A Foreground/Background Real-Time Scheduler for the IBM AT

Brück, Dag M.

1988

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT–7393)/1-029/(1988)

A Foreground/Background Real-Time Scheduler for the IBM AT

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
July 1988

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Report

Date of issue
1988-07-07

Document Number
CODEN:LUTFD2/(TFRT-7393)/1-029/(1988)

Author(s)
Dag M. Brück

Supervisor

Sponsoring organisation

Title and subtitle
A Foreground/Background Real-Time Scheduler for the IBM AT

Abstract

The combination of general purpose software and commonly available hardware is too slow for many control applications. The purpose of this report is to analyze some of the performance problems (given a fixed hardware), and to demonstrate a solution.

The foreground/background scheduler described in this report achieves high real-time performance by reducing complexity and functionality. Context switch time is 56 μs. The PID controller used as a case study achieves a sampling rate of 2 kHz with the scheduler.

The scheduler, written in assembler, is designed for a controller in C, but can be adopted to other languages. The scheduler is also compared with a general purpose real-time kernel.

Key words
Real-time programming; Process control; Modula-2; C.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
29

Recipient's notes

Security classification

1. Introduction

Computer implementations of control systems often require performance that is quite difficult to achieve with commonly available hardware, and general-purpose software. Robotics is a typical example: the speed loop requires a sampling rate of at least 1 kHz, the position loop a sampling rate of about 100 Hz. Luckily, the controller for most control applications is very simple: it consists of a single regulator loop and some sort of man-machine interface. The purpose of this report is to:

- Determine what operations in a controller are time consuming and therefore limit the sampling rate.
- Present one possible solution — a real-time scheduler for fast control applications. The scheduler is written in assembly language, and has been designed for a controller written in C.
- Present some practical results. A PID controller has been implemented, and an upper limit on the sampling rate has been determined.

The obvious way to get high performance is to buy faster hardware: a more powerful micro processor, a special-purpose control system, or even multi-processor architectures (HyperCubes, Transputers, etc.). This study is limited to a fixed hardware which is quite common — the IBM AT — and concentrates on the software issues.

In this report, the term *regulator* represents the computer realization of a control algorithm. The term *controller* denotes a complete program which also contains code for calculating regulator coefficients from the user's regulator parameters, some basic operator communication, and initializations.

2. Analysis of the controller

In order to analyze (and hopefully increase) the performance of the controller, we must look at a number of problem areas:

- The regulator algorithm.
- A/D and D/A conversion.
- Operator communication and graphics.
- The real-time kernel.
- The programming language.

Each of these five areas will be studied separately below. The basic characteristics will be outlined and improvements suggested.

The regulator algorithm

There is not much to do about the basic regulator algorithm, once it has been chosen. Two obvious optimizations are possible:

- Calculating regulator coefficients from the user's regulator parameters.
- Rearranging the order of calculations: in a PID regulator, the integral part can be calculated after the new output. This reduces the delay from input to output, but does not increase the sampling rate.

More drastic actions may also be called for. For example, it is possible to refrain from floating-point calculations and to use a fixed-point regulator instead; this approach works fine for PID regulators. Fixed-point arithmetic is difficult to use unless built into the language, see Section 5.

It is very important to identify truly time-critical operations, and to simplify the regulator as much as possible. For example, plotting should not be done by the regulator, but by a process at lower priority. In some cases, for example, when the scheduler described in this report is used, it may be very difficult to plot at low priority.

A/D and D/A conversion

The overhead associated with A/D and D/A conversion is usually determined by the hardware. The D/A conversion time is not critical, but with typical sample-and-hold A/D converters, the program must wait for the A/D converter to stabilize. This period could be used for other calculations, but so is not done here. In many industrial controllers, A/D conversion is started by the program, but actual data transfer is done by the converter using DMA.

It may also be necessary to convert between the user's engineering units and the range of the converters (-2048 to $+2047$ for a 12-bit converter), which may represent a voltage between -10 and $+10$ V. Conversion between integer and floating-point numbers is particularly time-consuming.

Operator communication

Operator communication can also be regarded as a real-time task, although with lesser demands on response time. Drawing graphics usually requires a lot of CPU resources, which are taken from the total available, even if executed at a lower priority. Graphics should not be attempted in a high-performance regulator. General purpose graphics systems (for example, GKS) are often too slow, and other packages with less functionality must be chosen.

Due to lack of hardware support, some simple graphics operations may require inordinate real-time support in software. For example, the mouse on an IBM AT should be tracked by a separate process which polls the mouse position at 50 Hz.

Overhead in the kernel

A proper kernel is flexible and have powerful real-time primitives. A typical kernel is the Modula-2 kernel developed by Leif Andersson [Andersson, 1988]. It provides a general process concept, priority levels and round robin scheduling, semaphores, events, time-outs, and interrupt processes. There is additional support for message passing using mailboxes, graphics and mouse.

This powerful real-time environment does not come for free. In a small example, a single process requested invocation at every tick of the system clock (Listing 1). This meant going back and forth between the user process and the null process (two context switches), requiring 800 μs. The time spent in the kernel was measured by looking at a signal with an oscilloscope. Every time the kernel was entered, a 1 V signal was generated on one of the D/A channels; the signal was removed just before leaving the kernel. The kernel is implemented almost without exception in Modula-2, and context switch is by means of `TRANSFER`. The machine code generated for the kernel routines looks reasonably efficient, so one cannot help asking: what does a `TRANSFER` in Modula-2 do that takes 400 μs?

Two strategies are available to decrease kernel overhead: writing a more efficient kernel with similar functionality, or simplifying the kernel at the expense of decreased functionality. The latter approach was taken in this study. Many

```
MODULE Sched;
(* Rescheduling of a process using the Modula-2 kernel *)

FROM Kernel IMPORT MaxPriority, Time, IncTime, WaitUntil,
  GetTime, SetPriority, InitKernel, CreateProcess;
FROM Terminal IMPORT WriteString, WriteLn;

PROCEDURE Process;
VAR   now : Time;
BEGIN
  SetPriority(10);
  GetTime(now);
  LOOP
    IncTime(now, 10);
    WaitUntil(now);
  END;
END Process;

BEGIN
  WriteString('Starting.  Waiting time 10 ms'); WriteLn;
  InitKernel;
  CreateProcess(Process, 2000);
  SetPriority(MaxPriority);
END Sched.
```

Listing 1. Program for measuring context switch time in Modula-2 kernel.

useful control applications not even need general purpose processes. The simplest possible scheme is called foreground/background scheduling. The design and implementation of a scheduler is discussed in Section 3, its performance is analyzed in Section 4, and an application using the scheduler is analyzed in Section 5.

