# LUND UNIVERSITY

## A Modula-2 Real-Time Scheduler

## Use and Implementation

Andersson, Leif

1989

Link to publication

*Total number of authors:*
1

# A Modula-2 Real-Time Scheduler
## Use and Implementation

Leif Andersson

| Department of Automatic Control Lund Institute of Technology P.O. Box 118 S-221 00 Lund Sweden | Document name INTERNAL REPORT |
|---|---|
| | Date of issue 1989-01-26 |
| | Document Number CODEN: LUTFD2/(TFRT–7414)/1-9/(1989) |

| Author(s) Leif Andersson | Supervisor |
|---|---|
| | Sponsoring organisation |

Title and subtitle
A Modula-2 Real-Time Scheduler—Use and Implementation

Abstract

Describes a simple foreground/background scheduler programmed in Modula-2. An example of its use is given, and the complete implementation is shown.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

| Language English | Number of pages 9 | Recipient's notes |
|---|---|---|
| Security classification | | |

# 1.  Introduction

The simplest possible base for a real-time program is the Foreground/Background Scheduler. Such programs have been used at the Department of Automatic Control since the late seventies [Matsson 1978]. This report describes an implementation in Modula-2 for the IBM PC/AT and compatibles. Also included is a very simple program that gives an example of its use. Section 2 contains the library description for the Scheduler and section 3 the example program. Section 4 contains the implementation module of the Scheduler, and sections 5 contains the definition and implementation modules of a lowest-level Clock Interrupt Driver that is used by the Scheduler. There is a bibliography in the last section.

# 2.  Scheduler Library Description

DEFINITION MODULE Scheduler;
   A simple foreground/background scheduler
EXPORT  QUALIFIED Schedule, Run, Stop, Lag;

PROCEDURE Schedule(FG:PROC);
   Initializes the scheduler and specifies which procedure to be called on each sampling instance. No other procedures of the module should be called before this.

PROCEDURE Run(period: CARDINAL);
   Make the scheduler call the foreground procedure specified in Schedule with an interval specified by period, expressed in milliseconds.

PROCEDURE Stop;
   Cancels the calling of the specified procedure. The procedure Run may be called later to resume the scheduling.

PROCEDURE Lag(): CARDINAL;
   If the foreground procedure is still running when it is due to be called the next time, then the scheduler will instead increment an internal counter. The foreground routine will then be recalled immediately on return and the counter decremented. Thus, if the foregroud routine is too long, then the counter will start to accumulate. The procedure Lag will return this counter, enabling the foreground procedure to check that it doesn't lag behind.

END Scheduler.

# 3.  The Example Program

As an example for the Foreground/Background Scheduler we choose a very simple proportional regulator. The only parameter that can be changed is the gain. The program contains a procedure Opcom that runs in a loop accepting real numbers, and a procedure Regul that is the regulator proper, and that is run regularly by the Scheduler. In order to simplify Opcom we use the convention that a gain with an absolute value less than 0.01 is a signal to exit the program.

A general problem with real-time programs is the sharing of variables between processes in a proper way so that they are not garbled by simultaneous access. The solution in this case is as follows. There are two copies of the variable, one that is changed exclusively by Opcom and one by Regul. There is also a flag, which is set by Opcom when it has changed the variable. Regul runs regularly, and each time it has performed its normal task, it checks the flag and copies the variable if the flag is set and then clears the flag. There is a very small probability that Opcom wants to change its variable before Regul has taken care of the previous change, and therefore Opcom waits for the flag to be reset before it changes its copy of the variable.

```
MODULE ExampleSchedule;

FROM Scheduler IMPORT Schedule, Run, Stop;

FROM ConvReal IMPORT StringToReal;

FROM AnalogIO IMPORT ADIn, DAOut;

FROM BIOSterminal IMPORT WriteString, ReadString, WriteLn;
```
The module BIOSterminal should be used instead of Terminal so that DOS does not interfere with the real-time operations.

