

Abstract

The work develops a microcontroller solution for controlling the pressure of pneumatic organs irrespective of the demand of volume flow rate. Prior investigations suggested that PID algorithm be used as a controller. The target hardware was also proposed to realize the algorithm. The present work makes use of those ideas and develops the code for the controller. It describes the programming tools to be used, introduces its development environments, determines hardware resources to be used, sketches the interfacing of the peripherals with the hardware and finally discusses the scopes of certain possibilities for changes that can be made. It has developed windows based embedded c code and has given illustrations of the important lines of the code. Several electrical peripheral circuits have also been built up within the scope.

Table of Contents

| | |
|--|-----------|
| Abstract..... | i |
| Table of Contents..... | ii |
| List of Symbols..... | iv |
| List of Abbreviations..... | iv |
| Acknowledgements..... | vi |
| Certification..... | vii |
| | |
| 1.Introduction..... | 1 |
| 1.1 Background..... | 1 |
| 1.2 Approach and Objectives..... | 1 |
| | |
| 2.Introductory Feedback Control..... | 3 |
| 2.1 Basic Principle of Feedback Control..... | 3 |
| 2.2 The Feedback Control System..... | 3 |
| 2.3 PID algorithm..... | 4 |
| 2.4 Digital PID algorithm..... | 6 |
| | |
| 3.Introduction to Microcontroller..... | 9 |
| 3.1 Microcontroller..... | 9 |
| 3.2 What is C167?..... | 9 |
| 3.3 Using the C167..... | 10 |
| 3.4 Programming the C167..... | 11 |
| 3.5 Embedded Systems Programming Concepts..... | 11 |
| | |
| 4.Understanding the Keil IDE..... | 12 |
| 4.1 The Keil IDE..... | 12 |
| 4.2 Creating New Source Files..... | 13 |
| 4.2 Adding and Configuring the Startup Code..... | 13 |
| 4.3 Setting Tool Options for Target..... | 14 |
| 4.4 Building Project and Creating a HEX File..... | 14 |
| 4.5 Downloading a Program to the minimodule-167..... | 15 |
| | |
| 5.miniMODUL-167: A Synopsis..... | 19 |
| 5.1 What is miniMODUL-167?..... | 19 |
| 5.2 Parallel Ports..... | 19 |
| 5.3 The Analog/Digital Converter..... | 22 |
| | |
| 6.Timers..... | 25 |
| 6.1 The General Purpose Timer Units..... | 25 |

| | | |
|---|--|-----------|
| 6.2 | Timer Block GPT1 | 25 |
| 6.3 | GPT1 Core Timer T3..... | 26 |
| 6.4 | Timer 3 Run Bit..... | 27 |
| 6.5 | Timer 3 in Timer Mode | 27 |
| 7.Interrupts | | 28 |
| 7.1 | Interrupts..... | 28 |
| 7.2 | Interrupt Control for GPT1 Timers..... | 28 |
| 8.The PID Code | | 29 |
| 8.1 | The PID Code | 29 |
| 8.2 | Code Illustration | 34 |
| 9.Interfacing with the Microcontroller | | 37 |
| 9.1 | The Sensor | 37 |
| 9.2 | D/A Converter | 38 |
| 9.3 | Connection Diagrams | 38 |
| 9.4 | Calibration | 39 |
| 10.Remarks on Results | | 41 |
| 11.Summary and Conclusions | | 42 |
| 11.1 | Introduction | 42 |
| 11.2 | Future Work..... | 42 |
| List of References | | 44 |

List of Symbols

| Symbol | Unit | Meaning |
|--------|------|--------------------------------------|
| e | - | Error |
| f | Hz | Frequency |
| i | - | Integer |
| e_i | - | Error at iT -th sample |
| K_p | - | Proportional constant |
| K_i | - | Integral constant |
| K_d | - | Derivative constant |
| P | Pa | Pressure |
| T | Sec | Sampling time |
| t | Sec | Time |
| u_i | - | Controller output at iT -th sample |

| Indices | Meaning |
|---------|-------------------|
| B | Binary |
| d | Derivative |
| D | Decimal |
| H | Hexadecimal |
| i | Integral, Integer |
| p | Proportional |

List of Abbreviations

| Abbreviation | Acronym |
|--------------|------------------------------|
| ADCON | ADC Control Register |
| ADCIR | ADC Interrupt Request Flag |
| A/D | Analog to Digital |
| ADBSY | ADC Busy Flag |
| ADCIN | ADC Channel Injection Enable |
| ADCSTC | ADC Sample Time Control |
| ADCTC | ADC Conversion Time Control |
| ADDAT | ADC Result Register |
| ADM | ADC Mode Selection |

| | |
|---------|--|
| ANSI | American National Standard Institute |
| ADST | ADC Start Bit |
| ADWR | ADC Wait for Read Control |
| CPU | Central Processing Unit |
| D/A | Digital to Analog |
| DPx | Port x Direction Control Register |
| GPTx | General Purpose Timer x |
| IDE | Integrated Development Environment |
| IO | Input Output |
| LSB | Least Significant Bit |
| MSB | Most Significant Bit |
| ODPx | Port x Open Drain Control Register |
| PID | Proportional, Integral, Derivative |
| Px | Port x Data Register |
| PxDIDIS | Port x Digital Input Disable Register |
| S0BG | ASC0 Baud Rate Generator |
| S0CON | ASC0 Control Register |
| S0RIC | ASC0 Receive Interrupt Control Register |
| S0TIC | ASC0 Transmit Interrupt Control Register |
| TxCON | GPT1 Timer x Control Register |
| TxI | GPT1 Timer x Input Selection |
| TxIC | GPT1 Timer x Interrupt Control Register |
| TxINT | Timer x Interrupt Vector |
| TxIR | Timer x Interrupt Request Flag |
| TxOE | Timer x Alternate Output Function Enable |
| TxOTL | Timer x Output Toggle Latch |
| TxM | GPT1 Timer x Mode Control |
| TxR | Timer x Run Bit |
| TxUD | Timer x Up/Down Control |
| TxUDE | Timer x External Up/Down Enable |

Acknowledgements

The author wishes to express thanks to his project advisor, Prof. Dr.-Ing Thomas Carolus, for his insight and essential aid throughout carrying out the whole project and writing of this report.

The author's wholehearted appreciation and gratitude is extended to Mr. Bernd Homrighausen for his continuous assistance with the project. The author extends special thanks and regards to Mr. Erhard Kluge for his invaluable assistance for building up electrical circuits.

Certification

This is to certify that the work presented in this thesis is the outcome of the investigation carried out by the candidate under the supervision of Prof. Dr. –Ing. Thomas Carolus, IFT, University of Siegen.

Signature of the candidate

◆ 1 ◆

Introduction

1.1 Background

Pneumatic organs require pressurized air supplied at constant pressure to the pipes -independent of the demand of volume flow rate. Traditionally this is ensured by a plenum and a mechanical valve with a mechanical feedback loop. In this project a microcontroller solution has been investigated to ensure constant pressure which is sensed by an electronic sensor. A by-pass valve is driven by an actuator in order for positioning it in accordance with the requirements. The actuator of the by-pass valve is an electric linear drive (Lin-Mot[®]).

1.2 Approach and Objectives

The investigation presented in this thesis is an extension of the work of a previous student who introduced the microcontroller concept for the regulation of the pressure of the pneumatic organ. On the basis of his investigation the microcontroller and the programming compiler for our project are selected.

In the previous investigation, the three term PID controller is proposed for the possible solution of the control algorithm. We have examined the concept and used the algorithm to control the actuator and hence, the pressure of the system. The algorithm is implemented in windows-based Keil software using ANSI C language. The target hardware is a Phytex[®] board which uses C167 Infineon microcontroller.

A description of the introductory feed-back control is presented in chapter 2 which summarizes the concept used in the controlling algorithm.

Chapter 3 gives a fundamental idea about microcontroller. It also describes some aspects of C167 and its programming tools.

The Keil IDE, programming tool of c167, is briefly described in chapter 4. As using this tool is an integral part of the programming, understanding the IDE is required.

Several C167 resources relevant to our program have been discussed in chapter 5. It gives an application specific description for clear understanding of the program.

Chapter 6 describes timers in a nutshell. General Purpose Timer GPT1 is used to associate triggering the interrupt, which is a topic of chapter 7.

The text version of the code is presented in chapter 8. Illustration of important lines is also discussed.

Chapter 9 emphasizes interfacing issues. The sensor, microcontroller, D/A converter are the topics of this chapter.

Conclusions are summarized in chapter 10. Recommendations for future work are given in the final chapter 11.

◆ 2 ◆

Introductory Feedback Control

2.1 Basic Principle of Feedback Control

In the feedback system, the *controlled variable*, i.e. the quantity to be acted upon, is measured and compared with a given reference input. The reference point is normally termed as *set point*. If a difference between these two quantities arises, then the controller acts on the controlled system in such a way that the difference is reduced or made zero.