Language considerations

Of the languages used in this study, Modula-2 is a good language for control applications. It is one of the few widely available languages where necessary low-level operations (such as, processes, mutual exclusion, input/output) are part of the language definition. Modula-2 has no facilities for process scheduling or processes waiting for a specified period of time. Regrettably, the user has little knowledge, and even less control, over some internal operations, notably TRANSFER; this precludes the use of Modula-2 for maximum performance applications on the IBM AT. In all other respects, the Logitech implementation of Modula-2 is very good.

On the other hand, C is not a real-time programming language, but C is very efficient and its run-time behaviour is easily predictable (partly because the user must provide all real-time primitives himself). Writing a kernel and interfacing with assembler is quite easy in C.

Even within the bounds of a single programming language, many improvements can be made. If optimization is addressed in an orderly manner, the resulting program may remain comprehensible. A number of techniques are available; low-level optimizations are important in control applications.

3. The foreground/background scheduler

This section describes the implementation of a foreground/background scheduler. Much of what is said here concerns a particular implementation on an IBM AT, and may be of limited application elsewhere. The code for the scheduler is listed in Appendix A.

What is a scheduler?

A scheduler can be regarded as a $1\frac{1}{2}$ process real-time kernel. The *foreground* process is a procedure that is called at fixed time intervals by the scheduler. The foreground process is always allowed to run to completion. When the foreground process is not running, the *background* process is said to be running, which is an ordinary sequential program. Typically, the foreground is the implementation of a regulator loop, while the background handles operator communication.

When the foreground process is started, processor status of the background process must be saved. A property of the foreground/background scheduler is that it uses only one stack: processor status is saved, and the foreground process is executed, using the stack of the background process. Stack and processor status are restored to original form when the foreground process terminates. A kernel, on the other hand, maintains a separate stack for every process.

The foreground/background scheduler is an old concept and has been used in a large number of applications. A scheduler for Digital Equipment Corporation's LSI-11 is described by Mattsson [1978].

Functionality not provided

The scheduler achieves its performance through decreased functionality and complexity; a number of features available in a proper kernel, as for example the Modula-2 kernel, are missing.

There are no "real" processes, so there is no need for process priorities or round-robin scheduling. The foreground process interrupts the background process, and therefore has higher priority by definition. Process communication can easily be accomplished using shared variables (see Section 5), and because the foreground process always runs to completion, mutual exclusion with semaphores and events is not needed.

Because the real-time application is limited to the regular invocation of the foreground process, interrupt processing and time-outs are uninteresting.

Implementation

In order to implement a foreground/background scheduler, the following three issues must be addressed:

- The scheduler needs a reliable time-reference with adequate resolution.
- The user must define the foreground process and the invocation interval.
- The scheduler must at regular intervals invoke the foreground process.

The time-reference is provided by a programmable hardware timer; the resolution of the timer is 0.84 μs. The timer is programmed to generate interrupts at a certain multiple of the resolution (this is called a tick). A tick is typically between 0.5 and 10 ms.

The application program initializes the scheduler by specifying what function should be invoked as the foreground process, and how often. The second parameter, called the period, is the number of ticks between foreground process invocations. This means that the hardware clock is scaled two times: Firstly, the scheduler's basic time unit (the tick) is generated by programming the timer. Secondly, the application can control the period between invocations of the foreground process, expressed in multiples of the tick.

When the scheduler is about to start the foreground process, the state of the background process is saved. The foreground process is assumed to use integer operations only, so floating point registers are not saved. If the foreground process for some reason has not finished after one period (i.e., the scheduler should have started it again), the lag is registered; the scheduler will immediately restart the foreground process in order to "catch up." Only one restart is done every time the foreground process is lagging, so lag does not accumulate.

The code of the scheduler has been divided into the following routines (see Appendix A):

`IntHandler` Scheduler interrupt handler, invoked for every timer interrupt. Updates the BIOS clock if necessary. Increments the tick counter, and checks if the foreground process should be started. Registers lag if necessary.

`FGstart` Saves processor status of the background process. Starts the foreground process at least once, or until no lag has been registered. Restores processor status and returns to the background process.

`BREAK_int` Interrupt handler for emergency stops. Invoked by the BIOS when CTRL-BREAK is pressed. Calls `reset` below, and terminates program.

`schedule` Called by the application program to initialize the scheduler. Saves the address of the function to execute as the foreground process. Initializes interrupt handlers and the hardware timer.

`reset` Called by the application program to disable the scheduler. Restores timer and original interrupt handlers.

`ADin` Returns measurement from specified A/D converter.

`DAout` Outputs signal to specified D/A converter.

It should be noted that certain operations cannot be performed in the foreground process. Floating point calculations are not possible, as explained above. BIOS routines can probably be called, but DOS routines cannot. Although ordinary C code is re-entrant, some routines in the C run-time library may not be re-entrant. Library routines with unknown time demands should of course be avoided.

Machine specific details

One of the early design constraints was to write the scheduler in assembler (Microsoft MASM) and to limit the scheduler to applications written in Microsoft C. As expected, C proved to be efficient and easy to use. Low-level optimizations (in particular in the PID regulator, Section 5) were easily checked in the object-code listings. The small memory model was chosen to get maximum performance. Interfacing C programs to assembler is well documented.

The IBM AT architecture is awkward, and the Intel 80286 gives an outdated impression compared to many other processors. Low-level program-

ming and interfacing with the hardware is error-prone; the machine must be rebooted very often because of the lack of protection mechanisms, so development is painfully slow. Low-level operations are not well documented.

A complication is that the scheduler must update the time-of-day clock in the BIOS at 18.2 Hz; the timer is normally used for this purpose. In emergency cases, it is possible to run the system without updating the clock.