| | |
|---|---|
| `VAR OpcomK: REAL;` | Gain variable used by Opcom. |
| `VAR RegulK: REAL;` | Gain variable used by Regul. The value of OpcomK is transferred here by Regul. |
| `VAR change: BOOLEAN;` | Flag to signal to Regul that a parameter change has taken place. |
| `CONST period = 10;` | The sampling period in milliseconds. |

```
TYPE string = ARRAY [0..79] OF CHAR;
PROCEDURE Opcom;
VAR s: string; pos: CARDINAL; val: REAL;
BEGIN
  REPEAT
    WriteString("> ");
    ReadString(s); WriteLn;
    pos:=0;
    StringToReal(s,pos,val);
    IF pos > 0 THEN
      WHILE change DO ; END;
```
> At this point we have an acceptable number. The WHILE-loop is to handle the (very rare) situation where a previous change has not yet been taken by Regul.
```
      OpcomK := val;
```
> Since any previous change has been taken care of it is OK to set the variable.
```
      change := TRUE;
```
> Now set the flag so that Regul can transfer the value to its variable.
```
    ELSE
```
> A bad number has been detected. Just complain and continue the loop.
```
      WriteString("Error: Bad number."); WriteLn;
    END;
```

```
      UNTIL ABS(OpcomK) < 0.001;
END Opcom;

PROCEDURE Regul;
VAR r, y, u: REAL;
BEGIN
   The following four statements constitute a very simple proportional regula-
   tor.
   r := ADIn(0);
   y := ADIn(1);
   u := RegulK*(r-y);
   DAOut(0,u);
      We now test if Opcom has set the flag, and if so transfer the variable value.
      Note that since Regul cannot be interrupted by Opcom there is no risk
      that Opcom can access the variables until Regul has finished.
   IF change THEN
      RegulK := OpcomK;
      change := FALSE;
   END;
END Regul;

BEGIN
   change := FALSE;
   OpcomK := 1.0; RegulK := 1.0;
   Schedule(Regul);
   Run(period);
   Opcom;
   Stop;
END ExampleSchedule.
```

## 4.  Scheduler Implementation

```
IMPLEMENTATION MODULE Scheduler;

FROM SYSTEM IMPORT
   ENABLE, DISABLE, CODE, ADDRESS, SETREG, ADR, DS, BX;
FROM InitError IMPORT Trap;
   The procedure Trap is an error-message-and-exit routine.

FROM FloatingUtilities IMPORT Float;
IMPORT ClockInterrupts;

CONST TICK=1;   Tick time in milliseconds

VAR
   ForeGround: PROC;   The procedure to be called on each sampling in-
                       stance. Set by the procedure Schedule.
   running: BOOLEAN;   Flag to indicate if a foreground process should run
                       or not. Changed by Run and Stop procedures.
   period: CARDINAL;   The sampling period, i.e. the number of ticks be-
                       tween each call of the foreground process. Set by the
                       procedure Run.
```

```
    time: CARDINAL;        The actual time within the period. This variable
                           is incremented each tick and compared to period.
                           When they are equal, the foreground is called and
                           time is reset.
    lagging: CARDINAL;     A counter for the number of times the foreground
                           was still active when it was due the next time. See
                           procedure Ticker.
    FParea: ARRAY [0..47] OF CARDINAL; Save area for the FP registers.
    started: BOOLEAN;      Flag set when procedure Schedule is called.

(*$R-*) (*$S-*) (*$T-*)

PROCEDURE Schedule(FG: PROC);
    Initialization procedure. It checks that it isn't called twice, sets the
    ForeGround procedure variable and starts the clock interrupt driver.
BEGIN
  IF started THEN
    Trap('Scheduler: Schedule called twice.');
  END;
  started := TRUE;
  ForeGround := FG;
  ClockInterrupts.Init(Ticker,Float(TICK));
END Schedule;

PROCEDURE Run(p: CARDINAL);
    Sets period from the input parameter, and sets the flag running so that the
    foreground will be called with the proper interval.
BEGIN
  IF NOT started THEN
    Trap('Scheduler: Run called before Schedule.')
  END;
  IF running THEN
    Trap('Scheduler: Run called twice without Stop.')
  END;
  DISABLE;
  time:=0;
  period := p;
  running:= TRUE;
  ENABLE;
END Run;

PROCEDURE Stop;
    Resets the running flag, which means that the foreground will no longer be
    called.
BEGIN
  IF NOT started THEN
    Trap('Scheduler: Stop called before Schedule.')
  END;
  IF NOT running THEN
    Trap('Scheduler: Stop called twice without Run.')
  END;
  DISABLE;
  running := FALSE;
```

```
    lagging := 0;
    ENABLE;
END Stop;

PROCEDURE Lag(): CARDINAL;
  Returns the variable lagging
BEGIN
  RETURN lagging;
END Lag;

PROCEDURE SaveFloat;
  Saves the floating point registers
VAR a: ADDRESS;
BEGIN
  a:=ADR(FParea);
  SETREG(DS,a.SEGMENT);
  SETREG(BX,a.OFFSET);
  (* FSAVE [BX] *)  CODE(0DDH,037H);
END SaveFloat;

PROCEDURE RestoreFloat;
  Restores the floating point registers
VAR a: ADDRESS;
BEGIN
  a:=ADR(FParea);
  SETREG(DS,a.SEGMENT);
  SETREG(BX,a.OFFSET);
  (* FRSTOR [BX] *)  CODE(0DDH,027H);
END RestoreFloat;

PROCEDURE Ticker;
```