The *set point* is presented by the voltage V_s , equivalent to the required pressure; the sensor read value is V_f . Their difference, the *error* or the actuating signal e , acts on the input of the controller comprising the microcontroller and D/A converter, so influencing the voltage of the actuator, that the difference is reduced or made zero. [reference: 1]

2.2 The Feedback Control System

It is seen from the Fig. 2.1 that the feedback control system is characterized by a closed sequence of effects. Any feedback circuit comprises a controller and a controlled system.

Controlled System

This is the part whose construction is determined by the actual purpose of the installation and in which the influence on the *controlled variable* takes place. In our case it is the plenum or broadly speaking, the pneumatic system itself.

Controller

This is the part introduced into the installation to influence the *controlled variable*. The microcontroller, the PID algorithm and D/A converter comprise the controller in our case. [reference: 1]

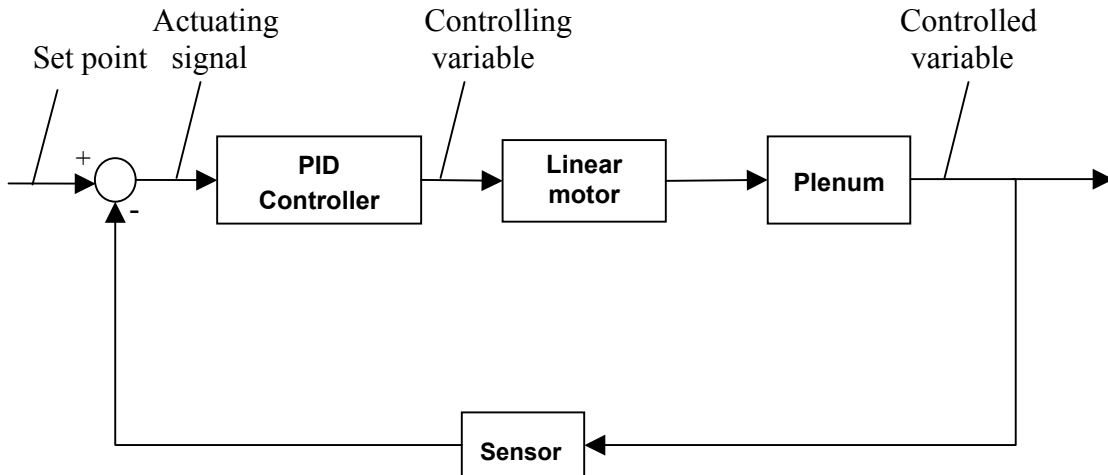


Figure 2.1 Block diagram of the feed back control system

2.3 PID algorithm

PID stands for Proportional-Integral-Derivative. This is a type of feedback controller whose output, a *controlling variable*, is generally based on the *error* between some user-defined *set point* and some measured *controlled variable* (*process variable*).

So, when everything is up and running, our PID controller compares the *process variable* to its *set point* and then calculates the difference between the two signals, the *error* (e).

Then, based on the *error*, the controller calculates an output that positions the actuator. If the actual pressure is above its set point, the controller will reduce the actuator position and vice versa.

P, I and D Control Modes

As we said above, a PID controller has proportional, integral and derivative control modes. These modes each react differently to the *error*, and also, the degree of control action is adjustable for each mode.

Proportional Control

The proportional control mode changes the controller output in proportion to the *error*. The adjustable setting here is called the *Controller Gain* (K_p), sometimes also referred to as a PID controller's P-setting or its proportional setting.

The control action is proportional to both the *controller gain* and the *error*. A higher controller gain will increase the amount of output action and so will a larger *error*.

Mathematically,

$$P_{Action} = K_p \cdot e, \quad (2.1)$$

where K_p is known as *proportional constant*.

The use of proportional control alone has a large drawback – *offset* or *steady state error*. *Offset* is a sustained error that cannot be eliminated by proportional control alone. For example, let's consider our system with a proportional-only controller.

As long as the speed of the motor remains constant, the pressure (which is our *process variable* in this case) will remain at its set point.

But, if the operator should decrease the speed of the motor, the pressure will begin to decrease. While the pressure decreases the *error* increases and our proportional controller increases the controller output proportional to this *error*. Consequently, the actuator controlling the pressure in the plenum moves further and hence a higher pressure develops inside the plenum.

As the pressure continues to decrease, the actuator continues to move forward until it reaches the steady state. At this point the pressure remains constant, and so does the error. Then, because the error remains constant, our P-controller will keep its output constant and the actuator will hold its position. The system now remains at balance with the pressure remaining below its set point. This residual error is called *offset*.

With our P-controller the *offset* will remain until the operator manually applies a bias to the controller's output to remove the *offset*. It is said that the operator has to manually "Reset" the controller. Or we can add integral action to our controller.

Integral Control

The integral control mode of a controller produces a long term corrective change in controller output, driving the *error* or *offset* to zero. Integral control eliminates *offset*. A PI controller simply adds together the output of the P and I modes of the controller. The integral action raises the controller output far enough to bring the level back to its *set point*. Fig. 2.2 shows the graphical representation of the integral action.

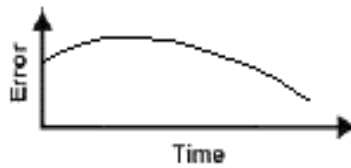


Figure 2.2 Graphical representation of the integral action²

Mathematically,

$$I_{Action} = K_i \int e \cdot dt \quad (2.2)$$

The integral of the error is multiplied by a gain, K_i . In many systems K_i is responsible for driving error to zero. This K_i is usually known as *Integral Constant*.

Derivative Control

The third control action in a PID controller is derivative. Derivative control is rarely used in controllers. However, derivative control can make a control loop respond faster and with less *overshoot*. The derivative control mode produces an output based on the rate of change of the *error*. Derivative action is sometimes called *Rate*. Its action is dependent on the rate of change (or slope) of the *error*. It has an adjustable setting called *Derivative Constant* (K_d).

$$D_{Action} = K_d \cdot \frac{de}{dt} \quad (2.3)$$

Therefore, the output of the PID controller becomes:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t) \cdot dt + K_d \cdot \frac{de(t)}{dt} \quad (2.4)$$

[reference: 2]

2.4 Digital PID algorithm

In digital control we use discrete signal. Let us consider Fig. 2.3.

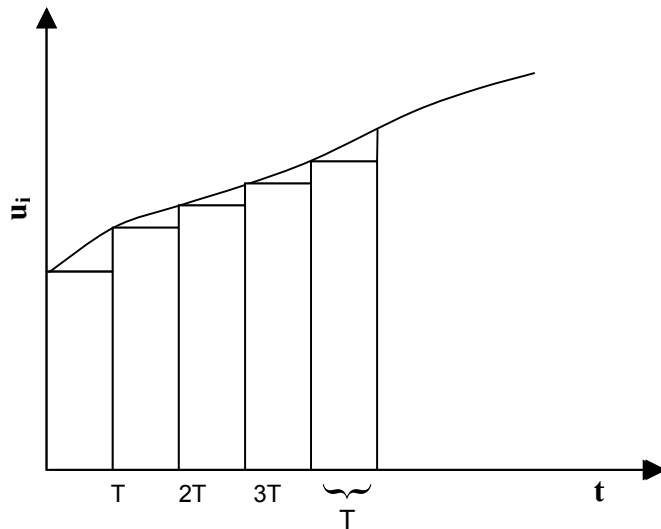


Figure 2.3 Digital outputs at sampled times³

The basic difference between an analog and digital controller is that the digital system operates on discrete signals (or samples of the sensed signal) rather than on continuous signals.

We assume that *sampling time*, T is constant. As we will proceed, this fact will be established. The *sampling time* must be considerably smaller than the data processing time.

Quasi-continuous Control

In quasi-continuous control, the control algorithms are realized on a digital controller, like the one in our case. However, the design procedures as well as the algorithms are the same as continuous control.

Now, we will deduce the quasi-continuous algorithm for the PID controller.

Following the equation 2.4 we know that the output is:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(t^*) dt^* + K_d \cdot \frac{de(t)}{dt} \tag{2.5}$$

Say,

$$t = i \cdot T \tag{2.6}$$

T is sample time and will be considered constant.

Then,

$$e(t) = e(i \cdot T) = e_i \quad (2.7)$$

Similarly,

$$u(t) = u(i \cdot T) = u_i \quad (2.8)$$

As we know, digital data are not continuous; then,

$$u_i = K_p e_i + K_i T \sum_{j=0}^{i-1} e(jT) + \frac{K_d}{T} [e(iT) - e[(i-1)T]] \quad (2.9)^3$$

For simplicity we would use the following form of the equation:

$$u_i = K_p e_i + K_i T \sum_{j=0}^i e(jT) + \frac{K_d}{T} [e(iT) - e[(i-1)T]] \quad (2.10)$$

[reference: 3]

◆ 3 ◆

Introduction to Microcontroller

3.1 Microcontroller

To realize the PID algorithm, in this project we have used the Phytex[®] board as the target hardware. We know that the brain of the board is a controller, in other words, a microprocessor. It is not a Gigahertz processor; a 20 MHz processor is sufficient for our purpose. In our case, it is the Infineon C167CR-LM (henceforth referred to as the C167).

3.2 What is C167?

Fig. 2.1 shows the location of the microcontroller on the minimodule -167 board from Phytex[®].