An early version of the scheduler was supposed to use two different interrupt vectors depending on whether the foreground process or the background process was executing. This idea had to be dropped: the DOS routine for setting the interrupt vector is much too slow, and the address for setting the interrupt vector directly could not be found. The current version must test a flag before starting the foreground process.

Changing interrupt vectors is a serious business, and the system will crash unless everything is reset prior to program termination. A special handler for the CTRL-BREAK key provides a crude emergency stop. Other errors that terminate the program (for example, CTRL-C) are not handled.

An amusing observation is that if the interrupt vector is not reset, the system will still work after the program has terminated; the system will not crash until the next program is loaded into memory.

Future extensions

A number of extensions could be considered in future versions of the scheduler:

- For non-integer regulators, the floating point registers must be saved before starting the foreground process.
- The foreground process executes in the stack currently in use by the background process. If the background process is performing input/output, or otherwise is using DOS, the foreground process will sometimes execute in the DOS stack.

  The size of the DOS stack is unknown but probably quite small, so there is a real risk of stack overflow. For large foreground processes, the scheduler would have to change stacks.
- Currently, only the small memory model is supported. Other memory models are needed to easily handle more than 64 KB data and 64 KB code.
- The tick is currently compiled into the scheduler; more flexibility would be gained if the application program could set the length of the tick.

All these extensions are relatively simple to realize. Another suggested extension is to allow multiple foreground processes. Multiple foreground processes are not necessarily an advantage: the scheduler is intended for simple control applications, and a proper kernel should be used for complex applications.

An important question is "can we make the scheduler faster?" Restricted by the 80286 processor and MS-DOS, there is not much room for improvement; registers must be saved and set up, lag must be registered, etc. One improvement is to disable the BIOS clock (see Section 4).

A significant change is to stop counting interrupts from the hardware timer, and invoke the foreground process at every tick. This would make the scheduler simpler, but the maximum period between invocations would be limited by the range of the hardware timer to 55 ms. If multiple timers are cascaded, the tick could be increased.

Operation	w clock	w/o clock
Timer interrupt handler	18 μs	16 μs
Start of foreground process	38 μs	38 μs
Updating BIOS clock	76 μs	—

Table 1. Execution times with test signal.

Operation	w clock	w/o clock
Timer interrupt handler	9.5 μs	7.5 μs
Start of foreground process	33.5 μs	33.5 μs
Updating BIOS clock	76 μs	—

Table 2. Execution times without test signal.

Voltage	Time	Instructions
0	4.0 μs	11
non-zero	4.5 μs	13

Table 3. Test signal generation times.

4. Scheduler performance

This section presents the results from measuring the actual performance of the basic scheduler, plus execution times for the A/D and D/A converter routines used by the PID controller in Section 5.

Conditions

These tests were conducted on a Tandon AT (IBM AT compatible) with an 8 MHz Intel 80286 processor. The scheduler is written in assembler and the processes (foreground and background) in Microsoft C version 4.0, using the small memory model. The foreground process is executing an empty function, i.e., the test includes the overhead of invoking a C-function.

The measurements were performed by looking at a test signal with an oscilloscope. A test signal of 1V is available when the interrupt handler is executing. A test signal of 2V is available when the foreground process is executing. Some extra overhead associated with entering and leaving the interrupt handler (37 cycles, 4.6 μs at 8 MHz), which can not be seen on the test signal, has been added to the figures. The inaccuracy of the results is about ± 2 μs.

Results

Table 1 gives the execution times of the scheduler, both with time-of-day clock handling, and without. With the clock, the scheduler overhead is normally 56 μs to start the foreground process, and 132 μs when the BIOS clock must be updated.

Table 2 gives estimated execution times for a scheduler without test signal. The minimum overhead that can be expected, without test signal and clock update, is 41 μs. The time required for generating the test signal has been calculated by counting instructions of the `DAOUT` macro (Table 3).

```
#include "fb.h"

main ()
{
  while (1) {
    DAout(1, 1024);
    ADin(1);
    DAout(1, -1024);
  }
}
```

Listing 2. Program for measuring A/D and D/A conversion times.

Performance of A/D and D/A conversions

Listing 2 gives the program which was used to measure the performance of one A/D conversion and one D/A conversion. On this hardware, total time is 68 μs. Clearly, the A/D conversion alone requires around 60 μs.

5. Case study: the PID controller

In order to test the scheduler under realistic conditions, a fast PID controller was developed. Some interesting observations on regulator implementation and floating point performance will be presented, and a practical upper limit on the sampling rate has been determined.

The regulator algorithm

The basic PID algorithm can generally be expressed as follows:

$$u(t) = k\left(e(t) + \frac{1}{T_i}\int e(s)\,ds + T_d\frac{de}{dt}\right)$$

where u is the control variable, k is the gain, $e = r - y$ is the control error, r is the set point, and y is the measured value.

The actual computer realization has been rewritten as follows:

$$\begin{aligned} P &= k(br - y) \\ D &= a_d D - b_d(y - y_{\text{old}}) \\ v &= P + I + D \\ I &= I + b_i(r - y) + b_r(u - v) \\ y_{\text{old}} &= y \end{aligned}$$

Parameter b can be used to reduce overshoot at step changes [Åström, 1987]. The regulator is also provided with anti-windup:

$$u = \begin{cases} u_{\min}, & \text{if } v < u_{\min} \\ v, & \text{if } u_{\min} \le v \le u_{\max} \\ u_{\max}, & \text{if } v > u_{\max} \end{cases}$$

The regulator coefficients are calculated from the user's regulator parameters:

$$b_i = \frac{kh}{T_i} \qquad a_d = \frac{T_d}{T_d + Nh} \qquad b_r = \frac{h}{T_r} \qquad b_d = \frac{kT_dN}{T_d + Nh}$$

where h is the sampling time, T_i is the integration time, T_r is the tracking time, T_d is the derivative time, and N is the maximum derivative gain.

Controller operation

The controller consists of a foreground process which implements the regulator described above, and a background process which handles operator communication. The background process runs in an endless loop, performing the following operations in sequence:

- Ask the user for regulator parameters.
- Calculate internal regulator coefficients from the parameters.
- Switch the regulator over to the new coefficients.
- Print the new parameters and coefficients.

