

Airflow measurement using the 83/87C752 and “C”

AN429

INTRODUCTION

This application note describes a low-cost airflow measurement device based on the Philips 83/87C752 microcontroller. Airflow measurement—determining the volume of air transferred per unit time (cubic feet per minute, or cfm)—is intrinsic to a variety of industrial and scientific processes.

Airflow computation depends on three simultaneous physical air measurements—velocity, pressure, and temperature. This design includes circuits and sensors allowing the 8XC752 to measure all three parameters.

The design also includes seven-segment LED displays, discrete LEDs, and pushbutton switches to allow selective display of airflow, temperature, and pressure. Furthermore, airflow is continuously compared with a programmer-defined setpoint. Should the measured airflow exceed the setpoint, an output relay is energized. In actual application, this relay output could be used to signal the setpoint violation (via lamp or audio annunciator) or otherwise control the overall process (e.g., emergency process shutdown). Of course, the setpoint, comparison criteria (greater, less than, etc.) and violation response (relay on, relay off) are easily changed by program modification to meet actual application requirements.

Referring to Figure 1, the overall operation of the airflow device is as follows.

Normally the unit continuously displays the airflow (in cfm) on the seven-segment

displays. The discrete CFM LED is also lit to confirm the parameter being displayed.

Pressing the TEMP pushbutton switches the display to temperature (in degrees C) and lights the TEMP LED. As long as the pushbutton remains pressed, the temperature is displayed. When the pushbutton is released, the display reverts to the default pressure display.

Similarly, pressing the PSI pushbutton displays the atmospheric pressure (in pounds per square inch) and lights the PSI LED. The pressure is displayed as long as the pushbutton is pressed, and the default airflow display resumes when the pushbutton is released.

Finally, pressing the SET-POINT pushbutton displays the programmed airflow setpoint (in cfm) and lights the SET-POINT LED. Again, releasing the pushbutton causes the display to revert to the default airflow measurement.

CONTROL PROGRAMMING IN “C”

While, thanks to advanced semiconductor processing, hardware price/performance continues to improve, software development technology has changed little over time. Thus, given ever-rising costs for qualified personnel, software “productivity” is arguably in decline. Indeed, for low-unit cost and/or low-volume applications, software development has emerged as the major portion of total design cost. Furthermore,

beyond the initial programming cost, “hidden” costs also arise in the form of life-cycle code maintenance and revision and lost revenue/market share due to excessive time-to-market.

Traditionally, control applications have been programmed in assembly language to overcome microcontroller resource and performance constraints. Now, thanks to more powerful microcontrollers and advanced compiler technology, it is feasible to program control applications using a High-Level Language (HLL).

The primary benefit of using an HLL is obvious—one HLL program “statement” can perform the same function as many lines of assembly language. Furthermore, a well-written HLL program will typically be more “readable” than an assembly language equivalent, resulting in reduced maintenance and revision/upgrade costs.

Of the many popular HLLs, the “C” language has emerged as the major contender for control applications. More than other languages, C gives the programmer direct access to, and control of, low-level hardware resources—a requirement for deterministic, real-time I/O applications. Furthermore, C is based on a “minimalist” philosophy in which the language performs only those functions explicitly requested by the programmer. This approach is well-suited for control applications, which are often characterized by strict cost and performance requirements.

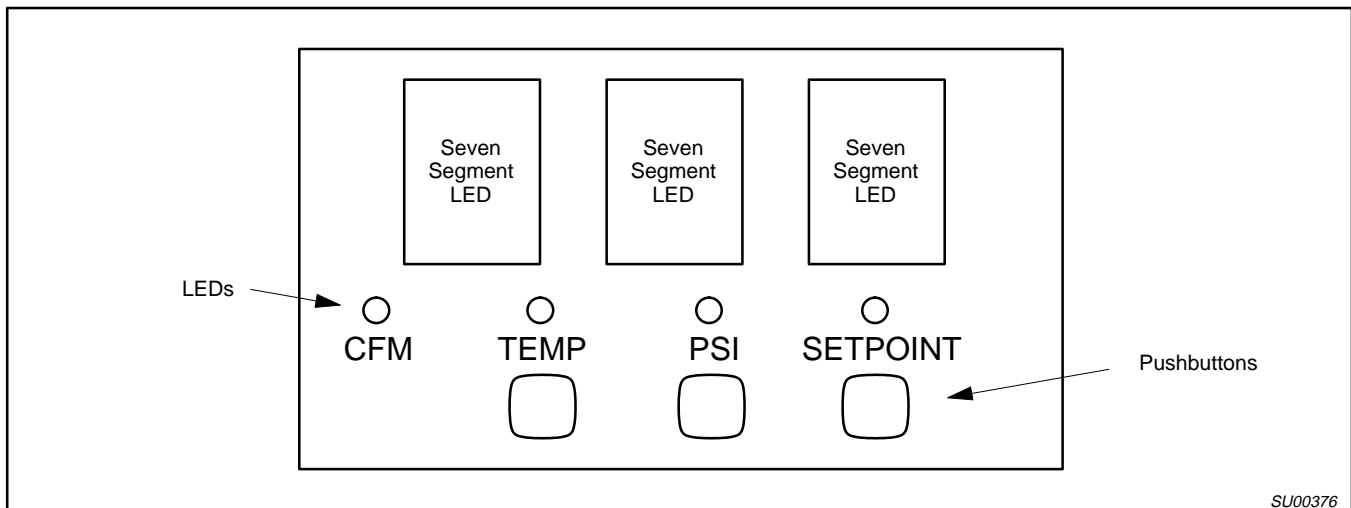


Figure 1. Airflow Meter Front Panel

Airflow measurement using the 83/87C752 and “C”

AN429

8XC752 OVERVIEW

The 83C752/87C752 (ROM/EPROM-based) combine the performance advantages of the 8-bit 80C51 architecture with the low cost, power consumption, and size/pin count of a 4-bit microcontroller. Therefore, the 8XC752 is uniquely capable of bringing high processing speed and HLL programming to even the most cost-sensitive applications such as handheld (battery driven) instruments, automotive distributed processing, “smart” appliances, and sophisticated consumer electronics.