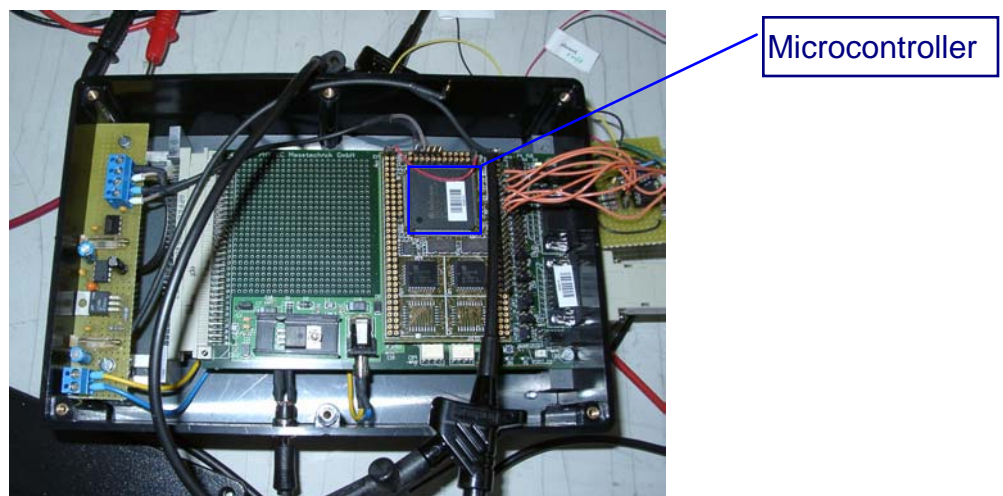


Figure 2.2 Microcontroller

The C167 is a microcontroller; it is used to control some process or aspect of the environment. Just like the more popular microprocessor (the Intel Pentium or the AMD Athlon), a microcontroller has a lot of stuff built into it – on-chip memory, an Arithmetic Logic Unit (ALU) etc. However, unlike the Pentium a microcontroller is not optimized for building PCs or laptops. For instance, the maximum clock speed of our C167 is 20 MHz. The latest Pentiums on the other hand have clock speeds in the Gigahertz range. But this speed is required only for computationally intensive software. The software controlling our system is not that complex– 20 MHz is a very fast clock speed for a microcontroller.

Clock speed is not the only area where microcontrollers differ from microprocessors. The C167 has other components suited for control that are built on chip, which the Pentium does not have. Examples include Analog-to-Digital converter (A/D module), the ASC0 (Asynchronous serial transmitter) and the PWM (Pulse Width Modulation unit). However, we will cover only those features that will be needed in our application. [reference: 4]

3.3 Using the C167

Using the C167 (or any microcontroller) is mainly a four-step process. First, we program the module we want (for instance, The General Purpose Timer Units). Then we download this code to the on-board RAM. We test this code. If there is an error, we start the debugging process. Once we are satisfied, the C167 is doing what we want; we have the option of burning our code to FLASH so our code is retained even when we turn off the power. [reference: 4]

Fig. 3.2 summarizes the build process.

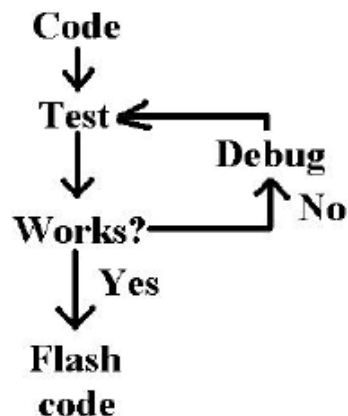


Figure 3.2 The design process⁴

⁴–reference: 4

3.4 Programming the C167

C167 Programming Tools

The Keil IDE (Integrated Development Environment) is used to program the C167 in the C programming language. It writes to the board RAM an executable version of our code. This tool is used to design and test our program. Once we are satisfied it is working correctly, we can “permanently burn” the code onto FLASH using Phytex FLASH tools. But, FLASH is reprogrammable so we can erase FLASH memory using the software and download new code; the number of times this can be done is around 100,000. [reference: 5]


3.5 Embedded Systems Programming Concepts

It is conceivable that we are writing a code for a board that has no monitor, no keyboard or hard disk. So, it may feel weird at first. Nevertheless, such code is usually very clean and simple, as we will see. This kind of programming is called *embedded systems programming*. We are writing a code that is going to run on an independent system. The system may encompass a wide spectrum of examples, for instance our microcontroller may control the flight control systems of a Boeing 747. Hence, unlike writing a code on a computer, one has to make sure that his program is not buggy. [reference: 5]

◆ 4 ◆

Understanding the Keil IDE

4.1 The Keil IDE

First, we start up Keil then do we double click on the  icon on desktop. To create a new **project file** we do the following:

1. We select from the μ Vision2 menu **Project – New Project....** This opens a standard Windows dialog that asks for the new project file name.
2. We use the icon **Create New Folder** in this dialog to get a new empty folder. Then we select this folder and enter the file name for the new project, i.e. **Project1**. μ Vision2 creates a new project file with the name **PROJECT1.UV2** that contains a default target and file group name. One can see these names in the **Project Window – Files**.
3. Now we use from the menu **Project – Select Device for Target** and select a CPU for our project. The **Select Device** dialog box shows the μ Vision2 device database. We just select the microcontroller we use. We are using for our example the C167CR-LM CPU. This selection sets necessary tool options for the C167CR-LM and simplifies in this way the tool configuration. Fig. 4.2 shows the CPU selection window. [reference: 6]

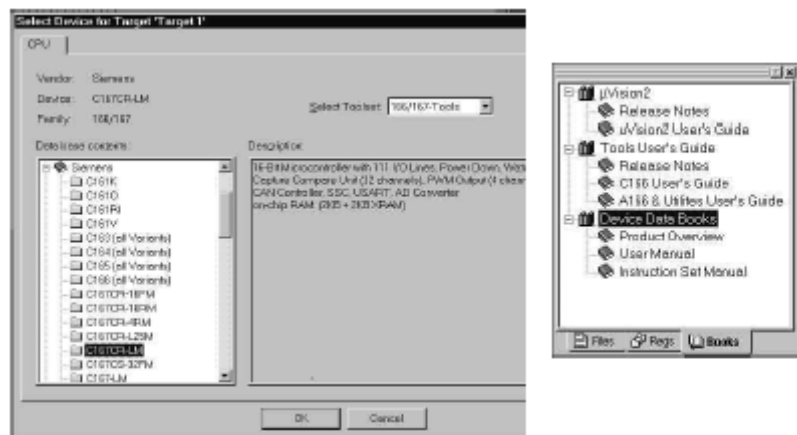


Figure 4.1 CPU selection screen

4.2 Creating New Source Files

We may create a new source file with the menu option **File – New**. This opens an empty editor window where we can enter our source code. μ Vision2 enables the C color syntax highlighting when we save our file with the dialog **File – Save As...** under a filename with the extension ***.C**.

Once we have created our source file we can add this file to our project. μ Vision2 offers several ways to add source files to a project. For example, we can select the file group in the **Project Window – Files** page and click with the right mouse key to open a local menu. The option **Add Files** opens the standard files dialog. Then we select the file the **.C** (e.g. main.c) we have just created. Fig. 4.2 shows the **Project Window** and the menu for adding the files. [reference: 6]



Figure 4.2 Project Window with Add Files menu

4.2 Adding and Configuring the Startup Code

Typically, a 166/ST10 program requires a CPU initialization code that needs to match the configuration of our hardware design. For most 166 / ST10 derivatives, we can use the **START167.A66** file as startup code. Since we need to modify that file to match our target hardware, we need to copy the **START167.A66** file from the folder **C:\KEIL\C166\LIB** to our project folder.

It is a good practice to create a new file group for the CPU configuration files. With **Project – Targets, Groups, Files...** we can open a dialog box where we add a group named **System Files** to our target. In the same dialog box we can use the **Add Files to Group...** button to add the **START167.A66** file to our project. Fig. 4.3 shows dialog box to create new groups and to add files to the new groups. [reference: 6]



The **Project Window – Files** lists all items of our project.



Figure 4.3 Screen for creating new groups

4.3 Setting Tool Options for Target

μ Vision2 lets us set options for our target hardware. The dialog **Options for Target** opens via the toolbar icon. In the **Target** tab we specify all relevant parameters of our target hardware and the on-chip components of the device we have selected. Fig. 4.4 the settings for our example are shown. [reference: 6]

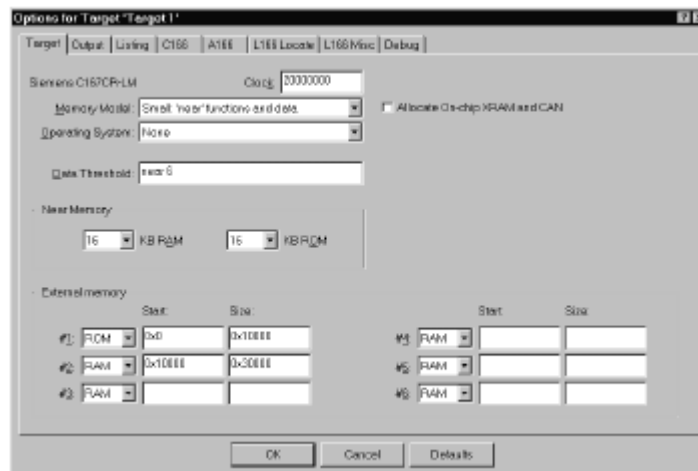


Figure 4.4 Window for setting options for target

4.4 Building Project and Creating a HEX File

Typically, the tool settings under **Options – Target** are all we need to start a new application. We may translate all source files and link the application with a click on the **Build Target** toolbar icon. When we build an application with syntax errors, μ Vision2 will display errors and warning messages in the **Output**

Window – Build page. A double click on a message line opens the source file on the correct location in a μ Vision2 editor window.

