notification
  adc ? adc.reading -- Start conversion of first channel

  -- Read each sensor signal and then scale and output --
  SEQ chan.index = 0 FOR num.adc.reads
  SEQ
  multiplexor ! sensor.sequence[ (chan.index + 1) REM num.adc.reads ]
  clock ? time
  clock ? AFTER (time PLUS 3) -- 3 us settling time for multiplexor

  end.of.conversion ? done -- Wait for conversion to complete
  adc ? adc.reading -- Read value from the ADC
  value := (adc.reading /\ #FFF0) -- 12 bit ADC, mask other bits

  CASE sensor.sequence[ chan.index]

  acc1.chan, acc2.chan, acc3.chan, acc4.chan
    ADC.signals.out ! ADC.acc.raw.t; INT16 value

  gap1.chan, gap2.chan, gap3.chan, gap4.chan
    ADC.signals.out ! ADC.gap.raw.t; INT16 value
:

```

```

#USE "system.tsr"

PROC DAC.Driver( CHAN OF converter.signal.p DAC.signals.in,
                PORT OF INT dac0, dac1, dac2, dac3,
                PORT OF INT watchdog )

-- DAC.Driver scales the magnet current demands and sends them to the DACs
-- It also resets the watchdog timer each time a current demand is processed.
--
-- This process uses no external variables and must be configured for
-- high priority operation on the DAC card processor (T212).

-- DAC specification --
VAL levels.per.fsd IS 4096 (INT32):
VAL volts.per.fsd IS 10 :
VAL min.DAC.level IS 0 :
VAL max.DAC.level IS 4095 :

-- Electromagnet current amplifier details --
VAL amps.per.volt IS 2 :

-- Calculate composite denominator term --
VAL INT32 mA.per.fsd IS INT32 (1000 * (amps.per.volt * volts.per.fsd)) :

BYTE magnet :
INT16 I.mA :
INT DAC.drive :

WHILE TRUE
SEQ
  DAC.signals.in ? CASE DAC.I.mA.t; magnet; I.mA

  DAC.drive := INT (((INT32 I.mA) * levels.per.fsd) / mA.per.fsd)
  IF
    (DAC.drive < min.DAC.level)
      DAC.drive := min.DAC.level
    (DAC.drive > max.DAC.level)
      DAC.drive := max.DAC.level
  TRUE
  SKIP

  CASE INT magnet
  0
    dac0 ! DAC.drive
  1
    dac1 ! DAC.drive
  2
    dac2 ! DAC.drive
  3
    dac3 ! DAC.drive

  watchdog ! 0
:

```

```

#USE "system.tsr"

PROC Switch.Monitor( PORT OF INT  switch,
                    CHAN OF BOOL  switch.pressed )

-- Switch.Monitor generates an exception message each time the push switch
-- on the front panel of the DAC card is pressed. Debouncing and hysteresis
-- are required before accepting a switch actuation.

-- This process uses no external variables and MUST be configured for low
-- priority operation.

VAL [5]BYTE message IS [0 (BYTE), 0 (BYTE), 68 (BYTE), 65 (BYTE), 67 (BYTE) ] :
VAL REAL32 zero IS 0.0 (REAL32) :
VAL INT sample.time.ticks IS 156 : -- approximately 10 ms

INT filter, time, input :
BOOL pressed, pressed.before :
TIMER clock :

SEQ
  filter := 0
  pressed := FALSE
  clock ? time

  WHILE TRUE
    SEQ
      switch ? input
      input := (input /\ 1) * 1000

      -- Debounce the switch output using a low-pass filter with Tc = 10 Ts
      filter := ( (filter * 9) + input ) / 10

      -- Detect switch transitions and apply debouncing --
      pressed.before := pressed
      pressed := (filter > 500)
      IF
        pressed AND (NOT pressed.before)
          SEQ
            filter := filter + 500
            switch.pressed ! TRUE
            (NOT pressed) AND pressed.before
            filter := filter - 500
          TRUE
          SKIP

      -- Wait for next sampling instant --
      time := time PLUS sample.time.ticks
      clock ? AFTER time

```

## E.2.4 Control processes Library

```

#USE "system.tsr"
#USE  snqlmath

PROC Reference.Signal.Generator( CHAN OF BOOL request.data.conversion.out,
                                CHAN OF signal.p ref.signals.out,

                                CHAN OF signal.gen.p parameters.in,
                                CHAN OF signal.gen.p status.out,
                                CHAN OF BOOL shutdown.request.in,
                                CHAN OF exception.p exception.report.out )

-- Reference.Signal.Generator() receives parameters and commands from the user
-- interface and exception report manager. It initiates each iteration of the
-- control system by requesting the ADC to read the sensor signals. It also
-- generates a time reference signal and a position refernce signal for a smooth
-- startup and shutdown as well as generating periodic signals for test purposes.
--
-- This process uses no external variables and must be configured for
-- high priority operation.

-- Worldly & other constants --
VAL two.pi IS 2.0 (REAL32) * 3.1415926 (REAL32) :
VAL set.down.on.track IS 0.008 (REAL32) :
VAL start.stop.time.constant IS 0.5 (REAL32) :

-- General variables --
CHAN OF signal.gen.p status.buffer :
BOOL buffered.status :

BOOL running, levitate, fully.up, fully.down, previously.fully.down :
BOOL dynamic.ref.selected, sine.wave.selected :
REAL32 sample.interval, mean.gap, ref.freq, ref.amplitude :
[num.modes]REAL32 mean.gap.coeffs, dyn.ref.coeffs, mode.refs :
REAL32 filter.decay.coeff, filter.input.coeff, filter1, filter2 :
REAL32 gap.ref, mean.ref, dyn.ref, time, sine.wave, cosine.wave :

-- Sampling values and variables --
VAL ticks.per.sec IS 1000000.0 (REAL32) :
INT sample.interval.ticks, sample.time, time.now :
REAL32 cycle.time :
TIMER clock :

PAR
  WHILE TRUE
    SEQ
      status.buffer ? CASE signal.gen.levitated; buffered.status
      status.out ! signal.gen.levitated; buffered.status
    SEQ
      running := FALSE
      previously.fully.down := TRUE
      WHILE TRUE
        PRI ALT
          ALT
            shutdown.request.in ? levitate
            levitate := FALSE
            parameters.in ? CASE
              signal.gen.levitate; levitate
            SKIP
            signal.gen.params; sample.interval; mean.gap; mean.gap.coeffs;
              dynamic.ref.selected; sine.wave.selected; ref.freq;
              ref.amplitude; dyn.ref.coeffs
          SEQ
            sample.interval.ticks := INT ROUND (ticks.per.sec * sample.interval)
            filter.decay.coeff := exp(-(sample.interval/start.stop.time.constant))
            filter.input.coeff := 1.0 (REAL32) - filter.decay.coeff
            ref.amplitude := ref.amplitude / 2.0 (REAL32) -- was pk-pk
            IF
              NOT running -- Then Initialise system --
                SEQ
                  running := TRUE
                  levitate := FALSE
                  filter1 := set.down.on.track
                  filter2 := filter1

```



```

        time := 0.0 (REAL32)
        clock ? sample.time
    TRUE
    SKIP

running & SKIP
SEQ
-- Set mean reference gap for either up or down --
IF
    levitate
    gap.ref := mean.gap
    NOT levitate
    SEQ
        gap.ref := set.down.on.track
        time := 0.0 (REAL32)

-- Ensure smooth startup & shutdown by filtering the reference signal --
filter1 := (filter1 * filter.decay.coeff)+(gap.ref * filter.input.coeff)
filter2 := (filter2 * filter.decay.coeff)+(filter1 * filter.input.coeff)
mean.ref := filter2
fully.up := (mean.ref < (mean.gap + 1.0E-4(REAL32)))
previously.fully.down := fully.down
fully.down := (mean.ref > (set.down.on.track - 1.0E-4(REAL32)))

-- Add dynamic reference to mean reference if required --
IF
    (fully.up AND dynamic.ref.selected)
    SEQ
        -- Advance time and limit domain to prevent loss of precision --
        IF
            (time < (1.0(REAL32) / ref.freq))
            time := time + sample.interval
            TRUE
            time := 0.0 (REAL32)

        -- Generate dynamic reference component --
        sine.wave := ref.amplitude * SIN( two.pi * (ref.freq * time) )
        cosine.wave := ref.amplitude * COS( two.pi * (ref.freq * time) )
        IF
            sine.wave.selected
            dyn.ref := sine.wave
            TRUE -- square wave --
            IF
                (sine.wave > 0.0 (REAL32))
                dyn.ref := ref.amplitude
            TRUE
                dyn.ref := - ref.amplitude
        TRUE
        dyn.ref := 0.0(REAL32)

-- Wait for next sampling time --
clock ? time.now
cycle.time := REAL32 TRUNC (time.now MINUS sample.time)
sample.time := sample.time PLUS sample.interval.ticks
IF
    (time.now AFTER sample.time)
    VAL [ ]BYTE message IS "Error: Sampling period too short, cycle time:"

    SEQ
        exception.report.out ! SIZE message::message; BYTE 0;
            (cycle.time / ticks.per.sec) * 1000.0 (REAL32); TRUE
        sample.time := time.now
    TRUE
    clock ? AFTER sample.time

IF
    NOT fully.down
    SEQ
        -- Request conversion of sensor signals by the ADC --
        request.data.conversion.out ! TRUE

        -- Output the time reference and mode reference signals --
        SEQ index = 0 FOR num.modes
            mode.refs[index] := (mean.ref * mean.gap.coeffs[index]) +
                (dyn.ref * dyn.ref.coeffs[index])
        --mode.refs[2] := (cosine.wave * dyn.ref.coeffs[2])

        ref.signals.out ! time.stamp.t; time
        ref.signals.out ! mode.pos.refs.t; mode.refs
    (fully.down AND (NOT previously.fully.down))
    status.buffer ! signal.gen.levitated; FALSE

```

```

TRUE
SKIP
:
#USE "system.tsr"

PROC Signal.Data.Manager( VAL BOOL head.manager,

  CHAN OF converter.signal.p from.analogue.converter,
    CHAN OF signal.p sig.gen.refs.in,
    []CHAN OF signal.p accs.to.mode.cont, gaps.to.mode.cont,
    []CHAN OF signal.p refs.to.mode.cont, force.from.mode.cont,
    []CHAN OF signal.p gaps.and.forces.to.mag.cont, current.from.mag.cont,
  CHAN OF converter.signal.p to.analogue.converter,

    CHAN OF signal.p exchange.data.out, exchange.data.in,

    []CHAN OF signal.p data.from.mode.cont,
    []CHAN OF signal.p force.from.magnet.cont,
    CHAN OF monitor.p data.to.monitor.buffer )

-- Signal.Data.Manager() runs on both main processors and controls
-- the flow of all signals from the ADC through to the DAC. The Boolean
-- 'head.manager' determines which processes are connected to this process.
--
-- This process uses no external variables and should be configured for
-- high priority operation.

VAL num.local.magnet.controllers IS SIZE gaps.and.forces.to.mag.cont :
VAL num.local.mode.controllers IS SIZE accs.to.mode.cont:

-- ADC input voltage range --
VAL volts.per.fsd IS 5.0 (REAL32) / 32768.0 (REAL32) :

-- Accelerometer details --
VAL Gs.per.volt IS 0.2 (REAL32):
VAL G IS 9.81 (REAL32) :
VAL acc.multiplier IS (volts.per.fsd * Gs.per.volt) * G :

-- Gap sensor details --
VAL metres.per.volt IS 0.00125 (REAL32) :
VAL gap.multiplier IS volts.per.fsd * metres.per.volt :
VAL sensor.offset IS 0.0032 (REAL32) : -- 3.2 mm
VAL net.track.gap IS 0.009 (REAL32) : -- 9 mm
VAL gap.offset IS (net.track.gap - sensor.offset) :

BYTE magnet, MAGNET, mode, MODE :
INT m, M :

INT16 magnet.acc.raw, magnet.gap.raw :

REAL32 time, magnet.acc, magnet.gap, magnet.current, MAGNET.CURRENT :
[num.magnets]REAL32 magnet.accs, magnet.gaps, magnet.currents :
REAL32 magnet.force, MAGNET.FORCE :
[num.magnets]REAL32 magnet.forces, MAGNET.FORCES :

REAL32 mode.force, MODE.FORCE :
[num.modes]REAL32 mode.pos.refs, mode.forces :

INT16 magnet.current.mA, MAGNET.CURRENT.mA :

REAL32 gap, acc, vel, pos, trk.pos, GAP, ACC, VEL, POS, TRK.POS :
[num.modes]REAL32 gaps, accs, vels, poss, trk.poss :

WHILE TRUE
  SEQ
    -- Get time stamp and reference signals from the signal generator --
    IF head.manager
      SEQ
        sig.gen.refs.in ? CASE time.stamp.t; time
        sig.gen.refs.in ? CASE mode.pos.refs.t; mode.pos.refs
        exchange.data.out ! time.stamp.t; time
        exchange.data.out ! mode.pos.refs.t; mode.pos.refs
      NOT head.manager
      SEQ
        exchange.data.in ? CASE time.stamp.t; time
        exchange.data.in ? CASE mode.pos.refs.t; mode.pos.refs

```

```

-- And send to the mode controllers --
SEQ controllers = 0 FOR num.local.mode.controllers
  refs.to.mode.cont[ controllers ] ! mode.pos.refs.t; mode.pos.refs

-- Get magnet acceleration measurements from the ADC / other processor --
SEQ magnet.index = 0 FOR num.magnets
  SEQ
  IF
  head.manager
  SEQ
  magnet := BYTE magnet.index
  from.analogue.converter ? CASE ADC.acc.raw.t; magnet.acc.raw
  magnet.acc := acc.multiplier * (REAL32 ROUND magnet.acc.raw)
  exchange.data.out ! magnet.acc.t; magnet; magnet.acc
  NOT head.manager
  exchange.data.in ? CASE magnet.acc.t; magnet; magnet.acc
  magnet.accs[ INT magnet ] := magnet.acc

-- And send to the mode controllers --
SEQ controllers = 0 FOR num.local.mode.controllers
  accs.to.mode.cont[ controllers ] ! magnet.accs.t; magnet.accs

-- Get magnet air gap measurements from the ADC / other processor --
SEQ magnet.index = 0 FOR num.magnets
  SEQ
  IF
  head.manager
  SEQ
  magnet := BYTE magnet.index
  from.analogue.converter ? CASE ADC.gap.raw.t; magnet.gap.raw
  magnet.gap := gap.offset - (gap.multiplier * (REAL32 ROUND magnet.gap.raw))
  exchange.data.out ! magnet.gap.t; magnet; magnet.gap
  NOT head.manager
  exchange.data.in ? CASE magnet.gap.t; magnet; magnet.gap
  magnet.gaps[ INT magnet ] := magnet.gap

-- And send to the mode controllers --
SEQ controllers = 0 FOR num.local.mode.controllers
  gaps.to.mode.cont[ controllers ] ! magnet.gaps.t; magnet.gaps

-- Get force demands from mode controllers and exchange with other processor --
SEQ controllers = 0 FOR num.local.mode.controllers
  SEQ
  force.from.mode.cont[controllers] ? CASE mode.force.t; mode; mode.force
  PAR
  exchange.data.out ! mode.force.t; mode; mode.force
  exchange.data.in ? CASE mode.force.t; MODE; MODE.FORCE
  mode.forces[ INT mode ] := mode.force
  mode.forces[ INT MODE ] := MODE.FORCE

-- And send to the magnet controllers --
SEQ controllers = 0 FOR num.local.magnet.controllers
  gaps.and.forces.to.mag.cont[ controllers ] ! magnet.gaps.t; magnet.gaps
SEQ controllers = 0 FOR num.local.magnet.controllers
  gaps.and.forces.to.mag.cont[ controllers ] ! mode.forces.t; mode.forces