It is interesting to note that there is no real mutual exclusion problem in the controller; the regulator uses one set of coefficients while the background process operates on another set. The background process swaps coefficients by simply changing an index. Since the foreground process always runs to completion, there is no risk that it will get new coefficients in the middle of the regulator loop. The C code for the controller is listed in Appendix B.

Fixed-point arithmetic

Floating point arithmetic is normally slower than integer arithmetic, so a fixed point regulator was chosen for the controller.

A fixed point number represents a (fixed) subrange of real numbers with limited precision, and is implemented using integers. The number is scaled (normally by a power of two) in order to get as much precision as possible for a given range; there is a trade-off between range and precision, of course. Fixed point arithmetic is discussed in some detail by Young [1982].

Ada is one of the few programming languages where fixed point numbers are defined. In other languages the user must use an integer data type and take care of the peculiarities himself; there are quite a few:

- The range of all numbers must be carefully estimated and an appropriate scaling chosen. This is often a difficult task.
- The terms of an addition or a subtraction must be aligned to get the same scale factor before the operation.
- The operands of a multiplication or a division rarely need to be scaled, but the operation should be performed in double precision (multiplication of two 16-bit integers gives a 32-bit result).
- The result of an operation must be scaled to fit the destination. In particular, multiplication and division accumulate the scale factors of the operands.
- The range of all numbers should be checked, but this is often impossible due to performance constraints.
- All arithmetic operations should be checked for overflow, but this is impossible in many languages.

A uniform scale factor is used for all variables in the PID regulator in order to reduce complexity. The risk of overflow is avoided by using 32-bit integers. In the regulator code, the result of a multiplication is scaled, but all other effects of fixed point arithmetic have been eliminated. The regulator coefficients are scaled on input, and the analog signals are scaled by the converters.

Optimizations

Many techniques can be applied to make a program faster. The first problem is to find the "hot-spots" in the program; the second problem is to apply the most cost-effective transformation of the code. A very good source of advice is *Writing Efficient Programs* by Jon Louis Bentley [1982].

The controller is fortunately quite easy to optimize. Most important, the area of interest is only about 20 lines of code. The regulator code is also quite simple, with a minimum of logic. Therefore, most optimizations are quite low-level, but there are two cases where logic has "replaced" arithmetic.

Data types and scaling. The most important data structure (from the performance point of view) is the regulator coefficient table. Firstly, the scale factor was chosen as a power of two, so scaling could be made with arithmetic shifts. Unfortunately, shifting a long integer is translated into a subroutine call by the Microsoft C compiler. By using the scale factor 2^{16}, the compiler can emit a single `mov` instruction, rather than a shift.

Secondly, some calculations must be performed with 32-bit (long) integers to avoid overflow, so 32-bit integers are used through-out to minimize the number of conversions.

Using logic instead of arithmetic. One of the most time-consuming operations in the regulator is to multiply two numbers, so minimizing the number of multiplications is important. The regulator always begins by selecting the "current" set of coefficients. This was initially done with the code in Listing 3 (generated machine code on the right).

```
par = &reg_par[current_par];        imul  ax,current_par,24
                                    mov   si, ax
                                    add   si,reg_par
```

Listing 3. Original pointer initialization.

Because `current_par` is always 0 or 1, the code could be rewritten using a simple test (Listing 4).

```
par = &reg_par[0];                  mov   si,reg_par
if (current_par != 0)               cmp   current_par,0
  par = &reg_par[1]                 je    label
                                    mov   si,reg_par+24
```

Listing 4. Improved pointer initialization.

Similarly, one can assume that the control variable rarely saturates, i.e., that $u - v = 0$. In this case, the calculation of the integral part can be simplified. These two optimizations reduced the execution time of the regulator by about 40 μs.

Additional optimizations. Three typical C optimizations have been applied: Firstly, regulator coefficients are accessed via a pointer to the "current" element of the coefficient table (see above). Secondly, the pointer and the control variable are declared as `register` variables; only two `register` variables are supported by the compiler. Thirdly, states are incremented using an assignment operator (`I += x;` rather than `I = I + x;`).

Processor Clock frequency	80386 20 MHz	80286 8 MHz	Relative performance
Floating point regulator	90 μs	570 μs	6.3
Integer regulator	48 μs	187 μs	3.9
Simple integer regulator	40 μs	159 μs	4.0
Floating point penalty	30%	205%	

Table 4. Performance of PID regulator, no A/D or D/A conversion.

Processor Clock frequency	80386 20 MHz	68020 16 MHz	Relative performance
Floating point regulator	90 μs	155 μs	1.7
Integer regulator	48 μs	43 μs	0.9
Floating point penalty	88%	260%	

Table 5. Comparison of Intel and Motorola processors.

Results

The performance of three different PID regulators is presented in Table 4. The first uses floating point arithmetic, the second long integers, and the third is a simplified regulator without parameter b. All tests were performed without the A/D and D/A conversions required in a real controller. The programs were run on a Compaq 386/20 and a Tandon AT (also see Section 4). Exactly the same executable code was run on both processors.

The second and third columns of Table 4 present the absolute execution times for the Intel 80386 and 80286 processors, with attached floating point processors (80387 and 80287). On the 80286, the convenience of floating point arithmetic will give a 205% increase in execution time; on the 80386 the increase is 88%. It is interesting to note that for a given clock frequency, the 80386 is only 60% faster on integer arithmetic, but 2.5 times faster on floating point arithmetic. Table 5 compares the performance of the Intel 80386 and the Motorola 68020 processors, with respective floating point processors (80387 and 68881).

The controller has been tested practically on a high-performance hydraulic servo. The PID controller performs two A/D conversions and one D/A conversion for each sample; this requires about 120 μs (see Section 4), which is significant compared to the execution time for the PID algorithm. The maximum sampling frequency on an 8 MHz Tandon AT is therefore chosen as 2 kHz, which also allows a minimum of time for the background process.

6. Conclusions

The real-time performance needed in some control applications is difficult to achieve with a fully featured kernel. The simplest, and therefore the most efficient, solution for control applications is a foreground/background scheduler.

Context switch time with the scheduler implemented on an 8 MHz Intel 80286 is 56 μs; context switch time using Modula-2's TRANSFER is 400 μs. With the hardware used, significantly higher performance cannot be achieved.