After we have tested our application, it is required to create an Intel HEX file to download the software into an EPROM programmer or simulator. μ Vision2 creates HEX files with each build process when **Create HEX file** under **Options for Target – Output** is enabled. Fig. 4.5 shows the window for creating HEX files. [reference: 6]

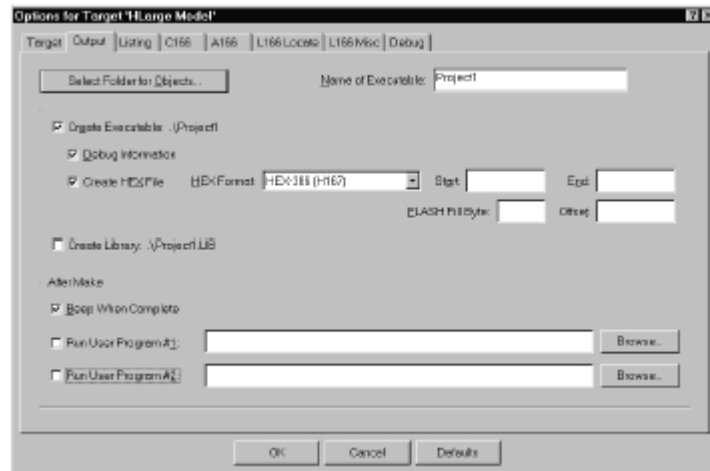


Figure 4.5 Window for creating HEX files

4.5 Downloading a Program to the minimodule-167

First, we have to *compile* the program: send it through Keil's C compiler that translates our program into object code that is then translated by a linker into executable code. It is very easy to do this; we just click on **Project** as shown in. Then we click on **Rebuild all Target files**. It should not produce any warnings or error messages. In order to actually download the program, we make sure our serial cable is connected to the RS-232 (serial interface) port of the minimodule.

Downloading Code with Flash Tools for Windows

- We Start Flash Tools for Windows by double-clicking on the Flash Tools for Windows icon or by selecting *Flash Tools for Windows* from within the *Programs/PHYTEC Flash Tools for Windows* program group.
- The Connect tab of the Flash Tools for Windows Worksheet window will now appear.
- We double click on *16-Bit RS232*

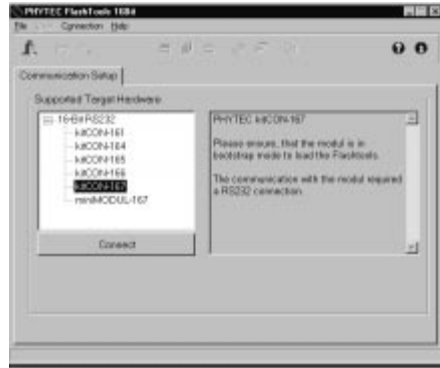


Figure 4.6A Window for the target hardware selection

- We select miniMODUL-167 from the list and press the *Connect* button.



Figure 4.6B Interface and baud rate selection window

- We select the correct serial port for our host-PC and a *9600 baud rate*.
- We click the *OK* button to load the module based part of the Flash Tools to the target hardware. Fig. 4.6A, 4.6B, 4.6C show windows for the above processes sequentially.

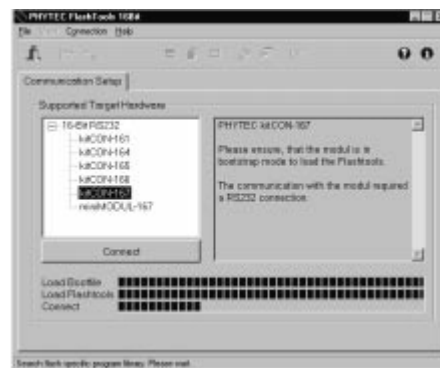


Figure 4.6C Mode selection and data transfer window

- After the data transfer we will see Flash Tools for Windows Worksheet window with the following tabs:

Fig. 4.7 shows the window for **Flash Information** which shows sector and address ranges in Flash-Memory:

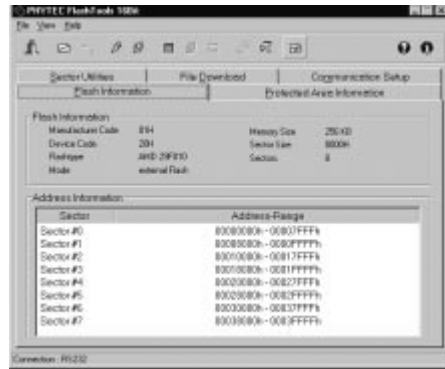


Fig 4.7 Window for **File Information** tab

Fig. 4.8 shows the window for **File Download** which downloads specified hex files to the target hardware:

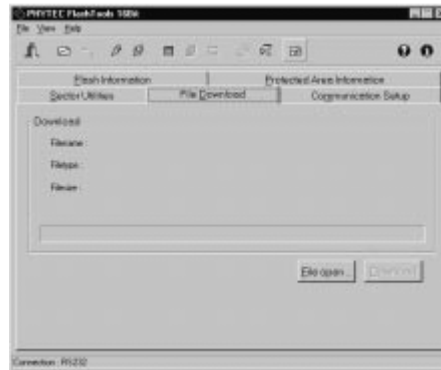


Fig 4.8 Window for **File Download** tab

Fig. 4.9 is the **Protected Areas Information** window showing protected areas of Flash-Memory:



Fig 4.9 Window for **Protected Area Information** tab

Sector Utilities allow erasure of individual sectors of Flash-Memory. Fig. 4.10 is the window for this tab:

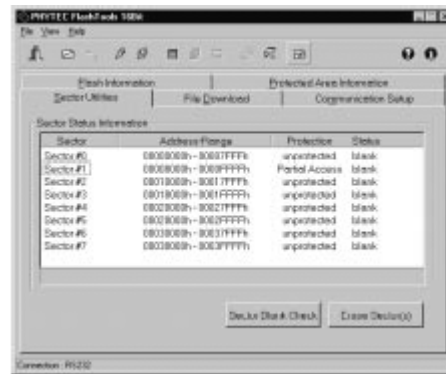


Fig 4.10 Window for **Sector Utilities** tab

Communication Setup provides us with the possibility to disconnect the target and reconnect with an equal one. Programming of several targets is possible for us thereby. Fig. 4.11 shows this window: [reference: 6]

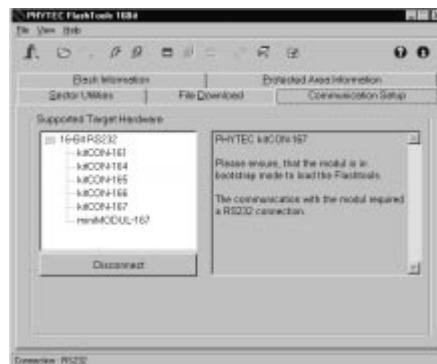


Fig 4.11 Window for **Communication Setup** tab

◆ 5 ◆

miniMODUL-167: A Synopsis

5.1 What is miniMODUL-167?

The miniMODUL-167 is a 16-bit microcontroller. Of its several features, we will discuss only about the ones we will use.

5.2 Parallel Ports

A port is a gateway for data flow (hence the name, “port”). Data can either come into the microcontroller from a port, or flow out of a microcontroller through a port to the outside world. Data comes into the microcontroller as a time varying voltage, $V(t)$. Analog to digital converters (A/D converters) in the C167 convert this analog voltage into digital data. The C167 then operates on this digital data and the result is sent to other port as needed for further use. In our application, we will use only port 5 and port 8.

Data in the digital world simply consists of two numbers: 0 and 1. The 0 usually refers to a switch (e.g. a transistor) being off (that is, connected to the lowest voltage in a circuit, usually ground) and the 1 refers to a switch being turned on (e.g. a transistor that is connected to the highest voltage in a circuit).

So, PORT5_1 refers to bit 1 on port 5 in the C167. The maximum number of bits within a port in the C167 is 16 (hence the C167 is a 16-bit microcontroller). It can be noticed that this is the maximum number; some ports in the C167 have only 8 bits (Port 8 and Port 7). Some ports are special and can handle only specific types of data, for instance Port 5 can handle only analog data.