-- Get current demands from the magnet controllers and exchange --
-- with the other processor --
SEQ controllers = 0 FOR num.local.magnet.controllers
  SEQ
  current.from.mag.cont[controllers] ? CASE magnet.current.t;
  magnet; magnet.current
  magnet.current.mA := INT16 ROUND (1000.0(REAL32) * magnet.current)
  IF
  head.manager
  PAR
  exchange.data.out ! magnet.current.t; magnet; magnet.current
  exchange.data.in ? CASE magnet.current.t; MAGNET; MAGNET.CURRENT
  NOT head.manager
  SEQ
  PAR
  exchange.data.out ! magnet.current.t; magnet; magnet.current
  exchange.data.in ? CASE magnet.current.t; MAGNET; MAGNET.CURRENT
  to.analogue.converter ! DAC.I.mA.t; magnet; magnet.current.mA
  MAGNET.CURRENT.mA := INT16 ROUND (1000.0(REAL32) * MAGNET.CURRENT)
  to.analogue.converter ! DAC.I.mA.t; MAGNET; MAGNET.CURRENT.mA
  magnet.currents[ INT magnet ] := magnet.current
  magnet.currents[ INT MAGNET ] := MAGNET.CURRENT

```

```

-- Get the internal signals from the magnet controllers --
SEQ controllers = 0 FOR num.local.magnet.controllers
  SEQ
    force.from.magnet.cont[controllers] ? CASE magnet.force.t;
                                         magnet; magnet.force
  PAR
    exchange.data.out ! magnet.force.t; magnet; magnet.force
    exchange.data.in ? CASE magnet.force.t; MAGNET; MAGNET.FORCE
    magnet.forces[ INT magnet ] := magnet.force
    magnet.forces[ INT MAGNET ] := MAGNET.FORCE

-- and the internal signals from the mode controllers --
SEQ controllers = 0 FOR num.local.mode.controllers
  SEQ
    data.from.mode.cont[controllers] ? CASE mode.block.t; mode; gap; acc;
                                         vel; pos; trk.pos
  PAR
    exchange.data.out ! mode.block.t; mode; gap; acc; vel; pos; trk.pos
    exchange.data.in ? CASE mode.block.t; MODE; GAP; ACC; VEL; POS; TRK.POS
    m, M := INT mode, INT MODE
    gaps[m], accs[m], vels[m], poss[m], trk.poss[m] := gap, acc, vel, pos, trk.pos
    gaps[M], accs[M], vels[M], poss[M], trk.poss[M] := GAP, ACC, VEL, POS, TRK.POS

-- And send ALL control system data signals to the data monitor's buffer --
IF head.manager
  [4][num.magnets]REAL32 magnet.data :
  PLACE magnet.data IN VECSPACE :
  [7][num.modes]REAL32 mode.data :
  PLACE mode.data IN VECSPACE :
  SEQ
    magnet.data[0] := magnet.gaps
    magnet.data[1] := magnet.accs
    magnet.data[2] := magnet.forces
    magnet.data[3] := magnet.currents
    mode.data[0] := mode.pos.refs
    mode.data[1] := gaps
    mode.data[2] := accs
    mode.data[3] := vels
    mode.data[4] := poss
    mode.data[5] := trk.poss
    mode.data[6] := mode.forces
    data.to.monitor.buffer ! time; magnet.data; mode.data
  TRUE
  SKIP
:

#USE "system.tsr"
#USE snglmath

PROC Vehicle.Mode.Controller( VAL INT mode.index,
  CHAN OF signal.p position.references.in,
  CHAN OF signal.p magnet.accs.in, magnet.gaps.in,
  CHAN OF signal.p mode.force.out,
  CHAN OF signal.p internal.signals.out,

  CHAN OF controller.p parameters.in,
  CHAN OF exception.p exception.out )

-- Calculate.Mode.Force() performs the following tasks:
-- 1. Receives the air gap and acceleration sensor signals for each magnet.
-- 2. Transforms the sensor signals to vehicle mode 'mode.index' signals.
-- 3. Estimates the mode velocity, position and track position.
-- 4. Filters the track position to form a reference track trajectory.
-- 5. Controls the absolute vehicle mode position using the above as a reference.
-- 6. Outputs the resultant vehicle mode force demand.

-- This process uses no external variables and should be configured for
-- high priority operation.

VAL pi IS 3.1415926 (REAL32) :
VAL two.pi IS 2.0 (REAL32) * pi :

VAL vel.limit IS 0.1 (REAL32) : -- 0.1 mm/ms velocity limit --
VAL [] BYTE warning.message IS "Warning: Excessive velocity, V = " :
```

```

REAL32 sample.interval :
REAL64 sample.interval.r64 :
[num.modes][num.magnets]REAL32 magnets.to.modes.transform :
[num.magnets]REAL32 mode.transform :
[num.modes]REAL32 integral.times, stiffnesses, dampings, massnesses :
REAL32 integral.time, stiffness, damping, massness :
INT control.mode :
BOOL track.ref.select :

REAL32 est.cutoff.Hz, traj.cutoff.Hz :
REAL64 est.input.factor, est.decay.factor :
REAL64 traj.input.factor, traj.decay.factor :

REAL64 mode.pos.ref :
[num.modes]REAL32 mode.pos.refs :
REAL64 track.pos.ref, pos.ref :
[num.magnets]REAL32 magnet.gaps, magnet.accs :
REAL32 gap, acc :
REAL64 acc.r64, acc.lp, acc.hp, vel.hp, pos.hp, pos.error :
REAL64 track.pos, traj.pos.f1, traj.pos.f2, track.trajectory :
REAL32 acc.feedback, vel.feedback, pos.feedback, int.feedback, pos.error.r32 :
REAL32 min.integral.force, max.integral.force :
REAL32 force.demand, acc.hp.r32, vel.hp.r32, pos.hp.r32, track.pos.r32 :

WHILE TRUE
  ALT
    parameters.in ? CASE
      mode.control.params; sample.interval; magnets.to.modes.transform;
      est.cutoff.Hz; traj.cutoff.Hz;
      integral.times; stiffnesses; dampings; massnesses;
      control.mode; track.ref.select
    SEQ
      -- Select relevant mode control parameters --
      mode.transform := magnets.to.modes.transform[ mode.index ]
      integral.time := integral.times[ mode.index ]
      stiffness:= stiffnesses[ mode.index ]
      damping := dampings[ mode.index ]
      massness := massnesses[ mode.index ]
      CASE control.mode
        local, user
          SEQ
            min.integral.force := 200.0 (REAL32)
            max.integral.force := 500.0 (REAL32)
          vehicle
            IF
              (mode.index = 0) -- heave mode
                SEQ
                  min.integral.force := 700.0 (REAL32)
                  max.integral.force := 2500.0 (REAL32)
                TRUE
                  SEQ
                    min.integral.force := -1000.0 (REAL32)
                    max.integral.force := 1000.0 (REAL32)

      -- Calculate filter coefficients --
      sample.interval.r64 := REAL64 sample.interval
      est.decay.factor := REAL64 exp((-two.pi)*(est.cutoff.Hz*sample.interval))
      est.input.factor := 1.0 (REAL64) - est.decay.factor
      traj.decay.factor:=REAL64 exp((-two.pi)*(traj.cutoff.Hz*sample.interval))
      traj.input.factor := 1.0 (REAL64) - traj.decay.factor

  initialise.system
  SEQ -- reset filters etc.
    acc.lp := 0.0 (REAL64)
    vel.hp := 0.0 (REAL64)
    pos.hp := 0.0 (REAL64)
    track.pos := 0.0 (REAL64)
    traj.pos.f1 := track.pos
    traj.pos.f2 := traj.pos.f1
    int.feedback := 0.0 (REAL32)

  position.references.in ? CASE mode.pos.refs.t; mode.pos.refs
  SEQ -- run controller
    mode.pos.ref := REAL64 mode.pos.refs[ mode.index ]
  IF
    track.ref.select
      track.pos.ref, pos.ref := mode.pos.ref, 0.0 (REAL64)
    TRUE
      track.pos.ref, pos.ref := 0.0 (REAL64), mode.pos.ref

```

```

PAR
  SEQ
    -- Input the magnet air gaps --
    magnet.gaps.in ? CASE magnet.gaps.t; magnet.gaps

    -- Transform the magnet gaps to obtain the mode gap --
    gap := 0.0 (REAL32)
    SEQ index = 0 FOR num.magnets
      gap := gap + (mode.transform[index] * magnet.gaps[index])

  SEQ
    -- Input the (high pass filtered) magnet accelerations --
    magnet.accs.in ? CASE magnet.accs.t; magnet.accs

    -- Transform the magnet accelerations to the mode acceleration --
    acc := 0.0 (REAL32)
    SEQ index = 0 FOR num.magnets
      acc := acc + (mode.transform[index] * magnet.accs[index])

    -- High-pass filter the acceleration to remove any offset --
    acc.r64 := REAL64 acc
    acc.lp := (acc.lp * est.decay.factor) + (acc.r64 * est.input.factor)
    acc.hp := acc.r64 - acc.lp
    acc.hp.r32 := REAL32 ROUND acc.hp

    -- Estimate the absolute velocity and position --
    vel.hp := (vel.hp * est.decay.factor) + (acc.hp * sample.interval.r64)
    vel.hp.r32 := REAL32 ROUND vel.hp
    pos.hp := (pos.hp * est.decay.factor) + (vel.hp * sample.interval.r64)
    pos.hp.r32 := REAL32 ROUND pos.hp

    -- Estimate the absolute track position --
    track.pos := (pos.hp + track.pos.ref) - (REAL64 gap)
    track.pos.r32 := REAL32 ROUND track.pos

    -- and filter to form the track trajectory --
    traj.pos.f1 := (traj.pos.f1*traj.decay.factor)+(track.pos*traj.input.factor)
    traj.pos.f2 := (traj.pos.f2*traj.decay.factor)+(traj.pos.f1*traj.input.factor)
    track.trajectory := traj.pos.f2

    -- Calculate the absolute position error --
    pos.error := (pos.hp - pos.ref) - track.trajectory
    pos.error.r32 := REAL32 ROUND pos.error

    -- Calculate the feedback components --
    acc.feedback := massness * acc.hp.r32
    vel.feedback := damping * vel.hp.r32
    pos.feedback := stiffness * pos.error.r32
    IF
      (int.feedback < min.integral.force)
        int.feedback := min.integral.force
      (int.feedback > max.integral.force)
        int.feedback := max.integral.force
      (integral.time > 0.0 (REAL32) )
        int.feedback := int.feedback +
          ( pos.error.r32*sample.interval ) * (stiffness/integral.time) )
    TRUE
      int.feedback := 0.0 (REAL32)

    -- Total the force requirement --
    force.demand := ((int.feedback + pos.feedback)+(vel.feedback + acc.feedback))

  PAR
    -- Output mode force demand --
    mode.force.out ! mode.force.t; BYTE mode.index; force.demand

    -- Output internal signals for monitoring --
    internal.signals.out ! mode.block.t; BYTE mode.index; gap; acc.hp.r32;
      vel.hp.r32; pos.hp.r32; track.pos.r32

    -- Check velocity signal magnitude --
    IF
      (vel.hp.r32 < (-vel.limit)) OR (vel.hp.r32 > vel.limit)
        exception.out ! SIZE warning.message::warning.message;
          BYTE mode.index; vel.hp.r32; TRUE
    TRUE
      SKIP

```

```

#USE "system.tsr"
#USE  snlmath

PROC Magnet.Force.Controller( VAL INT  magnet.index,
    CHAN OF signal.p  gaps.and.forces.in,
    CHAN OF signal.p  current.out,
    CHAN OF signal.p  magnet.force.out,

    CHAN OF controller.p  parameters.in,
    CHAN OF exception.p  exception.out )

-- Calculate.magnet.current uses the nonlinear magnet force model to determine
-- the required current for the demanded force at the present airgap.

-- Worldly constants --
VAL pi IS 3.1415926 (REAL32) :
VAL mu.air IS 1.2566E-6 (REAL32) : -- Permeability of free space
-- These are used to make the equations more readable --
VAL zero IS 0.0 (REAL32) :
VAL one IS 1.0 (REAL32) :
VAL two IS 2.0 (REAL32) :
VAL four IS 4.0 (REAL32) :

-- Electromagnet parameters --
VAL y IS 0.0015 (REAL32) : -- Lateral offset between magnet & track poles
VAL l IS 0.200 (REAL32) : -- Magnet length
VAL w IS 0.033 (REAL32) : -- Width between the magnet pole pieces
VAL h IS 0.063 (REAL32) : -- Height of the magnet pole pieces
VAL t IS 0.030 (REAL32) : -- Width between the track pole pieces
VAL p IS 0.0095 (REAL32) : -- Width of the pole pieces

VAL N.coils IS 274.0 (REAL32) : -- Number of coil turns
VAL min.current IS 0.0 (REAL32) : -- Impose current limit
VAL max.current IS 20.0 (REAL32) : -- Impose current limit
VAL mu.iron IS 2000.0 (REAL32) : -- Maximum permeability of core material
VAL flux.roll.start IS 1.0 (REAL32) : -- Start of permeability roll-off (/Tesla)
VAL flux.roll.stop IS 1.7 (REAL32) : -- End of permeability roll-off (/Tesla)

VAL T.eddy IS 0.0015 (REAL32) : -- Target eddy current time constant
VAL N.phase.lead IS 3.0 (REAL32) : -- Phase-lead ratio to achieve the above

-- Input and transformation variables --
REAL32 sample.interval, mag.force :
[num.magnets][num.modes]REAL32 modes.to.magnets.matrix :
[num.modes]REAL32 magnet.transform, mode.forces :
[num.magnets]REAL32 magnet.gaps :

-- Nonlinear magnet model variables --
REAL32 T.phase.lead, input.factor, decay.factor :
REAL32 flux.LP, flux.comp :
REAL32 R.track, R.leakage, R.airgap, R.magnet :
REAL32 force.demand, airgap, force.per.pole :
REAL32 airgap.flux, flux.squared, leakage.flux, magnet.flux :
REAL32 flux.density, mu.magnet :
REAL32 airgaps.plus.track.mmf, magnet.mmf :
REAL32 current.demand :

-- Calculate the track and leakage reluctances --
VAL R.track IS ( t + (four*p) ) / ( (mu.iron*mu.air) * (l*p) ) :
VAL R.leakage IS w / ( mu.air * ( (1+((two*w)/pi)) * ((h/two)+(w/pi)) ) ) :

WHILE TRUE
  ALT
    parameters.in ? CASE
      magnet.control.params; sample.interval; modes.to.magnets.matrix
      SEQ
        -- Select relevant magnet control parameters --
        magnet.transform := modes.to.magnets.matrix[ magnet.index ]

        -- Calculate phase-lead compensator coefficients --
        IF
          (T.eddy > sample.interval)
            T.phase.lead := T.eddy
          TRUE
            T.phase.lead := sample.interval

        decay.factor := exp(- (sample.interval / T.phase.lead))
        input.factor := one - decay.factor

```

```

initialise.system
    flux.LP := zero -- Reset phase-lead compensator

gaps.and.forces.in ? CASE magnet.gaps.t; magnet.gaps
SEQ
    -- Calculate airgap reluctance --
    airgap := magnet.gaps[ magnet.index ]
    R.airgap := airgap / ( (mu.air*1) * ( p + ((two/pi)*airgap) ) )

    gaps.and.forces.in ? CASE mode.forces.t; mode.forces
    mag.force := 0.0 (REAL32)
    SEQ index = 0 FOR num.modes
        mag.force := mag.force + (magnet.transform[index] * mode.forces[index])

    IF
        (force.demand < zero)
            force.demand := zero
        TRUE
            force.demand := mag.force

    -- Calculate the gross force per pole --
    force.per.pole := force.demand /
        ( two * ( one - ( (y*y)/airgap) / (airgap+((pi*p)/two)) ) )

    -- Calculate the required airgap flux --
    flux.squared := force.per.pole * ( two*airgap) / R.airgap )
    IF
        (flux.squared > zero)
            airgap.flux := SQRT( flux.squared )
        TRUE
            airgap.flux := zero

