A DSP Implementation of an Adaptive Controller

Åström, Karl Johan; Jagannathan, Kanniah

1992

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

ISSN 0280–5316
ISRN LUTFD2/TFRT--7494--SE

A DSP Implementation of an Adaptive Controller

Karl Johan Åström
Jagannathan Kanniah

Department of Automatic Control
Lund Institute of Technology
March 1992

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

Document name
REPORT

Date of issue
March 1992

Document Number
ISRN LUTFD2/TFRT--7494--SE

Author(s)
Karl Johan Åström
Jagannathan Kanniah

Supervisor

Sponsoring organisation

Title and subtitle
A DSP Implementation of an Adaptive Controller

Abstract

Limitations in computing speed have so far precluded the use of adaptive regulators for controlling fast processes. The advent of digital signal processors (DSP) has somewhat reduced the limitations, since they are specially designed for fast computations. Furthermore, filtering operations are quite straightforward to implement using a DSP. An adaptive regulator, which samples at 3.33 kHz, is described in this report. Particular attention has been given to prefiltering and postfiltering in the algorithm. The prefiltering requirements are different from standard digital filtering. Postfiltering is implemented as predictive first order hold (PFOH). The implementational details, timing considerations, and test results are also presented.

Key words
DSP. Adaptive control. Pre-filtering. Post-filtering. Predictive first-order hold. Digital Control.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
53

Recipient's notes

Security classification

A DSP Implementation of an Adaptive Controller

Karl Johan Åström
Department of Automatic Control
Lund Institute of Technology
Box 118, S–221 00, Lund
Sweden

Jagannathan Kanniah
Department of Electronics
and Communications Engineering
Singapore Polytechnic
Singapore 0513

Abstract. Limitations in computing speed have so far precluded the use of adaptive regulators for controlling fast processes. The advent of digital signal processors (DSP) has somewhat reduced the limitations, since they are specially designed for fast computations. Furthermore, filtering operations are quite straightforward to implement using a DSP. An adaptive regulator, which samples at 3.33 kHz, is described in this report. Particular attention has been given to prefiltering and post filtering in the algorithm. The prefiltering requirements are different from standard digital filtering. Post filtering is implemented as predictive first order hold (PFOH). The implementational details, timing considerations, and test results are also presented.

1. Introduction

Adaptive controllers have mostly been implemented using microprocessors and microcontrollers. The advent of microprocessors and microcontrollers reduced the cost and development time drastically. This is due to the fact that microcontrollers offered a whole bunch of additional hardware like ADCs, DACs, RAM, EPROM, and even PWM features, which are needed for real time data acquisition and control. Even with all these advancements, the sampling time has to be rather large, since it is necessary to perform identification and control computations between control changes while implementing adaptive regulators. With the current technology, it is possible to get sampling rates up to a few hundred Hz. These limitations are easing with the advent of digital signal processors. Now it is possible to implement adaptive regulators with substantially faster sampling rates.

Furthermore, the modern DSPs are capable of floating point computations without any additional overheads of programming, fast clock rates, and parallel processing. Digital Signal Processors are specially designed for filter applications. Hence while implementing an adaptive regulator, it is possible to include a suitable adaptive filtering as an integrated part of the software, by using multirate sampling techniques. In the following sections various considerations regarding adaptive prefiltering and post filtering for implementing an adaptive regulator are discussed. Actual implementational details and test results are also presented.

2. Adaptive Pre-Filtering

The importance of prefiltering can not be overemphasised [1,2]. The quality of control depends critically on the quality of the measured signals. In all digital control applications, it is important to have a proper prefiltering of signals before they are sampled. Because of aliasing problem connected with the sampling procedure, it is necessary to eliminate all the frequencies above the Nyquist frequency corresponding to the sampling rate. In adaptive systems there is an additional difficulty because the aliased signals may give rise to estimation errors. For high performance systems with fixed sampling rates, it is common practise to design analog prefilters, since the required cut-off frequency is known. For a general purpose systems, where the sampling rates may vary significantly, a fixed prefilter matched with the fastest sampling rate is typically used. Such a system may not have good performance at low sampling rates. It is fairly complicated to implement an analog filter with variable bandwidth. Such a filter is also costly.

Since filters are easily implemented in a DSP system, it is possible to include adaptive prefiltering as a part of the adaptive controller. A possible filtering scheme based on multirate sampling is shown in Figure 1. A fast sampler with period h is combined with a fixed analog prefilter. The adaptive prefilter which runs with sampling period h then performs digital filtering, which is matched to the variable sampling period T of the controller.

Filter Options

Suitable choices of anti-aliasing filters are second or fourth order Butterworth, IATE (Integral Time Asolute Error), or Bessel filters. While the effect of delay in the filter can be ignored in the audio applications, it has to be taken into account in control applications. All filters introduce additional dynamics in the system, which complicates the model. The Bessel filter can be easily approximated by a time delay, since the phase shift due to this filter is directly proportional to the frequency of the signal. This implies that the sampled data model can be assumed to contain an additional time delay compared to the process.

Prefilter Design

The purpose of design described here is to reduce signal components in the frequencies higher than the Nyquist frequency to negligible levels and to make the signal transmission up to the Nyquist frequency as close to unity as possible. Let us consider the case of a continuous time prefilter. The properties expressed above can be captured by finding a filter transfer function G that

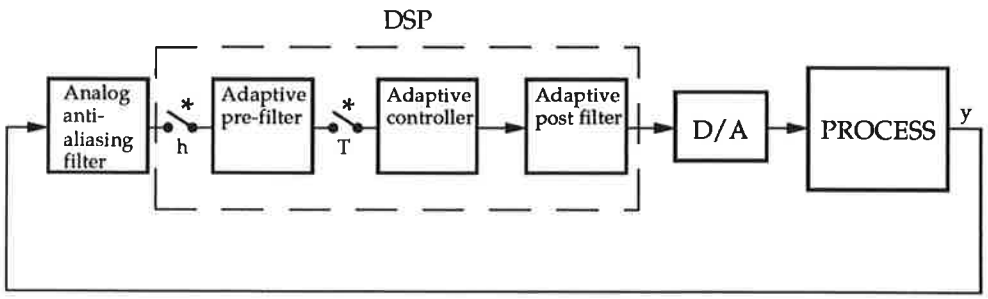


Figure 1. The filtering set-up.

minimizes the loss function

$$\begin{aligned} J(G) &= \int_0^{\omega_N} |G(iw) - 1|^2 dw + \int_{\omega_N}^{\infty} |G(iw)|^2 dw \\ &= \int_0^{\omega_N} |G(iw)|^2 dw - \int_0^{\omega_N} \big(G(iw) + G(-iw)\big) dw + w_N \\ &= \omega_N - 2 \int_0^{\omega_N} Re\{G(iw)\} dw + \int_0^{\infty} |G(iw)|^2 dw \end{aligned} \tag{1}$$

Now consider the case of a multirate prefilter. Let h be the fast sampling rate and $\omega_N = 1/2T$ be the Nyquist frequency associated with the slow sampling rate, corresponding to the control interval. If H is the pulse transfer function associated with the fast sampling rate, the problem of finding the prefilter coefficients can be stated as to find an H that minimizes

$$\begin{aligned} J(H) = &\int_0^{\pi/h} H(e^{-i\omega h}) H(e^{i\omega h}) dw \\ &- 2 \int_0^{\omega_N} Re\{H(e^{-i\omega h})\} dw + \omega_n \end{aligned} \tag{2}$$

If $H(i\omega)$ is an FIR filter, i.e.

$$H(z) = b_o + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-n} \tag{3}$$

We get

$$\begin{aligned} J(H) &= \frac{\pi}{h} \left(\sum_{k=0}^{n} {b_k}^2 \right) - 2 \sum_{k=0}^{n} b_k \int_0^{\omega_N} \cos(k\omega h) dw + \omega_N \\ &= \frac{\pi}{n} \sum_{k=0}^{n} {b_n}^2 - 2 \sum_{k=0}^{n} b_k \frac{1}{kh} \sin(\omega_N k h) + \omega_N \end{aligned} \tag{4}$$

The coefficients that minimize $J(H)$ are given by

$$\frac{\pi}{h} b_k = \frac{\sin(\omega_N k h)}{kh}$$

The filter coefficients are thus given by

$$\begin{aligned} b_o &= \frac{\omega_N h}{\pi} \\ b_k &= \frac{1}{\pi k} \sin(\omega_N k h) \qquad k = 1, 2, \ldots, n \end{aligned} \tag{5}$$

It is desirable that the prefilter has unit DC gain. In order to guarantee this, we can minimize the criterion J under the constraint that

$$\sum_{k=0}^{n} b_k = 1 \tag{6}$$

By introducing a Langrangian multiplier the criterion to be minimized then is

$$J(H) + \lambda\left(\sum_{k=0}^{n} b_k - 1\right)$$

Letting the partial derivatives with respect to b_k equal to zero we get

$$\frac{\pi}{k} b_k - \frac{1}{kh}\sin(\omega_N kh) + \lambda = 0$$

This gives

$$b_k{}' = \frac{1}{\pi k}\sin(\omega_N kh) + \frac{\lambda h}{\pi}$$

Let b_k be the values given by Equation 5. We then find that

$$b_k{}' = b_k + \alpha \tag{7}$$

where the parameter α should be determined in such a way that the coefficients sum up to one. Hence

$$b_k{}' = b_k + \frac{1}{n+1}\sum_{k=0}^{n} b_k \tag{8}$$

DSP Filter Code

Since there is a C compiler available for the DSP, we choose to write the filter code in C. The efficiency of such a program, measured in terms of time of execution, depends on programming skill. Two kinds of programming were attempted. The difference in performance differences were quite surprising. The filter output is given by the difference equation,

$$y(k) = b_0\, S(k) + b_1\, S(k-1) + \cdots + b_n\, S(k-n) \tag{9}$$

where $S(k), S(k-1), \ldots, S(k-n)$ are the measured signals, $b_0, b_1, \ldots, b_n$ are the filter coefficients, and n is the order of the filter. The software gets a new measurement each sampling time and shifts the data through the filter storage array. Then the filter output can be readily calculated. It is quite tempting to implement the filter as shown in the flowchart given in Figure 2. This filter program was implemented for a filter of order 100 and a timing test was conducted. While the system throughput was 33 Megaflops, one filter output computation took 240 μs. This can be compared with the time for 200 floating point operations which is 6 μs. When an optimizer was run on the C-code, the time taken reduced to 140 μs. One can easily see that large amount of time is spent on shifting data through the filter array.

An alternate approach is to reserve data buffer of double the size of filter array length to store the data vector, avoid the shifting the data, but only manipulate an address pointer. The method is shown in Figure 3. Whenever one data is taken in, the pointer is used to write it in two locations with addresses namely, the ($database$+i) and ($database$+n + i), Initially the value of i is zero. When i reaches a value equal to $n-1$, the data is written in the two locations and pointer is reset to zero.

When the first data is taken in, the point A has data and other locations below it up to and excluding A will have zeros. When the second sample is

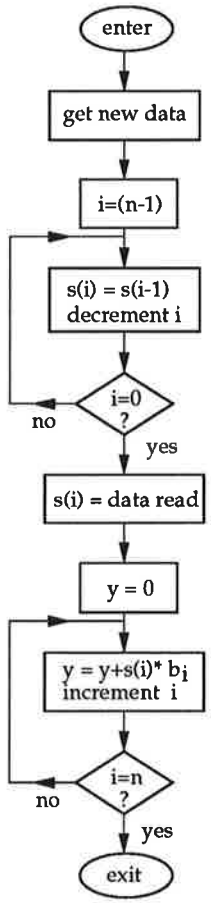


Figure 2. C-program filter flowchart.

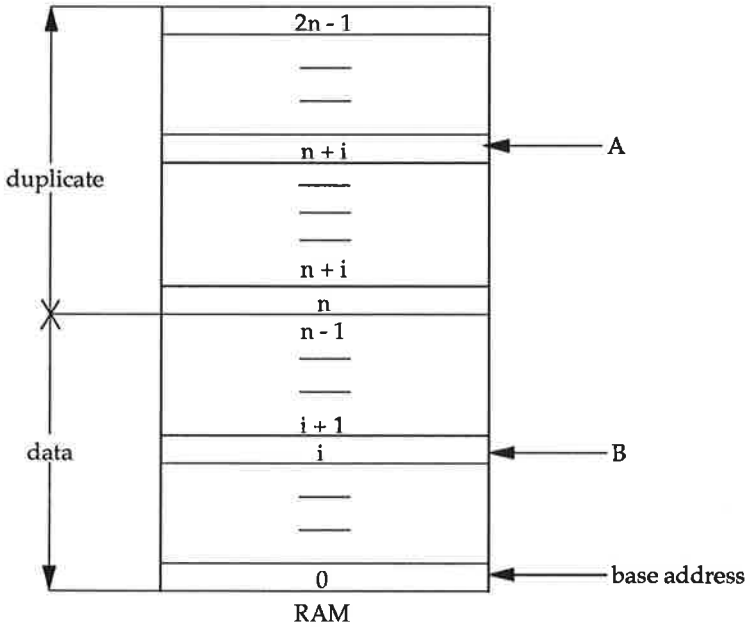


Figure 3. Fast FIR filter memory map implementation.

taken ($i = 1$), data at point A (which has moved one address higher) has the most recent data and the one below is one sample older. When this happens and i reaches the value $(n - 1)$, the locations from A to B will contain valid data, the most recent at A and the oldest just above location B. After this i is

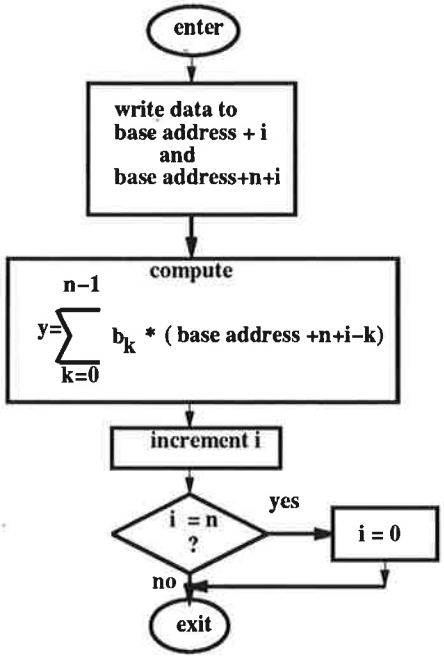


Figure 4. Fast FIR filter flowchart.

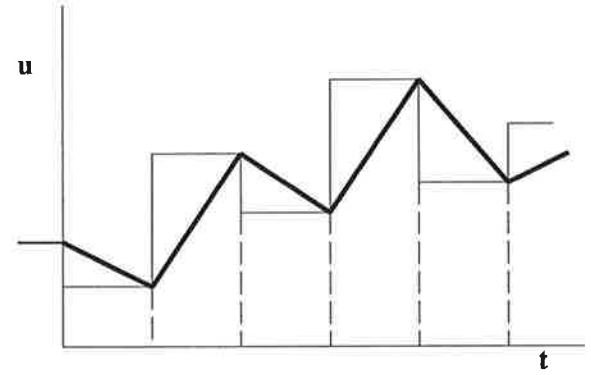
reset to zero. Then A moves back to location n and B moves back to location 0. Now again a scan from A to B will give the n most recent. The flowchart of this filter implementation is shown in Figure 4 The shifting process is avoided by this procedure. The update is to be done every sampling time. The filter output may be calculated as and when required.

Even though it may be implemented in C-language, it could be tricky to do it without any overheads. With some of knowledge about frame page pointers and calling mechanism involved while a C-routine calls an assembly routine, it is straightforward to implement it in assembly code. Such a code listing is given in Section 1 of the Appendix. Please note that the multiplication and addition are performed as parallel instructions, thus utilizing the TMS320c30 processor's ability for parallel processing. A 100th order filter took only 30 μs for computing one filtered output, thus improving the speed by a factor of 5. This shows that the choice of a proper algorithm is essential even while using a fast processor.