Port 5

This is a 16-bit port. This port can be used for the inputs of analog data (known as alternate function). Each line of Port 5 is connected to the input multiplexer of the Analog/Digital Converter. All port lines can accept analog signals (ANx) that can be converted by the ADC. [reference: 7]

Fig. 5.1 summarizes the above functions of Port 5.

| Port 5 Pin | Alternate Function |
|------------|--------------------|
| P5.0 | Analog Input AN0 |
| P5.1 | Analog Input AN1 |
| P5.2 | Analog Input AN2 |
| P5.3 | Analog Input AN3 |
| P5.4 | Analog Input AN4 |
| P5.5 | Analog Input AN5 |
| P5.6 | Analog Input AN6 |
| P5.7 | Analog Input AN7 |
| P5.8 | Analog Input AN8 |
| P5.7 | Analog Input AN9 |
| P5.10 | Analog Input AN10 |
| P5.11 | Analog Input AN11 |
| P5.12 | Analog Input AN12 |
| P5.13 | Analog Input AN13 |
| P5.14 | Analog Input AN14 |
| P5.15 | Analog Input AN15 |

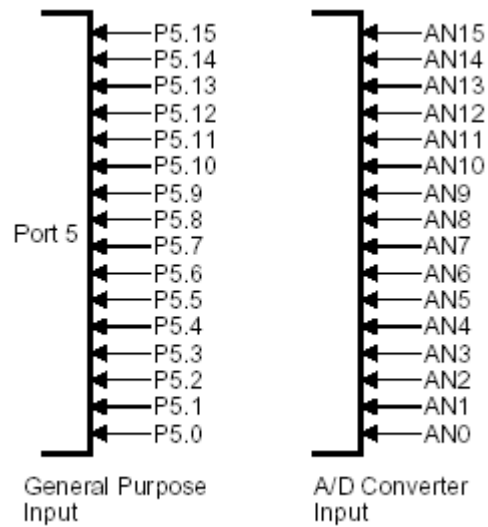


Figure 5.1 Port 8 IO and Alternate Functions

Port 8

If this 8-bit port is used for general purpose IO, the direction of each line can be configured via the corresponding direction register DP8. Fig. 5.2 shows P8 direction control register and Fig. 5.3 shows P8 IO functions. [reference: 7]

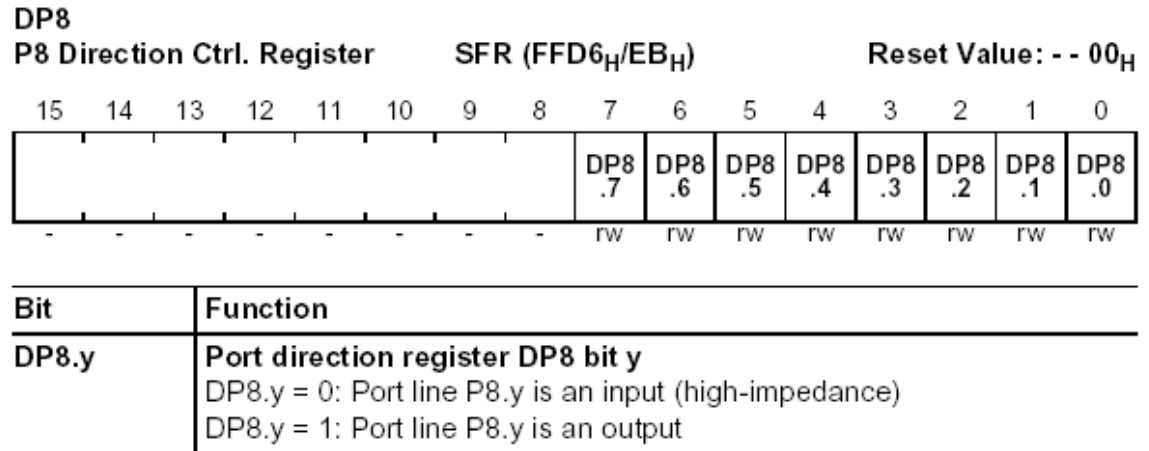


Figure 5.2 Port 8 Direction Control Register

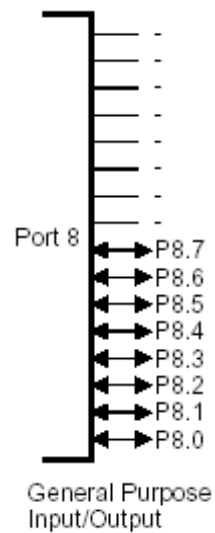


Figure 5.3 Port 8 IO Functions

5.3 The Analog/Digital Converter

The C167CR provides an Analog/Digital Converter with 10-bit resolution and a sample & hold circuit on-chip. A multiplexer selects between up to 16 analog input channels (alternate functions of Port 5) either via software (fixed channel modes) or automatically (auto scan modes). To fulfill most requirements of embedded control applications the ADC supports the following conversion modes:

- **Fixed Channel Single Conversion** produces just one result from the selected channel
- **Fixed Channel Continuous Conversion** repeatedly converts the selected channel
- **Auto Scan Single Conversion** produces one result from each of a selected group of channels
- **Auto Scan Continuous Conversion** repeatedly converts the selected group of channels
- **Wait for ADDAT Read Mode** start a conversion automatically when the previous result was read
- **Channel Injection Mode** insert the conversion of a specific channel into a group conversion (auto scan)

In our program we will use *Fixed Channel Single Conversion* Mode.

Mode Selection and Operation

As we told, the analog input channels AN15 ... AN0 are alternate functions of Port 5 which is an input-only port. The functions of the A/D converter are controlled by the bit-addressable A/D Converter Control Register ADCON. Its bit-fields specify the analog channel to be acted upon, the conversion mode, and also reflect the status of the converter. Fig. 5.4 shows ADCON register.

| ADCON | | SFR (FFA0 _H /D0 _H) | | Reset value: 0000 _H | | | | | | | | | | | |
|----------------------|----|---|----|--------------------------------|-------|------|-------|------|---|-----|---|------|---|---|---|
| ADC Control Register | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADCTC | | ADSTC | | ADCRQ | ADCIN | ADWR | ADBSY | ADST | - | ADM | | ADCH | | | |
| rw | | rw | | rwh | rw | rw | rwh | rwh | - | rw | | rw | | | |

| Bit | Function |
|-------|--|
| ADCH | ADC Analog Channel Input Selection Selects the (first) ADC channel which is to be converted. <i>Note: Valid channel numbers are 0_H to 7_H.</i> |
| ADM | ADC Mode Selection 00: Fixed Channel Single Conversion 01: Fixed Channel Continuous Conversion 10: Auto Scan Single Conversion 11: Auto Scan Continuous Conversion |
| ADST | ADC Start Bit 0: Stop a running conversion 1: Start conversion(s) |
| ADBSY | ADC Busy Flag 0: ADC is idle 1: A conversion is active. |
| ADWR | ADC Wait for Read Control |
| ADCIN | ADC Channel Injection Enable |
| ADCRQ | ADC Channel Injection Request Flag |
| ADSTC | ADC Sample Time Control (Defines the ADC sample time in a certain range) 00: $t_{BC} \times 8$ 01: $t_{BC} \times 16$ 10: $t_{BC} \times 32$ 11: $t_{BC} \times 64$ |
| ADCTC | ADC Conversion Time Control (Defines the ADC basic conversion clock f_{BC}) 00: $f_{BC} = f_{CPU} / 4$ 01: $f_{BC} = f_{CPU} / 2$ 10: $f_{BC} = f_{CPU} / 16$ 11: $f_{BC} = f_{CPU} / 8$ |

Figure 5.4 ADC Control Register

Bit-field ADCH specifies the analog input channel which is to be converted (first channel of a conversion sequence in auto scan modes). Bit-field ADM selects the operating mode of the A/D converter. A conversion (or a sequence) is then started by setting bit ADST. Clearing ADST stops the A/D converter after a certain operation which depends on the selected operating mode. The busy flag (read-only) ADBSY is set, as long as a conversion is in progress. The result of a conversion is stored in the result register ADDAT. Fig. 5.5 shows ADDAT register.

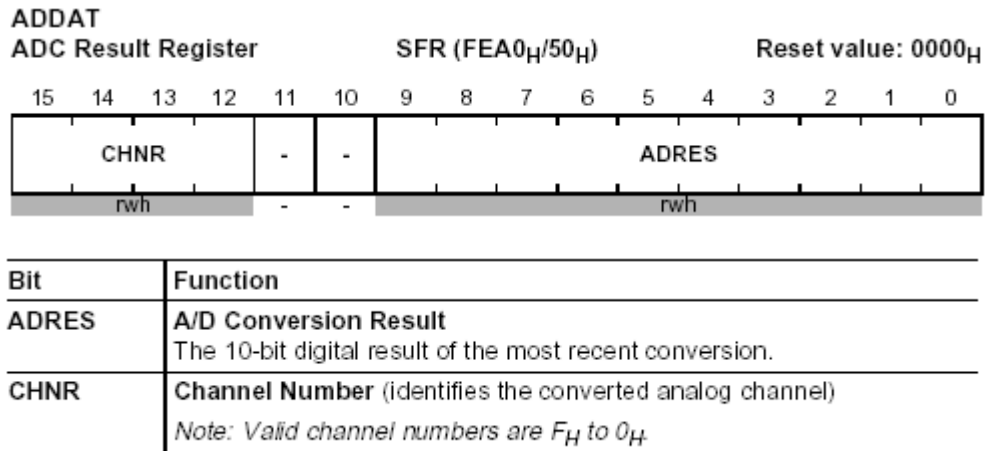


Figure 5.5 ADC result register

A conversion is started by setting bit ADST = ‘1’. The busy flag ADBSY will be set and the converter then selects and samples the input channel, which is specified by the channel selection field ADCH in register ADCON. The sampled level will then be held internally during the conversion. When the conversion of this channel is complete, the 10-bit result together with the number of the converted channel is transferred into the result register ADDAT and the interrupt request flag ADCIR is set. The conversion result is placed into bit-field ADRES of register ADDAT.