    -- Calculate the mmf required to overcome the airgap and track reluctances
    airgaps.plus.track.mmf := airgap.flux * ( two*R.airgap) + R.track )

    -- Calculate the consequential leakage flux --
    leakage.flux := airgaps.plus.track.mmf / R.leakage

    -- Calculate the resulting electromagnet flux --
    magnet.flux := airgap.flux + leakage.flux

    -- Apply phase lead compensation to the magnet flux demand --
    flux.LP := (flux.LP * decay.factor) + (magnet.flux * input.factor)
    flux.comp := (N.phase.lead * magnet.flux) - ((N.phase.lead-one) * flux.LP)

    -- Calculate the magnet reluctance at this flux level --
    flux.density := magnet.flux / (1 * p)
    IF
        (flux.density < flux.roll.start)
            mu.magnet := mu.iron
        (flux.density < flux.roll.stop)
            mu.magnet := mu.iron * (1.1(REAL32) - ( (flux.density-flux.roll.start)/
                (flux.roll.stop - flux.roll.start) ) )
        TRUE
            mu.magnet := mu.iron * 0.1(REAL32)
    R.magnet := ( two*h) + (w+(two*p)) ) / ( (mu.air*mu.magnet) * (1*p) )

    -- Calculate the mmf required to overcome the magnet reluctance --
    magnet.mmf := flux.comp * R.magnet

    -- Finally, calculate the current required to produce the total mmf --
    current.demand := (airgaps.plus.track.mmf + magnet.mmf) / N.coils
    IF
        (current.demand > max.current)
            current.demand := max.current
        (current.demand < min.current)
            current.demand := min.current
        TRUE
            SKIP

    -- Output magnet current demand (and force demand for monitoring) --
    PAR
        current.out ! magnet.current.t; BYTE magnet.index; current.demand
        magnet.force.out ! magnet.force.t; BYTE magnet.index; mag.force

```



```

#USE "system.tsr"

PROC Command.Message.Manager( VAL BOOL head.manager,
    CHAN OF controller.p commands.in,
    []CHAN OF controller.p magnet.controller.commands,
    []CHAN OF controller.p mode.controller.commands,
    CHAN OF controller.p commands.out,
    CHAN OF BOOL exception.manager.reset )

VAL num.local.magnet.controllers IS SIZE magnet.controller.commands :
VAL num.local.mode.controllers IS SIZE mode.controller.commands :

REAL32 sample.time, est.freq, traj.freq, mass :
[num.modes][num.magnets]REAL32 input.matrix :
[num.magnets][num.modes]REAL32 output.matrix :
[num.modes]REAL32 integral.T, stiffness, damping, massness :
INT control.mode :
BOOL ref.sig.injection.select :

WHILE TRUE
    commands.in ? CASE

        magnet.control.params; sample.time; output.matrix
        SEQ
        IF
            head.manager
                commands.out ! magnet.control.params;
                    sample.time; output.matrix
        TRUE
        SKIP

        SEQ channels = 0 FOR num.local.magnet.controllers
            magnet.controller.commands[ channels ] ! magnet.control.params;
                sample.time; output.matrix

        mode.control.params; sample.time; input.matrix; est.freq; traj.freq;
        integral.T; stiffness; damping; massness;
        control.mode; ref.sig.injection.select
        SEQ
        IF
            head.manager
                commands.out ! mode.control.params;
                    sample.time; input.matrix; est.freq; traj.freq;
                    integral.T; stiffness; damping; massness;
                    control.mode; ref.sig.injection.select
        TRUE
        SKIP

        SEQ channels = 0 FOR num.local.mode.controllers
            mode.controller.commands[ channels ] ! mode.control.params;
                sample.time; input.matrix; est.freq; traj.freq;
                integral.T; stiffness; damping; massness;
                control.mode; ref.sig.injection.select

        initialise.system
        SEQ
        IF
            head.manager
                commands.out ! initialise.system
        TRUE
        SKIP

        SEQ channels = 0 FOR num.local.magnet.controllers
            magnet.controller.commands[ channels ] ! initialise.system

        SEQ channels = 0 FOR num.local.mode.controllers
            mode.controller.commands[ channels ] ! initialise.system

        exception.manager.reset ! TRUE
    :

```

```

#USE "system.tsr"

PROC Exception.Report.Manager( VAL BOOL head.manager,
                               CHAN OF BOOL manager.reset,
                               CHAN OF exception.p exception.in,
                               []CHAN OF exception.p exceptions.in,
                               CHAN OF BOOL switch.pressed.in,
                               CHAN OF exception.p exception.out,
                               CHAN OF BOOL shutdown.request)

VAL num.exception.channels IS SIZE exceptions.in :

BOOL reset, send.report, report.received :

INT message.length, b.message.length :
[80]BYTE message :
PLACE message IN VECSPACE :
[80]BYTE b.message :
PLACE b.message IN VECSPACE :
BYTE index, b.index :
REAL32 value, b.value :
BOOL delevitate, b.delevitate :
BOOL switch.pressed :

CHAN OF exception.p exception.buffer :

SEQ
  report.received := FALSE
  send.report := TRUE
  PAR
    WHILE TRUE
      SEQ
        PRI ALT
          ALT channels = 0 FOR num.exception.channels
            exceptions.in[ channels ] ? message.length::message; index; value; delevitate
            report.received := TRUE
          exception.in ? message.length::message; index; value; delevitate
            report.received := TRUE
          switch.pressed.in ? delevitate
            SEQ
              message.length := 0
              report.received := TRUE
            manager.reset ? reset
            SEQ
              report.received := FALSE
              send.report := TRUE

          switch.pressed := report.received AND (message.length = 0)

        IF
          switch.pressed AND send.report
            SEQ
              exception.buffer ! message.length::message; index; value; delevitate
              report.received := FALSE
            report.received AND send.report
            SEQ
              exception.buffer ! message.length::message; index; value; delevitate
              send.report := FALSE
            TRUE
            SKIP

    WHILE TRUE
      SEQ
        exception.buffer ? b.message.length::b.message; b.index; b.value; b.delevitate
        IF
          head.manager AND b.delevitate
            shutdown.request ! TRUE
          TRUE
            SKIP
        exception.out ! b.message.length::b.message; b.index; b.value; b.delevitate

```

## E.2.5 Executive processes Library

```

#USE "system.tsr"

PROC Monitor.Data.Buffer( CHAN OF monitor.p monitor.data.in,
                        CHAN OF BOOL data.request.in,
                        CHAN OF monitor.p monitor.data.out,

                        CHAN OF data.buffer.p commands.and.params.in,
                        CHAN OF exception.p exception.report.out )

-- Monitor.Data.Buffer() receives all data signals from the control system for
-- each iteration of the loop. This data can be optionally stored at a
-- specified sampling rate and duration. The output data can then be either
-- from the on-line data or from that stored in the buffer.
--
-- This process uses no external variables and should be configured for
-- high priority operation.

BOOL logging, log.pending, data.complete, use.stored.data, request.data :

REAL32 time.stamp, sample.time, sample.interval, sample.duration :
[4][num.magnets]REAL32 magnet.data :
[7][num.modes]REAL32 mode.data :

VAL buffer.size IS 2000 :
INT write.index, read.index :

-- Place the large buffer arrays in slow off-processor memory --
[buffer.size]REAL32 buffered.time.stamp :
PLACE buffered.time.stamp IN VECSPACE :
[buffer.size][4][num.magnets]REAL32 buffered.magnet.data :
PLACE buffered.magnet.data IN VECSPACE :
[buffer.size][7][num.modes]REAL32 buffered.mode.data :
PLACE buffered.mode.data IN VECSPACE :

SEQ
  logging := FALSE
  log.pending := FALSE
  data.complete := FALSE

  -- Reset variable storage
  time.stamp := 0.0 (REAL32)
  SEQ var.index = 0 FOR 4
    SEQ magnet.index = 0 FOR num.magnets
      magnet.data[var.index][magnet.index] := 0.0 (REAL32)
  SEQ var.index = 0 FOR 7
    SEQ mode.index = 0 FOR num.modes
      mode.data[var.index][mode.index] := 0.0 (REAL32)

  WHILE TRUE
    PRI ALT
      monitor.data.in ? time.stamp; magnet.data; mode.data
      SEQ
        -- Log data into buffer when appropriate --
        logging := logging OR (log.pending AND (time.stamp = 0.0 (REAL32)))
        log.pending := log.pending AND (NOT logging)
        IF
          logging AND (time.stamp >= sample.time)
          SEQ
            write.index := write.index + 1
            buffered.time.stamp[write.index] := time.stamp
            buffered.mode.data[write.index] := mode.data
            buffered.magnet.data[write.index] := magnet.data
            data.complete := (write.index = (buffer.size-1)) OR
              (sample.time >= sample.duration)
            sample.time := sample.time + sample.interval
          logging
            data.complete := (time.stamp = 0.0 (REAL32))
          TRUE
            SKIP
        logging := logging AND (NOT data.complete)

      commands.and.params.in ? CASE
        data.buffer.select; use.stored.data -- Select output source
        SKIP
        data.buffer.params; sample.interval; sample.duration
        SKIP

```

```

data.buffer.log -- Start storing data to buffer
SEQ
  logging := FALSE
  log.pending := TRUE
  data.complete := FALSE
  sample.time := 0.0 (REAL32)
  write.index := -1
  read.index := -1

data.request.in ? request.data
IF
  use.stored.data AND (logging OR data.complete)
  SEQ
    read.index := (read.index + 1) REM (write.index + 1)
    monitor.data.out ! buffered.time.stamp[read.index];
                    buffered.magnet.data[read.index];
                    buffered.mode.data[read.index]
  TRUE
  monitor.data.out ! time.stamp; magnet.data; mode.data

:

#USE "system.tsr"
#USE  userio

PROC User.Interface( CHAN OF signal.gen.p  signal.gen.params.out, signal.gen.status.in,
                    CHAN OF controller.p  controller.params.out,
                    CHAN OF data.buffer.p  data.buffer.params.out,

                    CHAN OF INT  keyboard,
                    CHAN OF ANY  screen,

                    CHAN OF BOOL  data.request,
                    CHAN OF monitor.p  monitor.data.in,

                    CHAN OF exception.p  exception.reports.in )

-- User.Interface() provides the overall executive control of the suspension system.
-- A terminal can be connected (optically) to the vehicle to access a menu-driven
-- facility which permits levitation control as well as as modification of all
-- control system parameters. User.Interface() also provides data monitoring
-- facilities either asynchronously for fast sampling or synchronously to provide
-- real-time monitoring. The push button switch on the DAC card can also be used
-- to levitate and de-levitate the vehicle.

-- MENU MANIPULATION PROCEDURES --
PROC Menu.Title( VAL [] BYTE title )
-- Menu.Title() centres, underlines and displays the supplied text.
VAL offset IS (80 - (SIZE title)) / 2 :
SEQ
  goto.xy( screen, 0,0 )
  clear.eos( screen )
  goto.xy( screen, offset,0 )
  write.text.line( screen, title )
  goto.xy( screen, offset,1 )
  SEQ index = 1 FOR SIZE title
    write.char( screen, '~' )
  SEQ index = 1 FOR 3
    newline( screen )
:

PROC Menu.Line( VAL []BYTE option )
-- Menu.Line() offsets and displays the supplied text.
SEQ
  SEQ index = 1 FOR 20
    write.char( screen, ' ' )
  write.text.line( screen, option )
:

PROC Menu.Line.b( VAL []BYTE option, true.msg, false.msg, VAL BOOL value )
-- Menu.Line.b() offsets and displays the supplied option text.
-- It also uses the supplied Boolean value to selectively display a second
-- text message.
SEQ
  SEQ index = 1 FOR 20
    write.char( screen, ' ' )
  write.full.string( screen, option )
  IF
    (value = TRUE)

```

```

        write.text.line( screen, true.mesg )
        (value = FALSE)
        write.text.line( screen, false.mesg )
    :

PROC Menu.Line.r( VAL []BYTE option, units, VAL REAL32 value, VAL INT i,d )
-- Menu.Line.r() offsets and displays the supplied option text.
-- It also displays the supplied REAL32 value followed by the units text.
SEQ
SEQ index = 1 FOR 20
    write.char( screen, ' ' )
    write.full.string( screen, option )
    write.real32( screen, value, i,d )
    write.text.line( screen, units )
:

PROC Menu.Line.ra( VAL []BYTE option, units, VAL []REAL32 value,
                  VAL REAL32 divisor, VAL INT i,d )
-- Menu.Line.ra() offsets and displays the supplied option text.
-- It also displays the supplied []REAL32 values followed by the units text.
SEQ
SEQ index = 1 FOR 20
    write.char( screen, ' ' )
    write.full.string( screen, option )
    SEQ index = 0 FOR SIZE value
        SEQ
            write.real32( screen, value[index]/divisor, i,d )
            write.char( screen, ' ' )
        write.text.line( screen, units )
:

PROC Get.Option( INT char )
-- Get.Option() prompts the user for a single character input.
SEQ
write.full.string( screen, "*c*n          Select option: " )
keyboard ? char
char := (char /\ #DF)
:

PROC Get.Value.r( VAL []BYTE prompt, REAL32 new.value, VAL REAL32 factor )
-- Get.Value.r() prompts the user with the supplied prompt and then
-- inputs and scales returns a REAL32 value from the keyboard.
INT delimiter :
SEQ
write.text.line( screen, prompt )
write.full.string( screen, "          Enter new parameter value: " )
new.value := 0.0 (REAL32)
delimiter := INT ' '
read.echo.real32( keyboard, screen, new.value, delimiter )
new.value := new.value * factor
:

PROC Get.Value.ra( VAL []BYTE prompt, []REAL32 new.value, VAL REAL32 factor )
-- Get.Value.ra() prompts the user with the supplied prompt and then
-- inputs and scales returns []REAL32 values from the keyboard.
INT delimiter :
SEQ
write.text.line( screen, prompt )
write.full.string( screen, "          Enter new parameter values: " )
SEQ index = 0 FOR SIZE new.value
    SEQ
        new.value[index] := 0.0 (REAL32)
        delimiter := INT ' '
        read.echo.real32( keyboard, screen, new.value[index], delimiter )
        new.value[index] := new.value[index] * factor
:

-- VARIABLE DECLARATIONS TO HOLD COMPLETE SET OF SYSTEM PARAMETERS --

-- Reference signal generator variables --
REAL32 sample.interval, mean.gap :
BOOL    dynamic.ref.signal, sine.wave.ref.signal :
REAL32 ref.frequency, ref.amplitude :
[num.modes]REAL32 mean.gap.coeffs :
[num.modes]REAL32 pos.ref.coeffs :

-- Controller parameters --
INT control.mode :

[num.modes][num.magnets]REAL32 input.matrix :
```

```

[num.magnets][num.modes]REAL32 output.matrix :
[num.modes]REAL32 integral.T, stiffness, damping, massness :
REAL32 est.cutoff.Hz, traj.cutoff.Hz :
BOOL track.reference :

-- Data signal buffer parameters --
REAL32 buffer.interval, buffer.duration :
BOOL buffer.output :

#USE "Defaults.tsr"

PROC Initialise.Reference.Signal.Generator.Parameters ( )
-- This procedure initialises all parameters for the ref signal generator.
SEQ
sample.interval := default.sample.interval
mean.gap := default.mean.gap
dynamic.ref.signal := default.dynamic.ref.signal
sine.wave.ref.signal := default.sine.wave.ref.signal
track.reference := default.track.reference
IF
track.reference
ref.frequency := default.track.ref.frequency
NOT track.reference
ref.frequency := default.pos.ref.frequency
ref.amplitude := default.ref.amplitude
CASE control.mode
local
pos.ref.coeffs := default.local.ref.coeffs
vehicle
pos.ref.coeffs := default.vehicle.ref.coeffs
user
pos.ref.coeffs := default.user.ref.coeffs
:

PROC Reference.Signal.Generator.Menu ( )
-- This procedure enables interactive modification of the reference signal
-- generator parameters.
BOOL finished :
INT option :
SEQ
finished := FALSE
WHILE NOT finished
SEQ
Menu.Title( "MAGLEV VEHICLE SUSPENSION - Signal Generator Parameters" )

Menu.Line( "Reset all parameters to default values" )
newline( screen )
Menu.Line.r( "Sampling interval ( ", " ms )", sample.interval*kilo,3,2 )
Menu.Line.r( "Operating air gap ( ", " mm )", mean.gap*kilo,1,2 )
Menu.Line.b( "Dynamic reference signal ( ", "On )", "Off )", dynamic.ref.signal )
Menu.Line.b( "Waveform of dynamic reference signal ( ", "Sine )", "Square )",
sine.wave.ref.signal )
Menu.Line.b( "Track/Vehicle reference signal ( ", "Track )", "Vehicle )", track.reference
)

Menu.Line.r( "Frequency of reference signal ( ", " Hz )", ref.frequency,2,2 )
Menu.Line.r( "Amplitude of reference signal ( ", " mm )", ref.amplitude*kilo,2,2 )
Menu.Line.ra( "Mode distribution of reference ( ", " )", pos.ref.coeffs,one,1,1 )
newline( screen )
Menu.Line( "Quit this menu." )

Get.Option( option )
CASE BYTE option
'R'
Initialise.Reference.Signal.Generator.Parameters ( )
'S'
Get.Value.r( "Sampling interval", sample.interval, one/kilo )
'O'
Get.Value.r( "Air gap", mean.gap, one/kilo )
'D'
dynamic.ref.signal := NOT dynamic.ref.signal
'W'
sine.wave.ref.signal := NOT sine.wave.ref.signal
'T'
track.reference := NOT track.reference
'F'
Get.Value.r( "Reference frequency", ref.frequency, one )
'A'
Get.Value.r( "Reference amplitude", ref.amplitude, one/kilo )
'M'

```

```

        Get.Value.ra( "Factors for each mode reference", pos.ref.coeffs, one)
    'Q'
        finished := TRUE
    ELSE
        beep( screen )
    IF
        (sample.interval < 0.0001 (REAL32))
            sample.interval := 0.0001 (REAL32)
        (sample.interval > 0.99 (REAL32))
            sample.interval := 0.99 (REAL32)
    TRUE
    SKIP
:

PROC Initialise.Controller.Parameters( )
-- This procedure initialises all parameters for the mode and magnet controllers.
SEQ
CASE control.mode
    local
        SEQ
            input.matrix := local.matrix
            output.matrix := local.matrix
        vehicle
            SEQ
                input.matrix := veh.input.matrix
                output.matrix := veh.output.matrix
            user
                SEQ
                    input.matrix := user.input.matrix
                    output.matrix := user.output.matrix
        est.cutoff.Hz := default.est.cutoff.Hz
        traj.cutoff.Hz := default.traj.cutoff.Hz
CASE control.mode
    local
        SEQ
            integral.T := local.integral.T
            stiffness := local.stiffness
            damping := local.damping
            massness := local.massness
        vehicle
            SEQ
                integral.T := veh.integral.T
                stiffness := veh.stiffness
                damping := veh.damping
                massness := veh.massness
        user
            SEQ
                integral.T := user.integral.T
                stiffness := user.stiffness
                damping := user.damping
                massness := user.massness
-- Also set the mode reference coefficients for the ref signal generator --
CASE control.mode
    local
        pos.ref.coeffs := default.local.ref.coeffs
    vehicle
        pos.ref.coeffs := default.vehicle.ref.coeffs
    user
        pos.ref.coeffs := default.user.ref.coeffs
CASE control.mode
    local
        mean.gap.coeffs := default.local.gap.coeffs
    vehicle
        mean.gap.coeffs := default.vehicle.gap.coeffs
    user
        mean.gap.coeffs := default.user.gap.coeffs
:

PROC Controller.Menu( )
-- This procedure enables interactive modification of the mode and magnet
-- controller parameters.
BOOL finished :
INT option :
SEQ
    finished := FALSE
    WHILE NOT finished
        SEQ
            Menu.Title( "MAGLEV VEHICLE SUSPENSION - Controller Parameters" )

```

```

Menu.Line.b( "Local mode controller ", "- Selected", "", (control.mode=local) )
Menu.Line.b( "Vehicle mode controller ", "- Selected", "", (control.mode=vehicle) )
Menu.Line.b( "User defined controller ", "- Selected", "", (control.mode=user) )
newline( screen )
Menu.Line.r( "Estimation filter frequency ( ", " Hz )", est.cutoff.Hz,1,2)
Menu.Line.r( "Track trajectory filter frequency ( ", " Hz )", traj.cutoff.Hz,2,1 )
newline( screen )
Menu.Line( "Absolute Position Controller Parameters" )
Menu.Line.ra( "Integral time constant ( ", " s )", integral.T,one,1,1 )
Menu.Line.ra( "Stiffness ( ", " kg/mm )", stiffness,kilo.g,3,1 )
Menu.Line.ra( "Damping ( ", " kg/(mm/s) )", damping,kilo.g,3,1 )
Menu.Line.ra( "Massness ( ", " kg )", massness,one,3,1 )
newline( screen )
Menu.Line( "Quit this menu." )

Get.Option( option )
CASE BYTE option
  'L','V','U'
  SEQ
    CASE BYTE option
      'L'
        control.mode := local
      'V'
        control.mode := vehicle
      'U'
        control.mode := user
    Initialise.Controller.Parameters( )
  'E'
    Get.Value.r( "Estimation filter frequency", est.cutoff.Hz, one)
  'T'
    Get.Value.r( "Trajectory filter frequency", traj.cutoff.Hz, one )
  'I'
    Get.Value.ra( "Integral action time constant", integral.T, one )
  'S'
    Get.Value.ra( "Stiffness", stiffness, kilo.g )
  'D'
    Get.Value.ra( "Damping", damping, kilo.g )
  'M'
    Get.Value.ra( "Massness", massness, one )
  'Q'
    finished := TRUE
  ELSE
    beep( screen )
:

PROC Initialise.Signal.Buffer.Parameters( )
-- This procedure initialises all parameters for the signal data buffer.
SEQ
  buffer.interval := default.buffer.interval
  buffer.duration := default.buffer.duration
  buffer.output := default.buffer.output
:

PROC Signal.Buffer.Menu( )
-- This procedure enables interactive modification of the parameters for the
-- signal data buffer.
BOOL finished :
INT option :
SEQ
  finished := FALSE
  WHILE NOT finished
    SEQ
      Menu.Title( "MAGLEV VEHICLE SUSPENSION - Signal Buffer Parameters" )

      Menu.Line( "Reset all parameters to default values" )
      newline( screen )
      Menu.Line.r( "Interval between buffer samples( ", " ms )", buffer.interval*kilo,2,1 )
      Menu.Line.r( "Duration (maximum) of buffer sampling( ", " s )", buffer.duration,2,1)
      newline( screen )
      Menu.Line( "Log signals into buffer now" )
      newline( screen )
      Menu.Line.b( "Select buffer output source ", "- stored data",
                  "- real-time data", buffer.output )

      newline( screen )
      Menu.Line( "Quit this menu." )

      Get.Option( option )
      CASE BYTE option
        'R'

```



```

        Initialise.Signal.Buffer.Parameters ( )
    'I'
        Get.Value.r( "Buffer interval", buffer.interval, one/kilo )
    'D'
        Get.Value.r( "Buffer duration", buffer.duration, one )
    'L'
        SEQ
            data.buffer.params.out ! data.buffer.params;  buffer.interval;
                                     buffer.duration
            data.buffer.params.out ! data.buffer.log
    'S'
        buffer.output := NOT buffer.output
    'Q'
        finished := TRUE
    ELSE
        beep( screen )
:

-- LOCAL VARIABLE DECLARATIONS FOR USER.INTERFACE() --

-- Menu control variables --
BOOL levitate, key.pressed :
INT option :

-- Exception report variables --
INT message.length :
[80]BYTE message :
BYTE index :
REAL32 value :
BOOL delevitate, switch.pressed :

-- Procedure code for User.Interface() --

#USE "Monitor.tsr"
SEQ
    control.mode := vehicle -- Select multi-variable vehicle control mode.

    Initialise.Signal.Buffer.Parameters( )
    Initialise.Controller.Parameters( )
    Initialise.Reference.Signal.Generator.Parameters( )

    controller.params.out ! initialise.system

    levitate := FALSE
    key.pressed := FALSE

    WHILE TRUE
        SEQ
            controller.params.out ! mode.control.params;  sample.interval; input.matrix;
                                     est.cutoff.Hz; traj.cutoff.Hz; integral.T; stiffness;
                                     damping; massness; control.mode; track.reference
            controller.params.out ! magnet.control.params;  sample.interval; output.matrix
            data.buffer.params.out ! data.buffer.params;  buffer.interval; buffer.duration
            data.buffer.params.out ! data.buffer.select;  buffer.output
            signal.gen.params.out ! signal.gen.params;  sample.interval; mean.gap;
                                     mean.gap.coeffs; dynamic.ref.signal; sine.wave.ref.signal;
                                     ref.frequency; ref.amplitude; pos.ref.coeffs

        PRI ALT
            keyboard ? option
            SEQ
                key.pressed := TRUE
                option := (option /\ #DF)
                CASE BYTE option
                    'L'
                        IF
                            NOT levitate
                                SEQ
                                    levitate := TRUE
                                    controller.params.out ! initialise.system
                                    signal.gen.params.out ! signal.gen.levitate;  levitate
                                TRUE
                            SKIP
                    'D'
                        SEQ
                            levitate := FALSE
                            signal.gen.params.out ! signal.gen.levitate;  levitate
                    'C'
                        Controller.Menu( )
                    'R'

```

```

        Reference.Signal.Generator.Menu( )
'S'
    Signal.Buffer.Menu( )
'M'
    Data.Monitor( data.request, monitor.data.in, keyboard, screen,
data.buffer.params.out )
    ELSE
        beep( screen )

Menu.Title( "MAGLEV VEHICLE SUSPENSION CONTROL SYSTEM" )
Menu.Line.b("", "De-levitate vehicle.", "Levitate vehicle.", levitate )
Menu.Line( "Controller parameters." )
Menu.Line( "Reference signal parameters." )
Menu.Line( "Signal buffer parameters." )
Menu.Line( "Monitor controller signals." )
write.full.string( screen, "*c*n          Select option: " )

signal.gen.status.in ? CASE signal.gen.levitated; levitate
controller.params.out ! initialise.system

exception.reports.in ? message.length::message; index; value; delevitate
SEQ
switch.pressed := (message.length = 0)
IF
    switch.pressed
    SEQ
        levitate := NOT levitate
        signal.gen.params.out ! signal.gen.levitate; levitate
    NOT switch.pressed
    SEQ
        IF
            NOT key.pressed
            keyboard ? option
            TRUE
                newline( screen )
            newline( screen )
            write.len.string( screen, message.length, message )
            write.real32( screen, value, 3,3 )
            newline( screen )
            newline( screen )
            write.full.string( screen, "*c*n          Select option: " )

```

:

## E.2.6 Default parameters Library for Executive

```

-- DEFAULT VALUES FOR ALL CONTROL SYSTEM PARAMETERS --

-- Useful constants (to improve readability) --
VAL zero IS 0.0 (REAL32) :
VAL qrt IS 0.25 (REAL32) :
VAL half IS 0.5 (REAL32) :
VAL one IS 1.0 (REAL32) :
VAL kilo IS 1000.0 (REAL32) :
VAL g IS 9.81 (REAL32) :
VAL kilo.g IS kilo * g :

-- Default parameter values for the Reference Signal Generator --
VAL default.sample.interval IS 0.00125 (REAL32) : -- 1.25 ms
VAL default.mean.gap IS 0.003 (REAL32) : -- 3.0 mm
VAL default.local.gap.coeffs IS [ one, one, one, one ] :
VAL default.user.gap.coeffs IS [ one, one, one, zero ] :
VAL default.vehicle.gap.coeffs IS [ one, zero, zero, zero ] :

VAL default.dynamic.ref.signal IS FALSE : -- constant gap ref
VAL default.sine.wave.ref.signal IS TRUE :
VAL default.track.reference IS TRUE :

VAL default.pos.ref.frequency IS 10.0 (REAL32) : -- 10 Hz
VAL default.track.ref.frequency IS 0.5 (REAL32) : -- 0.5 Hz
VAL default.ref.amplitude IS 0.001 (REAL32) : -- 1 mm pk-pk

VAL default.local.ref.coeffs IS [ one, one, one, one ] :
VAL default.user.ref.coeffs IS [ one, one, one, zero ] :
VAL default.vehicle.ref.coeffs IS [ one, zero, zero, zero ] :

-- Default parameter values for the Mode & Magnet Controllers --
VAL local.matrix IS [ [ one, zero, zero, zero ],
                    [ zero, one, zero, zero ],
                    [ zero, zero, one, zero ],
                    [ zero, zero, zero, one ] ] :

VAL veh.input.matrix IS [ [ qrt, qrt, qrt, qrt ],
                        [ qrt, -qrt, -qrt, qrt ],
                        [ qrt, qrt, -qrt, -qrt ],
                        [ qrt, -qrt, qrt, -qrt ] ] :

VAL veh.output.matrix IS [ [ qrt, qrt, qrt, qrt ],
                          [ qrt, -qrt, qrt, -qrt ],
                          [ qrt, -qrt, -qrt, qrt ],
                          [ qrt, qrt, -qrt, -qrt ] ] :

VAL user.input.matrix IS [ [ one, zero, zero, zero ],
                          [ zero, one, zero, zero ],
                          [ zero, zero, half, half ],
                          [ zero, zero, zero, zero ] ] :

VAL user.output.matrix IS [ [ one, zero, zero, zero ],
                           [ zero, one, zero, zero ],
                           [ zero, zero, half, zero ],
                           [ zero, zero, half, zero ] ] :

VAL REAL32 default.est.cutoff.Hz IS 0.1 (REAL32) :
VAL REAL32 default.traj.cutoff.Hz IS 4.0 (REAL32) :

VAL veh.integral.T IS [ one, one, one, zero ] : -- seconds
VAL veh.stiffness IS
  [ 100.0E+4 (REAL32), 100.0E+4 (REAL32), 100.0E+4 (REAL32), zero ] : -- N/m
VAL veh.damping IS
  [ 2.6E+4 (REAL32), 2.6E+4 (REAL32), 2.6E+4 (REAL32), 2.6E+4 (REAL32) ] : -- N/m/s
VAL veh.massness IS
  [ 60.0 (REAL32), 60.0 (REAL32), 60.0 (REAL32), 60.0 (REAL32) ] : -- kg

VAL local.integral.T IS [ one, one, one, one ] : -- seconds
VAL local.stiffness IS
  [ 25.0E+4 (REAL32), 25.0E+4 (REAL32), 25.0E+4 (REAL32), 25.0E+4 (REAL32) ] : -- N/m
VAL local.damping IS
  [ 0.65E+4 (REAL32), 0.65E+4 (REAL32), 0.65E+4 (REAL32), 0.65E+4 (REAL32) ] : -- N/m/s
VAL local.massness IS
  [ 15.0 (REAL32), 15.0 (REAL32), 15.0 (REAL32), 15.0 (REAL32) ] : -- kg

```

```

VAL user.integral.T IS [ one, one, one, zero ] : -- seconds
VAL user.stiffness IS
  [ 25.0E+4 (REAL32), 25.0E+4 (REAL32), 50.0E+4 (REAL32), zero ] : -- N/m
VAL user.damping IS
  [ 0.65E+4 (REAL32), 0.65E+4 (REAL32), 1.3E+4 (REAL32), zero ] : -- N/m/s
VAL user.massness IS
  [ 15.0 (REAL32), 15.0 (REAL32), 30.0 (REAL32), zero ] : -- kg

-- Default parameter values for the Signal Buffer --
VAL default.buffer.interval IS 0.005 (REAL32) : -- 5 ms
VAL default.buffer.duration IS 99.000 (REAL32) : -- 99 s
VAL default.buffer.output IS FALSE : -- Select on-line signals

```

## E.2.7 Data Monitor process Library for Executive

```

#USE "system.tsr"
#USE userio
#USE strings

PROC Data.Monitor( CHAN OF BOOL data.request.out,
                  CHAN OF monitor.p monitored.data.in,

                  CHAN OF INT keyboard,
                  CHAN OF ANY screen,

                  CHAN OF data.buffer.p data.buffer.params.out )

-- Data.Monitor receives data from the control system via the data buffer.
-- Data selected by the keyboard is the sent to the screen. Each data item
-- is toggled on/off using its initial letter. 'B' toggles bar graph/tabular
-- output, and 'W' causes the output to wait briefly to allow the Network
-- Monitor EXE program to break free from the data stream, thus allowing the
-- Data Logger EXE program to be run to log data to disc.