Obviously, the 8XC752 can be used for cost-reduced versions of existing 8-bit applications. The device can also replace similarly priced 4-bit devices to achieve benefits of higher performance and, most importantly, easier s/w development including the use of HLL. Indeed, the component and system design costs associated with the 8XC752 are so low that it is a viable candidate for first-time computerization of formerly non-microcontroller-based designs.

Figure 2 shows the block diagram of the 8XC752. Major features of the device include the following.

Full-Function, High-Speed (to 16MHz) 80C51 CPU Core

The popular 80C51 architecture features 8- and 16-bit processing and high-speed execution. Most instructions execute in a single machine cycle (the slowest instructions require only two cycles). Though a streamlined architecture, the CPU core,

unlike 4-bit devices, includes all the basic capabilities (such as stack, multiply instruction, interrupts, etc.) required to support HLL compilation. The CPU core also includes a unique Boolean processor which is well-suited for the bit-level processing and I/O common to control applications.

Low-Power CMOS and Power-Saving Operation Modes

Thanks to the advanced CMOS process, the 8XC752 features extremely low power consumption, which helps to extend battery life in handheld applications and otherwise reduce power supply and thermal dissipation costs and reliability concerns. Low ACTIVE mode (full-speed operation) power consumption—only 11mA typical at 12MHz—is further complemented by two program-initiated power-saving operation modes—IDLE and POWER-DOWN.

In idle mode, CPU instruction processing stops while on-chip I/O and RAM remain powered. Power consumption drops to 1.5 μ A (typical, 12MHz) until processing is restarted by interrupt or reset. Power-down mode cuts power consumption further (to only 10 μ A typical at 12MHz) by stopping both instruction and I/O processing. Return to full-speed operation from power-down mode is via reset.

Note that power consumption can be further cut by reducing the clock frequency as much as application performance requirements allow, as shown in Figure 3.

Another virtue of the CMOS process is superior tolerance to variations in V_{CC} , a requirement inherent in the targeted applications. The EPROM-based device (87C752) operates over a V_{CC} range of 4.5V to 5.5V, while the ROM-based device (83C752) will operate from 4V to 6V.

On-Chip ROM (83C752), EPROM (87C752), and RAM

The 8XC752 integrates 2048 bytes of program ROM/EPROM and 64 bytes of data RAM. This relatively small amount of memory reflects the fact that the targeted applications, though they may require high-speed processing, are typically characterized by simple algorithms and data structures. High code efficiency of the architecture means even this small amount of memory can effectively support the use of C. If necessary, the judicious use of assembly language can help bypass code size (and performance) constraints.

Five-Channel 8-Bit A/D Converter

Most control applications are characterized by the need to monitor “real-world” (i.e., analog) parameters. To this end, the 8XC752 includes a medium-speed (40 clock cycle conversion) 8-bit analog-to-digital (A/D) converter. Five separate input lines are provided along with multiplexer logic to select an input for conversion. The A/D converters speed, resolution, and accuracy are more than adequate to measure temperature, pressure, and other common environmental parameters.

Airflow measurement using the 83/87C752 and "C"