If bit ADST is reset via software, while a conversion is in progress, the A/D converter will stop after the current conversion (fixed channel modes). Setting bit ADST while a conversion is running will abort this conversion and start a new conversion with the parameters specified in ADCON.

It can be noted that abortion and restart (see above) are triggered by bit ADST changing from ‘0’ to ‘1’, i.e. ADST must be ‘0’ before being set.

While a conversion is in progress, the mode selection field ADM and the channel selection field ADCH may be changed. ADM will be evaluated after the current conversion. ADCH will be evaluated after the current conversion (fixed channel modes).

Fixed Channel Conversion Modes

These modes are selected by programming the mode selection bit-field ADM in register ADCON to ‘00B’ (single conversion) or to ‘01B’ (continuous conversion). After starting the converter through bit ADST the busy flag ADBSY will be set and the channel specified in bit field ADCH will be converted. After the conversion is complete, the interrupt request flag ADCIR will be set. [reference: 7]

◆ 6 ◆

Timers

6.1 The General Purpose Timer Units

The General Purpose Timer Units GPT1 and GPT2 represent very flexible multifunctional timer structures which may be used for timing, event counting, pulse width measurement, pulse generation, frequency multiplication, and other purposes. They incorporate five 16-bit timers that are grouped into the two timer blocks GPT1 and GPT2.

We will use only GPT1 and hence will discuss about its various features and functionality associated with our code. However, the complete explanation will be deferred until we talk about our code. [reference: 7]

6.2 Timer Block GPT1

From a programmer's point of view, the GPT1 block is composed of a set of SFRs as summarized in Fig. 6.1. Those portions of port and direction registers which are used for alternate functions by the GPT1 block are shaded.

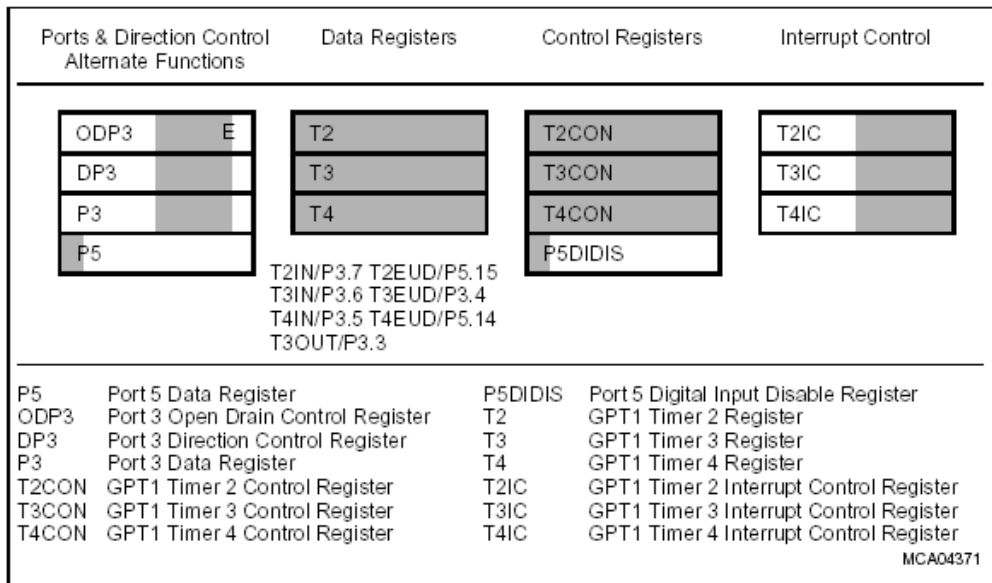


Figure 6.1 SFRs and port pins associated with timer block GPT1

Timer Block GPT1 has three timers, T2, T3 and T4. All three timers of block GPT1 can run in four basic modes, which are timer, gated timer, counter and incremental interface mode, and all timers can either count up or down. [reference: 7]

6.3 GPT1 Core Timer T3

The core timer T3 is configured and controlled via its bit addressable control register T3CON as shown in Fig. 6.2

| T3CON | | | | | | | | | | | | | | | |
|---|----|----|----|----|--------|-------|--------|-------|-----|---|-----|---|---|-----|---|
| Timer 3 Control Register | | | | | | | | | | | | | | | |
| SFR (FF42 _H /A1 _H) | | | | | | | | | | | | | | | |
| Reset value: 0000 _H | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| - | - | - | - | - | T3 OTL | T3 OE | T3 UDE | T3 UD | T3R | | T3M | | | T3I | |
| - | - | - | - | - | rwh | rw | rw | rw | rw | | rw | | | rw | |

| Bit | Function |
|-------|---|
| T3I | Timer 3 Input Selection Depends on the operating mode, see respective sections. |
| T3M | Timer 3 Mode Control (Basic Operating Mode) 000: Timer Mode 001: Counter Mode 010: Gated Timer with Gate active low 011: Gated Timer with Gate active high 100: <i>Reserved</i> . Do not use this combination. 101: <i>Reserved</i> . Do not use this combination. 110: Incremental Interface Mode 111: <i>Reserved</i> . Do not use this combination. |
| T3R | Timer 3 Run Bit 0: Timer/Counter 3 stops 1: Timer/Counter 3 runs |
| T3UD | Timer 3 Up/Down Control ¹⁾ |
| T3UDE | Timer 3 External Up/Down Enable ¹⁾ |
| T3OE | Alternate Output Function Enable 0: Alternate Output Function Disabled 1: Alternate Output Function Enabled |
| T3OTL | Timer 3 Output Toggle Latch Toggles on each overflow/underflow of T3. Can be set or reset by software. |

Figure 6.3 Timer 3 control register

1) For the effects of bits T3UD and T3UDE refer to the direction **Table 6-1**.

6.4 Timer 3 Run Bit

The timer can be started or stopped by software through bit T3R (Timer T3 Run Bit). If T3R = '0', the timer stops. Setting T3R to '1' will start the timer. In gated timer mode, the timer will only run if T3R = '1' and the gate is active (high or low, as programmed). [reference: 7]

| Pin TxEUD | Bit TxUDE | Bit TxUD | Count Direction |
|-----------|-----------|----------|-----------------|
| X | 0 | 0 | Count Up |
| X | 0 | 1 | Count Down |
| 0 | 1 | 0 | Count Up |
| 1 | 1 | 0 | Count Down |
| 0 | 1 | 1 | Count Down |
| 1 | 1 | 1 | Count Up |

Table 6.1 GPT1 Core Timer T3 Count Direction Control

6.5 Timer 3 in Timer Mode

Timer mode for the core timer T3 is selected by setting bit field T3M in register T3CON to '000B'. In this mode, T3 is clocked with the internal system clock (CPU clock) divided by a programmable prescaler, which is selected by bit field T3I. The input frequency f_{T3} for timer T3 and its resolution r_{T3} are scaled linearly with lower clock frequencies f_{CPU} , as can be seen from the following formula:

$$f_{T3} = \frac{f_{CPU}}{8 \times 2^{\langle T3I \rangle}}$$

$$r_{T3} [\mu s] = \frac{8 \times 2^{\langle T3I \rangle}}{f_{CPU} [MHz]}$$

[reference: 7]

7

Interrupts

7.1 Interrupts

An *interrupt*, as the name suggests, refers to an interruption of the microcontroller (or microprocessor) execution cycle. For each possible kind of interrupt, there will be an interrupt handler (or interrupt service routine). This is a piece of code that an operating system executes whenever an interrupt associated with the interrupt handler occurs.

Once an interrupt handler has finished executing, the processor starts off from where it was interrupted. Of course this is a very simple explanation of how an interrupt handler works.

Interrupts guarantee the execution of code when the interrupt occurs. This is useful in many contexts. For instance, suppose we want to sample a data only once every 10 milliseconds. We can write an *interrupt service routine* and hook it to the timers of our C167.

7.2 Interrupt Control for GPT1 Timers

When a timer overflows from $FFFF_H$ to 0000_H (when counting up), or when it underflows from 0000_H to $FFFF_H$ (when counting down), its interrupt request flag (T2IR, T3IR or T4IR) in register TxIC will be set. This will cause an interrupt to the respective timer interrupt vector (T2INT, T3INT or T4INT). There is an interrupt control register for each of the three timers.