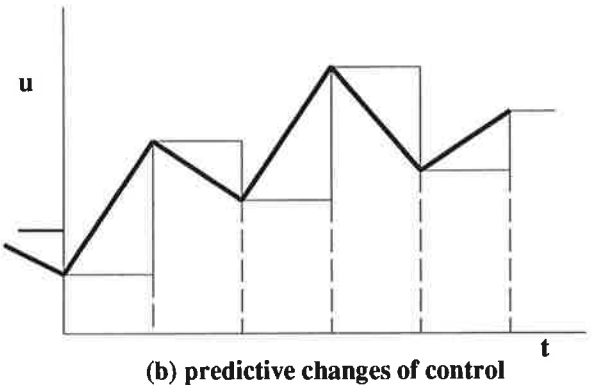
3. Adaptive Post-Filtering

In the traditional implementation of a digital controller the controller output is computed and written to a buffer/port connected to a digital to analog converter. The output is held constant between the consecutive control intervals. This causes the control variable to change in jumps. Such jumps will excite the high frequency modes in the controlled process, which results in inferior performance [3]. A simple analog pre-filter with response time of about one fourth of the control interval may somewhat alleviate this problem, but will not solve it. Intuitively speaking, a control output smoothing system, which gives a piecewise linear output, is more desirable.

Figure 5 shows two ways of doing this, with the same control sequence. Figure 5a can be easily implemented with any control strategy. This obviously produces a time delay in control. In Figure 5b, the $u(k+1)$ is computed at



(a) slow changes of control



(b) predictive changes of control

Figure 5. Smooth control changes.

instant k and control is intrapolated. This is called the predictive first order hold (PFOH). Such a hold circuit has been rejected as physically unrealisable, citing the violation of causality condition [4]. Let us assume that the control law to be implemented is given by

$$R(q)u(q) = T(q)u_c(q) - S(q)y(q) \tag{10}$$

In [5] it is shown that the PFOH is realizable if the following conditions hold.

$$\begin{aligned} \deg R &\geq \deg S + 1 \\ \deg R &\geq \deg T + 1 \end{aligned} \tag{11}$$

Process Model to Implement PFOH

In zero order hold sampling the control signal is piecewise constant. The pulse transfer function obtained by zero order hold sampling is

$$G(z) = (1 - z^{-1})\mathcal{Z}\mathcal{L}^{-1}\left(\frac{G(s)}{s}\right) \tag{12}$$

With a predictive first order hold the control signal is continuous and piecewise linear. Such a signal can be thought of as obtained from an integrator driven by a piecewise control signal, see Figure 6. The pulse transfer function of such

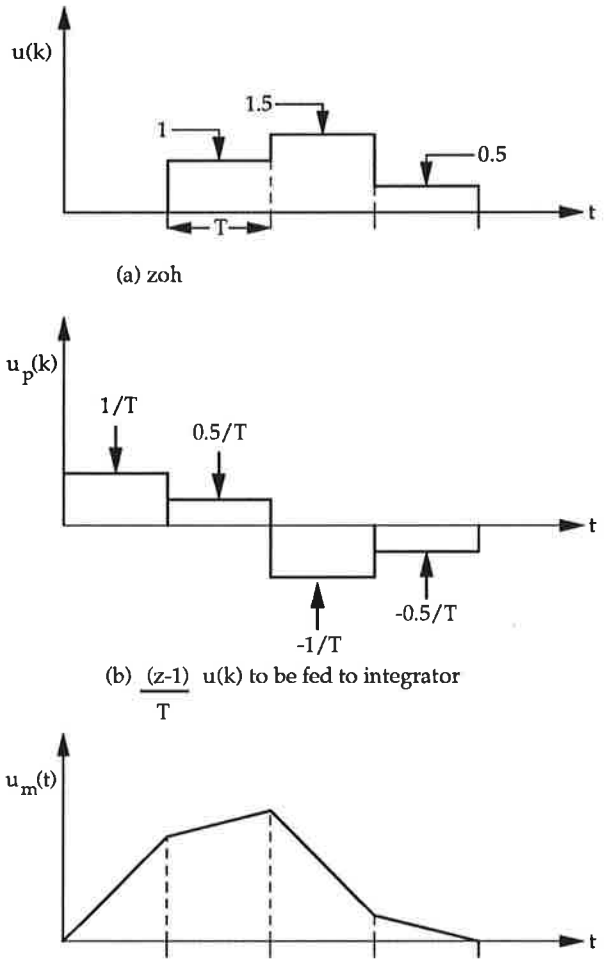


Figure 6. The physical interpretation of PFOH.

a system is

$$\begin{aligned} H(z) &= \frac{(z-1)}{T}\left((1-z^{-1})\mathcal{ZL}^{-1}\left(\frac{1}{s}\cdot\frac{G(s)}{s}\right)\right) \\ &= \frac{(z-1)^2}{Tz}\mathcal{ZL}^{-1}\left(\frac{G(s)}{s^2}\right) \end{aligned} \tag{13}$$

The input-output relation corresponding to this has the form

$$\begin{aligned} y(k) = &-a_1\,y(k-1) - a_2\,y(k-2) - \cdots - a_n\,y(k-n) \\ &+ b_0\,u(k) + b_1\,u(k-1) + \cdots + b_m\,u(k-m) \end{aligned} \tag{14}$$

Notice that $u(k)$ appears in the right hand side.

PFOH Implementation

The digital implementation of PFOH can be implemented using timer interrupts. Divide the control intervals into several smaller intervals. It is preferable to make the data sampling interval equal to the control smoothing intervals. Then a control staircase can be built bridging $u(k+1)$ and $u(k)$. At j:th

interrupt between major control intervals, the smoothing function is given by

$$u(j) = \frac{h}{T}\left(u(k+1) - u(k)\right) j \tag{15}$$

where h is the smoothing interval and T is the major control interval. The integer j is reset every major control interval. The control changes are in minute steps, thus producing a good approximation of the PFOH. A flow chart of the implementation is presented in a later section.

4. Identification

The process model used for identification is taken as

$$H(q) = \frac{B(q)}{A(q)} \tag{16}$$

Knowing the fact that PFOH control is to be implemented, $A(q)$ and $B(q)$ should be compatible with to Equation (14). The square root algorithm is used for identification [2].

5. Controller Design

Two types of controller designs are described in this section. In one design, no time delay is considered. In the second design the delay due to the filter is taken into account and a robust controller with integral action is implemented.

Plant with No Time Delay

The control law to be implemented is given in Equation (10) and the condition which it should satisfy for PFOH implementation is given in Equation (11). In order to evaluate the coefficients of the polynomials in Equation (10), the DAB (Diophantine Aryabhatta, and Bezout) equation has to be solved. It is given below.

$$\frac{BT}{AR + BS} = \frac{B_m A_0}{A_m A_0} \tag{17}$$

To find a solution to the above equation the additional conditions to be satisfied are [5]

$$\begin{aligned} \deg A_m - \deg B_m &\geq \deg A - \deg B + 1 \\ \deg A_0 &\geq 2\deg A - \deg A_m \end{aligned} \tag{18}$$

The response model can be chosen such that the process zeros are included in the model

$$H_m(q) = \frac{B(q)}{A_m(q)} \cdot \frac{A_m(1)}{B(1)} \tag{19}$$

Factorizing the polynomial B as $B^- B^+$, where B^- consists of the poorly the damped zeros and B^+ consists of the well damped zeros, the DAB equation can be rewritten as

$$\frac{B^+ B^- T}{B^+(AR' + B^- S)} = \frac{B^- B'_m A_0}{A_m A_0} \tag{20}$$

where $B = B^+B^-$, $B_m = B'_mB^-$, and $R = B^+R'$. Here it is assumed that no process zeros are cancelled. This is a safe practice, since zeros may drift from favourable positions to unfavorable position and may cause control oscillations. Hence we make

$$B^- = B \; ; \qquad B'_m = \frac{A_m(1)}{B(1)} \tag{21}$$

This gives

$$T(q) = B'_m A_0 \; ; \qquad AR' + B^-S = A_m A_0 \tag{22}$$

For example, if $\deg A = 2$ and $\deg B = 2$, then acceptable design will yield

$$\deg A_m = 3 \; ; \quad \deg A_0 = 1 \; ; \quad A_0 = q \tag{23}$$

The degrees of the controller polynomials are then

$$\deg R = \deg R' = 2 \; ; \quad \deg S = 1 \; ; \quad \deg T = 1 \tag{24}$$

The DAB equation must be solved in real time.

Robust Controller for System With Delay

A robust controller that includes integral action is described in this section. The time delay introduced by the filtering process is also taken into account. The aim is to test a reasonably complex controller on the DSP based system. When there is a time delay in the plant, degree of $A(q)$ increases. Then,

$$\deg A = 3$$

From Equation (18), we obtain

$$\deg A_m \geq 4 \tag{25}$$

The $R(q)$ polynomial must, however, include a $(q-1)$ term for integral action and the $S(q)$ polynomial must include a $(q+1)$ term, for achieving robustness. The DAB equation can be written as

$$\begin{aligned}(q^3 + a_1q^2 + a_2q)(q-1)R_m(q) \; + \\ (b_0q^2 + b_1q + b_2)(q+1)S_m(q) = A_m(q)A_0(q)\end{aligned} \tag{26}$$

To satisfy Equation (18) one must choose $\deg A_m + \deg A_0 = 8$. Hence the degrees of the polynomials can be obtained as

$$\deg R_m = 4 \; ; \qquad \deg S_m = 3 \tag{27}$$

and

$$\deg A_m = 5 \; ; \qquad \deg A_0 = 3 \tag{28}$$

After the modified controller polynomials $R_m(q)$ and $S_m(q)$ are solved, the actual controllers are given by

$$\begin{aligned}R(q) &= (q-1)R_m(q) \\ S(q) &= (q+1)S_m(q) \\ T(q) &= B\prime_m A_0\end{aligned} \tag{29}$$

6. Overall Software Structure

The primary tasks to be performed by the real time software are

1. data sampling
2. prefiltering
3. vector updating
4. control vector updating
5. composite data vector updating
6. identification
7. controller design
8. control computation
9. post filtering

The data sampling is done at the fast rate. The filter computations are needed when the data vector is to be updated. But the post filtering is to be more often for achieving a smooth variation in control. To make software less complex, the post filtering instants and the data sampling instants are made to coincide. The most time consuming segment of software is the identification and the least time consuming segment is the post filtering. There are several ways to implement the different functions. The data structure that represents the state of the controller are

$$\begin{aligned} uvect^T &= [uref(k)\ \ uref(k-1)\ \ldots\ldots &] \\ yvect^T &= [y(k)\ \ y(k-1)\ldots\ldots\ldots &] \\ ucon^T &= [u(k)\ \ u(k-1)\ \ldots\ldots\ldots &] \\ \phi^T &= [y(k)\ y(k-1)\ldots u(k)\ u(k-1)\ldots &] \\ \theta^T &= [-a_1\ \ -a_2\ldots b_0\ b_1\ldots &] \end{aligned} \tag{30}$$

Note that $u(k)$ is included in the *ucon* vector in order to implement the predictive first order hold. When there is additional time delay due to filtering, the first term in *ucon* vector would be $u(k-1)$. The lengths of the vectors depend on the number of the terms in the A and B polynomials. A constant named 'int interval' is used to initialize the timer-0 for interrupt generation. The clock takes δ μs to count each time. This determines the duration between the timer-0 interrupts, upon which the data sampling and control smoothing are performed. A variable, 'int count' counts the number of such interrupts and keeps track of the time. The control computation is performed when the variable reaches a specified number, 'int count max'. Then the data sampling time is given by

$$h = (int\_interval) \cdot \delta \tag{31}$$

The control interval is given by

$$T = (int\_count\_max) \cdot h \tag{32}$$

Operation of the software is centered around the timer-0 interrupts of the processor board used. It is obvious that the ϕ vector update, the identification and the DAB solution would have to be done in a sequence. These computations will take comparatively long time. The data sampling and the control smoothing have to be done at the exact instances, marked by the timer-0 interrupts. The data vector is updated every control interval. The filter output

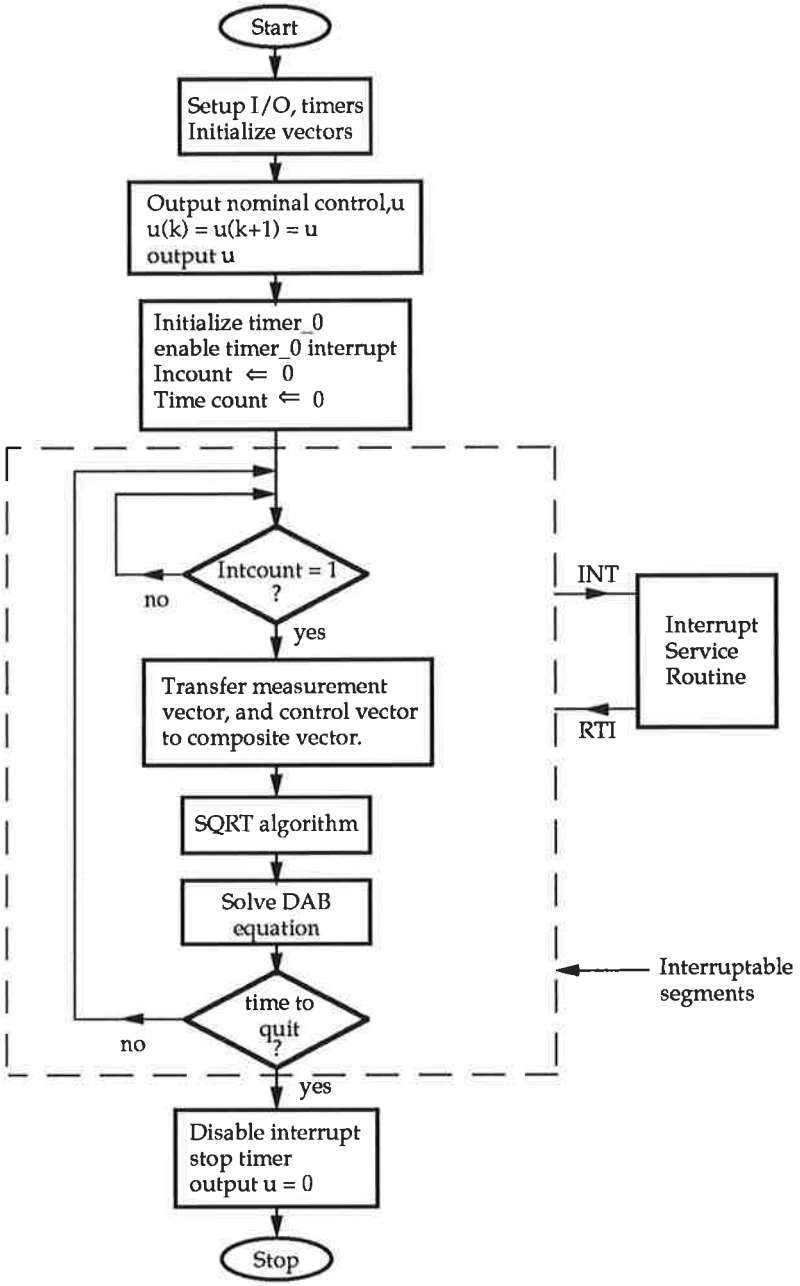


Figure 7. The overall flowchart of the software.

needs to be calculated only when the data vector is updated. In this implementation, however, the filter output is calculated every time data is read into the filter array for simplicity. For a 10:th order filter the additional time spent for calculation is small. The ϕ vector is updated by transferring data from the *yvect* and *ucon* vectors, just before the identification routine is entered.