-- Display variable tables etc. --
VAL num.vars IS 12 :

VAL one IS 1.0 (REAL32) :
VAL ten IS 10.0 (REAL32) :
VAL twenty IS 20.0 (REAL32) :
VAL kilo IS 1000.0 (REAL32) :

VAL [num.vars][2]BYTE var.name IS
  [ "T=", "G=", "A=", "F=", "I=", "r=", "g=", "a=", "v=", "p=", "t=", "f=" ] :
VAL [num.vars]REAL32 var.scaler IS
  [ one, kilo, one, one, kilo, kilo, one, one, kilo, kilo, one ] :
VAL [num.vars]REAL32 var.FSD IS
  [ one, ten, one, kilo, twenty, ten, ten, one, one, ten, ten, kilo ] :
VAL [num.vars]INT var.bp IS
  [ 2, 1, 1, 4, 2, 1, 1, 1, 1, 1, 1, 4 ] :
VAL [num.vars]INT var.ap IS
  [ 3, 3, 3, 1, 2, 1, 3, 3, 3, 3, 3, 1 ] :
VAL [num.vars]BOOL unipolar.var IS
  [ TRUE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE ] :
VAL [num.vars]BOOL initial.display.vars IS
  [ TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE ] :

[num.vars]BOOL display.on :
[num.vars][4]REAL32 display.table :

[80]BYTE display.line :
[200]BYTE string :
INT char, string.index, line.index :
REAL32 time :
BOOL bar.display.selected, buffer.output, finished :

SEQ
  display.on := initial.display.vars
  bar.display.selected := TRUE
  buffer.output := FALSE
  finished := FALSE

```

```

WHILE NOT finished
SEQ
  PRI ALT
  keyboard ? char
  SEQ
    CASE BYTE char
      'W','w'
        SEQ index1 = 1 FOR 30000
        SEQ index2 = 1 FOR 100
        SKIP
      'B','b'
        bar.display.selected := NOT bar.display.selected
      'L','l'
        data.buffer.params.out ! data.buffer.log
      'S','s'
        SEQ
          buffer.output := NOT buffer.output
          data.buffer.params.out ! data.buffer.select; buffer.output
      'Q','q'
        finished := TRUE
    ELSE
      SEQ index = 1 FOR num.vars - 1
      IF
        ((BYTE char) = var.name[index][0])
        display.on[index] := NOT display.on[index]
      TRUE
      SKIP

  TRUE & SKIP
  SKIP

-- Get some data from the controller --
data.request.out ! TRUE
monitored.data.in ? time; -- reference signal time stamp
                        [display.table FROM 1 FOR 4]; -- magnet signals
                        [display.table FROM 5 FOR 7] -- mode signals

IF
  (time = 0.0 (REAL32))
  write.endstream( screen )
TRUE
SKIP

-- Scale the variables before display --
SEQ var.index = 1 FOR num.vars - 1
IF
  display.on[ var.index ]
  SEQ index = 0 FOR 4
  display.table[var.index][index] := var.scaler[var.index] *
  display.table[var.index][index]
TRUE
SKIP

-- Display value of each selected variable --
IF
  bar.display.selected
  SEQ
    goto.xy( screen, 0, 24 )
    clear.eol( screen )
TRUE
SKIP

string[0] := '*c'
string[1] := '*n'
string.index := 2

-- Copy time stamp to display string --
append.text( string.index, string, var.name[0])
append.real32( string.index, string, time, var.bp[0], var.ap[0] )
string[ string.index ] := ' '
string.index := string.index + 1

-- Copy other required signals --
SEQ var.index = 1 FOR num.vars-1
IF
  display.on[ var.index ]
  SEQ
    append.text( string.index, string, var.name[ var.index ] )
    SEQ index = 0 FOR 4
    SEQ
      append.real32( string.index, string, display.table[var.index][index],

```

```

                var.bp[var.index], var.ap[var.index] )
        string[ string.index ] := ' '
        string.index := string.index + 1
    TRUE
    SKIP
write.len.string( screen, string.index, string )

IF
    bar.display.selected
    SEQ -- Display line graph for each selected variable --
    SEQ index = 0 FOR SIZE display.line
        display.line[ index ] := ' '
    SEQ index = 1 FOR 9
        display.line[ index * 8 ] := '.'
    display.line[ 0 ] := '|'
    display.line[ 40 ] := '|'

    SEQ var.index = 1 FOR num.vars - 1 -- exclude time --
    SEQ index = 0 FOR 4
        IF
            display.on[ var.index ]
            SEQ
                line.index := INT ROUND ((80.0 (REAL32) * display.table[var.index][index])
                    / var.FSD[var.index])

                IF
                    NOT unipolar.var[ var.index ]
                        line.index := line.index + 40
                    TRUE
                    SKIP
                IF
                    (line.index > 79)
                        line.index := 79
                    (line.index < 0)
                        line.index := 0
                    TRUE
                    SKIP
                display.line[line.index] := var.name[var.index][0]
                IF
                    (display.line[line.index] <> 'r')
                        display.line[ line.index ] := BYTE (index + 49)
                    TRUE
                    SKIP
            TRUE
            SKIP
        TRUE
        SKIP

    goto.xy( screen, 0, 23 )
    write.full.string( screen, display.line )
TRUE
SKIP
:

```

### E.3 Single magnet suspension controller

```

PROC Maglev.Controller( CHAN OF INT keyboard,
                      CHAN OF ANY screen,
                      CHAN OF INT16 ADC.in, ADC.out,
                      DAC.in, DAC.out )

#####

Single Magnet (Cantilever) Controller

Features: - Track trajectory filter
          - Absolute position controller (4 term)
          - Full non-linear magnet force model for current calculation
          - Real time data monitoring and logging

Author: Neil McLagan
Version 1.0
Last modified: 24-1-91

All variables are in elementary units (eg. secs, metres, Hz, metres/sec)
unless the variable name is post-fixed with an alternative, eg. airgap.mm
Airgap is positive, velocity and acceleration have same direction as airgap.

#####

#USE "Protocols.tsr"
#USE userio

-----

PROC Set.default.parameters(REAL32 integral.time, stiffness, damping, massness,
                           sampling.period, estim.cutoff.Hz, track.cutoff.Hz,
                           mass, operating.airgap,
                           airgap.frequency, airgap.amplitude, logging.period,
                           BYTE airgap.waveform )

-- Set default values for controller parameters --

SEQ
-- Default ride characteristics --
integral.time := 1.0 (REAL32) -- seconds
stiffness := 250000.0 (REAL32) -- Newtons/metre
damping := 6500.0 (REAL32) -- Newtons/(metre/second)
massness := 15.0 (REAL32) -- Newtons/(metre/second^2) = Kilogrammes

-- Initialise sampling period and filter cut-off frequencies --
sampling.period := 0.0005 (REAL32)
estim.cutoff.Hz := 0.1 (REAL32)
track.cutoff.Hz := 4.0 (REAL32)

-- Default mass --
mass := 13.0 (REAL32)

-- Default airgap reference characteristics --
operating.airgap := 0.003 (REAL32) -- 3 mm --
airgap.frequency := 0.05 (REAL32) -- 0.05 Hz --
airgap.amplitude := 0.001 (REAL32) -- 1 mm --
logging.period := 0.020 (REAL32) -- 20 ms --
airgap.waveform := 'C' -- Constant airgap --

:

-----

#USE "Protocols.tsr"

PROC Read.ADC( CHAN OF INT16 ADC.in, ADC.out,
              VAL INT mux.channel,
              REAL32 adc )

-- Read.ADC requests an ADC conversion for each selected channel.
-- It then scales the values appropriately before outputting them.

INT16 adc.value :
REAL32 adc.volts, sensor.gap, magnet.gap :

VAL REAL32 metres.per.volt IS 0.00125 (REAL32) :

```

```

VAL REAL32 sensor.offset IS 0.003500 (REAL32) :
VAL REAL32 magnet.offset IS 0.003740 (REAL32) :
VAL REAL32 magnet.sensor.length.ratio IS 1.446 (REAL32) :

SEQ
  ADC.out ! INT16 mux.channel
  ADC.in ? adc.value

  adc.volts := (REAL32 ROUND adc.value) / 6553.6(REAL32) -- convert to volts

  IF
    (mux.channel = 1) -- airgap channel request
      SEQ
        sensor.gap := (adc.volts * metres.per.volt) + sensor.offset
        magnet.gap := (sensor.gap - magnet.offset) * magnet.sensor.length.ratio
        adc := magnet.gap

    (mux.channel = 16) -- acceleration channel request
      SEQ
        adc := -(adc.volts / 5.0(REAL32)) -- convert to g's
        adc := adc * 9.81(REAL32) -- convert to m/s/s

:

-----

PROC Write.DAC( CHAN OF INT16 DAC.in, DAC.out,
               VAL REAL32 current.demand )

-- Write.DAC scales the requested current drive and sends it to the DAC.

VAL min.current IS 0.0 (REAL32) :
VAL max.current IS 19.9 (REAL32) :
REAL32 current :

SEQ
  current := current.demand
  IF
    current > max.current
      current := max.current
    current < min.current
      current := min.current
  TRUE
  SKIP

  DAC.out ! INT16 ROUND( current * 204.8(REAL32) )

:

-----

VAL num.airgap.chans IS 3 :
VAL num.acc.chans IS 2 :

PROC Interface.to.outside.world( CHAN OF INT16 ADC.in, ADC.out,
                                DAC.in, DAC.out,
                                CHAN OF control.p trigger,
                                CHAN OF data.p current,
                                [num.airgap.chans] CHAN OF data.p airgap.chan,
                                [num.acc.chans] CHAN OF data.p acc.chan,
                                CHAN OF control.p cycle.finished,
                                CHAN OF exception.p exception )

-- Interface.to.outside.world communicates the current demands and
-- sensor values to and from the DAC and ADC drivers.

-- A-D converter sensor addresses --
VAL acc.sensor IS 16 :
VAL airgap.sensor IS 1 :

BOOL final.trigger, final.current, start.converting :
REAL32 acc, airgap, current.demand :

SEQ
  final.current := FALSE
  WHILE NOT final.current
    ALT
      trigger ? final.trigger; start.converting
      SEQ
        -- Read in acceleration --

```



```

Read.ADC( ADC.in, ADC.out, acc.sensor, acc )
PAR i = 0 FOR num.acc.chans
  acc.chan[i] ! final.trigger; acc

  -- Read in position --
Read.ADC( ADC.in, ADC.out, airgap.sensor, airgap )
PAR i = 0 FOR num.airgap.chans
  airgap.chan[i] ! final.trigger; airgap

-- Output current demand to magnet current amplifier --
current ? final.current; current.demand
SEQ
  IF
    final.current
    Write.DAC( DAC.in, DAC.out, 0.0 (REAL32) )
    TRUE
    Write.DAC( DAC.in, DAC.out, current.demand )
  cycle.finished ! final.current; TRUE
:

-----
#USE "Protocols.tsr"
#USE dblmath

VAL num.force.chans IS 2 :
VAL num.velocity.chans IS 1 :

PROC Calculate.magnet.force( CHAN OF data.p airgap.in, airgap.ref.in, acc.in,
  [num.force.chans] CHAN OF data.p force.out,
  [num.velocity.chans] CHAN OF data.p velocity.out,
  CHAN OF exception.p exception.out,
  VAL REAL32 integral.time, stiffness, damping, massness,
  mass,
  sampling.period,
  est.cutoff.Hz, track.cutoff.Hz )

-- Calculate.magnet.force receives airgap and acceleration signals and performs
-- the functions of track trajectory filter and absolute position control.

VAL pi IS 3.1415926 (REAL64) :
VAL two.pi IS 2.0 (REAL64) * pi :
VAL gravity.acc IS 9.81 (REAL32) :

VAL int.force.limit IS 500.0 (REAL32) : -- 500 Newton integral limit
VAL vel.limit IS 0.2 (REAL32) : -- 0.2 mm/ms velocity limit --
VAL [] BYTE warning.message IS "Warning: Excessive velocity, V = " :

BOOL final.airgap, final.airgap.ref, final.acc, final.force :

REAL64 est.input.factor, est.decay.factor :
REAL64 track.input.factor, track.decay.factor :

REAL32 airgap, airgap.ref, acc.meas :
REAL64 acc.meas.r64, acc.lp, acc.hp, vel.hp, pos.hp, pos.error :
REAL64 track.pos, track.pos.fl, track.pos.f2, track.trajectory :
REAL32 acc.feedback, vel.feedback, pos.feedback, int.feedback :
REAL32 force.demand, vel.r32 :

SEQ
  -- Set filter factors --
  est.decay.factor := Dexp( (- two.pi) * (REAL64 (est.cutoff.Hz * sampling.period)) )
  est.input.factor := 1.0 (REAL64) - est.decay.factor

  track.decay.factor := Dexp( (- two.pi) * (REAL64 (track.cutoff.Hz * sampling.period)) )
  track.input.factor := 1.0 (REAL64) - track.decay.factor

  -- Initialise filters and integrators --
  PAR
    airgap.in ? final.airgap; airgap
    airgap.ref.in ? final.airgap.ref; airgap.ref
    acc.in ? final.acc; acc.meas

  final.force := (final.airgap AND (final.airgap.ref AND final.acc))
  PAR
    PAR i = 0 FOR num.force.chans
      force.out[i] ! final.force; 0.0 (REAL32)
    PAR i = 0 FOR num.velocity.chans
      velocity.out[i] ! final.force; 0.0 (REAL32)

  acc.lp := 0.0 (REAL64)

```

```

vel.hp := 0.0 (REAL64)
pos.hp := 0.0 (REAL64)

track.pos := REAL64 (airgap.ref - airgap)
track.pos.f1 := track.pos
track.pos.f2 := track.pos.f1

int.feedback := 0.0 (REAL32)

WHILE NOT final.force
  SEQ
    PAR
      -- Input the airgap and airgap reference --
      airgap.in ? final.airgap; airgap
      airgap.ref.in ? final.airgap.ref; airgap.ref

    SEQ
      -- Input the (high pass filtered) acceleration --
      acc.in ? final.acc; acc.meas

      -- Estimate the absolute acceleration --
      acc.meas.r64 := REAL64 acc.meas
      acc.lp := (acc.lp * est.decay.factor) + (acc.meas.r64 * est.input.factor)
      acc.hp := acc.meas.r64 - acc.lp

      -- Estimate the absolute velocity and position --
      vel.hp := (vel.hp * est.decay.factor) + (acc.hp * (REAL64 sampling.period))
      pos.hp := (pos.hp * est.decay.factor) + (vel.hp * (REAL64 sampling.period))

      -- Estimate the absolute track position --
      track.pos := (REAL64 (airgap.ref - airgap)) + pos.hp

      -- and filter to form the track trajectory --
      track.pos.f1 := (track.pos.f1*track.decay.factor) + (track.pos*track.input.factor)
      track.pos.f2 := (track.pos.f2*track.decay.factor) + (track.pos.f1*track.input.factor)
      track.trajectory := track.pos.f2

      -- Calculate the absolute position error --
      pos.error := (pos.hp - track.trajectory)

      -- Calculate the feedback components --
      acc.feedback := massness * (REAL32 ROUND acc.hp)
      vel.feedback := damping * (REAL32 ROUND vel.hp)
      pos.feedback := stiffness * (REAL32 ROUND pos.error)
    IF
      (int.feedback < (-int.force.limit))
        int.feedback := -int.force.limit
      (int.feedback > int.force.limit)
        int.feedback := int.force.limit
      (integral.time > 0.0 (REAL32) )
        int.feedback := int.feedback +
          ( ((airgap-airgap.ref)*sampling.period) * (stiffness/integral.time) )
    TRUE
      int.feedback := 0.0 (REAL32)

      -- Total the force requirement (and account for mass) --
      force.demand := ( (int.feedback + pos.feedback) +
        (vel.feedback + acc.feedback) ) + (mass * gravity.acc)

      -- Output force demand & velocity --
      final.force := (final.airgap AND (final.airgap.ref AND final.acc))
      vel.r32 := REAL32 ROUND vel.hp
    PAR
      PAR i = 0 FOR num.force.chans
        force.out[i] ! final.force; force.demand
      PAR i = 0 FOR num.velocity.chans
        velocity.out[i] ! final.force; vel.r32

      -- Check velocity signal magnitude --
    IF
      (vel.r32 < (-vel.limit)) OR (vel.r32 > vel.limit)
        exception.out ! SIZE warning.message::warning.message; vel.r32
    TRUE
      SKIP

```

:

```

-----
#USE "Protocols.tsr"
#USE snglmath

VAL num.current.chans IS 2 :

PROC Calculate.magnet.current( CHAN OF data.p airgap.in,
                               force.demand.in,
                               [num.current.chans] CHAN OF data.p current.required.out,
                               CHAN OF exception.p exception.out,
                               VAL REAL32 sampling.period )

-- Calculate.magnet.current uses the nonlinear magnet force model to determine
-- the required current for the demanded force at the present airgap.