We have used T3 in our program. So, our explanations will be limited to it. Fig. 7.1 shows the timer 3 control register. [reference: 7]

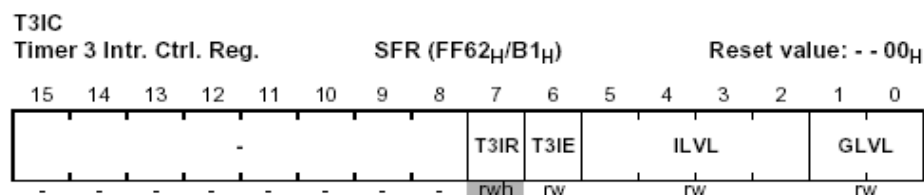


Figure 7.1 Timer 3 Control Register

◆ 8 ◆

The PID Code

8.1 The PID Code

Here is the full text version of our code:

```
1      #include <stdio.h>
2      #include <reg167.h>
3      #include<math.h>

4      // ***** GLOBAL VARIABLES AND CONSTANTS *****

5      int sensorValue ;      // for sensor

6      /// for PID control

7      int output ;

8      float Error=0;

9      float setPoint=717;

10     #define maxValue 32767

11     #define T3INT  0x23      // Interrupt vector

12     //***** FUNCTION PROTOTYPE *****

13     void Initialize(void);      // General function

14     void PID(float);      // control function

15     void AToDInitialize(void);  // A/D converter

16     unsigned int ReadSensor(unsigned int);

17     void TimerInIt(void);
```

```

18     void Interrupt(void);           // Interrupt routine

19     // ***** MAIN PROGRAM *****

20     void main(void) {

21         P3 |= 0x0400;               /* SET PORT 3.10 OUTPUT LATCH (TXD) */
22         DP3 |= 0x0400;              /* SET PORT 3.10 DIRECTION CONTROL (TXD OUTPUT) */
23         DP3 &= 0xF7FF;              /* RESET PORT 3.11 DIRECTION CONTROL (RXD INPUT) */
24         S0TIC = 0x80;               /* SET TRANSMIT INTERRUPT FLAG */
25         S0RIC = 0x00;               /* DELETE RECEIVE INTERRUPT FLAG */
26         S0BG = 0x40;                /* SET BAUDRATE TO 9600 BAUD */
27         S0CON = 0x8011;             /* SET SERIAL MODE */

28         Initialize();               // ALWAYS DO THIS FIRST!

29         while(1)

30             ;

31     }

32     void Initialize(void){

33         DP8=0xFF;                   // Use P8_0, P8_1..P8_8 as outputs

34         P8=0x00;

35         AToDInitialize();           // Initialize AnalogToDigital

36         TimerInit();                // Control Interrupt timer

37         IEN = 1;                    // GLOBALLY ENABLE INTERRUPTS

38     }

39     /***** Sensor functions - AToD module *****/

40     // Function: AToDInitialize

41     // Purpose: Initialize AToD converter module

```

```

42     // Destroys: ADCON
43     void AToDInitialize()
44     {
45         ADCON = 0x0000;           // ADM = 0x00, fixed channel SINGLE conversion
46     }

47     void TimerInit(void){
48         /// ***** Timer 3 Control Register *****/
49         /// timer 3 works in timer mode
50         /// prescaler factor is 8
51         /// up/down control bit is reset
52         /// external up/down control is disabled
53         /// alternate output (toggle T3OUT) function is disabled
54         T3CON = 0x0000;           // GPT1 Timer 3 Control Register*/

55         /* 00000| 0   | 0   | 0   | 0   | 0   | 000 | 000
56            |T3OTL| T3OE | T3UDE| T3UD |T3R |T3M | T3I */
57         //T3M=000 timer mode; T3I=000, hence preselector 8

58         T3 = 0x9E58;             // load timer 3 register; 10 ms
59         // GPT1 Timer 3 Register

60         /// enable timer 3 interrupt

61         /// timer 3 interrupt priority level(ILVL) = 13

62         /// timer 3 interrupt group level (GLVL) = 3

63         T3IC = 0x0077;           // GPT1 Timer 3 Interrupt Control Register
64         //00000000.0.1.1101.11:1101b=13d; 11b=3d

65         T3R = 1;                 // set timer 3 run bit

```

```

66     }

67     //***** INTERRUPT ROUTINE *****/
68     void Interrupt(void) interrupt T3INT {
69         sensorValue=ReadSensor(14);
70         Error=setPoint-sensorValue; //setPoint--float;sensorValue--unsigned int
71         PID(Error) ;
72     }
73     //*****Function: ReadSensor *****/
74     // Purpose:Reads a sensor value from the specified analog input port (Port P5_14).
75     // Arguments: input analog port to be read (0 to 15)
76     // Return values: converted digital value

77     //*****/
78     unsigned int ReadSensor(unsigned int port_no){
79         ADCON = (ADCON & 0xFFF0) | port_no;
80         ADST = 1;           // start conversion
81         while(ADBSY)       // wait for conversion
82             ;
83         ADCIR = 0;         // stop it
84         return (ADDAT & 0x03FF); // returns 10 bit result
    }

```

```

85      /***** Function: PID *****/
86      // Purpose: controller for driving the actuator
87      // Arguments: Kp, Kd, Ki

88      void PID(float currentError){
89          static float oldError = 0 ;
90          static float integralError = 0;
91          long temp ;
92          static float Kp =1.2;
93          static float Ki = 9.5;
94          static float Kd = 0.05;

95          integralError += currentError;      // Accumulate error for integral control
96          if(integralError>=maxValue)
97              integralError=maxValue;
98          if(integralError<=0)
99              integralError=0;

100         temp=(currentError) * Kp + (currentError - oldError)* Kd+Ki * integralError ;
101         if( temp < 0)                          // Sanity checks
102             output = 0 ;
103         else if(temp >= maxValue)
104             output = maxValue;
105         else

```

```

106     output = temp ;                // output is a 16-bit integer (int->16 bit)
107     output=((long)output*255)/32767; // Scale 0-255
108     output=((long)output*128)/255+127; // calibrate for D/A Converter
109     P8=output;
110     oldError = currentError ;      // Update error
111 }

```

8.2 Code Illustration

Line 2: # include<reg167.h> means include information about ports etc in our program. This helps us refer to port 2 as P2 (etc) and also has information about control and data registers. This line is known as a PREPROCESSOR DIRECTIVE.

Line 21-27: These lines are for serial initialization. If we want to see the output of the program on a hyper terminal then we need this part of code.

Line 28: Initialize () function initializes several other initialize functions.

Line 29: While (1) makes the loop infinity. In embedded programming, a loop never ends.

Line 33: DP8 = 0xFF tells the C167 that all the 8-bits in port 8 are to be used as output.

Line 34: P8 = 0x00 tells the C167 that all the 8-bits in port 8 have a value of 0.

Line 37: IEN = 1 means enable interrupts. We will discuss about interrupts later.

Line 45: ADCON is A/D Converter Control Register. As stated above, its reset value is 0000_H . Setting all bits zero sets the respective bits their respective modes as we can see from fig 4.4 ADCON register table.

Line 47: This is the beginning of TimerInit function. This function deals with Timer 3 Control Register T3CON. Most of the features are illustrated in the function itself. The timer here runs in Timer Mode. This timing is needed for the interrupt processing.

Line 58: T3 = 0x9E58 is equivalent of 10 ms. Every 10 ms the interrupt handling function will be invoked. Now, we will explain how T3 is equivalent of 10 ms. T3 is a GPT1 Timer 3 Register. We know from the formula,

$$r_{T3} [\mu s] = \frac{8 \times 2^{\langle T3 \rangle}}{f_{CPU} [MHz]}$$

Here,

$$T3I = 0_D ;$$

$$f_{CPU} = 20MHz$$

Therefore, $r_{T3} = 400ns$

It means that its resolution is 400 ns. Now, while counting up, T3 *overflows* from FFFF to 0000. Then it resets. When it counts 9E58, there is again a trigger. Therefore, it counts

$$FFFF - 9E58 = 61A7 \text{ times.}$$

$$61A7_H = 24999_D$$

Then, the period is:

$$24999 \times 400ns = 9.996ms \approx 10ms$$

Line 63: T3IC is Timer 3 Interrupt Control. In the program, it has been shown that interrupt priority level (ILVL) is 13 and interrupt group level (GLVL) is 3.

Line 68: This is the beginning line of interrupt handling function. This interrupt is triggered every 10 ms. This function contains **ReadSensor ()** function which returns a 10 bit digital value for further manipulation by the **PID ()** function.

Line 78: Here starts the ReadSensor () function which has an *argument* unsigned int port_no. We have used port 5_14. So, the value of the argument variable is 14.

Line 79: ADCON is Analog to Digital Control Register. It has 16 bits. ADCON & 0xFFF0 sets the last four digits zero as they have 'and' logic. This four digits then have 'or' relationship with **port_no**. Now,

$$14_D = 1110_B$$

Therefore,

$$0000 | 1110 = 1110$$

Line 84: A/D converter can convert 10 bit digital values. A/D converter Result Register ADDAT contains 10 bit the following way:

ADDAT & 03FF

gives 10 bit values from the right to the left. The maximum value it can give is *03FF* . ADDAT gets the result from A/D conversion and ‘&’ relation with *03FF* gives the return value from the function.