AN429

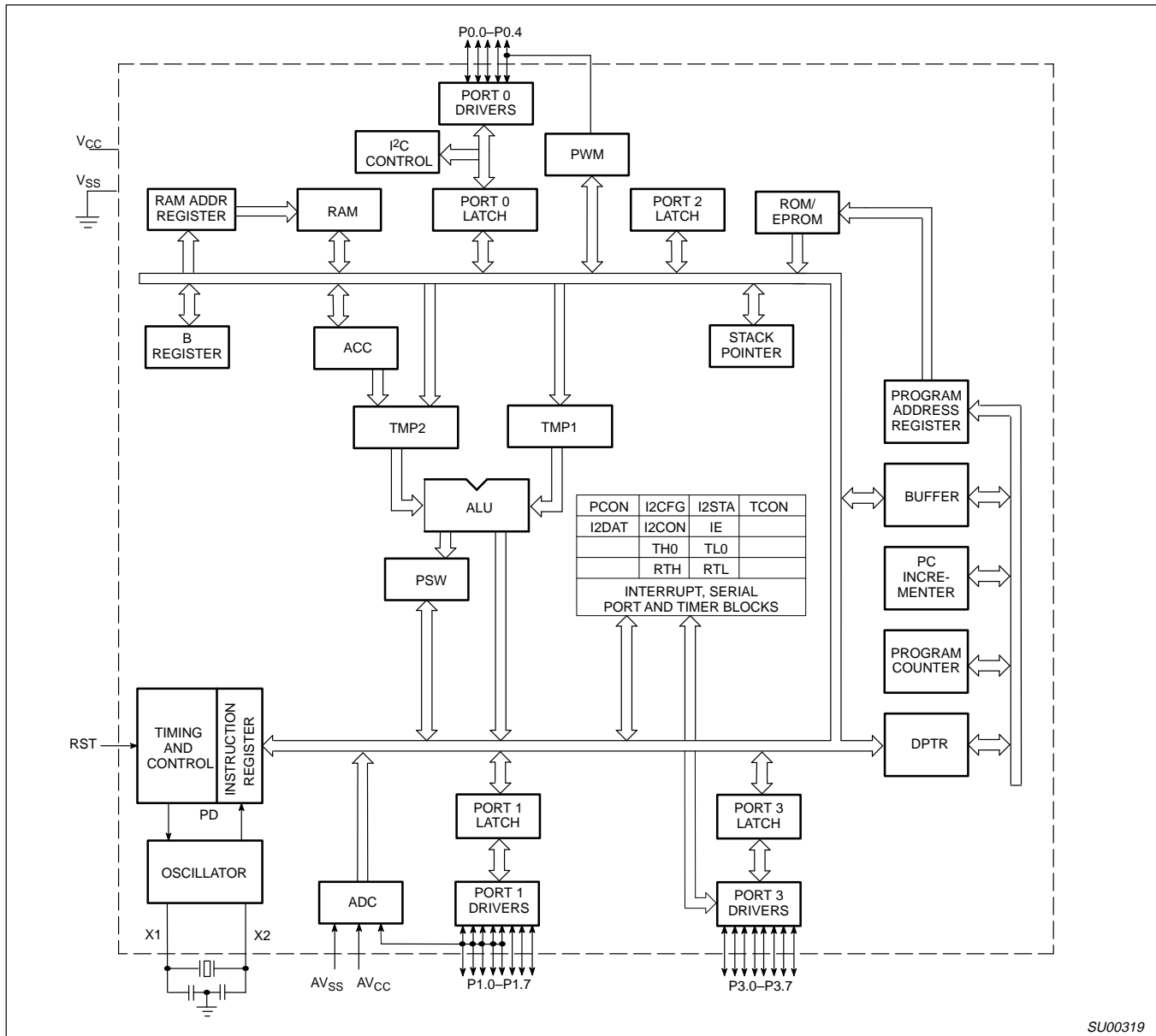


Figure 2. Block Diagram of the 8XC752

SU00319

Timer/Counters

Control applications, due to their "real-time" nature, invariably call for a variety of timing and counting capabilities. The 8XC752 meets the need by integrating three separate functions—a 16-bit auto-reload counter/timer, an 8-bit pulse width modulator (PWM) output/timer, and a fixed-rate timer for timebase generation. Together, these timing/counting resources can serve a range of tasks, including waveform generation, external event counting, elapsed time

calculation, periodic interrupt generation, and watchdog timer.

I²C Bus

The Inter-Integrated Circuit (I²C) bus is a patented serial peripheral interface. The virtue of I²C is the ability to expand system functionality with acceptable performance and minimum cost. Notably, the pin and interconnect count is radically reduced compared to expansion via a typical microprocessor bus—I²C requires only two

lines, while a parallel bus often consumes 20-30 lines and may call for extra glue logic (decoder, address latch, etc.). The 8XC752 I²C port allows easy connection to a wide variety of compatible peripherals such as LCD drivers, A/D and D/A converters, consumer/telecom and special-purpose memory (e.g., EEPROM). I²C can also be used to build distributed processing systems connecting multiple I²C-compatible microcontrollers.

Airflow measurement using the 83/87C752 and "C"