The scheduler could be developed further to provide more features, but this effort should not be pursued. For complex control applications, a real-time kernel will provide more appropriate primitives, and more expensive hardware can meet most performance requirements.

In real applications, a controller based on an integer PID regulator and the foreground/background scheduler will achieve a 2 kHz sampling rate on an 80286, while allowing for some operator communication. The 80286 is very slow on floating point calculations, so an integer regulator really pays off. The 80386 is not so much faster on integer calculations but has better floating point performance; the performance penalty for using a floating point regulator is only about 90%, so there is really no need to use an integer regulator.

Acknowledgements

I would like to thank Bjarne Toftegård at DtH for valuable comments on the manuscript and for testing the PID regulator on Intel 80386 machines. Kjell Gustavsson and Bo Bernhardsson kindly tested the controller on an hydraulic servo. Sven Erik Mattsson wrote the original LSI-11 scheduler on which this work is based. Leif Andersson is a real programmer, whose experiences with the Modula-2 kernel made the new scheduler possible. Professor Karl Johan Åström provided me with a high-performance regulator and checked its implementation. Careful and valuable comments were given by Per Hagander.

This study was conducted for the course "Computer Implementation of Control Systems" given at the Department of Automatic Control, Lund, during the spring semester 1988.

References

ANDERSSON, LEIF (1988): "Modula-2 kernel documentation," Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, To be published.

BENTLEY, JON L. (1982): *Writing Efficient Programs*, Prentice Hall, USA.

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

YOUNG, STEPHEN J. (1982): *Real Time Languages*, Ellis Horwood Limited, Chichester, England.