Line 88: It’s the beginning of the function to calculate PID output. In this program we have used previous K_p , K_i and K_d values. We have tested these values and these give good results.

We have used port 8.0-8.7 as the out put port.

9.2 D/A Converter

The output we get from the microcontroller is digital. In order to get analog output, we have used an 8 bit D/A converter. The schematic diagram of the circuitry is given in Fig. 9.2:

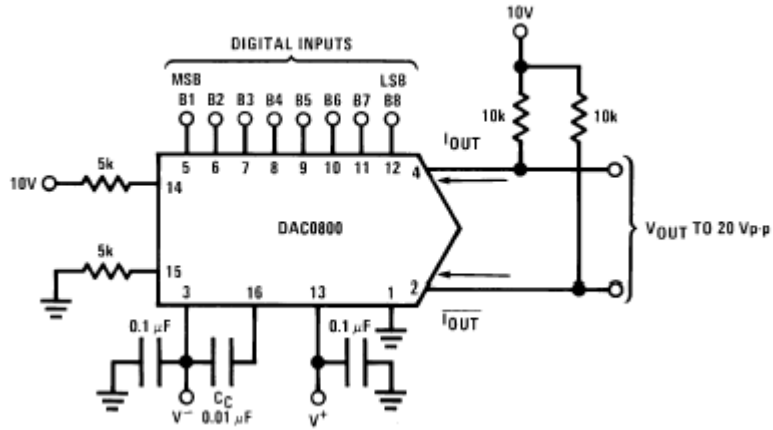


Figure 9.2 ± 20 VP-P Output digital-to-analog converter

9.3 Connection Diagrams

Figure 9.3 shows the pin lay-outs of our converter:

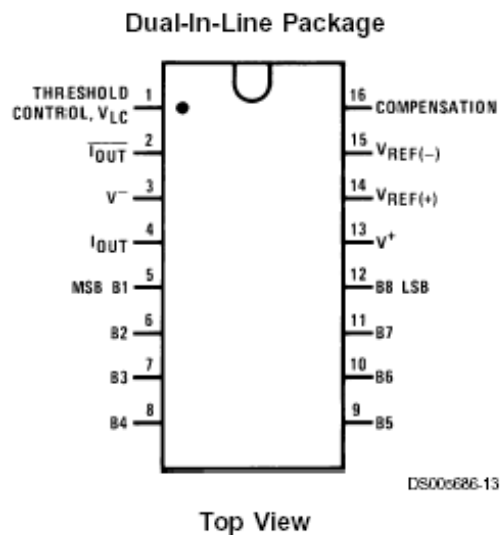


Figure 9.3 Connection diagram of the D/A converter

We know that an 8 bit D/A converter has a resolution of 1/256. It can give any value between 0 and 255. That is why we have limited our values from 0 to 255 in our program. However, Fig. 9.4 shows that we have to change a little in the program in order to adjust output of the controller with the D/A converter:

| | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | E_o | \bar{E}_o |
|---------------------|----|----|----|----|----|----|----|----|---------|-------------|
| Pos. Full Scale | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | -9.920 | +10.000 |
| Pos. Full Scale-LSB | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | -9.840 | +9.920 |
| Zero Scale+LSB | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -0.080 | +0.160 |
| Zero Scale | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.000 | +0.080 |
| Zero Scale-LSB | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | +0.080 | 0.000 |
| Neg. Full Scale+LSB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | +9.920 | -9.840 |
| Neg. Full Scale | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | +10.000 | -9.920 |

Figure 9.4 Basic bipolar output operation

9.4 Calibration

From the above figure we can see that for 10000000 the converter (E_o) gives 0 volt. But the microcontroller gives 0 voltage for 00000000. We can establish one relationship between these two outputs in such a way that when microcontroller output is 00000000, D/A converter gets 10000000 and for microcontroller output 11111111, D/A converter gets 00000000 and so on.

$$\text{Now, } \begin{aligned} 11111111_B &= 255_D \\ 10000000_B &= 128_D \end{aligned}$$

The straight-line equation:

$$y = \tan \theta \cdot x + c$$

Where y = D/A input, x = microcontroller output and $\tan\theta$ = slope.

Then

$$y = -\frac{128}{255}x + 128$$

This is what implemented in the program.

Fig. 9.5 will help us understand:

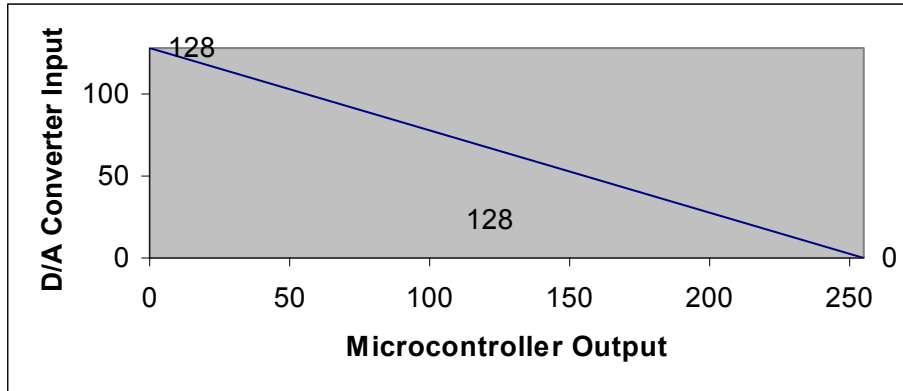


Figure 9.5 Calibration of the D/A Converter

◆ 10 ◆

Remarks on Results

Our task was to find an appropriate solution for controlling the pressure of the system shown in Fig. 10.1.



Figure 10.1 The overall system

We have developed the code for the PID algorithm and run it using the microcontroller. The actuator responds to the output of the microcontroller. The microcontroller can read the sensor data.

We have constructed several peripheral circuits such as interfacing sensor with the microcontroller, interfacing microcontroller output with the D/A converter, D/A converter output with the actuator. All the circuits worked well and gave us desirable results.

On top of all these, our system could be maintained at a constant pressure, which was the main objective of this project.

◆ 11 ◆

Summary and Conclusions

11.1 Introduction

The results presented in this thesis have been part of a continuing investigation into the pressure control mechanism of the pneumatic organs. The recent investigations employed a PC based on PID algorithm. Employing a PC in such an application makes the task tedious as one always needs to operate it. Moreover, as a PC is a general purpose computing machine, using it in the system makes it costlier.

Therefore, the next approach was to employ a microcontroller in place of a PC. This thesis is the outcome of this idea. In this, thesis we have used a microcontroller which can control the system automatically without the help of an operator. The PID code can permanently be burned in the FLASH memory or even can be changed in accordance with the need.

11.2 Future Work

As mentioned before, this work is a continuing investigation on how efficiently pressure can be controlled irrespective of the flow rate demand. We have used prior ideas in order to carry out this task. We have written the code for it and got essentially good result from our work. However, there are always scopes which can improve the result or even can modify the present findings.

We have used K_p , K_i and K_d values from the prior investigations. It is said that the actuator movement is a bit slower than what was when the PC was used. Therefore, we can investigate the fact more by changing these values. Standard tuning methods can be used with a view to determining the optimum values.

We have tried to keep our code simple and clean. Data types of the variables can be examined to get better result. One related issue is the assignment of maximum value of the PID output. Our program assumes maximum value to be 32767. It may take a considerable amount of time to reach the value, and hence the movement of the actuator. We have scaled the output from 0 to 255 as we have used 8 bit D/A converter. If the maximum value is taken less than what we did, the change of the PID output would be much quicker, and so would be the movement of the actuator.

The source voltage of the D/A converter has been taken to be ± 15 volts and the reference voltage to be 10 volts. All these voltage sources can be mounted on a single board to keep the controller compact. Our investigation showed that the actuator needs 0-10 volts for actuation. Further investigation might be helpful to authenticate the issue.

Other significant investigation might include optimum sample time calculation for triggering the interrupt handling function. Although in our program we have used this to be 10 ms, we have tried even lesser values, e.g. 2 ms. Apparently, it did not show much difference. Further investigation would be helpful to settle down the issue.

List of References

- [1] Buxbaum, A.; Schierau, K.; Straughen, A.: Design of Control Systems for DC drives, *Spinger-verlag*, 1990.
- [2] Smuts, J.: PID Controllers Explained, *Canadian Process Equipment & Control News*, February 2002. (http://www.pas.com/white_papers/pid_explained.pdf)
- [3] Roth, H.: Quasi Continuous Controller, *Lecture Notes at University of Siegen, Germany*, summer 2003.
- [4] Bharat, M. B.: Microcontroller Basics, *Robotics Notes at UC Berkley, USA*, August 2003.
(<http://inst.eecs.berkeley.edu/%7Eee40/calbot/pdf/ChapterTwo/ChapterTwo.pdf>)
- [5] Bharat, M. B.: Programming the 167, *Robotics Notes at UC Berkley, USA*, August 2003.
(<http://inst.eecs.berkeley.edu/%7Eee40/calbot/pdf/ChapterFour/ChapterFour.pdf>)
- [6] Keil User Manual
- [7] Infineon C167CR-LM Datasheet