-- Worldly constants --
VAL pi IS 3.1415926 (REAL32) :
VAL mu.air IS 1.2566E-6 (REAL32) : -- Permeability of free space
-- These are used to make the equations more readable --
VAL zero IS 0.0 (REAL32) :
VAL one IS 1.0 (REAL32) :
VAL two IS 2.0 (REAL32) :
VAL four IS 4.0 (REAL32) :

-- Electromagnet parameters --
VAL y IS 0.0015 (REAL32) : -- Lateral offset between magnet & track poles
VAL l IS 0.200 (REAL32) : -- Magnet length
VAL w IS 0.033 (REAL32) : -- Width between the magnet pole pieces
VAL h IS 0.063 (REAL32) : -- Height of the magnet pole pieces
VAL t IS 0.030 (REAL32) : -- Width between the track pole pieces
VAL p IS 0.0095 (REAL32) : -- Width of the pole pieces

VAL N.coils IS 274.0 (REAL32) : -- Number of coil turns
VAL min.current IS 0.0 (REAL32) : -- Impose current limit
VAL max.current IS 20.0 (REAL32) : -- Impose current limit
VAL mu.iron IS 2000.0 (REAL32) : -- Maximum permeability of the core material
VAL flux.roll.start IS 1.0 (REAL32) : -- Start of permeability roll-off (/Tesla)
VAL flux.roll.stop IS 1.7 (REAL32) : -- End of permeability roll-off (/Tesla)

VAL T.eddy IS 0.0015 (REAL32) : -- Target eddy current time constant
VAL N.lead.lag IS 3.0 (REAL32) : -- Lead lag ratio to achieve the above

-- Nonlinear magnet model variables --
REAL32 T.lead.lag, lead.lag.input.factor, lead.lag.decay.factor :
REAL32 flux.low.pass, flux.lead.lag :
REAL32 R.track, R.leakage, R.airgap, R.magnet :
REAL32 force.demand, airgap, force.per.pole :
REAL32 airgap.flux, flux.squared, leakage.flux, magnet.flux :
REAL32 flux.density, mu.magnet :
REAL32 airgaps.plus.track.mmf, magnet.mmf :
REAL32 current.demand :

BOOL final.airgap, final.force, final.output :

SEQ
-- Determine lead lag compensator coefficients and initialise --
IF
  (T.eddy > sampling.period)
  T.lead.lag := T.eddy
  TRUE
  T.lead.lag := sampling.period

lead.lag.decay.factor := exp( - (sampling.period / T.eddy) )
lead.lag.input.factor := one - lead.lag.decay.factor
flux.low.pass := zero

-- Calculate the track and leakage reluctances --
R.track := ( t + (four*p) ) / ( (mu.iron*mu.air) * (l*p) )
R.leakage := w / ( mu.air * ( 1+((two*w)/pi) ) * ((h/two)+(w/pi) ) )

final.output := FALSE
WHILE NOT final.output
  SEQ
  PAR
    force.demand.in ? final.force; force.demand
    airgap.in ? final.airgap; airgap

  IF
    (force.demand < zero)
    force.demand := zero

```

```

TRUE
SKIP

-- Calculate airgap reluctance --
R.airgap := airgap / ( (mu.air*1) * ( p + ((two*airgap)/pi) ) )

-- Calculate the gross force per pole --
force.per.pole := force.demand /
  ( two * ( one - ( (y*y)/airgap) / (airgap+((pi*p)/two)) ) )

-- Calculate the required airgap flux --
flux.squared := force.per.pole * ( (two*airgap) / R.airgap )
IF
  (flux.squared > zero)
    airgap.flux := SQRT( flux.squared )
  TRUE
    airgap.flux := zero

-- Calculate the mmf required to overcome the airgap and track reluctances
airgaps.plus.track.mmf := airgap.flux * ( (two*R.airgap) + R.track )

-- Calculate the consequential leakage flux --
leakage.flux := airgaps.plus.track.mmf / R.leakage

-- Calculate the resulting electromagnet flux --
magnet.flux := airgap.flux + leakage.flux

-- Apply lead lag compensation teh the magnet flux demand --
flux.low.pass := (flux.low.pass * lead.lag.decay.factor) +
  (magnet.flux * lead.lag.input.factor)
flux.lead.lag := (N.lead.lag * magnet.flux) - ( (N.lead.lag-one) * flux.low.pass )

-- Calculate the magnet reluctance at this flux level --
flux.density := magnet.flux / (1 * p)
IF
  (flux.density < flux.roll.start)
    mu.magnet := mu.iron
  (flux.density < flux.roll.stop)
    mu.magnet := mu.iron * ( 1.1(REAL32) - ( (flux.density - flux.roll.start) /
      (flux.roll.stop - flux.roll.start) ) )
  TRUE
    mu.magnet := mu.iron * 0.1(REAL32)
R.magnet := ( (two*h) + (w+(two*p)) ) / ( (mu.air*mu.magnet) * (1*p) )

-- Calculate the mmf required to overcome the magnet reluctance --
magnet.mmf := flux.lead.lag * R.magnet

-- Finally, calculate the current required to produce the total mmf --
current.demand := (airgaps.plus.track.mmf + magnet.mmf) / N.coils
IF
  (current.demand > max.current)
    current.demand := max.current
  (current.demand < min.current)
    current.demand := min.current
  TRUE
    SKIP

final.output := (final.airgap AND final.force)
PAR i = 0 FOR num.current.chans
  current.required.out[i] ! final.output; current.demand
:

```

```

-----
#USE "Protocols.tsr"

PROC Handle.exceptions( [] CHAN OF exception.p exceptions.in,
                       CHAN OF BOOL stop.system.out,
                       [] BYTE message,
                       INT message.length,
                       REAL32 message.value )

-- Handle.exceptions initiates a controlled system shutdown upon receiving
-- any exception message (including a user requested shutdown).

VAL number.of.error.channels IS SIZE exceptions.in :
VAL terminate IS 0 :

[80] BYTE ignore.message :
INT ignore.length :
REAL32 ignore.value :

BOOL finished :

SEQ
-- Get first exception from the system --
ALT i = 0 FOR number.of.error.channels
  exceptions.in[i] ? message.length::message; message.value
  SKIP

finished := FALSE
PAR
-- Instruct system to shut down --
stop.system.out ! TRUE

-- Accept (but ignore) any further exceptions --
WHILE NOT finished
  ALT i = 0 FOR number.of.error.channels
    exceptions.in[i] ? ignore.length::ignore.message; ignore.value
    finished := (i = terminate)
:

```

```

-----
#USE "Protocols.tsr"

PROC Data.Buffer( CHAN OF data.p v1.in,v1.out, v2.in,v2.out, v3.in,v3.out,
                 v4.in,v4.out, v5.in,v5.out, v6.in,v6.out,
                 CHAN OF BOOL output.required )

-- Data.Buffer buffers data between the control system and the data
-- monitoring system, allowing them to run asynchronously.

REAL32 v1.data, v2.data, v3.data, v4.data, v5.data, v6.data :
BOOL v1.finished, v2.finished, v3.finished, v4.finished, v4.finished,
v5.finished, v6.finished :

BOOL ready, output.notified:

SEQ
v1.finished := FALSE
output.notified := FALSE

PAR
WHILE NOT (v1.finished AND output.notified)
-- Continuously read in all variables --
PRI ALT
  output.required ? ready
  PAR
    v1.out ! v1.finished; v1.data
    v2.out ! v2.finished; v2.data
    v3.out ! v3.finished; v3.data
    v4.out ! v4.finished; v4.data
    v5.out ! v5.finished; v5.data
    v6.out ! v6.finished; v6.data
    output.notified := v1.finished

  NOT v1.finished & SKIP
  PAR
    v1.in ? v1.finished; v1.data
    v2.in ? v2.finished; v2.data
    v3.in ? v3.finished; v3.data
    v4.in ? v4.finished; v4.data
    v5.in ? v5.finished; v5.data

```

```

v6.in ? v6.finished; v6.data
:

-----
#USE "Protocols.tsr"
#USE userio
#USE strings

PROC Data.Monitor( CHAN OF data.p gap.in, vel.in, acc.in,
                  force.in, current.in, gap.ref.in,
                  CHAN OF BOOL data.input.trigger,
                  CHAN OF exception.p exception.out,
                  CHAN OF INT keyboard,
                  CHAN OF ANY screen,
                  VAL REAL32 logging.period )

-- Data.Monitor receives data from the control system via the data buffer.
-- Data selected by the keyboard is sent to the screen. Each data item
-- is toggled on/off using its initial letter. 'T' toggles tabular/graph
-- output, and 'P' causes the output to pause briefly to allow the Network
-- Monitor EXE program to break free from the data stream, thus allowing the
-- Data Logger EXE program to be run to log data to disc.

INT char :
BYTE option :

BOOL final.gap, final.vel, final.acc,
      final.force, final.current, final.gap.ref,
      finished :

-- Display variable tables etc. --
VAL num.vars IS 7 :

VAL [num.vars] [2] BYTE var.name IS
[ "T=", "R=", "G=", "V=", "A=", "I=", "F=" ] :
VAL [num.vars] INT var.scaler IS
[ 1, 1000, 1000, 1, 1, 1, 1 ] :
VAL [num.vars] INT var.FSD IS
[ 1, 10, 10, 1, 1, 20, 1000 ] :
VAL [num.vars] INT var.bp IS
[ 2, 1, 1, 1, 1, 2, 4 ] :
VAL [num.vars] INT var.ap IS
[ 3, 1, 3, 3, 3, 2, 1 ] :
VAL [num.vars] BOOL initial.display.vars IS
[ TRUE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE ] :

[num.vars] BOOL display.on :
[num.vars] REAL32 display.table :

VAL ticks.per.second IS 15625.0 (REAL32) :
TIMER clock :
[80] BYTE display.line :
INT display.time.interval, display.time, time, last.time, line.index :
REAL32 time.interval, real.time :
REAL32 gap.ref, last.gap.ref :
BOOL bar.display.selected :

INT string.index :
[200] BYTE string :

SEQ
display.time.interval := INT TRUNC ( logging.period * ticks.per.second )
display.on := initial.display.vars

SEQ var.index = 0 FOR num.vars
display.table[ var.index ] := 0.0 (REAL32)

bar.display.selected := TRUE
finished := FALSE
clock ? time
clock ? display.time
real.time := 0.0 (REAL32)
last.gap.ref := 0.1 (REAL32)

WHILE NOT finished
SEQ
PRI ALT
keyboard ? char
SEQ
option := BYTE (char /\ #DF)

```

```

IF
  (option = 'L')
    write.endstream( screen )
  (option = 'P')
    SEQ index1 = 1 FOR 30000
      SEQ index2 = 1 FOR 100
        SKIP
  (option = 'Q')
    exception.out ! 0::""; 0.0 (REAL32)
  (option = 'T')
    bar.display.selected := NOT bar.display.selected
  TRUE
    SEQ index = 1 FOR num.vars - 1
      IF
        (option = var.name[index][0])
          display.on[index] := NOT display.on[index]
      TRUE
        SKIP

  TRUE & SKIP
  SKIP

clock ? AFTER display.time
display.time := display.time PLUS display.time.interval

-- Get some data from the controller --
data.input.trigger ! TRUE

last.time := time
clock ? time
time.interval := (REAL32 TRUNC (time MINUS last.time))/ticks.per.second

PAR
  gap.ref.in ? final.gap.ref; display.table[1]
  gap.in ? final.gap; display.table[2]
  vel.in ? final.vel; display.table[3]
  acc.in ? final.acc; display.table[4]
  current.in ? final.current; display.table[5]
  force.in ? final.force; display.table[6]
finished := final.gap

gap.ref := display.table[1]
IF
  ((gap.ref - 0.00009 (REAL32)) > last.gap.ref)
  SEQ
    write.endstream( screen )
    display.table[1] := last.gap.ref
    clock ? display.time
    real.time := 0.0 (REAL32)
  TRUE
    real.time := real.time + time.interval
last.gap.ref := gap.ref

display.table[0] := real.time

-- Scale the variables before display --
SEQ var.index = 1 FOR num.vars - 1
  IF
    display.on[ var.index ]
    display.table[ var.index ] := display.table[ var.index ] *
      (REAL32 TRUNC var.scaler[ var.index ])
  TRUE
    SKIP

-- Display value of each selected variable --
IF
  bar.display.selected
  SEQ
    goto.xy( screen, 0, 24 )
    clear.eol( screen )
  TRUE
    SKIP

string[0] := '*c'
string[1] := '*n'
string.index := 2
SEQ var.index = 0 FOR num.vars
  IF
    display.on[ var.index ]
    SEQ
      append.text( string.index, string, var.name[ var.index ] )

```

```

        append.real32( string.index, string,
            display.table[var.index], var.bp[var.index], var.ap[var.index] )
        string[ string.index ] := ' '
        string.index := string.index + 1
    TRUE
    SKIP
write.len.string( screen, string.index, string )

IF
    bar.display.selected
    SEQ -- Display line graph for each selected variable --
    SEQ index = 0 FOR SIZE display.line
        display.line[ index ] := ' '
    SEQ index = 1 FOR 9
        display.line[ index * 8 ] := '.'
    display.line[ 0 ] := '|'

    SEQ var.index = 1 FOR num.vars - 1 -- exclude time --
    IF
        display.on[ var.index ]
        SEQ
            line.index := INT ROUND ((80.0 (REAL32) * display.table[var.index]) /
                (REAL32 TRUNC var.FSD[var.index]))
            IF
                (line.index > 79)
                    line.index := 79
                (line.index < 0)
                    line.index := 0
            TRUE
            SKIP
            display.line[ line.index ] := var.name[ var.index ][ 0 ]
        TRUE
        SKIP

        goto.xy( screen, 0, 23 )
        write.full.string( screen, display.line )
    TRUE
    SKIP
:

-----
#USE "Protocols.tsr"
#USE snglmath

VAL num.ref.chans IS 2 :

PROC Controller( CHAN OF control.p cycle.finished.in,
                CHAN OF control.p start.cycle.out,
                [num.ref.chans] CHAN OF data.p airgap.reference.out,
                CHAN OF BOOL stop.system.in,
                CHAN OF exception.p exception.out,
                VAL REAL32 sampling.period,
                VAL REAL32 operating.airgap, airgap.frequency,
                airgap.amplitude,
                VAL BYTE airgap.waveform )

-- Controller controls the scheduling, start up, and shutdown of the magnet.
-- It also checks the execution time and generates the reference signal.