ÅSTRÖM, KARL JOHAN (1987): "Implementation of PID Regulators," CODEN: LUTFD2/TFRT-7344, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Appendix A
Listing of foreground/background scheduler

```
        NAME    FB
        TITLE   Foreground/Background scheduler for C on IBM AT
        PAGE    50,130
;-------------------------------------------------------------------------
;
; Overview.
;
; This is a foreground/background scheduler for simple control applications.
; A major design objective is to reduce overhead and to determine the maximum
; practical sampling rate.
;
; The program is normally executing the background task, which in a typical
; case is responsible for operator communication.  At regular time intervals
; defined by the user, the scheduler starts the foreground task.  Typically,
; the foreground task is the realization of a PID controller.
;
; The scheduler runs on Tandon AT and compatible machines (e.g., IBM AT).
; It has been designed for a user program written in Microsoft C,
; using the small memory model.  It can easily be re-written for other
; programming languages.  The scheduler is written in Microsoft Macro
; Assembler (MASM).
;
;
;
; Using the scheduler.
;
; The following declarations must be included in the user's program:
;
;       extern void schedule();
;       extern void reset();
;
; The code executed by the foreground task must be defined as a void
; function (called a procedure in Pascal).  This function, and all
; functions called by the foreground task, MUST be compiled with stack
; checking turned off.  The best way to do this is to use a pragma:
;
;       #pragma check_stack -
;       void foreground()
;       {
;          /* foreground implementation */
;       }
;       #pragma check_stack +
;
; The scheduler is invoked at fixed intervals (ticks), determined by the
; constant TIMER_PER defined below.  A typical tick is 10 milliseconds.
; Timing is based on a hardware clock of 1.19318 MHz; the clock is scaled
; by a hardware timer which generates interrupts every TIMER_PER clock cycles.
;
; The user's program must install the foreground procedure by calling
; schedule().  The second parameter determines how often the foreground
; task is executed (in multiples of the tick).  Example:
;
;       schedule(foreground, 5);
;
; The following should be noted:
;
;   1.  The foreground task will always run to completion.  If it needs
;       more than one tick (i.e., it should have been started again), the
;       lag is registered, and the foregound task is immediately started
```

```
;       again when it has completed.  Lag does not accumulate, so only
;       one restart is remembered.
;
;   2.  The foreground task can be re-scheduled at any time by calling
;       schedule() again.  In this way a new controller algorithm can
;       be installed instead of the old one.
;
;   3.  A period of 0 ticks has a special meaning: the foreground task
;       will not be executed, but the scheduler is still running.
;       This is more efficient than calling reset(), see below.
;
; Before terminating, reset() MUST be called.  Otherwise the computer
; will crash.  Note that it is possible to restart the scheduler, even
; after a reset.  If the program crashes without calling reset(), all
; hope is lost.  For emergency stop, CTRL-BREAK is treated specially
; and will terminate the program more gracefully.  Do not terminate
; the program by pressing CTRL-C.
;
;
;
; Caveats.
;
; There are a number of problems, potential problems and future
; extensions to consider:
;
;   1.  The registers of the floating-point coprocessor (Intel 80287)
;       are not saved when the foreground task is started.  This
;       requires a modification in FGstart.
;
;   2.  The foreground task executes in the currently active stack,
;       so the internal DOS stack is sometimes used.  We do not know
;       the size of the DOS stack, so there is a risk of stack overflow.
;
;   3.  The scheduler is tailored to the small memory model, which
;       normally only handles 64 KB code and 64 KB data.  To overcome
;       this limitation, use FAR and HUGE keywords; see "Microsoft C
;       Compiler User's Guide", section 8.3.
;
;   4.  There is only one foreground task.
;
;   5.  It would be nice to be able to set the tick value from the
;       program.  This requires changes in schedule() and IntHandler.
;
;   6.  Many operations may not be possible to perform in the foreground
;       task, e. g., calling DOS routines.
;
; The scheduler has been tested on a hydraulic servo, without errors.
;
;
;
; References.
;
; BURR-BROWN (1986): "The Handbook of Personal Computer Instrumentation",
;       Burr-Brown Corporation, Tuscon, Arizona, USA.
;
; IBM (March 1984): "Technical Reference (AT)", IBM Corporation, USA.
;
; IBM (May 1984): "Disk Operating System Version 3.00 Technical Reference",
;       IBM Corporation, USA.
;
; INTEL (1985): "iAPX 286 Programmer's Reference Manual", Intel Corporation,
;       Santa Clara, California, USA.
;
```

```
; MATTSSON, SVEN ERIK (1978): "A Simple Real-Time Scheduler", TFRT-7156,
;       Department of Automatic Control, Lund.
;
; MICROSOFT (1986): "Microsoft C Compiler User's Guide", Microsoft
;       Corporation, USA.
;
;-----------------------------------------------------------------------
; History:
;
; 1988-01-28  DB  Created.
; 1988-02-03  DB  First working version, no BIOS clock.
; 1988-02-04  DB  BIOS clock updated, handles CTRL-BREAK gracefully.
; 1988-02-05  DB  Improved documentation.
; 1988-03-06  DB  Added A/D and D/A conversion routines for C.
; 1988-05-10  DB  Sends 0 to channel 1, in _reset().
; 1988-05-13  DB  Time with interrupts off reduced in ADin.


                .286p


_TEXT   SEGMENT  BYTE PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP



;-----------------------------------------------------------------------
;
; Constant definitions.
;
; TSIGNAL       The scheduler will normally send a test signal to
;               D/A converter channel 0: 1V when the interrupt handler
;               is executing, and 2V when the FG task is executing.
;               Setting TSIGNAL zero will disable this feature.
;
; CLOCK         We should normally update the DOS clock at 18.2 Hz.
;               A small speed-up is acheived by setting CLOCK zero.
;
; TIMER_PER     Determines the period between clock interrupts.
;               TIMER_PER = ROUND(T * 1193.18), where T is the
;               desired period in milliseconds.
;
; TIMER_INT     This is the timer interrupt number, see BIOS manual.
;
; BIOS_CLOCK_INT        The BIOS clock handler should be called at a
;               rate of about 18.2 Hz.  Normally the clock interrupt
;               does this, so we have to install the original clock
;               interrupt handler in a new place.  BIOS_CLOCK_INT
;               is a free interrupt number we use for this purpose.
;
; ADBASE        Base address of A/D converter registers.  The other
;               ADxxxx symbols are offsets from ADBASE.
;
; DAC           Base address of D/A converter registers.
```

```
;
TSIGNAL         EQU     1
; CLOCK         EQU     1
CLOCK           EQU     0
; TIMER_PER     EQU     11932           ; approximately 10 ms
; TIMER_PER     EQU     1193            ; approximately 1 ms
TIMER_PER       EQU     597             ; approximately 0.5 ms
TIMER_INT       EQU     8
BIOS_CLOCK_INT  EQU     0E5H

ADBASE          EQU     0300H
ADSTATUS        EQU     ADBASE
ADCHAN          EQU     ADBASE+1
ADCONV          EQU     ADBASE+2
ADLOW           EQU     ADBASE+3
ADHIGH          EQU     ADBASE+4

DAC             EQU     0310H


;-------------------------------------------------------------------------
;
; Scheduler data
;
_DATA           SEGMENT

FG_ADDR         DW      0       ; pointer to FG procedure
PERIOD          DW      0       ; # clock ticks between FG starts, 0 = no FG
ICNT            DW      0       ; 0 = no FG task, 1 = start FG
FG_ACTIVE       DW      0       ; <>0 = FG active
LAG             DW      0       ; >0 = FG task is lagging

INIT_FLAG       DW      0       ; 1 = scheduler initialized
BIOS_TIMER      DW      0       ; Bios clock int must be called at 18 Hz

BIOS_VEC        DW      0       ; original interrupt vector for int 8
                DW      0

_DATA           ENDS


;-------------------------------------------------------------------------
;
; SENDEOI - Send End-Of-Interrupt to interrupt controller
;
; Note:         AX is destroyed.
;
SENDEOI MACRO
        MOV     AL,20H
        OUT     20H,AL
        ENDM


;-------------------------------------------------------------------------
;
; DAOUT  -  output integer voltage on D/A-converter channel 0
;
; Note:         AX is destroyed.
;               VOLTAGE must be a constant expression.
```

```
;

DAOUT   MACRO   VOLTAGE
      IF        TSIGNAL
        PUSH    DX
        CLI
        MOV     DX,DAC
        SUB     AX,AX           ;; channel 0
        OUT     DX,AL
        INC     DX
        IF VOLTAGE NE 0
          MOV   AX,205*VOLTAGE
          OUT   DX,AL
          MOV   AL,AH
        ELSE
          OUT   DX,AL