The overall system flowchart is shown in Figure 7. Initially several hardware and software initializations need to be done. Other details like outputting a nominal control are also done, before entering the main loop. There is a 'Time count' to keep track of the number of control intervals. This is needed in laboratory situation. The first box in the main loop is used to synchronize the starting of the execution with the specific interrupt where the *int_count* changes from 0 to 1. Then the process starts executing the *slow* routines, while being interrupted by the timer. The interrupt service routine is shown

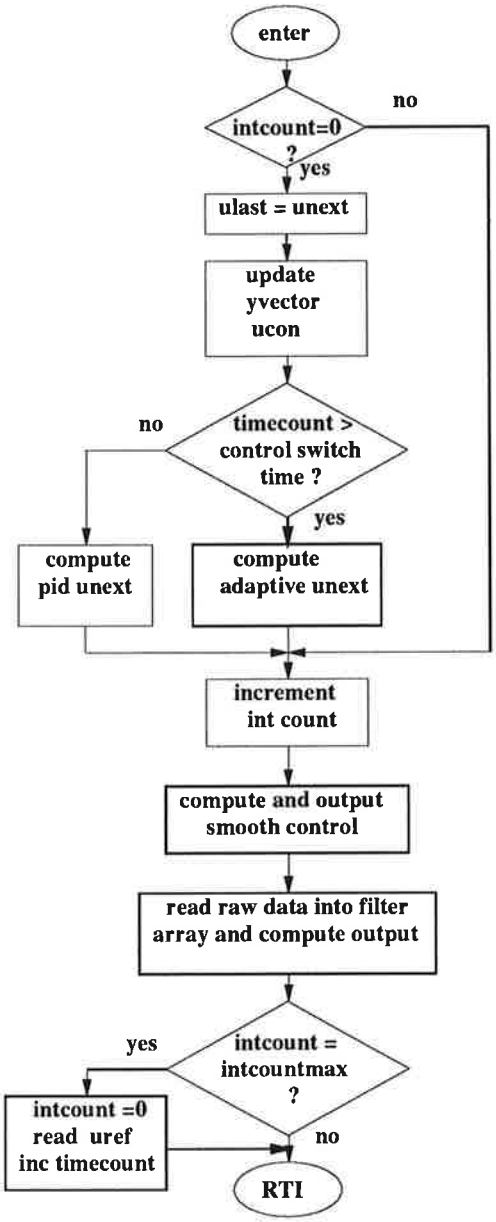


Figure 8. Interrupt service routine flowchart.

in Figure 8. Reading *Uref* can not be done in the main loop, since it might cause an interrupt during interrupt. Computing *unext* which is $u(k+1)$ may be in the main loop. We prefer to keep it in the interrupt service routine to keep 'what happens when' clear to understand. Data sampling and control smoothing is done every occurrence of the timer-0 interrupt.

Please note that when *int_count* changes from (*int_count_max* -1) to (*int_count_max*) it is reset to 0. Immediately upon entering the interrupt service routine, the next step in control staircase is computed and sent to the control port. Since the reading of the data takes a certain delay (40 μs in the hardware used), this is postponed until after the control smoothing. This is just a compromise.

Timing Considerations

By looking at the two flowcharts, one can see that it will be desirable that the interrupt during which *int_count* changes from (*int_count_max* -1) to (*int_count_max*)and then rest to (0), occurs at the end of the main loop. Then the processor should arrive at the first statement of the loop to wait for the interrupt in which *int_count* changes from 0 to 1. This is the ideal timing situation.

If a larger control interval is required, the variable *int_count_max* may be increased. Let us assume that a shorter control interval is required and the *int_count_max* is correspondingly reduced. The interrupt in which *int_count* changes from 0 to 1 would have occurred even before loop is executed once. In this case the *ucon* and *yvect* vectors are updated at the right instants. But the identification is completed without disturbing ϕ, even as further interrupt operations continue. Then synchronization with the beginning of the loop occurs when the interrupt in which *int_count* changes from 0 to 1 occurs again. The ϕ vector is updated by transferring *ucon* and *yvect* vectors to it. Now the next identification calculations are performed. This technique of keeping *ucon* and *yvec* vectors which are updated regularly, and a separate ϕ vector, which is updated with the most recent coherent data when required, adds a flexibility and avoids time conflicts. The price paid is that the parameter update is done only when it is possible, resulting in a compromise of tracking.

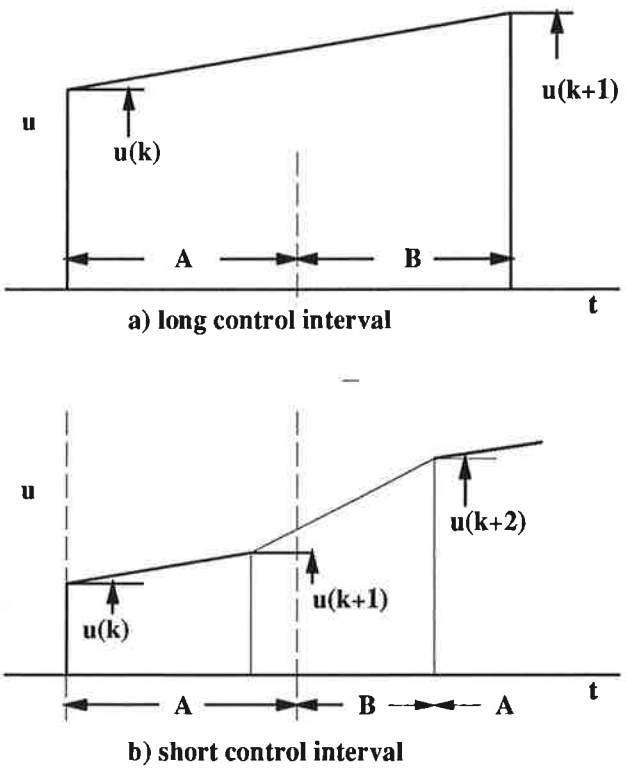


Figure 9. The time distribution.

Figure 9a shows the time taken by the identifier as duration A and the waiting time at the main loop's first instruction as duration B. In this case the control interval is sufficiently large. This is the case when a moderately fast process is controlled. During both periods interrupts continue to occur and control smoothing is done each interrupt.

Routine	Analog_In	update vector s & compute unext	change phi	Id_filter	DAB	filer size=100	others
time taken in microsec	40	20	5	230	40	30	15

Figure 10. Time taken by individual routines.

When a fast process is being controlled, the control interval is shorter. The situation is shown in Figure 9b. The duration A is larger than the control interval. The waiting time B is chosen to synchronize with the next start of the control sweep.

The filtering routine is listed in section 1 of the Appendix. The identification and control routines are given in Sections 2, 3, and 4 of the Appendix. The routines for data input and data output routines are given in Section 5 of the appendix. The other routines required for initializing and I/O operation etc are given in Section 6. Timer control support software is listed in Section 7 of the Appendix.

The time taken by each routine is listed in the table given in Figure 10. Even when the identification is done every control interval, the control interval can be as short as 400 μs, without PFOH. The filtering time depends on the order of the filter. Every interrupt causes an overhead of 15 μs in addition to the actual service time. The control rate can, however, be even faster, due to the flexibility of the software described earlier. This results in an adaptive controller whose data sampling is around 10 kHz and control speed is around 4 kHz.

7. Development Environment

The Hardware Setup

The hardware used for this project consists of:

a) The National Instruments Board NB-M10-16 which provides 16 channels of analog input with 12 bit resolution and 2 channels of analog output [6].

b) The National Instrument Board NB-DSP2300, which has the TMS320c30 digital signal processor [7].

c) The Macintosh II computer, which is the host for development. The items a and b are installed in the mother board slots of the host.

Development Tools

The software development tools used for this work consists of the following [8,9]:

a) A C-compiler which produces TMS320c30 assembly code. A 'shell' program and an 'interlist' utility are included in the compiler package. This compiler comes with a set of 'Tools', which consists of the batch files needed to run the compiler and 'header' files(.h files) needed for run time support, and 'libraries' which consists of support object and source libraries.

b) An assembler which translates assembly code into machine code (COFF) object files.

c) The archiver which enables a collection of files to be grouped into a single archive library. An rts.lib consists of standard run time support routines, compiler utility routines, and mathematical routines.

d) The linker which combines the object files, into a single executable object module, performing the relocation and resolution of external references. Such development produces a module that can be executed on a TMS320c30 target system.

e) The Debug2300 is the real time debugging program for the object module.

The compiler package runs on the Macintosh-II under the Macintosh Programmers Workshop (MPW). The above tools can not be run on a Macintosh without the MPW.

8. Experiments

The DSP based adaptive regulator has been used to control two different processes, one a moderately fast process (a servo) and the other a fast process (an analog mock up), to evaluate the performance. Experimental results are presented for both process.

Servo System

A simplified model of the servo is given by

$$G(s) = \frac{K\ a}{s(s+a)} \tag{33}$$

The pulse transfer function of the process with PFOH implementation is

$$G(q) = \frac{B(q)}{A(q)} = K\frac{b_0q^2 + b_1q + b_2}{q^2 - (1+e^{-aT})q + e^{-aT}} \tag{34}$$

where

$$b_0 = \frac{1}{a^2h} + \frac{h}{2} - \frac{1}{a} - \frac{\alpha}{a^2h}$$

$$b_1 = \frac{(2\alpha - 2)}{a^2h} + \frac{h(1-\alpha)}{2} + \frac{(1+\alpha)}{a} \tag{35}$$

$$b_2 = \frac{(1-\alpha)}{a^2h} - \frac{\alpha}{a} - \frac{h\alpha}{2}$$

and T is the control interval. The model for the desired behaviour used for control is

$$H_m(q) = \frac{B(q)}{q(q^2 + p_1q + p_2)} \cdot \frac{(1+p_1+p_2)}{B(1)} \tag{36}$$

The values of the model parameters p_1 and p_2 are selected to give $\zeta = 0.7$ and $\omega = 8$, corresponding to a sampling time of 0.1 sec. The controller polynomials are given as

$$\begin{aligned} R(q) &= r_0q^2 + r_1q + r_2 \\ S(q) &= s_0q + s_1 \\ \textit{and}\quad T(q) &= t_0q \end{aligned} \tag{37}$$

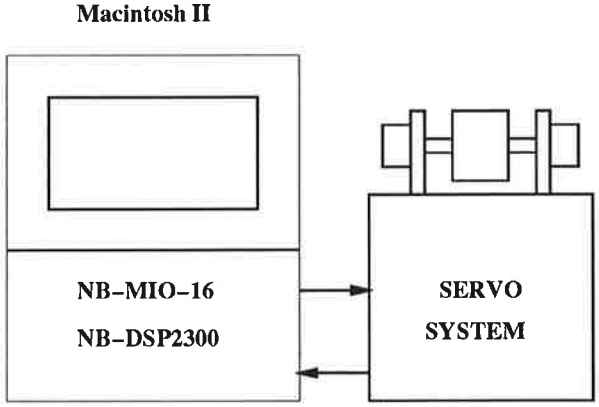


Figure 11. Experimental setup for servo.

For the purpose of experimental verification the plant parameters for the transfer function given in equation (33) were measured. The approximate values are $K = 31$, and $a = 0.62$. With these values, the parameters of the difference equation including the gain factors become

$$\begin{aligned} a_1 &= -1.94 & b_0' &= Kb_0 = 0.0315 \\ a_2 &= +0.94 & b_1' &= Kb_1 = 0.1242 \\ & & b_2' &= Kb_2 = 0.0305 \end{aligned} \tag{38}$$

By referring to Equation (31) and using the fact the clock period $\delta = 0.24$ micro-seconds the timing control variables can be set as

$$int\_interval = 576; \qquad int\_count\_max = 723 \tag{39}$$

The experimental setup is shown in Figure 11. A prefilter of order 100 with a cutoff frequency of 180 Hz was used, since there was very little low frequency noise. The filter delay was only $(100 \cdot 576 \cdot 0.24 = 6.92\text{ms})$. This is negligible in relation to the control interval of 0.1 sec. The parameters converged to the values given below.

$$\begin{aligned} a_1 &= -1.945 & b_0 &= 0.017 \\ a_2 &= \ \ 0.914 & b_1 &= 0.122 \\ & & b_2 &= 0.057 \end{aligned}$$

It can be noted that they are quite close to their true values. The results obtained are shown in Figure 12. The top part shows the servo output and the lower figure shows the input voltage to the servo. The output is accordance with the specifications, $\omega = 8$, and $\zeta = 0.7$. The controller parameters converge to

$$\begin{aligned} r_0 &= 1.000 & s_0 &= \ \ 7.116 \\ r_1 &= 0.860 & s_1 &= -4.905 \\ r_2 &= 0.303 & t_0 &= \ \ 1.863 \\ & & t_1 &= \ \ 0.000 \end{aligned}$$

In this case the high frequency gain of the controller is approximately 27. The effectiveness of the prefilter is clearly seen in Figure 12. With the noise level at the measured signal, the control signal would saturate without the prefilter. The effect of the PFOH for a few control intervals are shown in Figure 13. Since the number of smoothing steps is very large (723), the outputs appears linear without visible steps.

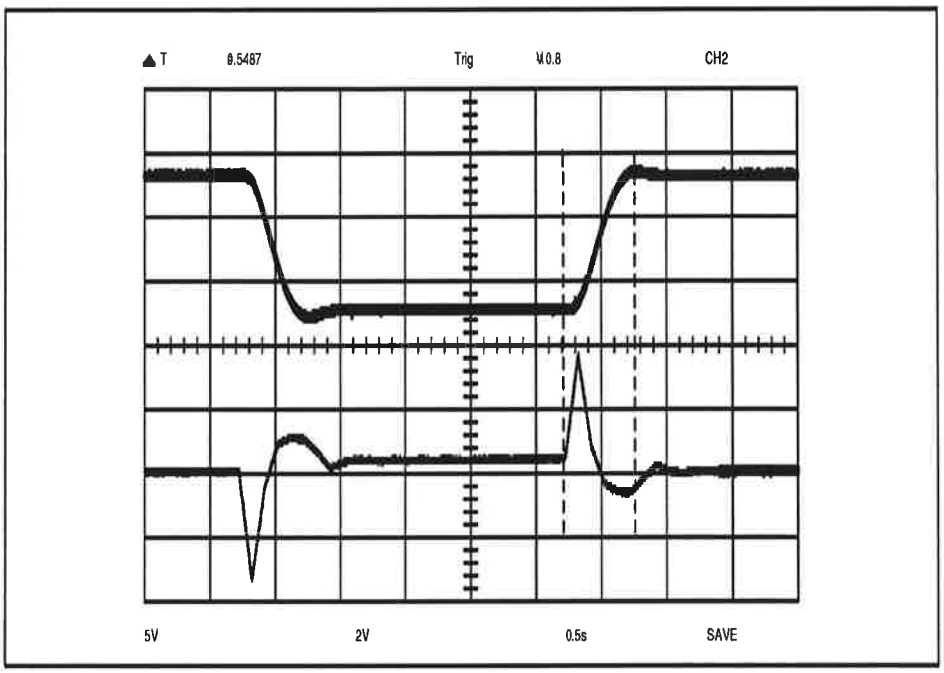


Figure 12. Test results of servo control.