AN429

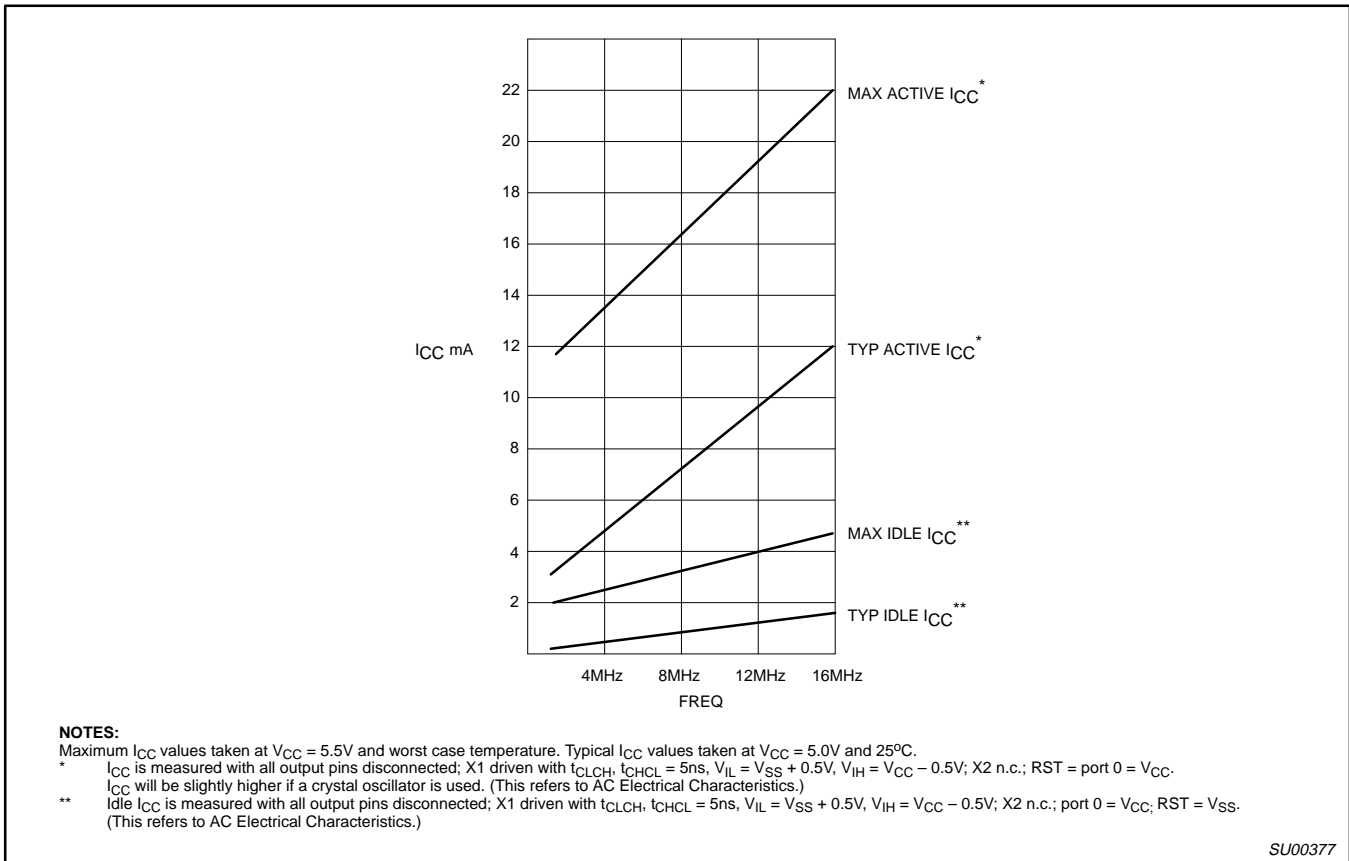


Figure 3. I_{CC} vs. FREQ

8XC752 PIN FUNCTIONS

Since the 8XC752 is packaged in a cost/space-saving 28-pin package DIP or PLCC, a flexible mapping of I/O functions to pins is required to ensure the widest possible application coverage.

Of the 28 pins, seven pins are allocated to basic functions, including digital power (V_{CC}, V_{SS}), analog reference (AV_{CC}, AV_{SS}), clock oscillator (X1, X2), and reset (RST). Thus, 21 pins, organized into three ports (5-bit port 0, 8-bit ports 1 and 3), are available for user I/O.

Figure 4 shows the alternative uses for these 21 lines. As shown, the mapping is quite versatile, which maximizes the access to on-chip I/O functions and helps ensure full pin utilization.

P0.0	TTL IN/OUT (open drain), I ² C clock (SCLK)
P0.1	TTL IN/OUT (open drain), I ² C data (SDA)
P0.2	TTL IN/OUT (open drain)
P0.3	TTL IN/OUT (internal pull-up)
P0.4	TTL IN/OUT (internal pull-up), PWM output
P1.0	TTL IN/OUT (internal pull-up), A/D input channel 0
P1.1	TTL IN/OUT (internal pull-up), A/D input channel 1
P1.2	TTL IN/OUT (internal pull-up), A/D input channel 2
P1.3	TTL IN/OUT (internal pull-up), A/D input channel 3
P1.4	TTL IN/OUT (internal pull-up), A/D input channel 4
P1.5	TTL IN/OUT (internal pull-up), INT0 interrupt input
P1.6	TTL IN/OUT (internal pull-up), INT1 interrupt input
P1.7	TTL IN/OUT (internal pull-up), TIMER 0 (T0) input
NOTE: P1.0–P1.4 may only be changed as a group, i.e., either all TTL I/O or all A/D inputs. However, when selected as A/D inputs, P1.0–P1.4 may also be used as TTL inputs.	
P3.0	TTL IN/OUT (internal pull-up)
P3.1	TTL IN/OUT (internal pull-up)
P3.2	TTL IN/OUT (internal pull-up)
P3.3	TTL IN/OUT (internal pull-up)
P3.4	TTL IN/OUT (internal pull-up)
P3.5	TTL IN/OUT (internal pull-up)
P3.6	TTL IN/OUT (internal pull-up)
P3.7	TTL IN/OUT (internal pull-up)

SU00378

Figure 4. 8XC752 I/O Port Description

Airflow measurement using the 83/87C752 and “C”

AN429

AIRFLOW METER CIRCUIT DESCRIPTION

Figure 5 is the schematic diagram of the airflow meter circuit. As shown, the 8XC752 is connected to the following function blocks.

Discrete and Seven-Segment LED Display

The seven-segment LEDs display the parameter of interest (airflow, temperature, pressure, or setpoint). A discrete LED associated with each parameter is lit when that parameter is being displayed.

The seven-segment LEDs are identified as X0.1, X1, and X10, reflecting their decimal position (tenths, ones, and tens, respectively). Each display has eight data inputs (the seven segments and a decimal point) and common terminals which allow the display to be enabled or blanked. The eight data inputs, and the four discrete LEDs, are driven from port 3 of the 8XC752 via high-current driver U2 and current limiting resistors RP1.