-- Worldly constants --
VAL two.pi IS 2.0 (REAL32) * 3.1415926 (REAL32) :

-- Airgap --
VAL airgap.ramp.limit IS 0.010 (REAL32) :
VAL airgap.ramp.vel IS 0.005 (REAL32) :
REAL32 t, sine, airgap, amplitude.factor :

-- Sampling --
VAL ticks.per.sec IS 1000000.0 (REAL32) :
INT sample.interval.ticks, sample.time, time :
REAL32 cycle.time :
TIMER sampler :

BOOL ramp.up, ramp.down, time.to.wind.down, finished, done :

SEQ
    -- Prepare status --
    t := 0.0 (REAL32)
    amplitude.factor := airgap.amplitude / 2.0 (REAL32) -- was pk-pk --

```



```

airgap := airgap.ramp.limit
ramp.up := TRUE
ramp.down := FALSE
finished := FALSE

-- Allow for all PROCs to initialise etc before timing --
start.cycle.out ! finished; TRUE
PAR i = 0 FOR num.ref.chans
    airgap.reference.out[i] ! finished; airgap
cycle.finished.in ? finished; done

-- Initialise sampling timer --
sample.interval.ticks := INT ROUND( ticks.per.sec * sampling.period)
sampler ? sample.time

WHILE NOT finished
    SEQ
        PRI ALT
            stop.system.in ? time.to.wind.down
            SEQ
                ramp.up := FALSE
                ramp.down := TRUE
            TRUE & SKIP
            SEQ
                ramp.up := ramp.up AND (airgap > operating.airgap)
                finished := ramp.down AND (airgap > airgap.ramp.limit)

        IF
            ramp.up
            airgap := airgap - (airgap.ramp.vel * sampling.period)
            ramp.down
            airgap := airgap + (airgap.ramp.vel * sampling.period)
            TRUE
            SEQ
                t := t + sampling.period
                sine := SIN( two.pi * (airgap.frequency * t) )
                CASE airgap.waveform
                    'I' -- sine wave --
                    airgap := operating.airgap + (amplitude.factor * sine)
                    'Q' -- square wave --
                    IF
                        (sine > 0.0 (REAL32))
                        airgap := operating.airgap + amplitude.factor
                    TRUE
                        airgap := operating.airgap - amplitude.factor
                    ELSE -- constant airgap --
                        airgap := operating.airgap

-- Wait for next sampling time --
sampler ? time
cycle.time := REAL32 TRUNC (time MINUS sample.time)
sample.time := sample.time PLUS sample.interval.ticks
IF
    (time AFTER sample.time)
    VAL [] BYTE message IS "Error: Sampling period is too short, cycle time was:" :
    SEQ
        exception.out ! SIZE message::message; (cycle.time / ticks.per.sec) * 1000.0 (REAL32)
        sample.time := time
    TRUE
    sampler ? AFTER sample.time

-- Initiate an iteration of the control loop --
start.cycle.out ! finished; TRUE
PAR i = 0 FOR num.ref.chans
    airgap.reference.out[i] ! finished; airgap
cycle.finished.in ? finished; done
:

-----

PROC Levitate( CHAN OF INT keyboard,
              CHAN OF ANY screen,
              CHAN OF INT16 ADC.in, ADC.out, DAC.in, DAC.out,
              VAL REAL32 integral.time, stiffness, damping, massness,
                  sampling.period,
                  est.cutoff.Hz, track.cutoff.Hz,
                  mass, operating.airgap,
                  airgap.frequency, airgap.amplitude, logging.period,
              VAL BYTE airgap.waveform )

```

```

-- Levitate executes the component processes of the control and monitoring
-- system, ensuring that appropriate process priorities are allocated and
-- that a controlled system shutdown is achieved.

[3] CHAN OF data.p gap :
[1] CHAN OF data.p vel :
[2] CHAN OF data.p acc, gap.ref, force, current :
CHAN OF data.p gap.buf, vel.buf, acc.buf, force.buf, current.buf, gap.ref.buf :
[7] CHAN OF exception.p exception :
CHAN OF control.p start.cycle, cycle.finished :
CHAN OF BOOL stop.system, buffer.output.trigger :

VAL release.exception.handler IS 0 :
VAL [] BYTE nothing IS "" :

INT message.length, anychar :
REAL32 message.value :
[80] BYTE message :

SEQ
  PAR
    SEQ
      PRI PAR
        PAR
          Interface.to.outside.world( ADC.in, ADC.out,
                                     DAC.in, DAC.out,
                                     start.cycle, current[1],
                                     gap, acc,
                                     cycle.finished,
                                     exception[1] )

          Calculate.magnet.force( gap[1], gap.ref[1], acc[1],
                                 force, vel,
                                 exception[3],
                                 integral.time, stiffness, damping, massness,
                                 mass, sampling.period,
                                 est.cutoff.Hz, track.cutoff.Hz )

          Calculate.magnet.current( gap[2], force[1],
                                   current,
                                   exception[4],
                                   sampling.period )

          Controller( cycle.finished,
                     start.cycle, gap.ref,
                     stop.system,
                     exception[5],
                     sampling.period,
                     operating.airgap, airgap.frequency, airgap.amplitude,
                     airgap.waveform )

          Data.Buffer( gap[0], gap.buf, vel[0], vel.buf, acc[0], acc.buf,
                     force[0], force.buf, current[0], current.buf,
                     gap.ref[0], gap.ref.buf,
                     buffer.output.trigger )

          Data.Monitor( gap.buf, vel.buf, acc.buf, force.buf, current.buf,
                      gap.ref.buf,
                      buffer.output.trigger,
                      exception[6], keyboard, screen,
                      logging.period )

          exception[ release.exception.handler ] ! 0::""; 0.0(REAL32)

          Handle.exceptions( exception,
                            stop.system,
                            message, message.length, message.value )

        IF
          (message.length > 0)
            SEQ
              newline( screen )
              newline( screen )
              write.len.string( screen, message.length, message )
              write.real32( screen, message.value, 3,3 )
              newline( screen )
              read.char( keyboard, anychar )
            TRUE
          SKIP

```

```

-----
PROC Menu( CHAN OF INT keyboard,
           CHAN OF ANY screen,
           CHAN OF INT16 ADC.in, ADC.out, DAC.in, DAC.out,
           REAL32 integral.time, stiffness, damping, massness,
                    sampling.period, est.cutoff.Hz, track.cutoff.Hz,
           mass, operating.airgap,
           airgap.frequency, airgap.amplitude, logging.period,
           BYTE airgap.waveform )

-- Menu permits the setting of system parameters and initiates a control system run.

-----
PROC Write.menu( VAL [] BYTE option,
                 VAL REAL32 value,
                 VAL INT i, d,
                 VAL [] BYTE units )
SEQ
write.full.string( screen, " " )
write.full.string( screen, option )
write.real32( screen, value, i, d )
write.full.string( screen, units )
newline( screen )
:

-----
PROC Get.value( VAL [] BYTE text,
               REAL32 new.value,
               VAL REAL32 factor )

INT delimiter :

SEQ
write.text.line( screen, text )
write.full.string( screen, " Enter new parameter value: " )
new.value := 0.0 (REAL32)
delimiter := INT ' '
read.echo.real32( keyboard, screen, new.value, delimiter )
new.value := new.value * factor
:

VAL kilo IS 1000.0 (REAL32) :
VAL kilo.g IS 9810.0 (REAL32) :

INT option.int :
BYTE option :
REAL32 factor :
BOOL finished :

SEQ
finished := FALSE
WHILE NOT finished
SEQ
write.endstream( screen )

-- Write menu on screen --
goto.xy( screen, 0,0 )
clear.eos( screen )

write.text.line( screen, " MAGLEV CONTROLLER" )
write.text.line( screen, " ~~~~~" )
newline( screen )

Write.menu("I ---- Integral time constant ( ", integral.time, 1,1, " s )" )
Write.menu("S ---- Stiffness ( ", stiffness/kilo.g, 2,1, " kg/mm )" )
Write.menu("D ---- Damping ( ", damping/kilo.g, 1,2, " kg/(mm/s)" )
Write.menu("M ---- Massness ( ", massness, 2,1, " kg )" )
newline( screen )

Write.menu("P ---- sampling Period ( ", sampling.period*kilo, 1,2, " ms )" )
Write.menu("E ---- Estimation cutoff frequency ( ", est.cutoff.Hz, 1,1, " Hz )" )
Write.menu("T ---- Track cutoff frequency ( ", track.cutoff.Hz, 2,1, " Hz )" )
Write.menu("L ---- suspended Load ( ", mass, 2,1, " Kg )" )
newline( screen )

Write.menu("G ---- airGap ( ", operating.airgap*kilo, 1,2, " mm )" )
Write.menu("R ---- Reference signal frequency ( ", airgap.frequency, 2,2, " Hz )" )
Write.menu("A ---- reference signal Amplitude ( ", airgap.amplitude*kilo, 1,2, " mm pk-pk )" )

```

```

Write.menu("O ---- data logging period ( ", logging.period*kilo, 2,1, " ms )")
write.text.line(screen,"                               W ---- Waveform ")
newline( screen )

write.text.line(screen,"                               V ---- leVitate")
write.text.line(screen,"                               Q ---- Default parameters")

write.full.string(screen,"*c*n                               Select option: ")
read.char(keyboard, option.int )

-- Convert to uppercase and BYTE --
option := BYTE (option.int /\ #DF)

CASE option
'I'
  Get.value( "Integral time constant", integral.time, 1.0 (REAL32) )
'S'
  Get.value( "Stiffness", stiffness, kilo.g )
'D'
  Get.value( "Damping", damping, kilo.g )
'M'
  Get.value( "Massness", massness, 1.0 (REAL32) )
'P'
  SEQ
    Get.value( "Sampling period", sampling.period, 0.001(REAL32))
    IF
      (sampling.period < 0.0001 (REAL32))
        sampling.period := 0.0001 (REAL32)
    TRUE
    SKIP
'E'
  Get.value( "Estimation cut-off frequency", est.cutoff.Hz, 1.0(REAL32))
'T'
  Get.value( "Track cut-off frequency", track.cutoff.Hz, 1.0(REAL32))
'L'
  Get.value( "Load", mass, 1.0 (REAL32) )
'G'
  Get.value( "Airgap", operating.airgap, 0.001 (REAL32) )
'R'
  Get.value( "Reference signal frequency", airgap.frequency, 1.0 (REAL32) )
'A'
  Get.value( "Reference signal amplitude", airgap.amplitude, 0.001 (REAL32) )
'O'
  Get.value( "Data logging period", logging.period, 0.001(REAL32))
'W'
  SEQ
    write.text.line( screen, "Waveform" )
    write.full.string( screen, "                               None, sIne or sQuare ? " )
    read.echo.char( keyboard, screen, option.int )
    airgap.waveform := BYTE (option.int /\ #DF)
'V'
  SEQ
    write.text.line( screen, "Levitating" )
    Levitate( keyboard, screen,
      ADC.in, ADC.out, DAC.in, DAC.out,
      integral.time, stiffness, damping, massness,
      sampling.period, est.cutoff.Hz, track.cutoff.Hz,
      mass, operating.airgap,
      airgap.frequency, airgap.amplitude, logging.period,
      airgap.waveform )
'Q'
  finished := TRUE
ELSE
  beep( screen )
:

```

```

-----
PROC Kernel( CHAN OF INT keyboard,
             CHAN OF ANY screen,
             CHAN OF INT16 ADC.in, ADC.out, DAC.in, DAC.out )

-- Kernel is the control system kernel process.

-- Ride characteristics --
REAL32 integral.time, stiffness, damping, massness :
REAL32 sampling.period, accel.cutoff.Hz, track.cutoff.Hz :
REAL32 mass, operating.airgap :
REAL32 airgap.frequency, airgap.amplitude, logging.period :
BYTE airgap.waveform :

SEQ
  WHILE TRUE
    SEQ
      Set.default.parameters( integral.time, stiffness, damping, massness,
                              sampling.period, accel.cutoff.Hz, track.cutoff.Hz,
                              mass, operating.airgap,
                              airgap.frequency, airgap.amplitude, logging.period,
                              airgap.waveform )

      Menu( keyboard, screen,
            ADC.in, ADC.out, DAC.in, DAC.out,
            integral.time, stiffness, damping, massness,
            sampling.period, accel.cutoff.Hz, track.cutoff.Hz,
            mass, operating.airgap,
            airgap.frequency, airgap.amplitude, logging.period,
            airgap.waveform )
    :

-----
-- Procedure code for Maglev.Controller()

SEQ
  Kernel( keyboard, screen,
          ADC.in, ADC.out,
          DAC.in, DAC.out )
:

```

---

```
VAL event IS 8 :
```

```
PROC ADC.driver( CHAN OF INT16 in, out )
```

```
INT16 channel, volts :
INT adc, mux, value :
BOOL done :
CHAN OF BOOL ready :
```

```
PLACE ready AT event :
PLACE adc AT #6000 :
PLACE mux AT #6100 :
```

```
-- The ADC outputs the last conversion result and starts a new
-- conversion each time it is read. It then generates an interrupt
-- upon the end of each conversion which is notified by 'ready'.
```

```
SEQ
```

```
value := adc -- start conversions & interrupts rolling
WHILE TRUE
```

```
SEQ
```

```
in ? channel -- wait for adc channel request
```

```
mux := INT channel -- select required channel
```

```
ready ? done
```

```
value := adc -- start conversion
```

```
ready ? done
```

```
value := adc -- read adc result
```

```
out ! INT16 (value /\ #FFF0) -- output adc result
```

```
:
```

---

```
PROC DAC.driver( CHAN OF INT16 in, out )
```

```
INT16 current :
BOOL kill.power :
INT dac1, switch, disable :
```

```
PLACE switch AT #6000 :
```

```
PLACE dac1 AT #6100 :
```

```
PLACE disable AT #6700 :
```

```
SEQ
```

```
WHILE TRUE
```

```
SEQ
```

```
in ? current -- wait for DAC output request
```

```
kill.power := (switch /\ 1) = 1 -- check front panel switch
```

```
IF
```

```
kill.power
```

```
dac1 := 0
```

```
((INT current) < #1000)
```

```
dac1 := (INT current)
```

```
TRUE
```

```
dac1 := #0FFF
```

```
disable := 0 -- reset watchdog timer
```

```
:
```

```
-----  
-- Process and channel Configuration  
  
-- Transputer link addresses --  
VAL link0out IS 0 :  
VAL link1out IS 1 :  
VAL link2out IS 2 :  
VAL link3out IS 3 :  
VAL link0in IS 4 :  
VAL link1in IS 5 :  
VAL link2in IS 6 :  
VAL link3in IS 7 :  
  
-- Define inter-processor links --  
CHAN OF INT host.to.controller :  
CHAN OF ANY controller.to.host :  
  
CHAN OF INT16 controller.to.DAC, controller.to.ADC :  
CHAN OF INT16 ADC.to.controller, DAC.to.controller :  
  
-- Configure process allocation and links --  
PLACED PAR  
  
PROCESSOR 1 T8 -- Controller  
  PLACE controller.to.host AT link0out :  
  PLACE host.to.controller AT link0in :  
  
  PLACE controller.to.ADC AT link1out :  
  PLACE ADC.to.controller AT link1in :  
  
  PLACE controller.to.DAC AT link2out :  
  PLACE DAC.to.controller AT link2in :  
  
  Maglev.Controller( host.to.controller, controller.to.host,  
                    ADC.to.controller, controller.to.ADC,  
                    DAC.to.controller, controller.to.DAC )  
  