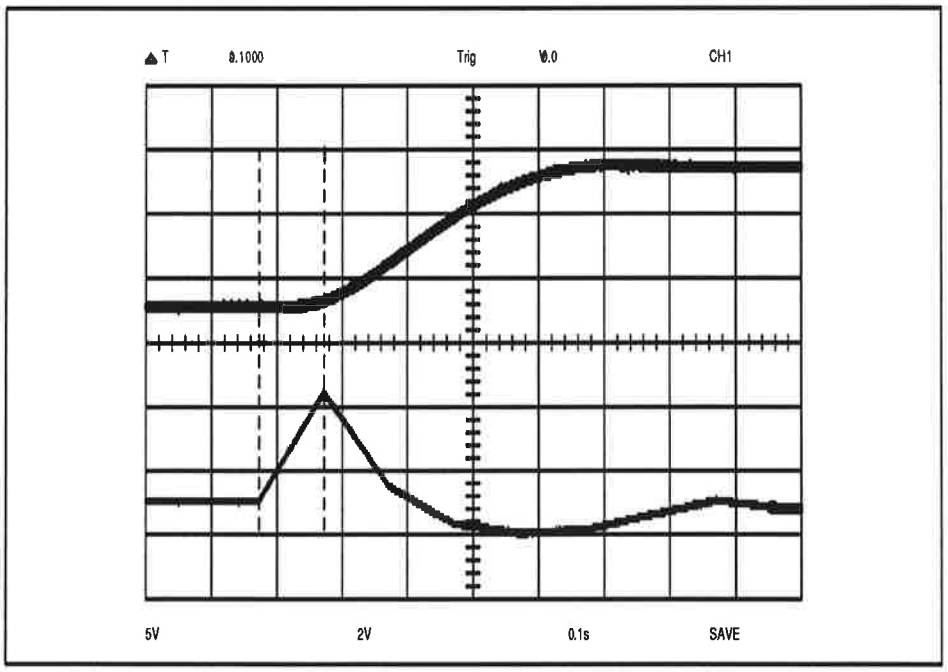


Figure 13. PFOH effect-723 steps between control changes.

Fast Process

In order to test the viability of controlling fast processes using the DSP based adaptive controller, a fast process using OP-Amps was built. The process transfer function is given by

$$G(s) = \frac{27.03}{s} \cdot \frac{270.3}{(s + 270.3)} \tag{40}$$

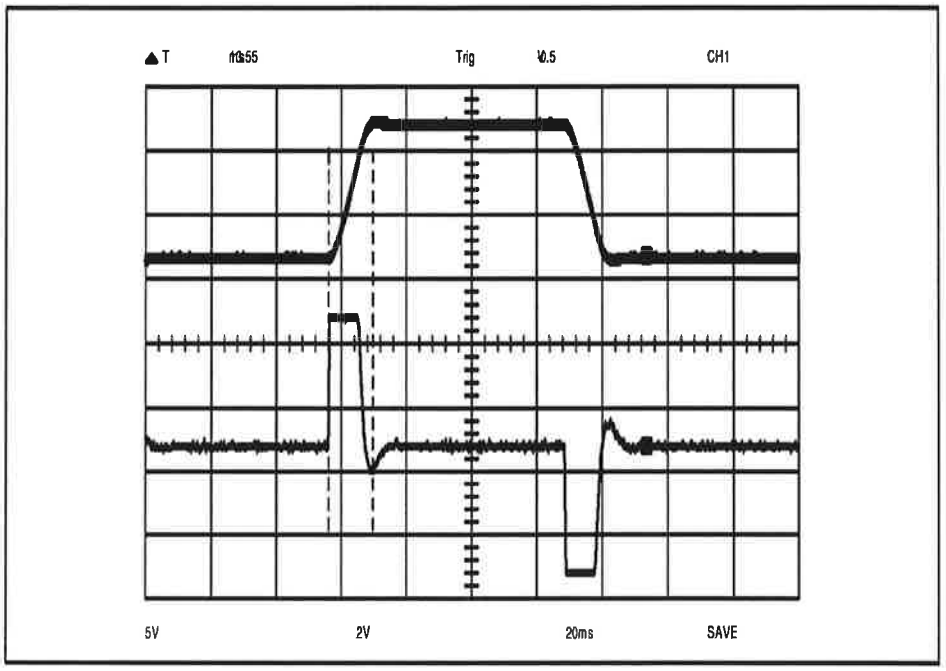


Figure 14. Response of the fast process.

The process has a time constant of 3.7 ms and an integrator. A desirable sampling period is 0.5 ms. Hence by referring to the Equations (31) and (32), the smoothing and control interval parameters are as given by

$$int\_interval = 417; \qquad int\_count\_max = 5$$

The desired model behaviour is chosen as in Equation (36), with $\zeta = 0.7$ $\omega = 600$. No zeros were cancelled. The controller and process polynomials identified are similar to the servo control described before. For this plant the parameters of the difference equation are

$$\begin{aligned} a_1 &= -1.8736 \\ a_2 &= \ \ 0.8736 \\ b_0' &= Kb_0 = 0.29441 \cdot 10^{-3} \\ b_1' &= Kb_1 = 1.13890 \cdot 10^{-3} \\ b_2' &= Kb_2 = 0.27518 \cdot 10^{-3} \end{aligned} \tag{41}$$

The identified parameters converged to the following values.

$$\begin{aligned} a_1 &= -1.789 & b_0 &= 0.651 \cdot 10^{-3} \\ a_2 &= \ \ 0.789 & b_1 &= 2.940 \cdot 10^{-3} \\ & & b_2 &= 1.114 \cdot 10^{-3} \end{aligned}$$

The controller parameters are

$$\begin{aligned} r_0 &= 1.000 & s_0 &= \ \ 50.522 \\ r_1 &= 0.171 & s_1 &= -35.003 \\ r_2 &= 0.049 & t_0 &= \ \ 15.524 \\ & & t_1 &= \ \ 0.000 \end{aligned}$$

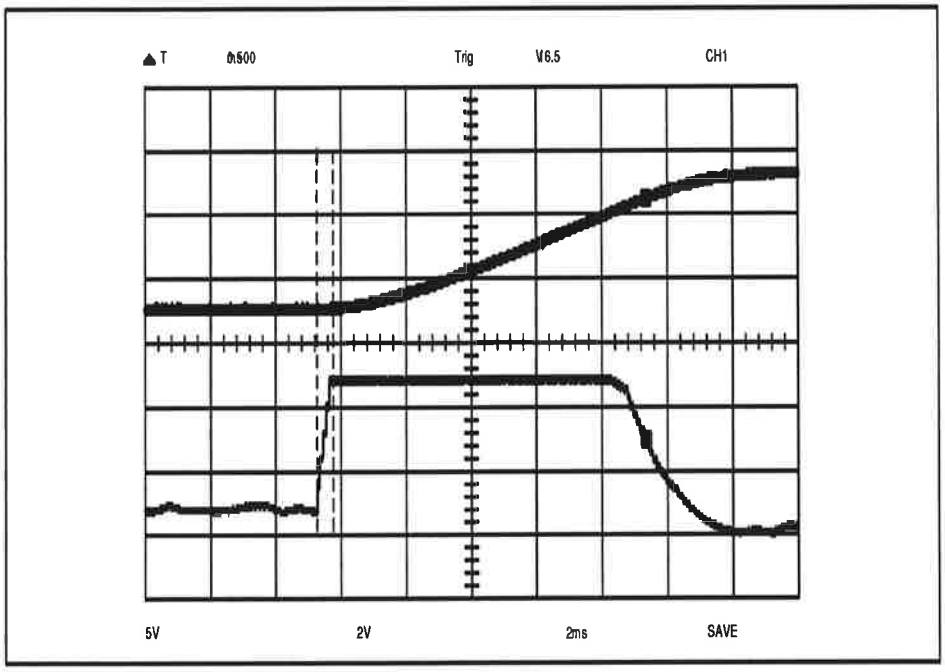


Figure 15. Effect of PFOH-5 steps between control changes.

The experimental results are shown in Figure 14, which shows the process input and output. The control smoothing effect is shown in Figure 15. The steps are visible, since $T = 5\text{h}$.

3.33 hHz Controller

It is possible to decrease the control interval further. If *int_count_max* is fixed to 3, the control interval would be 0.3 ms. The model parameters are

$$\begin{aligned} a_1 &= -1.922 & b_0 &= 0.107 \cdot 10^{-3} \\ a_2 &= 0.922 & b_1 &= 0.421 \cdot 10^{-3} \\ & & b_2 &= 0.103 \cdot 10^{-3} \end{aligned}$$

The identifier parameters converge to the following values.

$$\begin{aligned} a_1 &= -1.868 & b_0 &= 0.251 \cdot 10^{-3} \\ a_2 &= 0.868 & b_1 &= 0.771 \cdot 10^{-3} \\ & & b_2 &= 0.732 \cdot 10^{-3} \end{aligned}$$

The controller parameters are

$$\begin{aligned} r_0 &= 1.000 & s_0 &= 77.450 \\ r_1 &= 0.100 & s_1 &= -61.123 \\ r_2 &= 0.052 & t_0 &= 16.316 \\ & & t_1 &= 0.000 \end{aligned}$$

The response of the controller for this case is shown in Figure 16. The software setup is such that the identifier skips an update if the computation time

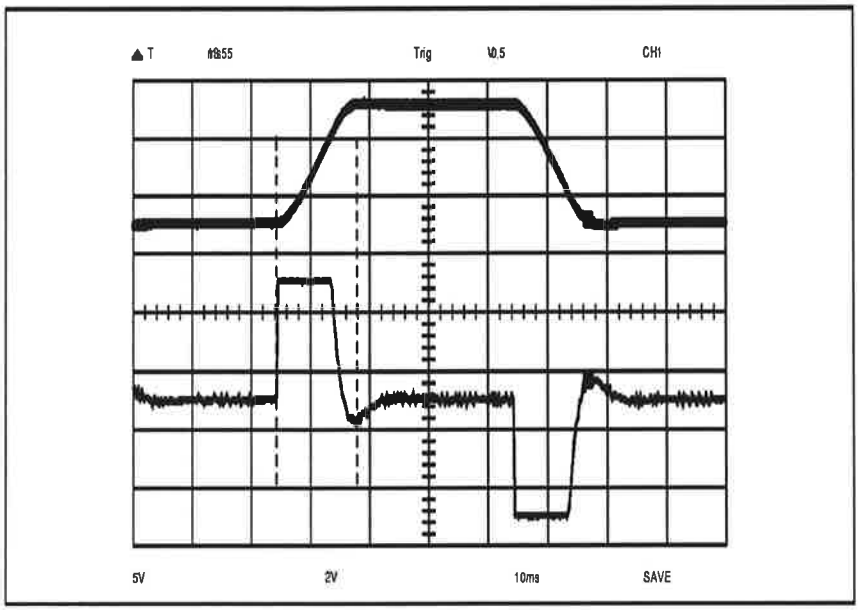


Figure 16. 3.33 kHz controller response.

exceeds the control interval. This is achieved by keeping a separate ϕ vector independent of control and output vectors and by updating it only when next identification begins. This allows the controller to work even at 4KHz. If it is pushed any further, the tracking process and hence the controller performance will deteriorate.

Fast Process Control with Filtering

In this section the test results of robust control with integral action and filtering is described. A filter with 10 coefficients is used to filter the data. The data sampling is done 8 KHz. The control interval is fixed to 0.5 ms. The delay due to filtering amounts to 1 control interval. The controller design is described in the last part of Section 5. The interval control parameters are set as

$$int\_interval \ = 521; \qquad int\_count\_max \ = 4$$

Hence control smoothing interval is 0.125 ms, which is same as the data sampling interval. The performance specification are taken as

$$\omega = 200 \ ; \ \zeta = 0.7$$

The observer poles are fixed to

$$q_1 = 0 \ ; \ q_2 = 0.2 \ ; \ q_3 = 0.1$$

The solution of the DAB equation requires the inverse of a $[9 \times 9]$ matrix in real time. Due to the absence of lower degree terms in the polynomial product $A_m(q)A_0(q)$, the matrix could be nicely partitioned into $[3 \times 3]$ and solved in real time. The actual parameters are given in Equation (41). The identified parameters converge to the values given below.

$$\begin{aligned} a_1 &= -1.869 & b_0 &= 0.327 \cdot 10^{-3} \\ a_2 &= \ \ 0.871 & b_1 &= 1.269 \cdot 10^{-3} \\ & & b_2 &= 1.504 \cdot 10^{-3} \end{aligned}$$

It can be seen that the parameter tracking has improved due to filtering. This is achieved in spite of the difficulty in tracking very small b-parameters. Also notice that the model structure used in the identification is not correct in this case. With the filtering there should actually be more b-parameters. A consequence of this is that there is more variability of the estimates. This is due to the fact that there is no set of parameters that will match all operating conditions. The parameters will change to make the best fit in each case. One set of controller parameters corresponding to the identified values given above are

$$\begin{aligned} r_0 &= 1.0000 & r_1 &= -0.2914 \\ r_2 &= -0.0085 & r_3 &= -0.1841 \\ r_4 &= -0.3332 & r_5 &= -0.1828 \\ s_0 &= 124.623 & s_1 &= -104.748 \\ s_2 &= -123.54 & s_3 &= 105.830 \\ s_4 &= 0.000 & t_0 &= 3.010 \\ t_1 &= -0.903 & t_2 &= 0.060 \\ t_3 &= 0.000 \end{aligned}$$

Another set was

$$\begin{aligned} r_0 &= 1.0000 & r_1 &= -0.390 \\ r_2 &= -0.065 & r_3 &= -0.190 \\ r_4 &= -0.239 & r_5 &= -0.115 \\ s_0 &= 63.616 & s_1 &= -49.157 \\ s_2 &= -62.859 & s_3 &= 49.914 \\ s_4 &= 0.000 & t_0 &= 2.070 \\ t_1 &= -0.621 & t_2 &= 0.041 \\ t_3 &= 0.000 \end{aligned}$$

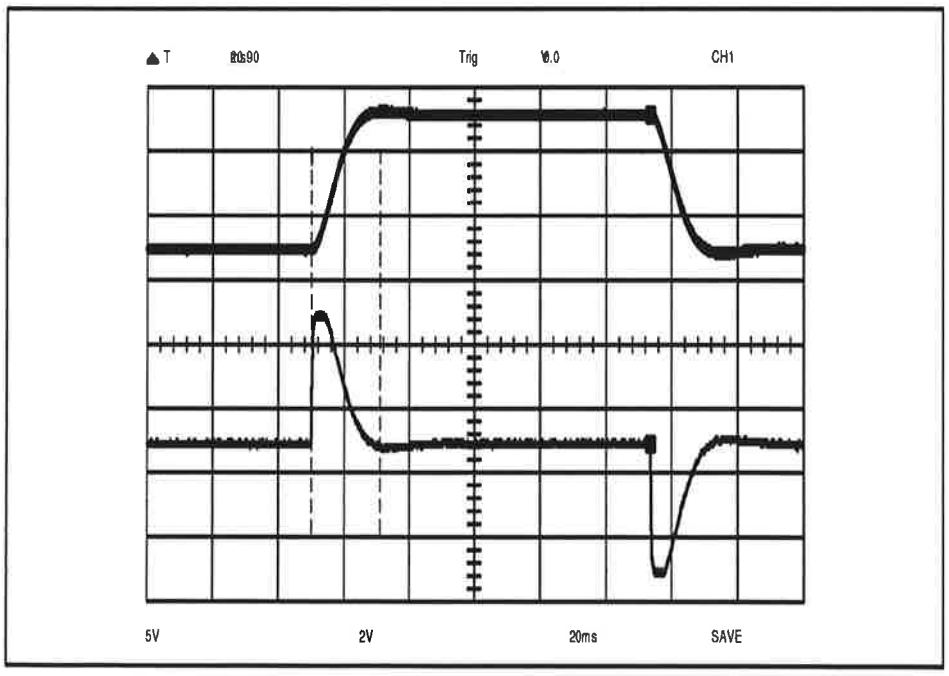


Figure 17. Response of the robust controller.

This gives an indication of the variability.

The high frequency gain of the controller is zero. The response is shown in Figure 17. It can be seen that the response of the process is good and according to specifications.

9. Conclusions

DSP implementation of an adaptive regulator has been described in this report. Since DSPs are specially designed fast fast filter computations, an adaptive prefilter can easily be incorporated as a part of the software. The idea of predictive first order hold has also been implemented. Multirate sampling is required to implement all the these features and it is achieved using the timer-0 available on the NB-DSP2300 system board. To keep the hardware and software systems simple, the adaptive filter sampling and the control smoothing required for the PFOH are both done at the same high frequency of around 10 kHz and control and identification calculations are done at a slower rate. It has been demonstrated that adaptive controllers operating at the rate of 3.33 khz can be easily achieved, even while performing adaptive prefiltering and post filtering and identifying 5 parameters. A higher rate of 4kz is possible. This opens up the possibility of implementing adaptive controllers for a variety of fast processes, which so far could not be controlled adaptively.