Since all the segmented and discrete LEDs share common data lines, data display must be time multiplexed. Transistors Q1-Q4 connect to separate output lines of port 0, allowing a particular seven-segment LED or the discrete LEDs (as a group) to be individually enabled for display. This type of LED multiplexing is quite common since, at a fast enough refresh rate, the switching between displays is not perceptible by the operator. The major benefit is the reduction of I/O lines required (without multiplexing, 28, rather than 8, data lines would be required).

Pushbutton Switch Inputs

Three pushbuttons select the parameter to be displayed—temperature, pressure, or setpoint (when no button is pressed, airflow is displayed). The four states (SW1, SW2, SW3, or no button pressed) are effectively encoded onto two port 1 input lines (taking advantage of the capability to use port 1 lines configured as A/D for TTL input) as follows:

	P1.3	P1.4
No button pressed	HIGH	HIGH
SW1 (TEMP) pressed	LOW	HIGH
SW2 (PSI) pressed	HIGH	LOW
SW3 (SETPOINT) pressed	LOW	LOW

The only impact of this encoding scheme is that SW3 has a higher priority than the other pushbuttons—a factor of no concern in this simple application. Similarly, latching, debouncing, rollover, or other conditioning of the pushbutton inputs is not required.

Setpoint Control

This is simply a variable resistor voltage divider which serves to establish an analog voltage corresponding to an airflow threshold at which action is taken. It connects to a port 1 A/D input.

Relay Output

When an airflow setpoint violation is detected, DPDT relay K1 is energized via P1.6, which is configured as a TTL output, buffered by transistor Q5.

Flowmeter Input

Measurement of the air velocity is via an air turbine tachometer connected, via optoisolator U7, to P1.5, which is configured as a TTL input. The tachometer input is

assumed to be a negative-going pulse train with less than 10% duty cycle.

Air Pressure Sensor

To determine airflow, the air velocity must be factored by ambient pressure—for a given velocity (and temperature), lower/higher atmospheric pressure will correspond with lower/higher airflow. The pressure sensor, U3, outputs a voltage differential corresponding to the pressure. Amplifier U4 conditions the pressure sensor output to the range of AV_{SS} to AV_{CC} (the analog references for the 8XC752 A/D converter). The conditioned pressure sensor output is presented to A/D input P1.0.

To calibrate the pressure sensor, press the PSI pushbutton and adjust the gain pot (R1) until the display matches the local atmospheric pressure in pounds per square inch (14.7 at sea level).

Air Temperature Sensor

Similar to pressure, ambient temperature also affects the airflow calculation. For a given air velocity (and pressure), higher/lower temperature will correspond with lower/higher airflow. Temperature sensor U5 outputs an absolute voltage corresponding to temperature. Amplifier U6 conditions the temperature sensor output to the range AV_{SS} to AV_{CC} for connection to A/D input P1.1.

To calibrate the temperature sensor, adjust the gain pot (R5) so that the display (while pressing the TEMP pushbutton) matches the measured output of U5 (LM35).

Figure 6 summarizes the usage of the 8XC752 I/O lines in this application.

Airflow measurement using the 83/87C752 and "C"