        ENDIF
        INC     DX
        OUT     DX,AL
        STI
        POP     DX
      ENDIF
        ENDM


;-------------------------------------------------------------------------
;
; TIMER  -  Program timer
;
; Note:         AX is destroyed.
;
TCC     EQU     043H    ; Timer/counter control word
TC0     EQU     040H    ; Timer 0

TIMER   MACRO   PERIOD
        CLI
        MOV     AL,036H
        OUT     TCC,AL
        MOV     AX,PERIOD
        OUT     TC0,AL
        MOV     AL,AH
        NOP
        OUT     TC0,AL
        STI
        ENDM


_TEXT   SEGMENT
;-------------------------------------------------------------------------
;
; IntHandler - interrupt handler when scheduler is used.
;
; The interrupt handler must perform the following functions:
;
;   1.  Save registers and set up the DS register.
;   2.  Update the BIOS clock at 18.2 Hz, or send an EOI signal.
;   3.  Determine if the FG task should be run, otherwise
;       restore registers and return.
;
; See also:     The Handbook of Personal Computer Instrumentation,
;               Burr-Brown Corporation, USA (1986).  Page 9-30.
```

```
;
; Note on timing:  The first and last instructions of the interrupt
; handler (before and after the calls to DAOUT) require 10 + 27
; clock cycles, i.e., 4.6 us at 8 MHz.
;

IntHandler:
        PUSH    AX
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX

        DAOUT   1

        ; Increment the BIOS clock timer and check if we must
        ; call the original interrupt handler.

        IF      CLOCK
        ADD     BIOS_TIMER,TIMER_PER
        JNC     NoBios
        INT     BIOS_CLOCK_INT
        JMP     CheckFG                 ; BIOS does a SENDEOI
        ENDIF

NoBios:
        SENDEOI

        ; Count all interrupts.  ICNT = 0 means that FG task should not run.

CheckFG:
        CMP     ICNT,0                  ; if (icnt == 0) return;
        JE      IntReturn
        DEC     ICNT                    ; if (--icnt > 0) return;
        JA      IntReturn

        ; ICNT = 0, so we should start the FG task.  If FG task is already
        ; running, special handling is required.

        MOV     AX,PERIOD
        MOV     ICNT,AX
        CMP     FG_ACTIVE,0
        JE      FGstart

        ; There has been 'PERIOD' interrupts while FG task was active,
        ; so the FG task is lagging.  Flag this, and restart FG task
        ; as soon as it has finished.  NOTE: FG is only restarted once,
        ; even if many interrupts were missed.

        MOV     LAG,1

        ; Return from interrupt, FG task has not been started.

IntReturn:
        DAOUT   0
        POP     DS
        POP     AX
        IRET



;-------------------------------------------------------------------------
;
; FGstart - suspend BG task and start FG task
```

```
;
; Note that there are two potential problems:
;
;       1.      The FP registers of the 80287 are not saved.
;
;       2.      The currently active stack is used, so the FG task
;               is sometimes run in the DOS stack which size
;               we do not know.
;
FGstart:
        MOV     FG_ACTIVE,1

        ; Save status.  Flags, CS, IP, AX and DS are already on the
        ; stack in the interrupt frame or saved entering IntHandler.

        PUSH    ES
        PUSHA

        ; Call the FG task procedure once, or as long as the FG task
        ; is lagging.  A short call is possible because CS was
        ; loaded when invoking the interrupt handler.

LagLoop:
        DAOUT   2
        MOV     LAG,0                   ; initially no lag
        MOV     BX,FG_ADDR
        CALL    BX
        CMP     LAG,0                   ; call until no more lag
        JNZ     LagLoop

        MOV     FG_ACTIVE,0

        ; Restore registers of BG task and return.

        POPA
        POP     ES

        DAOUT   0

        POP     DS
        POP     AX
        IRET



;-------------------------------------------------------------------------
;
; BREAK_int - CTRL-BREAK interrupt handler.
;
; This routine is invoked by BIOS through interrupt 1BH when
; CTRL-BREAK is pressed.  We must under all circumstances
; restore the original interrupt handler for the timer.
; In addition, the program is terminated.
;
; See also: IBM AT Technical Reference manual, page 5-7.
;           DOS Version 3.00 Technical Reference, page 5-129.
;

BREAK_int:
        PUSHA
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
```

```
        SENDEOI
        SENDEOI
        CALL    _reset

        MOV     AH,4CH
        MOV     AL,1
        INT     21H             ; should not return...

        POP     DS
        POPA
        IRET


;-------------------------------------------------------------------------
;
; _schedule - register FG procedure and period
;
; C syntax:  extern void schedule(procedure FG, unsigned short period);
;
; Note:  The C function MUST be compiled with stack checking turned off.
; The best way is put "#pragma check_stack-" immediately before the
; function, and "#pragma check_stack+" immediately after.  This also
; applies for functions called by the FG task.
;
; See also: DOS Version 3.00 Technical Reference Manual, p. 5-67 and p. 5-82.
;

                PUBLIC _schedule
_schedule       PROC NEAR

OFF1    EQU     4                       ; offset of FG
OFF2    EQU     6                       ; offset of period

        ENTER   0,0

        ; Setup scheduler: take address of procedure (void function)
        ; and the period from the main program in C.

        MOV     AX,[BP+OFF1]
        MOV     FG_ADDR,AX
        MOV     AX,[BP+OFF2]
        MOV     PERIOD,AX
        MOV     ICNT,AX                 ; icnt = period
        MOV     FG_ACTIVE,0

        ; Check if scheduler has been initialized.

        CMP     INIT_FLAG,1
        JE      NoInit

        ; Get the original interrupt handler from DOS and save it.

        MOV     INIT_FLAG,1
        MOV     AH,35H
        MOV     AL,TIMER_INT
        INT     21H
        MOV     BIOS_VEC,ES
        MOV     BIOS_VEC+2,BX

        ; Install it as handler for an unused interrupt, in order
        ; to update the BIOS clock.
```

```
        PUSH    DS
        MOV     AX,ES
        MOV     DS,AX
        MOV     DX,BX
        MOV     AH,25H
        MOV     AL,BIOS_CLOCK_INT
        INT     21H

        ; Catch CTRL-BREAK to make sure we reset the interrupt

        MOV     AX,CS
        MOV     DS,AX
        MOV     DX,OFFSET BREAK_int
        MOV     AH,25H
        MOV     AL,1BH
        INT     21H


        ; Install private interrupt handler for scheduler

        MOV     AX,CS
        MOV     DS,AX
        MOV     DX,OFFSET IntHandler
        MOV     AH,25H
        MOV     AL,TIMER_INT
        INT     21H
        POP     DS

        ; Program timer

        TIMER   TIMER_PER

NoInit:
        LEAVE
        RET
_schedule       ENDP


;-------------------------------------------------------------------------
;
; _reset - reset interrupt handler and timer
;
; C syntax:  extern void reset();
;
        PUBLIC  _reset
_reset  PROC    NEAR
        ENTER   0,0

        ; Reset some output signals...

        PUSH    0
        PUSH    1
        CALL    _DAout

        ; Restore the original interrupt handler

        PUSH    DS
        MOV     AX,BIOS_VEC
        MOV     DX,BIOS_VEC+2
        MOV     DS,AX
        MOV     AH,25H
```

```
        MOV     AL,TIMER_INT
        INT     21H
        POP     DS

        ; Reset timer and final cleanup

        TIMER   0

        MOV     ICNT,0
        MOV     PERIOD,0
        MOV     FG_ADDR,0
        MOV     INIT_FLAG,0

        LEAVE
        RET
_reset  ENDP


;-------------------------------------------------------------------------
;
; _ADin - Analog in for RTI-800.
;
; C syntax:  int ADin(channel)
;            int channel;
;

        PUBLIC  _ADin
_ADin   PROC    NEAR

        ENTER   0,0
        CLI                             ; interrupts off

        MOV     DX,ADCHAN
        MOV     AX,[BP+OFF1]            ; channel
        OUT     DX,AL

        MOV     DX,ADCONV               ; start conversion
        SUB     AX,AX
        OUT     DX,AL
        STI

        ; Loop until conversion done

        MOV     DX,ADSTATUS
LOOP:
        IN      AL,DX
        AND     AX,0040H                ; bit 6 busy bit
        JZ      LOOP

        ; Read result and return in AX

        MOV     DX,ADHIGH
        IN      AL,DX
        MOV     AH,AL
        MOV     DX,ADLOW
        IN      AL,DX

        LEAVE
        RET
_ADin   ENDP
```

```
;-------------------------------------------------------------------------
;
; _DAout - Analog out for RTI-800.
;
; C syntax:  void DAout(channel, value)
;            int channel;
;            int value;
;

        PUBLIC  _DAout
_DAout  PROC    NEAR

        ENTER   0,0
        CLI

        MOV     DX,DAC
        MOV     AX,[BP+OFF1]            ; channel
        OUT     DX,AL

        INC     DX
        MOV     AX,[BP+OFF2]            ; value
        OUT     DX,AL

        INC     DX
        MOV     AL,AH
        OUT     DX,AL

        STI
        LEAVE
        RET
_DAout  ENDP


_TEXT   ENDS
        END
```