10. References

[1] ÅSTRÖM, K. J. and B. WITTENMARK (1984): *Computer Controlled Systems: Theory and Design*, second edition, Prentice Hall International Inc.

[2] ÅSTRÖM, K. J. and B. WITTENMARK (1989): *Adaptive Control*, Addison-Wesley Publishing Company.

[3] ROHR, C., VALAVANI, L. S., ATHANS, M., and STEIN, G. (1985): "Robustness of continuous-time adaptive control algorithm in the presence of unmodelled dynamics," *IEEE Transactions on Automatic Control*, **AC-30**, 881–889.

[4] KUO, B. (1977): *Digital Control Systems*, SRL Publishing Company, Illinois.

[5] BERNHARDSSON, B. (1990): "The Predictive First Order Hold Circuit," *Proceedings of the 29th IEEE Conference on Decision and Control*, Honolulu, pp. 1890–1891.

[6] ANONYMOUS (Aug 1990): *NB-DSP2300 User Manual*, National Instruments Corporation, Austin, Texas, USA.

[7] ANONYMOUS (Nov 1989): *NB-MIO-16 User Manual*, National Instruments Corporation, Austin, Texas, USA.

[8] ANONYMOUS (Aug 1990): *TMS320c30 Reference Guide*, Texas Instruments Corporation, Texas, USA.

[9] ANONYMOUS (1986): *Digital Signal Processing with TMS320 family.*, Texas Instruments Corporation, Texas, USA.

Appendix

The various routines developed for this project are listed in the following subsections. The support routines supplied by vendors are included in the last section, for completion

1. Filter

The 'firfilt.asm' is the assembly language routine that calculates the filtered output.

```
*
*firfilt(current sample,filterlength,coefficients,sample arrray)
* -FP(2), -FP(3), -FP(4), -FP(5)
*
FP .set AR3
.globl _firfilt

_firfilt:
PUSH FP
LDI  SP,FP ;R0,R1,R2,R3,AR1,AR2 used; no need to preserve them

LDF  0.0,R0
LDF  0.0,R2
*
*here we update AR6 to point the current sample by using sampleshift variable
*
LDI *-FP(5),AR1  ;beginning address of sample array in AR1
ADDI @sampleshift,AR1 ;point to the current sample fillup location
*
*fill in the sample
*
LDF  *-FP(2),R3 ;take the sample
STF  R3,*AR1 ;fill  the current sample
*
*here we update AR1 to point the current duplicate using the sampleshift
*
ADDI *-FP(3),AR1 ;shift your pointer up by filter length
*
* fill duplicate
*
STF  R3,*AR1 ; sample from R3 to duplicate area
*
* prepare address of first coefficient in AR2
*
LDI *-FP(4),AR2 ;address of first filter coefficient in AR2
*
* calculate the filter output
*
LDI *-FP(3),RC ;load repeat counter filterlength
SUBI 1,RC ;reduce by one
RPTS RC
MPYF3 *AR2++(1),*AR1--(1),R2 ;multiply and move increment pointer
|| ADDF3 R2,R0,R0
lastadd: ADDF R2,R0 ; last product added and filter output is in R0
*
* update sampleshift to point next location
*
```

```
LDI @sampleshift,AR1
ADDI 1,AR1
CMPI *-FP(3),AR1
BNZ  not_through; if sampleshift reaches filterlength,reset to 0
LDI  0,AR1
not_through:
STI AR1,@sampleshift
*
* restore registers
*
POP FP
RETSU
*********************************************
* define initialised variable "sampleshift"  *
*********************************************
.bss sampleshift,1
.sect ".cinit"
.word 1,sampleshift
.word 00000000h
.end
```

2. Servo Control Routine

The 'filt pfoh SLOW.c' is the C-language programme that is used to control the servo system. This programme implements a controller with a filter of order 100 and the predictive first order hold. The programme listing is given below.

```
#include "DSP.h"
#include "int_DAQ.h"
#include "Analog.h"


long *nb_page0 = (long *) 0x805000;
int32 slot_base = 0xfd000000;

/* we just want to use the PID controller initially */

float past_integral,integral,con_coef_0,con_coef_1;
float Kp=0.7500;
float Ti=0.9;
float Td=0.135;

/* filter sampling time, control interval,runtime specifications    */
/*__________________________________________________________________*/
float h;
/* filter sampling interval =(clock = 0.24 micro-sec)x int_interval */
int32 int_interval = 576; /* timer-0 parameter */
int32 int_count_max = 723;
/* this decides the major control interval */
int32   run_time = 900; /* the total run time */
int32 control_switch_time =20;
/* when to switch from pid to adaptive control */
float ester;
float ester_min=0.00095;  /* equation error low limit */
float ester_max=2.00;
float dynamic_max;     /* variable upper limit for ester */

/*    identifier forgetting factor covariance etc.             */
/*____________________________________________________________*/
float lambda = 0.98;
```

```
float covare = 20000.0;

/*  adaptive controller performance specifications        */
/*_____________________________________________________________*/
float omega =8.0;
float zeta =0.7;
float   am[5];
/* to check any value using analog channel-0 in real time     */
float multiplier =1.0;
float probe=0.0;
int32 display_parameter =4;


typedef void (*funcpointer)();   /* define type function pointer */
funcpointer *Timer0vector = (funcpointer*) 0x001000f8;
/*user timer0 interrupt */
int32 timecount;      /* used for timing out the programme */


float fir100[] = {
0.000512854372582206, 0.000516450462259756, 0.000518204762982123,
0.000515450080616448, 0.000504004154825930, 0.000478336804016702,
0.000431852158133572, 0.000357278064969900, 0.000247148811983553,
9.43617917041610e-05, -0.000107216074233148, -0.000362120203258178,
-0.000672669271820524, -0.00103838733503633, -0.00145549925350262,
-0.00191653001825939, -0.00241003558951859, -0.00292048851294244,
-0.00342833599434955, -0.00391024150320646, -0.00433951359100031,
-0.00468671774693478, -0.00492045909327197, -0.00500831588420940,
-0.00491789645457208, -0.00461798579340490, -0.00407974259062006,
-0.00327790367905508, -0.00219195047381307, -0.000807191437446020,
0.000884284154197781, 0.00288282280617221, 0.00518063061306771,
0.00776143473774329, 0.0106004052421958, 0.0136643510817312,
0.0169121944326875, 0.0202957174770252, 0.0237605657005187,
0.0272474820682747, 0.0306937375059100, 0.0340347152960946,
0.0372056006176556, 0.0401431217742675, 0.0427872868837090,
0.0450830590554292, 0.0469819144235257, 0.0484432307925528,
0.0494354599825375, 0.0499370440382912, 0.0499370440382912,
0.0494354599825375, 0.0484432307925528, 0.0469819144235257,
0.0450830590554292, 0.0427872868837090, 0.0401431217742675,
0.0372056006176556, 0.0340347152960946, 0.0306937375059100,
0.0272474820682747, 0.0237605657005187, 0.0202957174770252,
0.0169121944326875, 0.0136643510817312, 0.0106004052421958,
0.00776143473774329, 0.00518063061306771, 0.00288282280617221,
0.000884284154197782, -0.000807191437446021, -0.00219195047381307,
-0.00327790367905509, -0.00407974259062006, -0.00461798579340490,
-0.00491789645457208, -0.00500831588420941, -0.00492045909327198,
-0.00468671774693479, -0.00433951359100031, -0.00391024150320647,
-0.00342833599434955, -0.00292048851294244, -0.00241003558951859,
-0.00191653001825939, -0.00145549925350262, -0.00103838733503633,
-0.000672669271820525, -0.000362120203258179,-0.000107216074233148,
9.43617917041610e-05, 0.000247148811983553, 0.000357278064969901,
0.000431852158133572, 0.000478336804016702, 0.000504004154825930,
0.000515450080616448, 0.000518204762982124, 0.000516450462259756,
0.000512854372582206
 };
extern float firfilt(float j,int32 filterlength,float fir100[],
                          float samplebuffer[]);
int32 filterlength = 100;
float samplebuffer[200]; /* twice the size of filter length */

/*   Needed for Identifier routines : all are global variables */

float phi[6],theta[6],eltrans[6][6],diag[6];
```

```
float r[6],s[6],t[6];   /* more than necessary */
float yvector[5],uref[5],ucon[5],error[5]; /* more than necessary */
float uread,uout,ulast,unext,yraw,yfilt,ic,icmax,hm;
float hm;  /* hm is the control interval */ /* forgetting factor */
int32 int_count;/* tracks the number of interrupts that occured  */
int na=2;    /* number of a-parameters or a1,a2 parameters*/
int nb=3;        /* number of b-parameters or b0,b1,b2 parameters */
int n;   /*   total number of parameters to be identified   */


   /*-----------------------------*/

void c_int01();

main()
{
  Init_MIO16(slot_base);
  Reset_MIO16(slot_base);
  Setup_MIO16(5, 0, 5, 2, 0, 0, 0);

 *Timer0vector = c_int01;   /* address of c_int01 placed in vector  */
  *(int*)Timer0vector |= 0x60000000; /* convert it to a branch instr */

 n = na+nb;
 h = int_interval *0.24e-06;
 init_uy_vectors();

 init_ident_matrices();

 compute_model();


     /* first time operation  */

  Analog_In(InChannel_1,gain_1,&yraw); /*  servo position  */
yfilt = yraw;
Analog_In(InChannel_0,gain_1,&uread);
uout= Kp *( uread -yfilt); /*first control is proportional */
ulast=uout;
Analog_Out(OutChannel_1,uout); /*control to plant  */

/* precalculate the pid- controller coefficiants */

    con_coef_0 =  (Td + (int_count_max * h))/(int_count_max *h) +1.0
       + ((int_count_max * h )/Ti);
con_coef_1 = -(Td + (int_count_max * h))/(int_count_max *h);
past_integral = 0.0;

  /* setup and start the timer-0 and enable that interrupt */

  Setup_Timer(0,0x000003c3,int_interval,0x00000000);
  asm(" OR 0100h,IE");  /* enable Timer-0 interrupt */
  int_count=0;
  timecount=0;
        /*  The beginning of the main loop */

  while (timecount < run_time) /* until timeout */
  {
   while( int_count != 1 ) {} ;
change_phi(); /* prepare consolidated data vector*/
ld_filter(n); /* the identifier routine: */
solve_dab();  /* compute the controller-rst parameters:*/
probe=theta[display_parameter]* multiplier;
```

```
Analog_Out(OutChannel_0,probe);
   };  /*  The main loop section */


time_is_up:
  asm(" AND 0feffh,IE ");  /* disable timer-0 interrupt */
  Hold_Timer(0);
  Analog_Out(OutChannel_1,0.0); /* output zero before quiting */
}

      /* The Timer-0 interrupt service routine */
  /* ------------------------------------ */
void c_int01()
{
if(int_count ==0)
{  ulast=unext;
   Update_vectors();
if( timecount < control_switch_time)
{ Compute_pid_unext(); }
else
{ Compute_adaptive_unext(); }
   }

int_count++;
Smooth_control();
Analog_In(InChannel_1,gain_1,&yraw);
yfilt=firfilt(yraw,filterlength,fir100,samplebuffer);
if(int_count ==int_count_max )
{ Analog_In(InChannel_0,gain_1,&uread);
int_count =0;
timecount++;
}
}

       /* The routine to calculate adaptive u(t+j.h) */
/*_________ R*u =T*u-S*y _____________________*/
Compute_adaptive_unext()
{
 unext=t[0] * uref[0]-s[0]*yvector[0]-s[1]*yvector[1];
 unext=unext - r[1]*ucon[0]-r[2]*ucon[1];
if(unext > 0.99) unext = 0.99;
if(unext < -.99) unext =-0.99;
}
     /* The routine to calculate pid controller u(t+j.h) */
     /* ---------------------------------------------- */
Compute_pid_unext()
{
 integral = ((error[0]+ error[1] )/(2.0 * Ti)) *(icmax * h);
 unext = Kp *(con_coef_0 * error[0]+ con_coef_1 * error[1]);
 unext = unext + Kp *( integral+ past_integral);
 if(unext < -0.990 ) unext = -0.990;
 if(unext > 0.990 ) unext = 0.990;
 past_integral = past_integral + integral;
 }
   /*       Smooth control                    */
   /* ------------------------------------  */
Smooth_control()
{
icmax = int_count_max;
ic=int_count;
uout = ulast + ((unext-ulast)*ic)/icmax;
if(uout> 0.98) uout = 0.98;
if(uout< -0.98) uout =-0.98;
```

```
Analog_Out(OutChannel_1,uout);
}
       /*    Initialise uy vectors                  */
/*_____________________________________________*/
init_uy_vectors()
{ int i;
for( i=0; i<5; i++)
{ yvector[i] =0.0;
  uref[i] = 0.0;
  ucon[i] = 0.0;
  error[i]= 0.0;
 }
}
       /* initialise matrices needed for identifier */
       /* _____________________________________________*/
init_ident_matrices()
{ int i,j;
for(i=0; i<6; i++)
{ phi[i] = 0.0;
  theta[i]= 0.001;
  r[i] = 0.1;
  s[i] = 0.1;
  t[i] = 0.1;
  diag[i] = covare;
  for(j=0; j<6;j++)
    { if(i == j)
      eltrans[i][j] = 1.0;
 else
  eltrans[i][j] = 0.0;
}
 }
  diag[0]= lambda;
 }
   /*   Update uy vectors               */
    /*_____________________________________________*/
Update_vectors()
{ int i;
for( i= 2 ; i>0; i--)
    { uref[i] = uref[i-1];
   }
for(i=2; i>0; i--)
 {  yvector[i]= yvector[i-1];
  error[i]= error[i-1];/* needed for pid control */
  }


for(i=3; i>0; i--)
 { ucon[i] = ucon[i-1];
   }
  error[0]= uread - yfilt;
  uref[0] = uread;
  yvector[0]= yfilt;
  ucon[0]= ulast;  /* control computed last time is ucon(t) now */
}

    /*  the square root identification routine */
/*_____________________________________________*/
ld_filter( int en)
{  int i,j; /* local variables  */
float e,w;

  /* diag(6),eltrans(6)(6),theta(6),phi(6) are global variables */
  /* lambda: global variable only used but not affected */
```

```
diag[0]= lambda;
e = phi[0];

for(i=1;i<=en;i++)
{e=e-theta[i]*phi[i];
}
ester=e;
if(ester > 0)ester=ester;
if(ester < 0)ester=-ester;
if(ester< ester_min) return;
dynamic_max=ester_max*(10000.0+timecount)/(10000.+100.*timecount);
      /*initially we accept large errors for correction; later we don't*/
if(ester> dynamic_max) return;

for( i=1; i<=en; i++)
 { w = phi[i];
   for(j=i+1; j<=en;j++)
    { w = w + phi[j]*eltrans[i][j];
 }
   eltrans[0][i] = 0.0;
   eltrans[i][0] = w;
  }
 for( i =en ; i>=1; i--)

 dyadic_reduction(0,i,en);

/* called with i varying from en down to 1 */
/* call involves and alters 0-th row and i-th row of [eltrans]*/
/* diag(0) and diag(i) are also altered by the call */

for(i=1; i<=en ; i++)

  { theta[i]= theta[i] + eltrans[0][i] * e ;
   diag[i] = diag[i] / lambda;
   }
}

    /*    the dyadic reduction routine called by ld_filter  */
/*_____________________________________________________*/
dyadic_reduction(int i0, int i1, int i2)
{
/* eltrans[0-th row] and eltrans[ i1-th row],diag(0),diag(i1) are passed */

int  j;      /* local variables */
float w1,w2,b1,gama;
float mzero = 1.0e-10;

if( diag[i1] < mzero)/* diag[i1] is "beta" the ith element */
  { diag[i1] = 0.0 ;
  }

b1 = eltrans[i1][i0];
/* 0-th element of b-vector i-e ith row [eltrans] */
w1 = diag[i0];
/* "alpha", the top of the diagonal:i0 = 0 always  */
w2 = diag[i1] * b1; /* w2 = beta * b1       */
diag[i0] = diag[i0]+ w2 *b1;
if( diag[i0] > mzero)
 { diag[i1] = diag[i1]*w1 /diag[i0];
   gama = w2 / diag[i0];
   for( j=i1; j<= i2; j++)
    { eltrans[i1][j] = eltrans[i1][j] - b1 * eltrans[i0][j];
```

```
      eltrans[i0][j] = eltrans[i0][j] + gama * eltrans[i1][j];
 }
  }
 }
    /* the dab equation is solved only for a 2nd order case */
    /*_____________________________________________________*/
solve_dab()
{ float ex1,ex2,ex3,y1,y2,y3,a1,a2,b0,b1,b2,bmprime;
a1 = -theta[1];
a2 = -theta[2];
b0 =  theta[na+1];
b1 =  theta[na+2];
b2 =  theta[na+3];
ex1 = b1-a1*b0;
ex2 = b0-b2/a2;
ex3 = am[2] - a2 - a1*am[1] + a1*a1;
y1  = b2 - a2*b0;
y2  = b1 - a1*b2/a2;
y3  = -a2*am[1] + a2*a1;
s[0] = (ex3*y2 - ex2*y3)/(ex1*y2 - y1 *ex2);
s[1] = ( ex3 - ex1*s[0] )/ ex2;
r[0] =1.0;
r[1] = am[1]-a1-b0*s[0];
r[2] =-b2 * s[1] / a2;
bmprime = (1. + am[1] +am[2])/ (b0 + b1 + b2);
t[0] = bmprime;
}
   /*  update the phi vector used by the ld_filter  */
    /*_____________________________________________*/
change_phi()
{  int i;
for( i=0; i<=na ; i++)
    { phi[ i ] = yvector[i];
   }
for( i=0; i< nb ; i++)
   { phi[na+1+i] = ucon[i];
   }
}
compute_model()
{ hm = h * int_count_max;
am[1] = -2.0 * exp( -zeta * omega *hm)*
cos(sqrt(1.0-zeta*zeta)*omega*hm);
am[2] = exp(-2.0 *zeta *omega*hm);
}
       /*------------------- END -------------------*/
```