AN429

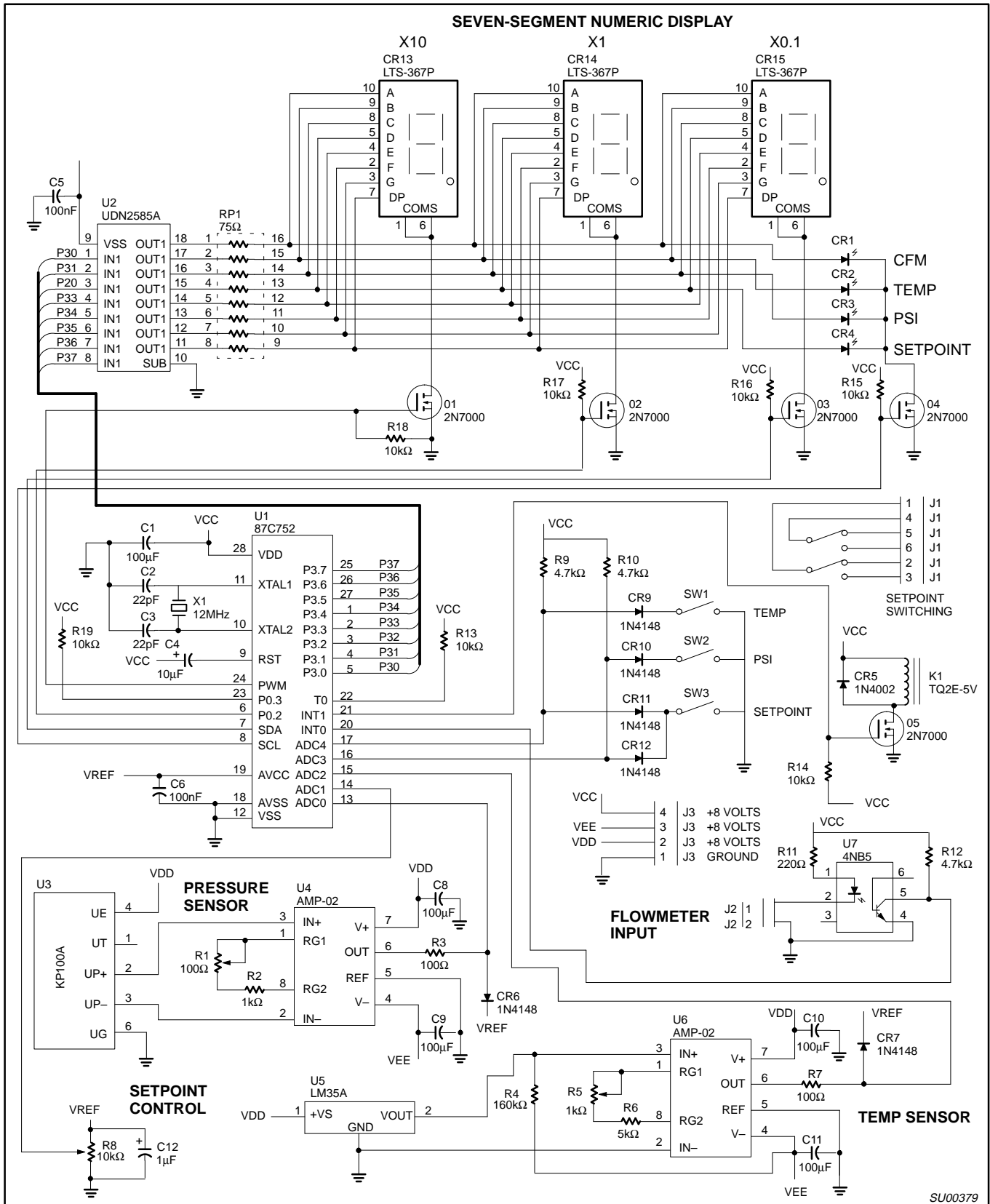


Figure 5. Schematic Diagram of the Airflow Meter Circuit

SU00379

Airflow measurement using the 83/87C752 and "C"

AN429

P0.0	TTL OUT—Enables the discrete LEDs
P0.1	TTL OUT—Enables the tenths digit seven-segment LED
P0.2	TTL OUT—Enables the ones digit seven-segment LED
P0.3	Pulled up
P0.4	TTL OUT—Enables the tens digit seven-segment LED
P1.0	A/D input—Connected to analog air pressure sensor
P1.1	A/D input—Connected to analog air temperature sensor
P1.2	A/D input—Connected to analog setpoint control
P1.3	TTL IN—One of two pushbutton input lines
P1.4	TTL IN—The second pushbutton input line
P1.5	INT0 interrupt input—Air turbine tachometer input
P1.6	TTL OUT—Setpoint relay control
P1.7	Pulled up
NOTE: P1.0–P1.4 may only be changed as a group, i.e., either all TTL I/O or all A/D inputs. However, when selected as A/D inputs, P1.0–P1.4 may also be used as TTL inputs.	
P3.0	TTL OUT—Seven-segment LEDs segment A, CFM discrete LED
P3.1	TTL OUT—Seven-segment LEDs segment B, TEMP discrete LED
P3.2	TTL OUT—Seven-segment LEDs segment C, PSI discrete LED
P3.3	TTL OUT—Seven-segment LEDs segment D, SETPOINT discrete LED
P3.4	TTL OUT—Seven-segment LEDs segment E
P3.5	TTL OUT—Seven-segment LEDs segment F
P3.6	TTL OUT—Seven-segment LEDs segment G
P3.7	TTL OUT—Seven-segment LEDs segment DP

SU00380

Figure 6. Airflow Meter I/O Port Usage

SOFTWARE DEVELOPMENT PROCEDURE

The airflow meter application software is almost entirely written in C using a development package from Franklin Software. The Franklin Software C compiler is a cross-compiler that runs on the IBM PC (and compatibles) while generating code suitable for execution by any 80C51-based product, including the 8XC752. For more information, contact:

Franklin Software
888 Saratoga Ave., #2
San Jose, CA 95129

The process of developing a C program using the Franklin package (the process is similar for other third-party cross-compilers) is as follows:

1. The program is entered/edited on the PC using the programmer's preferred text editor.
2. The program is compiled on the PC with the Franklin C compiler.
3. Should compile errors (also known as syntax errors) occur, they are corrected by returning to step 1 until an error-free compile is achieved.
4. Before testing the compiled program, it needs to be combined, using the Franklin-supplied linker, with any required assembly language routines. Besides routines explicitly written by the programmer, every Franklin C program requires an assembly language startup routine (supplied by Franklin and, if necessary, edited by the programmer) which performs basic reset initialization and configuration operations before transferring control to the C program.
5. The compiled object code is tested for correct operation. This can either be accomplished by using an 80C51-family simulator running on the PC or by downloading the object code to an in-circuit emulator. The simulator approach has the virtues of low cost and consolidation of all work on the PC at the cost of non-real-time operation/debug constraints (the simulator may execute 100-1000 times slower than the microcontroller). The in-circuit emulator provides real-time operation and the additional benefit of assisting hardware design debug at somewhat higher cost.
6. Should program execution prove faulty (known as semantic errors), return to step 1 until error-free operation is achieved.
7. The error-free (syntax and semantic) and linked object code, in the form of a .HEX file, is transferred to an EPROM programmer. Fitted with a suitable adaptor, the EPROM programmer can "burn" the object file into the targeted EPROM-based 80C51-family device. For ROM-based devices, the object file is transferred to the factory for custom masking.

Airflow measurement using the 83/87C752 and “C”

AN429

PROGRAM DESCRIPTION

Figure 7 is a flowchart of the program; following the flowchart is the program listing. The flowchart shows the basic processing and flow, while the listing documents the details of the program's implementation.