Appendix B
Listing of PID controller

```
/*
 * PID7.C
 *
 * Fast integer PID regulator using foreground/background scheduler.
 *
 */


#include <stdio.h>
#include "fb.h"



/* A/D and D/A converter channels */

#define         CH_R    0
#define         CH_Y    1
#define         CH_U    1



/* Other compile-time constants */

#define         UHIGH   2047            /* converter limits */
#define         ULOW    (-UHIGH)
#define         BITS    16              /* parameter scale factor */
#define         SCALE   65536           /* 2 ** BITS */
#define         ISCALE  (1.0 / SCALE)
#define         TIMER_PER       0.5     /* Scheduler period, ms */



/* Regulator parameter data structure */

typedef struct {                /* NOTE: all parameters are scaled, */
  long  K;                      /* i.e., SCALE * the real value.    */
  long  b;
  long  bi;
  long  br;
  long  ad;
  long  bd;
} regulator_par;


regulator_par reg_par[2];
int current_par = 0;            /* used to swap parameter set (0 or 1) */


float   Tr, Ti, Td, N;          /* real regulator parameters */
float   h;                      /* sampling period, in ms */


/* Regulator states */

int     yold;           /* unscaled state */
long    I, D;           /* scaled states */
```

```
/*
 * print_param()
 *
 * Print current parameters and regulator coefficients.
 *
 */

void print_param()
{
  register regulator_par        *par;

  par = &reg_par[current_par];

  printf("-----------------------------------------------------------------\n");
  printf("Gain (K):              %g\n", par->K * ISCALE);
  printf("Magic factor (b):      %g\n", par->b * ISCALE);
  printf("Tracking time (Tr):    %g\n", Tr);
  printf("Integral time (Ti):    %g\n", Ti);
  printf("Derivative time (Td): %g\n", Td);
  printf("Max deriv. gain (N):   %g\n", N);
  printf("\n");
  printf("bi:   %g\n", par->bi * ISCALE);
  printf("br:   %g\n", par->bi * ISCALE);
  printf("ad:   %g\n", par->ad * ISCALE);
  printf("bd:   %g\n", par->bd * ISCALE);
  printf("-----------------------------------------------------------------\n");
}



/*
 * input()
 *
 * Read a value with prompt and range checking.
 *
 */

float input(prompt, min, max)
char *prompt;
float min, max;
{
  float x;

again:
  printf("%s:   ", prompt);
  scanf("%f", &x);
  if (x < min || x > max) {
    printf("Value out of range (%f..%f)\n", min, max);
    goto again;
  }

  return x;
}



/*
 * read_param()
 *
 * Read new parameters and calculate regulator coefficients.
 *
 */
```

```
void read_param()
{
  float K, b;
  register regulator_par        *par;

  /* Copy old parameters to other data area */

  if (current_par == 0)
    par = &reg_par[1];
  else
    par = &reg_par[0];
  *par = reg_par[current_par];

  /* Read regulator parameters */

  K = input("Gain (K)", 0.0, 1000.0);
  par->K = SCALE * K;
  b = input("Magic factor (b)", 0.0, 1.0);
  par->b = SCALE * b;
  Ti = input("Integral time (Ti)", 0.1, 1000.0);
  Tr = input("Tracking time (Tr)", 0.1, Ti);
  Td = input("Derivative time (Td)", 0.0, 1000.0);
  N  = input("Max derivative gain (N)", 0.0, 30.0);

  /* Calculate regulator coefficients */

  par->bi = SCALE * K * h / Ti;
  par->br = SCALE * h / Tr;
  par->ad = SCALE * Td / (Td + N * h);
  par->bd = SCALE * (K * Td * N) / (Td + N * h);

  if (par->bi == 0)
    printf("Warning - bi = 0.\n");

  /* Switch parameter data areas */

  if (current_par == 0)
    current_par = 1;
  else
    current_par = 0;
}



/*
 * foreground()
 *
 * Regulator foreground process.
 *
 */

#pragma check_stack-

void foreground()
{
  register regulator_par  *par;

  int   r, y, v;                        /* unscaled variables */
  register int u;
  long  P;

  par = &reg_par[0];
  if (current_par != 0)
```

```
    par = &reg_par[1];

  /* Regulator loop */

  r = ADin(CH_R);
  y = ADin(CH_Y);

  P = par->K * (((par->b * r) >> BITS) - y);
  D = ((par->ad * D) >> BITS) - par->bd * (y - yold);
  u = v = (P + I + D) >> BITS;

  if (v < ULOW)
    u = ULOW;
  else if (v > UHIGH)
    u = UHIGH;
  DAout(CH_U, u);

  I += par->bi * (r - y);
  if (u != v)
    I += par->br * (u - v);
  yold = y;
}

#pragma check_stack+



/*
 * main()
 *
 * Main program.
 *
 */


main()
{
  /* Initialize states */

  I = D = 0;
  yold = ADin(CH_Y);

  printf("Sampling period (milliseconds): ");
  scanf("%f", &h);
  read_param();
  schedule(foreground, (int) (h / TIMER_PER));

  while (reg_par[current_par].K != 0) {
    print_param();
    read_param();
  };

  reset();
}
```