3. Fast System Control Routine

The 'nofilt pfoh FAST.c' is the C-programme that is used to control the fast system without filtering. The listing is given below.

```
#include "DSP.h"
#include "int_DAQ.h"
#include "Analog.h"


long *nb_page0 = (long *) 0x805000;
int32 slot_base = 0xfd000000;

      /* we just want to retain the PID controller in the begininig */
```

```
float past_integral,integral,con_coef_0,con_coef_1;
float Kp=0.7500;
float Ti=0.9;
float Td=0.135;

/* filter sampling time, control interval,runtime specifications    */
 /*-------------------------------------------------------------------*/
float h;
/* filter sampling interval = (clock = 0.24 micro-sec) x int_interval */
int32 int_interval = 417;
/* timer-0 parameter */
int32 int_count_max = 3;/* this decides the major control interval */
int32   run_time = 120000;
/* the total run time */
int32 control_switch_time =250;/* when to switch to adaptive control */
float ester;
float ester_min=0.00095; /* lower limit for estimation error */
float ester_max=2.00;
float dynamic_max;/* variable upper limit for estimation error */

/*    identifier forgetting factor covariance etc.              */
/*-------------------------------------------------------------*/
float lambda = 0.998;
float covare = 20000.0;

/*  adaptive controller performance specifications        */
/*------------------------------------------------------*/
float omega =600.0;
float zeta =0.7;
float   am[5];
/*      to check any value using analog channel -0            */
float multiplier =10.0;
float probe=0.0;
int32 display_parameter =4;


typedef void (*funcpointer)();
/* define type function pointer */
funcpointer *Timer0vector = (funcpointer*) 0x001000f8;
/*user timer0 interrupt */
int32 timecount;      /* used for timing out the programme */


/* Needed for Identifier routines : all are global variables */

float phi[6],theta[6],eltrans[6][6],diag[6];
/* to identify 5 parameters */
float r[6],s[6],t[6];   /* more than necessary */
float yvector[5],uref[5],ucon[5],error[5];/* more than necessary */
float uread,uout,ulast,unext,yraw,yfilt,ic,icmax,hm;
float hm;     /* hm is the control interval */
int32 int_count; /* tracks the number of interrupts that occured  */
int na=2;    /* number of a-parameters or a1,a2 parameters*/
int nb=3;    /* number of b-parameters or b0,b1,b2 parameters */
int n;   /* total number of parameters to be identified   */


  /*-----------------------------*/

void c_int01();

main()
{
```

```
  Init_MIO16(slot_base);
  Reset_MIO16(slot_base);
  Setup_MIO16(5, 0, 5, 2, 0, 0, 0);

 *Timer0vector = c_int01;
/* address of c_int01 placed in vector  */
  *(int*)Timer0vector |= 0x60000000;
/* convert it to a branch instr */

 n = na+nb;
 h = int_interval *0.24e-06;
 init_uy_vectors();

 init_ident_matrices();

 compute_model();


    /* first time operation  */

  Analog_In(InChannel_1,gain_1,&yraw);
yfilt = yraw;
Analog_In(InChannel_0,gain_1,&uread); /* reference value*/
uout= Kp *( uread -yfilt);  /*first control proportional*/
ulast=uout;
Analog_Out(OutChannel_1,uout);
/*control to plant via the out-channel-1  */

/* precalculate the pid- controller coefficiants: */

   con_coef_0 =  (Td + (int_count_max * h))/(int_count_max *h) +1.0
+ ((int_count_max * h )/Ti);
   con_coef_1 = -(Td + (int_count_max * h))/(int_count_max *h);
   past_integral = 0.0;

  /* setup and start the timer-0 and enable that interrupt */

  Setup_Timer(0,0x000003c3,int_interval,0x00000000);
  /* set up and start timer-0 */
  asm(" OR 0100h,IE");      /* enable Timer-0 interrupt */
  int_count=0;
  timecount =0;
       /*  The beginning of the infinite loop */

  while (timecount < run_time) /* until timeout */
  {
   while( int_count != 1 ) {} ;
change_phi();
/* prepare consolidated data vector   */
ld_filter(n); /* the identifier routine:  */
solve_dab();  /* compute the controller-rst parameters:*/
probe=theta[display_parameter]* multiplier;
Analog_Out(OutChannel_0,probe);
   };         /*  The main loop section */

time_is_up:
asm(" AND 0feffh,IE "); /* disable timer-0 interrupt */
   Hold_Timer(0);
   Analog_Out(OutChannel_1,0.0);
        /* output zero before quiting */
}

     /* The Timer-0 interrupt service routine */
```

```
      /* ____________________________________ */
void c_int01()
{ if(int_count ==0)
{  ulast=unext;
   Update_vectors();
if( timecount < control_switch_time)
{ Compute_pid_unext(); }
else
{ Compute_adaptive_unext(); }
   }

int_count++;
Smooth_control();
if(int_count ==int_count_max )
{ Analog_In(InChannel_1,gain_1,&yraw);
yfilt=yraw;
Analog_In(InChannel_0,gain_1,&uread);
int_count =0;
timecount++;
}
}

      /* The routine to calculate adaptive u(t+j.h) */
      /*_________ R*u =T*u-S*y _____________________*/
Compute_adaptive_unext()
{
 unext=t[0]*uref[0]-s[0]*yvector[0];
 unext= unext-s[1]*yvector[1]-r[1]*ucon[0]-r[2]*ucon[1];
if(unext > 0.99) unext = 0.99;
if(unext < -.99) unext =-0.99;
}
     /* The routine to calculate pid controller u(t+j.h) */
     /* ____________________________________________ */
Compute_pid_unext()
{
 integral = ((error[0]+ error[1] )/(2.0 * Ti)) *(icmax * h);
 unext = Kp *(con_coef_0 * error[0]+ con_coef_1 * error[1]);
 unext = unext + Kp *( integral+ past_integral);
 if(unext < -0.990 ) unext = -0.990;
 if(unext > 0.990 ) unext = 0.990;
 past_integral = past_integral + integral;
 }
    /*       Smooth control                    */
  /* ____________________________________  */
Smooth_control()
{
icmax = int_count_max;
ic=int_count;
uout = ulast + ((unext-ulast)*ic)/icmax;
if(uout> 0.98) uout = 0.98;
if(uout< -0.98) uout =-0.98;
Analog_Out(OutChannel_1,uout);
}
    /*    Initialise uy vectors                */
    /*_____________________________________________*/
init_uy_vectors()
{ int i;
for( i=0; i<5; i++)
{ yvector[i] =0.0;
  uref[i] = 0.0;
  ucon[i] = 0.0;
  error[i]= 0.0;
 }
```

```
}
   /* initialise matrices needed for identifier */
   /* ----------------------------------------*/
init_ident_matrices()
{ int i,j;
for(i=0; i<6; i++)
{ phi[i] = 0.0;
  theta[i]= 0.001;
  r[i] = 0.1;
  s[i] = 0.1;
  t[i] = 0.1;
  diag[i] = covare;
  for(j=0; j<6;j++)
    { if(i == j)
      eltrans[i][j] = 1.0;
 else
  eltrans[i][j] = 0.0;
}
 }
  diag[0]= lambda;
 }
       /*   Update uy vectors              */
  /*---------------------------------------*/
Update_vectors()
{ int i;
for( i= 2 ; i>0; i--)
    { uref[i] = uref[i-1];
   }
for(i=2; i>0; i--)
 {  yvector[i]= yvector[i-1];
  error[i]= error[i-1]; /* nedded for the pid */
  }

for(i=3; i>0; i--)
 { ucon[i] = ucon[i-1];
   }
  error[0]= uread - yfilt;
  uref[0] = uread;
  yvector[0]= yfilt;
  ucon[0]= ulast;
/* control computed last time is ucon(t) now */
}

 /*  the square root identification routine */
/*---------------------------------------*/
ld_filter( int en)
{  int i,j; /* local variables  */
float e,w;

  /* diag(6),eltrans(6)(6),theta(6),phi(6) are global*/

diag[0]= lambda;
e = phi[0];

for(i=1;i<=en;i++)
{e=e-theta[i]*phi[i];
}
ester=e;
if(ester > 0)ester=ester;
if(ester < 0)ester=-ester;
if(ester< ester_min) return;
dynamic_max=ester_max*(10000.0+timecount)/
                        (10000.+100.*timecount);
```

```
if(ester> dynamic_max) return;
/* variable upper limit for ester */

for( i=1; i<=en; i++)
 {
   w = phi[i];
   for(j=i+1; j<=en;j++)
    { w = w + phi[j]*eltrans[i][j];
 }
   eltrans[0][i] = 0.0;
   eltrans[i][0] = w;
  }
 for( i =en ; i>=1; i--)

 dyadic_reduction(0,i,en);

/* called with i varying from en down to 1 */
/* call involves and alters 0-th row and i-th row of [eltrans]*/
/* diag(0) and diag(i) are also altered by the call */

for(i=1; i<=en ; i++)

  { theta[i]= theta[i] + eltrans[0][i] * e ;
   diag[i] = diag[i] / lambda;
   }
}

      /* the dyadic reduction routine called by ld_filter */
       /*_____________________________________________________*/
dyadic_reduction(int i0, int i1, int i2)
{
/* eltrans[0-th row] and eltrans[ i1-th row],diag(0),diag(i1) */

int  j;      /* local variables */
float w1,w2,b1,gama;
float mzero = 1.0e-10;

if( diag[i1] < mzero) /* diag[i1] is "beta" the ith element*/
  { diag[i1] = 0.0 ;
  }

b1 = eltrans[i1][i0];
/* 0-th element of b-vector i-e ith row [eltrans] */
w1 = diag[i0];
w2 = diag[i1] * b1; /* w2 = beta * b1       */
diag[i0] = diag[i0]+ w2 *b1;
if( diag[i0] > mzero)
 { diag[i1] = diag[i1]*w1 /diag[i0];
   gama = w2 / diag[i0];
   for( j=i1; j<= i2; j++)
    { eltrans[i1][j] = eltrans[i1][j] - b1 * eltrans[i0][j];
      eltrans[i0][j] = eltrans[i0][j] + gama * eltrans[i1][j];
 }
  }
 }
/* the dab equation is solved only for 2nd order case */
/*_____________________________________________________*/
solve_dab()
{ float ex1,ex2,ex3,y1,y2,y3,a1,a2,b0,b1,b2,bmprime;
a1 = -theta[1];
a2 = -theta[2];
b0 =  theta[na+1];
b1 =  theta[na+2];
```

```
b2 =  theta[na+3];
ex1 = b1-a1*b0;
ex2 = b0-b2/a2;
ex3 = am[2] - a2 - a1*am[1] + a1*a1;
y1  = b2 - a2*b0;
y2  = b1 - a1*b2/a2;
y3  = -a2*am[1] + a2*a1;
s[0] = (ex3*y2 - ex2*y3)/(ex1*y2 - y1 *ex2);
s[1] = ( ex3 - ex1*s[0] )/ ex2;
r[0] =1.0;
r[1] = am[1]-a1-b0*s[0];
r[2] =-b2 * s[1] / a2;
bmprime = (1. + am[1] +am[2])/ (b0 + b1 + b2);
t[0] = bmprime;
}
/*  update the phi vector used by the ld_filter */
/*_____________________________________________*/
change_phi()
{  int i;
for( i=0; i<=na ; i++)
    { phi[ i ] = yvector[i];
   }
for( i=0; i< nb ; i++)
   { phi[na+1+i] = ucon[i];
   }
}
compute_model()
{  hm = h * int_count_max;
am[1] = -2.0 * exp( -zeta * omega *hm)*
cos(sqrt(1.0-zeta*zeta)*omega*hm);
am[2] = exp(-2.0 *zeta *omega*hm);
}
    /*------------------- END -------------------*/
```