The program consists of four interrupt-driven (i.e., foreground) routines and a main program (i.e., background). The background program is entered at reset and executes forever, interrupted periodically by the foreground interrupts. Communication between the background program and the foreground handlers is via shared variables.

The four interrupt routines are as follows.

- `multiplex ()` (INT3)

Free-running Timer 1 generates an interrupt at approximately 1000Hz and is used to multiplex the seven-segment and discrete LED display data. In a round-robin manner, at each interrupt, the program turns off the previously enabled display and writes data to, and enables, the next display. Finally, the interrupt routine sets a pointer to the next display—at the next interrupt, that display will be refreshed. Thus, each display (tens, ones, tenths, discrete LEDs) will be refreshed every fourth interrupt, which is more than fast enough for a flicker-free display.

- `read_switch ()` (INT6)

The PWM prescaler is configured to generate a periodic interrupt (INT6) at about 97Hz. The program counts these interrupts, and every 32nd interrupt sets an “update” variable. The main program will change the display data when it detects that “update” is set and clear “update” to prepare for the next display cycle. Thus, display change frequency is about 33Hz (i.e., 33ms), which eliminates display glitches associated with pushbutton switch bounce.

- `calc_cfm ()` (INT0)

The air velocity turbine tachometer drives the 8XC752 INT0 interrupt pin. At each interrupt, the program reads Timer 0, which keeps track of the elapsed time (the low 16 bits of a 24-bit count in microseconds) between INT0 interrupts. The high-order 8-bit elapsed time count is cleared for possible updating by the following routine.

- `overflow ()` (INT1)

When Timer 0 overflows (generating an interrupt), the program increments the high-order 8 bits of a 24-bit variable, counting the microseconds between tachometer interrupts (handled by the previous routine). If this 8-bit value becomes too large (i.e., tachometer interrupts stop), a NOFLOW

variable is set, which will cause the main program to display an EEE out-of-range indicator on the seven-segment LEDs.

With the interrupt handlers executing the low-level timing and I/O, the main program, which is entered on reset and executes forever, consists of only three major steps.

The temperature/pressure compensated airflow is calculated. First, the “base” cfm rate, as tracked by the `calc_cfm ()` tachometer interrupt is adjusted by removing the execution time of the `calc_cfm ()` handler itself. Next, the temperature is determined (A/D channel 1), and airflow is compensated. Similarly, the air pressure is determined (A/D channel 0) and airflow compensated again.

Now that the true airflow is calculated, it is compared with the setpoint (adjusted with the variable resistor), which is determined by reading A/D channel 2. If the airflow is greater than the setpoint, the relay is closed. Otherwise, the relay is opened.

Finally, the UPDATE flag (set by the 33Hz `read_switch ()` interrupt) is checked. If it is time to update, the data to be displayed is determined based on the pushbutton status and the state of the NOFLOW flag. The updated display data is initialized for later display on the LEDs by the `multiplex ()` display refresh interrupt handler.

Airflow measurement using the 83/87C752 and "C"