PROCESSOR 2 T2 -- ADC  
  PLACE ADC.to.controller AT link2out :  
  PLACE controller.to.ADC AT link2in :  
  
  ADC.driver( controller.to.ADC, ADC.to.controller )  
  
PROCESSOR 3 T2 -- DAC  
  PLACE DAC.to.controller AT link1out :  
  PLACE controller.to.DAC AT link1in :  
  
  DAC.driver( controller.to.DAC, DAC.to.controller )
```

## E.4 Single electromagnet real-time simulator

```

REAL64 FUNCTION Magnet.Force( VAL REAL64 airgap, current )

  VAL num.current.points IS 8 :
  VAL [num.current.points] REAL64 current.point IS
    [ 2.0 (REAL64), 3.0 (REAL64), 4.0 (REAL64), 5.0 (REAL64),
      6.0 (REAL64), 7.0 (REAL64), 8.0 (REAL64), 9.0 (REAL64) ] :

  VAL num.airgap.points IS 4 :
  VAL [num.airgap.points] REAL64 airgap.point IS
    [ 0.002 (REAL64), 0.003 (REAL64), 0.004 (REAL64), 0.005 (REAL64) ] :

  VAL [num.airgap.points][num.current.points] REAL64 force.table IS
    [[24.0 (REAL64), 75.0 (REAL64), 140.0 (REAL64), 212.0 (REAL64),
      300.0 (REAL64), 405.0 (REAL64), 540.0 (REAL64), 690.0 (REAL64)],
     [12.0 (REAL64), 46.0 (REAL64), 85.0 (REAL64), 132.5 (REAL64),
      185.0 (REAL64), 257.0 (REAL64), 357.5 (REAL64), 485.0 (REAL64)],
     [ 8.0 (REAL64), 20.0 (REAL64), 38.5 (REAL64), 69.0 (REAL64),
      108.0 (REAL64), 156.0 (REAL64), 212.5 (REAL64), 277.5 (REAL64)],
     [ 6.0 (REAL64), 12.5 (REAL64), 23.5 (REAL64), 42.0 (REAL64),
      63.0 (REAL64), 93.0 (REAL64), 125.0 (REAL64), 170.0 (REAL64)]]:

  INT current.index, airgap.index :
  REAL64 extra.current.ratio, extra.airgap.ratio :
  REAL64 corner.current, delta.current :
  REAL64 corner.airgap, delta.airgap :

  REAL64 F.Amin.Imin, F.Amax.Imin, F.Amin.Imax, F.Amax.Imax :
  REAL64 F.Amin, F.Amax, F :

  VALOF
  SEQ
  -- Find current index and delta

  current.index := 0
  SEQ each.current = 0 FOR num.current.points - 1
  IF
    (current >= current.point[ each.current ])
    current.index := each.current
  TRUE
  SKIP
  corner.current := current.point[ current.index ]
  delta.current := (current.point[ current.index + 1 ] - corner.current)
  extra.current.ratio := (current - corner.current) / delta.current

  -- Find airgap index and delta

  airgap.index := 0
  SEQ each.airgap = 0 FOR num.airgap.points - 1
  IF
    (airgap >= airgap.point[ each.airgap ])
    airgap.index := each.airgap
  TRUE
  SKIP
  corner.airgap := airgap.point[ airgap.index ]
  delta.airgap := (airgap.point[ airgap.index + 1 ] - corner.airgap)
  extra.airgap.ratio := (airgap - corner.airgap) / delta.airgap

  F.Amin.Imin := force.table[ airgap.index ][ current.index ]
  F.Amin.Imax := force.table[ airgap.index ][ current.index + 1 ]
  F.Amax.Imin := force.table[ airgap.index + 1 ][ current.index ]
  F.Amax.Imax := force.table[ airgap.index + 1 ][ current.index + 1 ]

  F.Amin :=F.Amin.Imin+((F.Amin.Imax-F.Amin.Imin)*extra.current.ratio)
  F.Amax :=F.Amax.Imin+((F.Amax.Imax-F.Amax.Imin)*extra.current.ratio)

  F := F.Amin + ( (F.Amax - F.Amin) * extra.airgap.ratio)
  RESULT F
:

```



```

-----
PROC Simulate.outside.world( CHAN OF INT16  ADC.in, ADC.out,
                             DAC.in, DAC.out,
                             CHAN OF control.p  trigger,
                             CHAN OF data.p    current,
                             [num.airgap.chans] CHAN OF data.p  airgap.chan,
                             [num.acc.chans]   CHAN OF data.p  acc.chan,
                             [num.ext.ref.chans] CHAN OF data.p  ext.ref.chan,
                             CHAN OF control.p  cycle.finished,
                             CHAN OF exception.p exception,
                             VAL REAL32       sampling.period )

-- Worldy constants --
VAL gravitational.acc IS 9.81 (REAL64) :

-- Load characteristics --
VAL mass IS 11.0 (REAL64) :

-- Magnet characteristics --
VAL L.magnet IS 0.011 (REAL64) :
VAL R.magnet IS 0.79 (REAL64) :
VAL F.min IS 1.0 (REAL64) :
VAL F.max IS 600.0 (REAL64) :

-- Current amplifier characteristics --
VAL k.I.err IS 33.0 (REAL64) :
VAL V.min IS -1.0 (REAL64) :
VAL V.max IS 24.0 (REAL64) :
VAL I.min IS 0.0 (REAL64) :
VAL I.max IS 15.0 (REAL64) :

-- Disturbance characteristics --
VAL airgap.min IS 0.002 (REAL64) :
VAL airgap.max IS 0.006 (REAL64) :
VAL airgap.start IS airgap.max :

-- Feedback sensors time constants and quantization --
VAL T.airgap.transducer IS 0.0005 (REAL64) :
VAL quanta.airgap IS 0.000005 (REAL64) :

VAL w.accel IS 234.4 (REAL64) :
VAL Z.accel IS 0.55 (REAL64) :
VAL quanta.accel IS 0.0005 (REAL64) :

VAL quanta.I IS 0.01 (REAL64) :

-- Simulation calculations --
REAL64 simulation.calculation.interval :
VAL num.steps IS 5 :

-- State variables (and others) --
BOOL final.trigger, final.current, start.converting :
REAL64 acceleration, accel.transducer, accel.tmp :
REAL64 velocity :
REAL64 airgap, airgap.transducer :
REAL64 I.demand, V.magnet, I.magnet, F.magnet :
REAL32 I.demand.real32, airgap.FB, accel.FB :

VAL kilo IS 1000.0 (REAL32) :
VAL ext.ref.sensor.mm IS 3 :
REAL32 ext.gap.ref.mm, ext.gap.ref :
BOOL ext.gap.ref.aborted :

```

```

-----
PROC Integrate( REAL64 y,
               VAL REAL64 x )
  SEQ
  y := y + (x * simulation.calculation.interval)
  :

```

```

PROC Real.pole( REAL64 y,
               VAL REAL64 x, time.constant )

-- G(s) = 1/(1+sT) --

REAL64 t :
SEQ
  t := (x-y) / time.constant
  Integrate( y, t )
:

PROC Complex.pole( REAL64 y, y.dot,
                  VAL REAL64 x, p, q )

-- G(s) = 1/(p.s^2 + q.s + 1) --

REAL64 t :
SEQ
  t := ((x-y) - (q*y.dot)) / p
  Integrate( y.dot, t )
  Integrate( y, y.dot )
:

PROC Double.integrate( REAL64 x, x.dot, x.dot.dot,
                      VAL REAL64 lower.limit, upper.limit )

SEQ
  Integrate( x.dot, x.dot.dot )
  Integrate( x, x.dot )
  IF
    (x < lower.limit)
    SEQ
      x := lower.limit
      x.dot := 0.0 (REAL64)
      x.dot.dot := 0.0 (REAL64)
    (x > upper.limit)
    SEQ
      x := upper.limit
      x.dot := 0.0 (REAL64)
      x.dot.dot := 0.0 (REAL64)
  TRUE
  SKIP
:

-----

REAL64 FUNCTION Bound( VAL REAL64 x, lower.limit, upper.limit )

REAL64 x.bounded:

VALOF
  IF
    (x < lower.limit)
    x.bounded := lower.limit
    (x > upper.limit)
    x.bounded := upper.limit
  TRUE
  x.bounded := x
  RESULT x.bounded
:

-----

REAL64 FUNCTION Quantize( VAL REAL64 x, quanta )

INT32 i:

VALOF
  i := INT32 ROUND (x / quanta)
  RESULT (REAL64 ROUND i) * quanta
:

```

```

SEQ
-- Initialise state variables --
I.magnet := 0.0 (REAL64)

airgap := airgap.start
velocity := 0.0 (REAL64)
acceleration := 0.0 (REAL64)

accel.tmp := 0.0 (REAL64)
accel.transducer := 0.0 (REAL64)
airgap.transducer := airgap.start

airgap.FB := REAL32 ROUND airgap.start
accel.FB := 0.0 (REAL32)

simulation.calculation.interval :=
    (REAL64 ROUND sampling.period) / (REAL64 ROUND num.steps)

final.current := FALSE
WHILE NOT final.current
    ALT
    -- Feedback signals wanted ? --
    trigger ? final.trigger; start.converting
    PAR
    SEQ -- Read in external airgap reference --
    Read.ADC( ADC.in, ADC.out,
        ext.ref.sensor.mm, ext.gap.ref.mm, ext.gap.ref.aborted)
    ext.gap.ref := (ext.gap.ref.mm / kilo)
    PAR i = 0 FOR num.ext.ref.chans
        ext.ref.chan[i] ! final.trigger; ext.gap.ref
    Write.DAC( DAC.in, DAC.out, -(airgap.FB * kilo))

    PAR i = 0 FOR num.airgap.chans
        airgap.chan[i] ! final.trigger; airgap.FB
    PAR i = 0 FOR num.acc.chans
        acc.chan[i] ! final.trigger; accel.FB

    -- Magnet current demand available ? --
    current ? final.current; I.demand.real32
    SEQ
    I.demand := Bound( REAL64 ROUND I.demand.real32, I.min, I.max )
    I.demand := Quantize( I.demand, quanta.I )

    -- Simulate magnet acceleration and airgap --
    SEQ each.step = 1 FOR num.steps
    SEQ
    V.magnet := Bound( k.I.err*(I.demand-I.magnet), V.min, V.max)

    Real.pole( I.magnet, V.magnet/R.magnet, L.magnet/R.magnet )

    F.magnet := Bound(Magnet.Force(airgap,I.magnet), F.min, F.max)

    acceleration := gravitational.acc - (F.magnet / mass)

    Double.integrate( airgap, velocity, acceleration,
        airgap.min, airgap.max )
    Complex.pole( accel.transducer, accel.tmp, acceleration,
        1.0(REAL64)/(w.accel*w.accel),
        2.0(REAL64)*(Z.accel/w.accel) )
    Real.pole( airgap.transducer, airgap, T.airgap.transducer )

    accel.FB := REAL32 ROUND Quantize(accel.transducer, quanta.accel)
    airgap.FB := REAL32 ROUND Quantize(airgap.transducer, quanta.airgap)

    cycle.finished ! final.current; TRUE

Write.DAC( DAC.in, DAC.out, 0.0 (REAL32) )
:

```

## E.5 Transputer network monitor

```
#####
Network Monitor

Features: - Transfers TDS keyboard and screen across to root transputer
          - Communication may be finished at any time using control-A

Author: Neil McLagan
Version: 1.0
Last modified: 21-12-88
#####

VAL link0out IS 0 :
VAL link1out IS 1 :
VAL link2out IS 2 :
VAL link3out IS 3 :
VAL link0in IS 4 :
VAL link1in IS 5 :
VAL link2in IS 6 :
VAL link3in IS 7 :

[1] CHAN OF ANY from.app :
CHAN OF ANY to.app :
PLACE from.app AT link2in :
PLACE to.app AT link2out :

#USE userio
#USE reinit
#USE interf

PROC Monitor([1] CHAN OF ANY from.app,
             CHAN OF ANY to.screen, to.app,
             CHAN OF INT from.keyboard,
             CHAN OF ANY from.kernel, to.kernel )

VAL k.get.abort.state IS 15 :
VAL abort.polling.interval IS 15 : -- 1 ms

INT key.to.app, dummy, time, state :
BOOL aborted, going :
CHAN OF INT kill, stop :
TIMER clock :

SEQ
  newline( to.screen )
  write.text.line( to.screen, "#####" )
  write.text.line( to.screen, "## Use control-A to return to TDS ##" )
  write.text.line( to.screen, "#####" )
  newline( to.screen )

  aborted := FALSE
  going := TRUE
  PRI PAR
  PAR
    WHILE going
    SEQ
      clock ? time
      clock ? AFTER (time PLUS abort.polling.interval)
      to.kernel ! k.get.abort.state
      from.kernel ? state
      going := (state = 0)
      IF
        going
        SKIP
      TRUE
      PAR
        stop ! 1
        kill ! 1
    WHILE NOT aborted
    ALT
      from.keyboard ? key.to.app
      [4] BYTE key.to.app.table RETYPES key.to.app :
      OutputOrFail.c( to.app, key.to.app.table, kill, aborted )
      kill ? dummy
      aborted := TRUE
    scrstream.multiplexor( from.app, to.screen, stop )
  :
SEQ
  Monitor( from.app, screen, to.app, keyboard, from.kernel, to.kernel )
```

## E.6 Transputer network data logger

```
#####
Network Logger

Features: - Transfers data from root transputer to DOS file

Author: Neil McLagan
Version: 1.0
Last modified: 26-12-88

#####

VAL link0out IS 0 :
VAL link1out IS 1 :
VAL link2out IS 2 :
VAL link3out IS 3 :
VAL link0in IS 4 :
VAL link1in IS 5 :
VAL link2in IS 6 :
VAL link3in IS 7 :

[1] CHAN OF ANY from.app :
CHAN OF ANY to.app :

PLACE from.app AT link2in :
PLACE to.app AT link2out :

#USE userio
#USE interf
#USE krnlhdr

PROC Logger([1] CHAN OF ANY from.app,
            CHAN OF ANY to.screen, to.app,
            CHAN OF INT from.keyboard,
            CHAN OF ANY from.filer, to.filer )

VAL [] BYTE path.name IS "D:\Neil\spreads\" :
[abs.id.size] BYTE file.name :
INT name.len, result, anychar :

CHAN OF ANY buffer.chan :
[30000] BYTE buffer :

SEQ
  newline( to.screen )
  write.text.line( to.screen, "Network Data Logger" )
  write.text.line( to.screen, "======" )
  newline( to.screen )
  newline( to.screen )

  write.full.string( to.screen, "Enter data file name: " )
  read.echo.text.line( from.keyboard, to.screen, name.len, file.name, anychar )
  name.len := name.len - 1

  newline( to.screen )
  write.text.line( to.screen, "Getting data from root transputer ..." )
  newline( to.screen )

PAR
  to.app ! INT 'B'
  SEQ
    scrstream.sink( from.app[0] )
    scrstream.sink( from.app[0] )
    scrstream.sink( from.app[0] )
    scrstream.to.array( from.app[0], buffer )

PAR
  to.app ! INT 'B'
  scrstream.sink( from.app[0] )

write.text.line( to.screen, "Writing data file to DOS ..." )
newline( to.screen )

PAR
  SEQ
    scrstream.from.array( buffer, buffer.chan )
    write.endstream( buffer.chan )
```

```
scrstream.to.server( buffer.chan, from.filer, to.filer,  
                    name.len, file.name, result )  
  
IF  
  (result = 0)  
    write.full.string( to.screen, "File transfer successful, file: " )  
  TRUE  
    write.text.line( to.screen, "File transfer failed, file:" )  
  
write.len.string( to.screen, name.len, file.name )  
newline( to.screen )  
newline( to.screen )  
  
write.full.string( to.screen, "Hit space to finish.")  
  
read.char( from.keyboard, anychar )  
  
:  
  
SEQ  
  Logger( from.app, screen, to.app, keyboard, from.filer, to.filer )
```