4. Fast System control Routine with Filtering

The 'filt pfoh FAST.c' is the C-programme that implements a robust controller with prefilter and predictive first order hold. The programme listing is given below.

```
#include "DSP.h"
#include "int_DAQ.h"
#include "Analog.h"


long *nb_page0 = (long *) 0x805000;
int32 slot_base = 0xfd000000;

/* we just want to use the PID controller in the beginiing */

float past_integral,integral,con_coef_0,con_coef_1;
float Kp=0.7500;
float Ti=0.9;
float Td=0.135;

/* filter sampling time, control interval,runtime specifications    */
/*__________________________________________________________________*/
float h;/* filter sampling interval=(clock=0.24 micro-sec)xint_interval*/
int32 int_interval = 521; /* timer-0 parameter */
int32 int_count_max = 4;    /* this decides the major control interval */
int32   run_time = 180000; /* the total run time */
```

```
int32 control_switch_time =2000;/* when to switch  to adaptive control */
float ester;
float ester_min=0.00095;/* lower limit for estimation error */
float ester_max=2.00;
float dynamic_max;      /*  variable upper limit for ester */

      /*    identifier forgetting factor covariance etc.             */
      /*_____________________________________________________________*/
float lambda = 0.998;
float covare = 20000.0;

    /*  adaptive controller performance specifications:3 delays   */
    /*_____________________________________________________________*/
float omega =200.0;
float zeta =0.7;
float  am[5];

  /*        observer polynomial with one pole at the origin  */
    /*_____________________________________________________________*/
float ao[]={1.0,0.0,0.0}; /* Ao(q) = q**3 + ao[1] q**2 + a[2] q  */
int32 polemod =1;    /* if polemod =1 poles,are fixed for observer*/
float q[] ={ 0.2,0.10};  /* else omega specification is used   */
float zobs=0.80;
float omega_obs=1200.0;


     /* to check any value using analog channel -0            */
  /*_____________________________________________________________*/
float multiplier =0.100;
float probe=0.50;
int32 display_parameter =3;


typedef void (*funcpointer)();    /* define type function pointer */
funcpointer *Timer0vector = (funcpointer*) 0x001000f8;
/*user timer0 interrupt */
int32 timecount;      /* used for timing out the programme */


float fir10[] = {0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1};
extern float firfilt(float j,int32 filterlength,float fir10[],
                        float samplebuffer[]);
int32 filterlength = 10;
float samplebuffer[20]; /* twice the size of filter length */

/*    Needed for Identifier routines : all are global variables */

float phi[6],theta[6],eltrans[6][6],diag[6];
float r[6],s[6],t[6],rm[6],sm[6];
float yvector[6],uref[6],ucon[6],error[6];
float uread,uout,ulast,unext,yraw,yfilt,ic,icmax,hm;
float hm;      /* hm is the control interval */
int32 int_count;/* tracks the number of interrupts occured  */
int na=2;  /* number of a-parameters or a1,a2 parameters*/
int nb=3;      /* number of b-parameters or b0,b1,b2 parameters */
int n; /*  total number of parameters to be identified   */

/* some variables used for DAB solution */

float amod[3],rseg[3],ahi[3][3],bhi[3][3],blo[3][3],alo[3][3]
                                                  ,bhi_in[3][3];
float bhi_inahi[3][3],bba[3][3],abba[3][3],abba_in[3][3];
```

```
    /*------------------------------*/

void c_int01();

main()
{
  Init_MIO16(slot_base);
  Reset_MIO16(slot_base);
  Setup_MIO16(5, 0, 5, 2, 0, 0, 0);

 *Timer0vector = c_int01;  /* address of c_int01 placed in vector  */
  *(int*)Timer0vector |= 0x60000000;/*convert it to a branch instr */
 n = na+nb;
 h = int_interval *0.24e-06;
 init_uy_vectors();

 init_ident_matrices();

 compute_model();


   /* first time operation  */

  Analog_In(InChannel_1,gain_1,&yraw);
yfilt = yraw;
Analog_In(InChannel_0,gain_1,&uread); /* reference value */
uout= Kp *( uread -yfilt);/*first control is proportional */
ulast=uout;
Analog_Out(OutChannel_1,uout); /*control to plant */

/* precalculate the pid- controller coefficiants:   */

con_coef_0 =  (Td + (int_count_max * h))/(int_count_max *h) +1.0
  + ((int_count_max * h )/Ti);
con_coef_1 = -(Td + (int_count_max * h))/(int_count_max *h);
past_integral = 0.0;

  /* setup and start the timer-0 and enable that interrupt */

  Setup_Timer(0,0x000003c3,int_interval,0x00000000);
  /* set up and start timer-0 */
  asm(" OR 0100h,IE");  /* enable Timer-0 interrupt */
  int_count=0;
  timecount =0;
         /*  The beginning of the main loop */

  while (timecount < run_time) /* until timeout */
  {
   while( int_count != 1 ) {} ;
change_phi();   /* prepare consolidated data vector */
ld_filter(n); /* the identifier routine:  */
solve_dab();    /* compute the controller-rst parameters */
probe= r[display_parameter]* multiplier;
Analog_Out(OutChannel_0,probe);
   };        /*  The main loop section */


time_is_up:
  asm(" AND 0feffh,IE ");   /* disable timer-0 interrupt */
  Hold_Timer(0);
  Analog_Out(OutChannel_1,0.0); /* output zero before quiting */
}
```

```
     /* The Timer-0 interrupt service routine */
      /* ------------------------------------ */
void c_int01()
{
if(int_count ==0)
{  ulast=unext;
Update_vectors();
if( timecount < control_switch_time)
{ Compute_pid_unext(); }
else
{ Compute_adaptive_unext(); }
   }

int_count++;
Smooth_control();
Analog_In(InChannel_1,gain_1,&yraw);
yfilt=firfilt(yraw,filterlength,fir10,samplebuffer);
if(int_count ==int_count_max )
{ Analog_In(InChannel_0,gain_1,&uread);
int_count =0;
timecount++;
}
}

     /* The routine to calculate adaptive u(t+j.h) */
     /*_________ R*u =T*u-S*y _____________________*/
Compute_adaptive_unext()
{ int i;
  unext=0.0;

  for(i=0;i<=2;i++)
  { unext=unext+t[i]*uref[i+1];}

  for(i=0;i<=4;i++)
  {unext=unext-s[i]*yvector[i];}

  for(i=1;i<=5;i++)
  {unext=unext-r[i]*ucon[i-1]; }

if(unext > 0.99) unext = 0.99;
if(unext < -.99) unext =-0.99;
}
  /* The routine to calculate pid controller u(t+j.h) */
  /* ------------------------------------------------ */
Compute_pid_unext()
{
 integral = ((error[0]+ error[1] )/(2.0 * Ti)) *(icmax * h);
 unext = Kp *(con_coef_0 * error[0]+ con_coef_1 * error[1]);
 unext = unext + Kp *( integral+ past_integral);
 if(unext < -0.990 ) unext = -0.990;
 if(unext > 0.990 ) unext = 0.990;
 past_integral = past_integral + integral;
 }
  /*       Smooth control                    */
  /* ---------------------------------------  */
Smooth_control()
{
icmax = int_count_max;
ic=int_count;
uout = ulast + ((unext-ulast)*ic)/icmax;
if(uout> 0.98) uout = 0.98;
if(uout< -0.98) uout =-0.98;
Analog_Out(OutChannel_1,uout);
```

```
}
  /*    Initialise uy vectors                */
        /*_____________________________________*/
init_uy_vectors()
{ int i;
for( i=0; i<6; i++)
{ yvector[i] =0.0;
  uref[i] = 0.0;
  ucon[i] = 0.0;
  error[i]= 0.0;
 }
}
   /* initialise matrices needed for identifier */
   /* _____________________________________*/
init_ident_matrices()
{ int i,j;
for(i=0; i<6; i++)
{ phi[i] = 0.0;
  theta[i]= 0.001;
  r[i] = 0.1;
  s[i] = 0.1;
  t[i] = 0.1;
  diag[i] = covare;
  for(j=0; j<6;j++)
    { if(i == j)
      eltrans[i][j] = 1.0;
 else
  eltrans[i][j] = 0.0;
}
 }
  diag[0]= lambda;
 }
 /*        Update uy vectors              */
 /*_____________________________________*/
Update_vectors()
{ int i;
for( i= 5 ; i>0; i--)
    { uref[i] = uref[i-1];
   }
for(i=5; i>0; i--)
 {  yvector[i]= yvector[i-1];
  error[i]= error[i-1];
  }       /* necessary only for the pid */


for(i=5; i>0; i--)
 { ucon[i] = ucon[i-1];
   }
  error[0]= uread - yfilt;
  uref[0] = uread;
  yvector[0]= yfilt;
  ucon[0]= ulast;
/* control output last time is ucon(t) now */
}

       /*  the square root identification routine */
       /*_____________________________________*/
ld_filter( int en)
{  int i,j; /* local variables  */
float e,w;

  /* diag(6),eltrans(6)(6),theta(6),phi(6) are global */
  /* lambda: global variable only used but not affected */
```

```
diag[0]= lambda;
e = phi[0];

for(i=1;i<=en;i++)
{e=e-theta[i]*phi[i];
}
ester=e;
if(ester > 0)ester=ester;
if(ester < 0)ester=-ester;
if(ester< ester_min) return;
dynamic_max=ester_max*(10000.0+timecount)/
                    (10000.+100.*timecount);
if(ester> dynamic_max) return;

for( i=1; i<=en; i++)
 {
   w = phi[i];
   for(j=i+1; j<=en;j++)
    { w = w + phi[j]*eltrans[i][j];
 }
   eltrans[0][i] = 0.0;
   eltrans[i][0] = w;
  }
 for( i =en ; i>=1; i--)

 dyadic_reduction(0,i,en);

/* called with i varying from en down to 1 */
/* call involves and alters 0-th row and i-th row of [eltrans]*/
/* diag(0) and diag(i) are also altered by the call */

for(i=1; i<=en ; i++)

  { theta[i]= theta[i] + eltrans[0][i] * e ;
   diag[i] = diag[i] / lambda;
   }
}

/*   the dyadic reduction routine called by ld_filter   */
/*_____________________________________________________*/
dyadic_reduction(int i0, int i1, int i2)
{
/* eltrans[0-th row] and eltrans[ i1-th row],diag(0),diag(i1)*/

int  j;
float w1,w2,b1,gama;
float mzero = 1.0e-10;

if( diag[i1] < mzero) /* diag[i1] is "beta" */
  { diag[i1] = 0.0 ;
  }

b1 = eltrans[i1][i0]; /* 0-th element of b-vector */
w1 = diag[i0];
w2 = diag[i1] * b1;    /* w2 = beta * b1   */
diag[i0] = diag[i0]+ w2 *b1;
if( diag[i0] > mzero)
 { diag[i1] = diag[i1]*w1 /diag[i0];
   gama = w2 / diag[i0];
   for( j=i1; j<= i2; j++)
   { eltrans[i1][j] = eltrans[i1][j] - b1 * eltrans[i0][j];
   eltrans[i0][j] = eltrans[i0][j] + gama * eltrans[i1][j];
```

```
 }
  }
 }
/* the dab equation is solved by partitioning [9x9] matrix */
/*_____________________________________________________*/
solve_dab()
{ float a1,a2,a3,b0,b1,b2,b3,bmprime,mag,sum,coff[3][3];
int i,j,k;

a1 = -theta[1];
a2 = -theta[2];
b0 =  theta[na+1];
b1 =  theta[na+2];
b2 =  theta[na+3];
bmprime = (1. + am[1] +am[2])/ (b0 + b1 + b2);
t[0] = bmprime;
t[1] = bmprime*ao[1];
t[2] = bmprime*ao[2];

a3 = -a2;
a2 = a2-a1;
a1 = a1-1;/* coefficients of A(q)(q-1) to include integrator */

b3 = b2;
b2 = b2+b1;
b1 = b1+b0;/*coefficients of B(q)(q+1) for robust controller */

   /* easy values   */
rm[0]=1.;
rm[1]=am[1]+ao[1]-a1;
sm[3] = 0.0;


amod[0]=am[2]+ao[1]*am[1]+ao[2]-a2*rm[0]-a1*rm[1];
amod[1]=ao[1]*am[2]+ao[2]*am[1]-a3*rm[0]-a2*rm[1];
amod[2]=ao[2]*am[2]-a3*rm[1];

for(i=0;i<3;i++)
{ for(j=0;j<3;j++)
 {  ahi[i][j]=0;
bhi[i][j]=0;
alo[i][j]=0;
blo[i][j]=0; } /* clear coefficient matrices */
}

for(i=0;i<3;i++)
{ alo[i][i] = 1.0;
  blo[i][i] = b0;
  bhi[i][i] = b3;
  ahi[i][i] = a3; }

alo[1][0] = a1;
alo[2][1] = a1;
alo[2][0] = a2;

blo[1][0] = b1;
blo[2][1] = b1;
blo[2][0] = b2;

bhi[0][1] = b2;
bhi[1][2] = b2;
bhi[0][2] = b1;
```

```
ahi[0][1] = a2;
ahi[1][2] = a2;
ahi[0][2] = a1; /* non-diagonal elements */

/* bhi_in  = inverse of bhi;   by hand computation  */

mag=bhi[0][0]*bhi[1][1]*bhi[2][2];
bhi_in[0][0] = bhi[1][1]*bhi[2][2]/mag;
bhi_in[1][1] = bhi[2][2]*bhi[0][0]/mag;
bhi_in[2][2] = bhi[0][0]*bhi[1][1]/mag;
bhi_in[0][1] = - bhi[0][1]/(bhi[0][0]*bhi[1][1]);
bhi_in[1][2] = - bhi[1][2]/(bhi[1][1]*bhi[2][2]);
bhi_in[0][2] = (bhi[0][1]*bhi[1][2]-bhi[0][2]*bhi[1][1])/mag;
bhi_in[2][0] = 0;
bhi_in[1][0] = 0;
bhi_in[2][1] = 0;


/* bhi_inahi = bhi_in * ahi  */

for(i=0;i<3;i++)
{ for(j=0;j<3;j++)
   { sum= 0.0;
 for(k=0;k<3;k++)
   {sum = sum + bhi_in[i][k]*ahi[k][j];}
 bhi_inahi[i][j] = sum;
}
 }

/*  bba = blo * bhi_inahi    */

for(i=0;i<3;i++)
{ for(j=0;j<3;j++)
{ sum=0.0;
  for(k=0;k<3;k++)
    { sum =sum + blo[i][k]*bhi_inahi[k][j]; }
  bba[i][j] = sum;
 }
 }

/*  abba = alo - bba      */

for(i=0;i<3;i++)
{ for(j=0;j<3;j++)
{ abba[i][j] = alo[i][j] - bba[i][j];  }
 }

/* abba_in = inverse of abba ; by hand compuation */

 coff[0][0] =abba[1][1]* abba[2][2]-abba[2][1]*abba[1][2];
 coff[0][1] =-abba[1][0]* abba[2][2]+abba[2][0]*abba[1][2];
 coff[0][2] =abba[1][0]* abba[2][1]-abba[2][0]*abba[1][1];
 coff[1][0] =-abba[0][1]* abba[2][2]+abba[2][1]*abba[0][2];
 coff[1][1] =abba[0][0]* abba[2][2]-abba[2][0]*abba[0][2];
 coff[1][2] =-abba[0][0]* abba[2][1]+abba[2][0]*abba[0][1];
 coff[2][0] =abba[0][1]* abba[1][2]-abba[1][1]*abba[0][2];
 coff[2][1] =-abba[0][0]* abba[1][2]+abba[1][0]*abba[0][2];
 coff[2][2] =abba[0][0]* abba[1][1]-abba[1][0]*abba[0][1];
 mag =abba[0][0]*coff[0][0]+abba[0][1]*coff[0][1]+
          abba[0][2]*coff[0][2];
for(i=0;i<3;i++)
{ for(j=0;j<3;j++)
abba_in[i][j]=coff[j][i]/mag;
```

```
}

/*  rseg = abba_in * amod   */

for(i=0;i<3;i++)
 { sum=0.0;
    for(k=0;k<3;k++)
    {sum = sum+abba_in[i][k]*amod[k]; }
   rseg[i]=sum;
   rm[i+2]=sum;
  }

/*  s = - bhi_inahi* rseg       */

for(i=0;i<3;i++)
 { sum =0.0;
    for(k=0;k<3;k++)
 { sum = sum - bhi_inahi[i][k]*rseg[k]; }
sm[i] = sum;
 }


/* r-parameters including integrator */

r[5]=-rm[4];
r[4]= rm[4]-rm[3];
r[3]= rm[3]-rm[2];
r[2]= rm[2]-rm[1];
r[1]= rm[1]-rm[0];
r[0]= rm[0];

/* s-parameters including (q+1)      */

s[4]=sm[3];
s[3]=sm[3]+sm[2];
s[2]=sm[2]+sm[1];
s[1]=sm[1]+sm[0];
s[0]=sm[0];
}
     /*   update the phi vector used by the ld_filter */
  /*_____________________________________________*/
change_phi()
{  int i;
for( i=0; i<=na ; i++)
    { phi[ i ] = yvector[i];
   }
for( i=0; i< nb ; i++)
   { phi[na+1+i] = ucon[i+1]; /* we take delay into account */
   }
}
  /* compute the model and observer as per specifications */
  /*_____________________________________________________*/
compute_model()
{   hm = h * int_count_max;
am[0] = 1.0;
am[1] = -2.0 * exp( -zeta * omega *hm)*
       cos((sqrt(1.0-zeta*zeta))*omega*hm);
am[2] = exp(-2.0 *zeta *omega*hm);
    if(polemod > 0)
  {
  ao[0]=1.0;
ao[1]= -(q[0]+q[1]);
ao[2]=(-q[0])*(-q[1]);
```

```
   }
else
  {
ao[0]=1.0;
ao[1] = -2.0 * exp (-zobs * omega_obs* hm)*
    cos(sqrt(1.0-zobs*zobs)*omega_obs*hm);
ao[2] = exp(-2.0 *zobs *omega_obs*hm);
   }
}
  /*------------------- END -------------------*/
```

5. Input and Output Routines

Analog.c and Read One Sample.c are the C-language subroutines used to get data in and out, Analog.h is the support file. The 'float and Scale.c' routine is used for converting data. The programs are listed below.