This is the main workhorse of the scheduler. It is called every clock tick by the clock interrupt driver. It counts ticks until the foreground is due, then it saves the floating point registers, calls the foreground procedure, and restores the floating point registers. There is also some interlocking, handled with the global variable lagging, to ensure that the foreground is not called while it is still running.

```
BEGIN
  IF NOT running THEN RETURN END;
  INC(time,TICK);
  IF time >= period THEN
    time := time - period;
    INC(lagging);
```

The variable lagging is 0 when everything starts. It is then incremented above, and decremented below. If it has a value > 1 here, then we arrive here *before* we have finished the foreground procedure the previous time. We should thus *not* call the foreground. It is instead recalled when it returns, because of the while statement.

```
    IF lagging = 1 THEN
      SaveFloat;
      WHILE lagging > 0 DO
        ENABLE;
        ForeGround;
```

```
            DISABLE;
            DEC(lagging);
          END;
          RestoreFloat;
        END;
      END;
  END Ticker;

BEGIN
  started := FALSE;
  running := FALSE;
  lagging := 0;
END Scheduler.
```

# 5.  ClockInterrupt Definition and Implementation

```
DEFINITION MODULE ClockInterrupts;
```
Low level clock interrupt driver.

```
EXPORT QUALIFIED Init;
```

```
PROCEDURE Init(P: PROC; tick: REAL);
```
Initialization procedure.

P     the procedure to be called on each clock interrupt.

tick the clock interrupt period expressed in ms.

```
END  ClockInterrupts.
IMPLEMENTATION MODULE ClockInterrupts;
```
The module ClockInterrupts uses the system clock of the computer to give interrupts regularly. The system clock normally interrupts ca. 18 times/second ($2^{64}$ times/hour). The hardware clock registers may be changed to interrupt at a higher rate, which is utilized here. Furthermore, the clock interrupt vector is changed so that a procedure in this module handles the interrupt. In order to maintain the system software clock on time the interrupt routine maintains a counter so that the standard interrupt routine may be called with the correct frequency. In order to call the standard interrupt routine, the original interrupt vector must be copied to an auxiliary software vector. An arbitrary choice of vector 229 has been made. If conflicts should arise, this number appears in one and only one place, in the CONST section below.

```
FROM SYSTEM IMPORT CODE, ADDRESS, OUTBYTE, DISABLE, ENABLE;
FROM Devices IMPORT SaveInterruptVector, RestoreInterruptVector;
FROM RTSMain IMPORT InstallTermProc;
FROM FloatingUtilities IMPORT Round;

CONST
  SavedClockVector = 229;   Auxiliary software interrupt vector
  BaseFrequency = 1193.18;  Frequency driving the counter/timer
  TCC = 043H;               Timer/counter control word
  TC0 = 040H;               Timer 0
  ClockMode = 036H;         Clock Mode 3, 16 bits, binary
```

```
VAR
    period: CARDINAL;        The value to set in the hardware counter/timer.
                             Also used to determine when to call the sys-
                             tem clock interrupt routine. Set once by Init
                             procedure.
    timer: CARDINAL;         The counter for calling the system clock interrupt
                             routine.
    ClockProcedure: PROC;    The procedure to call on each clock interrupt.
```