5.1 Analog I/O:

```
#include "DSP.h"
#include "int_DAQ.h"
#include "Analog.h"

extern int32 slot_base;

void Analog_In(enum InChannel channel, enum Gain gain, float* value)
{
  int32 intvalue;
  Send_NuBus ((gain | channel), slot_base + Mux_Gain);
  Read_One_Sample(slot_base, &intvalue);
  float_and_scale(&intvalue, value); /* convert to floating point */
}

void Analog_Out(enum OutChannel channel,float j)
{
  int32 i,Dac;
  Dac = slot_base + channel;
  j=j*(2048*0x010000); /* scale it back */
  i=j; /* convert back to integer */
  i += (2048*0x10000);  /* add the offset to suit DAC */
  Send_NuBus(i, Dac);
}
```

5.2 Read One Sample.c

```
/*
 * Read_One_Sample(board_firstadd,sample)
 *
 * Read one sample from the ADFIFO of the MIO if no error
 *
 * 12-18-90 DL DSP Group
 * 05-09-91 DL overRunErr, ADClear
 *  National Instruments
 */

#include "DSP.h"
#include "int_DAQ.h"

Read_One_Sample(board_firstadd,sample)
```

```
int32 board_firstadd,*sample;
{
register int32 status, StartConvert;
int32 temp;
StartConvert = board_firstadd + 0x10;
;Send_NuBus(0, StartConvert);
while (((status = Get_NuBus(board_firstadd)) & 0x20000000) == 0);
if ((status & 0x03000000)!= 0) {
     temp = ((status & 0x01000000)!= 0) ? overRunErr : overFlowErr;
     Send_NuBus(0x0,board_firstadd + ADClear);
     return(temp);
}
*sample = Get_NuBus(board_firstadd + AD_FIFO);
return(0);
}
```

5.3 Analog.h:

```
#ifndef _ANALOG_H_
#define _ANALOG_H_

enum InChannel { InChannel_0 = 0x00000, InChannel_1 = 0x10000,
           InChannel_2 = 0x20000, InChannel_3 = 0x30000 };
enum OutChannel { OutChannel_0 = DAC0, OutChannel_1 = DAC1 };
enum Gain { gain_1   = 0x000000, gain_10  = 0x400000,
    gain_100 = 0x800000, gain_500 = 0xc00000 };

void Analog_In(enum InChannel channel, enum Gain gain, float* value);
void Analog_Out(enum OutChannel channel,float j);

#endif
```

5.4 Float and Scale.c:

```
#include "DSP.h"
#include "int_DAQ.h"
/* this routine floats and scales the value read from the ADC */
/* to make any further processing convenient*/
float_and_scale(int32 *i,float *j)
{
*j=*i;
*j=*j/(2048*0x10000);

}
```

---

6. Support Routines for I/O

Programs given files Init MIO16.c, Send Nubus.asm, Get Nubus.asm, and Setup MIO16.c are required to support the I/O operations. The 'timer.c' is the C-programme needed for timer initialization and control.

6.1 Init MIO16.c

```
/*************************************************************
 * Initialize the board according to NB-MIO-16 User Manual p3-35
 * Board must be in factory setting !
 *
```

```
 * 05-06-91 DL
 *  National Instruments
 */

#include "DSP.h"
#include "int_DAQ.h"
int32 cmd2Reg, cmd1Reg;

Init_MIO16(board_firstadd)
int32 board_firstadd;
{
int32  AMCommand,AMData,NBadd,MIOstatusReg;
int32 Status,ADFifo,i;
int32 MIOstatus;

asm(" AND     0FFFDh,IE    ");/*disable interrupt*/
AMCommand = board_firstadd + AMC;
AMData = board_firstadd + AMD;
ADFifo = board_firstadd + AD_FIFO;
MIOstatusReg = board_firstadd + STAT;

cmd1Reg = 0x0;
Send_NuBus(cmd1Reg,board_firstadd + Command1);
cmd2Reg = 0x0;
Send_NuBus(cmd2Reg,board_firstadd + Command2);
Send_NuBus(0x0,board_firstadd + Mux_Gain);

/* 4- initialize RTSI bus switch */
for(i=0;i<56;i++) Send_NuBus(0x0,board_firstadd + MIO_R_SH);
Send_NuBus(0x0,board_firstadd + MIO_R_ST);

/* 5- initialize Am9513A */
Send_NuBus(0xffff0000,AMCommand);
Send_NuBus(0xffef0000,AMCommand);
Send_NuBus(0xff170000,AMCommand);
Send_NuBus(0xf0000000,AMData);

for(i=1;i<=5;i++){
Send_NuBus(0xff000000+i<<16,AMCommand);
Send_NuBus(0x00040000,AMData);
Send_NuBus(0xff080000+i<<16,AMCommand);
Send_NuBus(0x00030000,AMData);
}
Send_NuBus(0xff5f0000,AMCommand);
for(i=0;i<1000;i++);
Send_NuBus(0x0,board_firstadd + ADClear);

/* 6- initialize analog output circuitry */
Send_NuBus(0x0, board_firstadd + DAC0);
Send_NuBus(0x0, board_firstadd + DAC1);

/* 7- initialize digital output register */
Send_NuBus(0x0, board_firstadd + digOut);
}
```

6.2 Send Nubus.asm

```
* Send_NuBus(value,NuBus_Address) sends value to a NuBus
* address expressed
* in the 32-bit format (example:0x0f99000000)
*
* 12/15/90 by DL, DSP Group
```

```
FP .set AR3
.globl _nb_page0
.globl _Send_NuBus

_Send_NuBus:
PUSH FP
LDI SP,FP
PUSH AR0

LDI *-FP(3),R1
ASH -22,R1
AND 03ffh,R1
LDI @_nb_page0,AR0
STI R1,*AR0
LDI *-FP(3),R1
ASH -2,R1
AND @CONST+0,R1
OR @CONST+1,R1
LDI R1,AR0
LDI *-FP(2),R0
STI R0,*AR0

POP AR0
POP FP
RETS
*******************************************************
* DEFINE CONSTANTS                                    *
*******************************************************
.bss CONST,2
.sect ".cinit"
.word 2,CONST
.word  0fffffh          ;0
.word  0c00000h         ;1
.end


;#include "DSP.h"
;extern int32 *nb_page0;
;Send_NuBus(value,nb_addr)
;int32 nb_addr,value;
;{
;register int32 *r0;
;*nb_page0 = (nb_addr >> 22) & 0x000003ff;
;r0 = (int32*) (0xc00000 | ((nb_addr>>2) & 0x000fffff));
;*r0 =value;
;return(0);
;}
```

6.3 Get Nubus.asm

```
* [NuBus_Address] = Get_NuBus(NuBus_Address) returns
* the content of a NuBus address
*
* 11/7/90 by DL, DSP Group
*

FP .set AR3
.globl _nb_page0
.globl _Get_NuBus

_Get_NuBus:
PUSH FP
LDI SP,FP
```

```
PUSH AR0

LDI *-FP(2),R0
ASH -22,R0
AND 03ffh,R0
LDI @_nb_page0,AR0
STI R0,*AR0
LDI *-FP(2),R0
ASH -2,R0
AND @CONST+0,R0
OR @CONST+1,R0
LDI R0,AR0
LDI *AR0,R0

POP AR0
POP FP
RETS
*****************************************************
* DEFINE CONSTANTS                                  *
*****************************************************
.bss CONST,2
.sect ".cinit"
.word 2,CONST
.word  0fffffh          ;0
.word  0c00000h         ;1
.end



;equivalent C code:
;#include "DSP.h"
;extern int32 *nb_page0;
;Get_NuBus(nb_addr)
;int32 nb_addr;
;{
;register int32 *r0;
;*nb_page0 = (nb_addr >> 22) & 0x000003ff;
;r0 = (int32*) (0xc00000 | ((nb_addr>>2) & 0x000fffff));
;return(*r0);
;}
```

6.4 Setup MIO16.c

```
/*
 * Setup_MIO16(board_slot,gain,timebase,interval,channel,samples,RTSInt)
 *
 * Set up the MIO for data acquisition. Init_MIO16 and Reset_MIO16 must
 * be run previously (Init defines globals)
 * Continuous sampling: the MIO 16 is set up for continuous
 * sampling if samples <=0 or samples >65535
 * (see NB-MIO-16 User Manual p 3-40):
 * Gain: 0 to 3 in the bit 6 and 7 of Gain_Mux
 * Timebase: 5 (100Hz) to 1 (1MHz) in Counter 3
 *
 * Interrupt service (NB-MIO_16X p 3-79):
 * set CONVINTEN in Command1 if RTSInt != 0
 * set NBINTDIS  in Command2 : MIO16 doesn't assert interrupt on NuBus
 *
 * 12-02-90 DL DSP Group
 * 12-17-90 DL Setup_AD_MIO, channel selection
 * 12-18-90 DL continuous sampling selection
 * (LabDriver:Data_Acq.c:SetUpSampCtr)
 * 12-23-90 DL take samples parameter out
```

```
 * 12-25-90 DL board_slot instead of board_firstadd
 * 05-10-91 DL Send_NuBus, RTSInt, cmd1Reg,cmd2Reg.
 *  National Instruments
 */

#include "DSP.h"
#include "int_DAQ.h"
extern int32 cmd1Reg,cmd2Reg;
int32 Alloc_Mem();

Setup_MIO16(board_slot,gain,timebase,interval,channel,samples,RTSInt)
 int32 board_slot,gain,timebase,interval,channel,samples,RTSInt;
 {
 int32  err,n,m,AMCommand,AMData,NBadd,timeout;
 int32 i,delay,board_firstadd,temp;

board_firstadd = (board_slot * 0x01100000) + 0xF8800000;
if(gain >3 || gain < 0) return(outOfRangeErr);
if(channel>15 || channel<0) return(outOfRangeErr);
if (interval<2 || interval>65535) return(outOfRangeErr);
if (samples == 1) return(outOfRangeErr);

/**** 1- Select channel and gain ****/
Send_NuBus (0x0,board_firstadd + Mux_Count);
Send_NuBus ((gain<<22) | (channel<<16),board_firstadd + Mux_Gain);

/**** 2- Program Sample-Interval Counter ****/
AMCommand = board_firstadd + AMC;
AMData = board_firstadd + AMD;

Send_NuBus (0xFF030000,AMCommand);  /* select Counter 3 Mode Register */

switch(timebase){
case 1 : temp = 0x8b250000;break;   /*   1 MHz */
case 2 : temp = 0x8c250000;break;   /* 100 kHz */
case 3 : temp = 0x8d250000;break;   /*  10 kHz */
case 4 : temp = 0x8e250000;break;   /*   1 kHz */
case 5 : temp = 0x8f250000;break;   /* 100 Hz  */
default: return(outOfRangeErr);break;
}
Send_NuBus (temp,AMData);      /* select timebase */

Send_NuBus (0xFF0B0000,AMCommand);  /* select Counter 3 Load Register */
Send_NuBus (0x00020000,AMData);     /* Counter 3 Load value */
Send_NuBus (0xFF440000,AMCommand);  /* load Counter 3 */
Send_NuBus (0xFFF30000,AMCommand);  /* step Counter 3 to 1 */
Send_NuBus (interval<<16,AMData);   /* select sample interval */
Send_NuBus (0xFF240000,AMCommand);   /* arm Counter 3 */

/**** 3-  Program the Sample Counter if non continuous ****/
if (samples >0 && samples < 65536){
Send_NuBus (0xFF040000,AMCommand);/* select Counter 4 Mode Register */
Send_NuBus (0x10010000,AMData);   /* store Counter 4 Mode value */
Send_NuBus (0xFF0C0000,AMCommand);/* select Counter 4 Load Register */
Send_NuBus (samples<<16,AMData);
Send_NuBus (0xFF680000,AMCommand);/* arm Counter 4 */
}
else{ /* set output CTR4 to low */
Send_NuBus(0xff040000,AMCommand);
Send_NuBus(0x0,AMData);
Send_NuBus(0xff0c0000,AMCommand);
Send_NuBus(0x00030000,AMData);
}
```

```
/*
 * 4- Clear tha A/D circuitry and enable continuous data acquisition.
 *    MIO-16 User Manual p3-43.
 */
Send_NuBus (0x0,board_firstadd + ADClear);     /* clear ADFIFO  */

cmd1Reg = (RTSInt) ? 0x00500000 : 0x00100000 ;
Send_NuBus (cmd1Reg,board_firstadd + Command1);

/* Command1:
   x x x x x x x DAQ INT  CONV INT  DMA  DAQ  SCAN SCAN 16*-32  2SC*
                   EN        EN     EN   EN    EN  DIV CNT   ADC*
   0 0 0 0 0 0 0   0         1      0    1     0   0      0     0  */

cmd2Reg =  0x00800000;
Send_NuBus (cmd2Reg,board_firstadd + Command2);
/* Command2:
 x x x x x DOUTB DOUTA  NBINT   DMA  DMA  DMA   A4   A4    A2    A2
 EN  EN     DIS     A2   A1   A0    RCV  DRV RCV   DRV
 0 0 0 0 0   0  0     1       0     0    0     0    0      0     0    */

return(0);
}
```

6.5 Timer.c

```
/************************************************************************
* Setup_Timer   - call to set up timer's control, period, and counter
*                 registers.  Use either Setup_Timer or Start_Timer
*                 but not both.
*
* Start_Timer   - resets and starts timer
* Hold_Timer    - holds timer in present state
* Restart_Timer - restarts timer from previous count
* Timer_Count   - returns 32-bit timer count
*
*   Copyright  1990 National Instruments Corporation.
*   All rights reserved.
*
************************************************************************/


unsigned long   *timer_0_control = (unsigned long *) 0x808020,
                *timer_0_count   = (unsigned long *) 0x808024,
                *timer_0_period  = (unsigned long *) 0x808028,
*timer_1_control = (unsigned long *) 0x808030,
                *timer_1_count   = (unsigned long *) 0x808034,
                *timer_1_period  = (unsigned long *) 0x808038;

Setup_Timer(timer,control,period,counter)
long  control,period,counter;
{
  if(timer)   {
*timer_1_period  = (unsigned) period;
*timer_1_control = (unsigned) control;
*timer_1_count   = (unsigned) counter;
}
  else {
*timer_0_period  = (unsigned) period;
*timer_0_control = (unsigned) control;
*timer_0_count   = (unsigned) counter;
}
```

```
}


Start_Timer(timer)
long  timer;
{
  if(timer)  {
*timer_1_period = (unsigned) 0xffffffff;
*timer_1_control = 0x03C3;   /* zero and start counter */
}
  else   {
*timer_0_period = (unsigned) 0xffffffff;
*timer_0_control = 0x03C3;
}
}

Restart_Timer(timer)
long  timer;
{
  if(timer)  *timer_1_control = 0x0383;   /* restart counter */
  else           *timer_0_control = 0x0383;
}

Hold_Timer(timer)
long  timer;
{
if(timer)  *timer_1_control = 0x0303;   /* hold counter */
else *timer_0_control = 0x0303;
}

unsigned long Timer_Count(timer)
long  timer;
{
if(timer)   return(*timer_1_count);
else             return(*timer_0_count);
}
```