(*$O+*)(*$R-*)(*$S-*)(*$T-*)
PROCEDURE ClockInterrupt;
    This is the Clock Interrupt Service Routine. Is job is to save the registers
    and call the higher level clock interrupt handler. It also maintains a counter
    so that the original Interrupt Service Routine is called at approximately the
    correct interval.
BEGIN

```
(*      PUSH AX *) CODE(050H);
(*      PUSH CX *) CODE(051H);
(*      PUSH DX *) CODE(052H);
(*      PUSH BX *) CODE(053H);
(*      PUSH SI *) CODE(056H);
(*      PUSH DI *) CODE(057H);
(*      PUSH DS *) CODE(01EH);
(*      PUSH ES *) CODE(006H);
```

At this point all registers are saved. The purpose of the next statement is
to increment the counter, but also to set the Carry flag if the increment
overflows. The carry is then tested in the next CODE-statement. This is ugly
programming, but it works provided there is only MOV-instructions after the
ADD-instruction in the Modula-statement. This should be checked with each
new version.

```
        timer:=timer+period;
(*      JNC L1 *)                  CODE(073H, 004H);
(*      INT SavedClockVector *) CODE(0CDH, SavedClockVector);
(*      JMP L2 *)                  CODE(0EBH, 004H);
(* L1: SENDEOI *)                  CODE(0B0H, 020H, 0E6H, 020H);
(* L2:          *)
```

All interrupt administration is done. Call the higher level interrupt routine
and restore the register.

```
        ClockProcedure;
(*      POP ES *) CODE(007H);
(*      POP DS *) CODE(01FH);
(*      POP DI *) CODE(05FH);
(*      POP SI *) CODE(05EH);
(*      POP BX *) CODE(05BH);
(*      POP DX *) CODE(05AH);
(*      POP CX *) CODE(059H);
(*      POP AX *) CODE(058H);
(*      LEAVE  *) CODE(0C9H);
(*      IRET   *) CODE(0CFH);
END ClockInterrupt;
```

```
PROCEDURE Init(P: PROC; tick: REAL);
VAR IV: ADDRESS; phigh, plow: CARDINAL;
BEGIN
  InstallTermProc(Stop);
  ClockProcedure:=P;
    Compute the number of clock cycles between each interrupt. We need it
    in high-byte/low-byte form.
  period:=Round(tick * BaseFrequency);
  plow:=period MOD 256;
  phigh := period DIV 256;
    Save the original clock interrupt vector and set the vector to point to the
    ClockInterrupt procedure of this module. The rest of the initialization
    is done with interrupts off.
  DISABLE;
  SaveInterruptVector(8,IV);
  RestoreInterruptVector(SavedClockVector,IV);
  RestoreInterruptVector(8,ADDRESS(ClockInterrupt));
    We reprogram the system timer/counter to give interrupts with the rate
    determined by tick. The reason for the do-nothing Delay procedure is
    that things may malfunction if two OUT-instructions are placed too close
    to each other.
  OUTBYTE(TCC,ClockMode); Delay;
  OUTBYTE(TCO,plow); Delay;
  OUTBYTE(TCO,phigh); Delay;
  ENABLE;
END Init;

PROCEDURE Stop;
VAR IV: ADDRESS;
BEGIN
  DISABLE;
    Reset the clock interrupt vector
  SaveInterruptVector(SavedClockVector,IV);
  RestoreInterruptVector(8,IV);
    Reset the system timer/counter to its normal value of 18 interrupts per
    second.
  OUTBYTE(TCC,ClockMode); Delay;
  OUTBYTE(TCO,0); Delay;
  OUTBYTE(TCO,0); Delay;
  ENABLE;
END Stop;

PROCEDURE Delay;
  Does nothing
BEGIN
END Delay;

END ClockInterrupts.
```

# 6. References

Matsson, S. E. (1978): "A Simple Real-Time Scheduler," CODEN: LUTFD2/TFRT-7156, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Brück, D. M., "A Foreground/Background Real-Time Scheduler for the IBM AT," CODEN: LUTFD2/TFRT-7393, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.