AN429

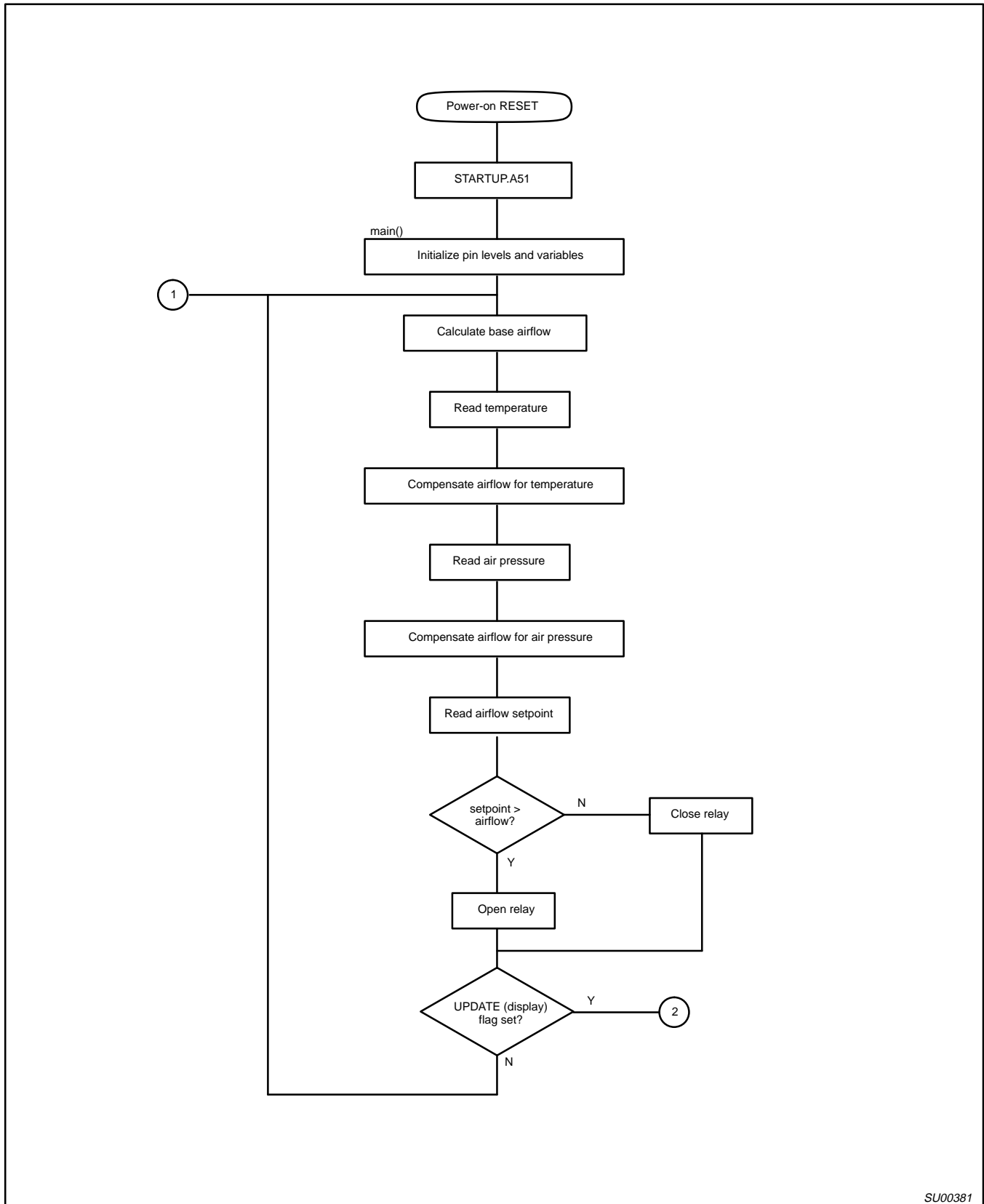


Figure 7. Program Flowchart

SU00381

Airflow measurement using the 83/87C752 and "C"

AN429

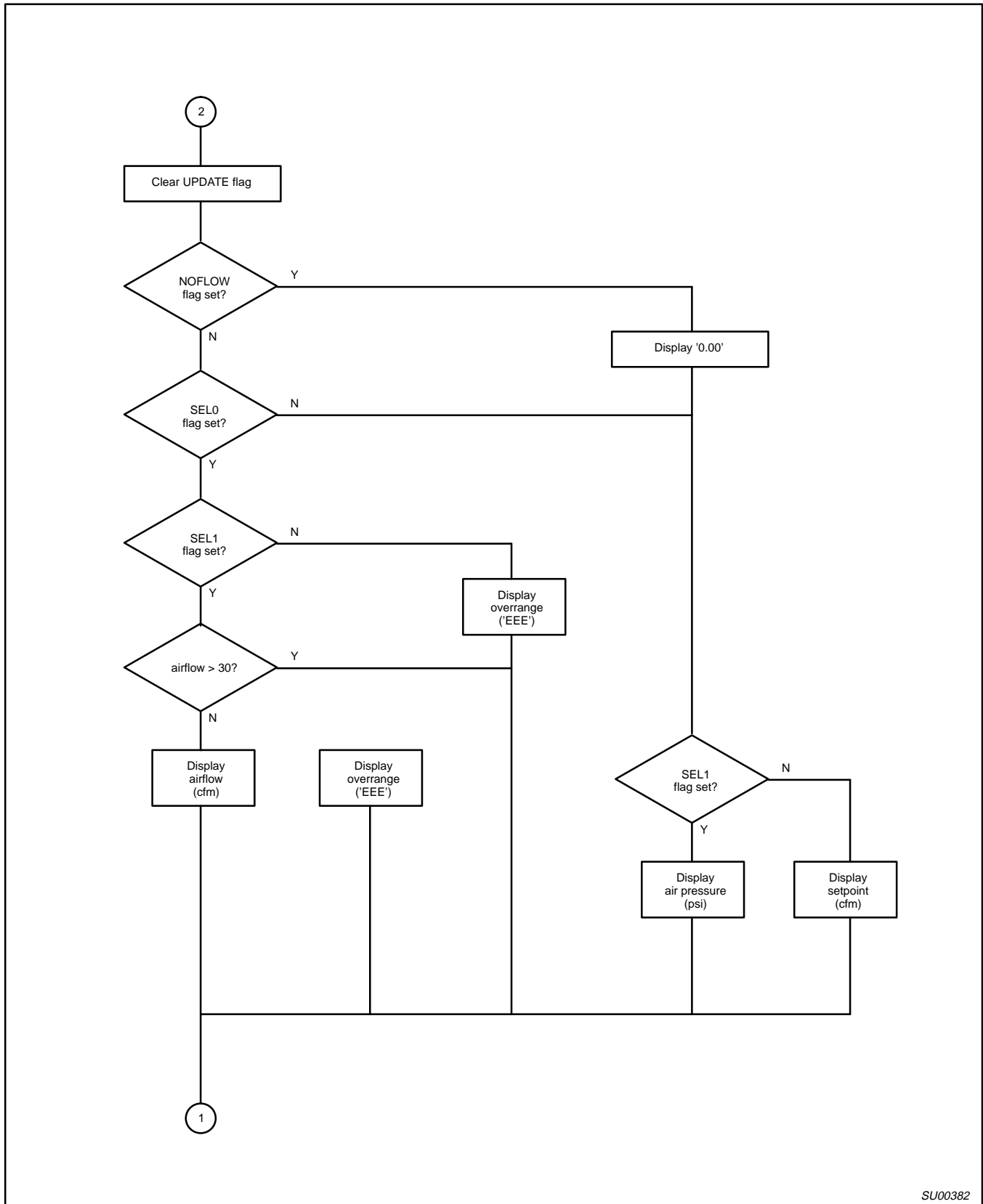
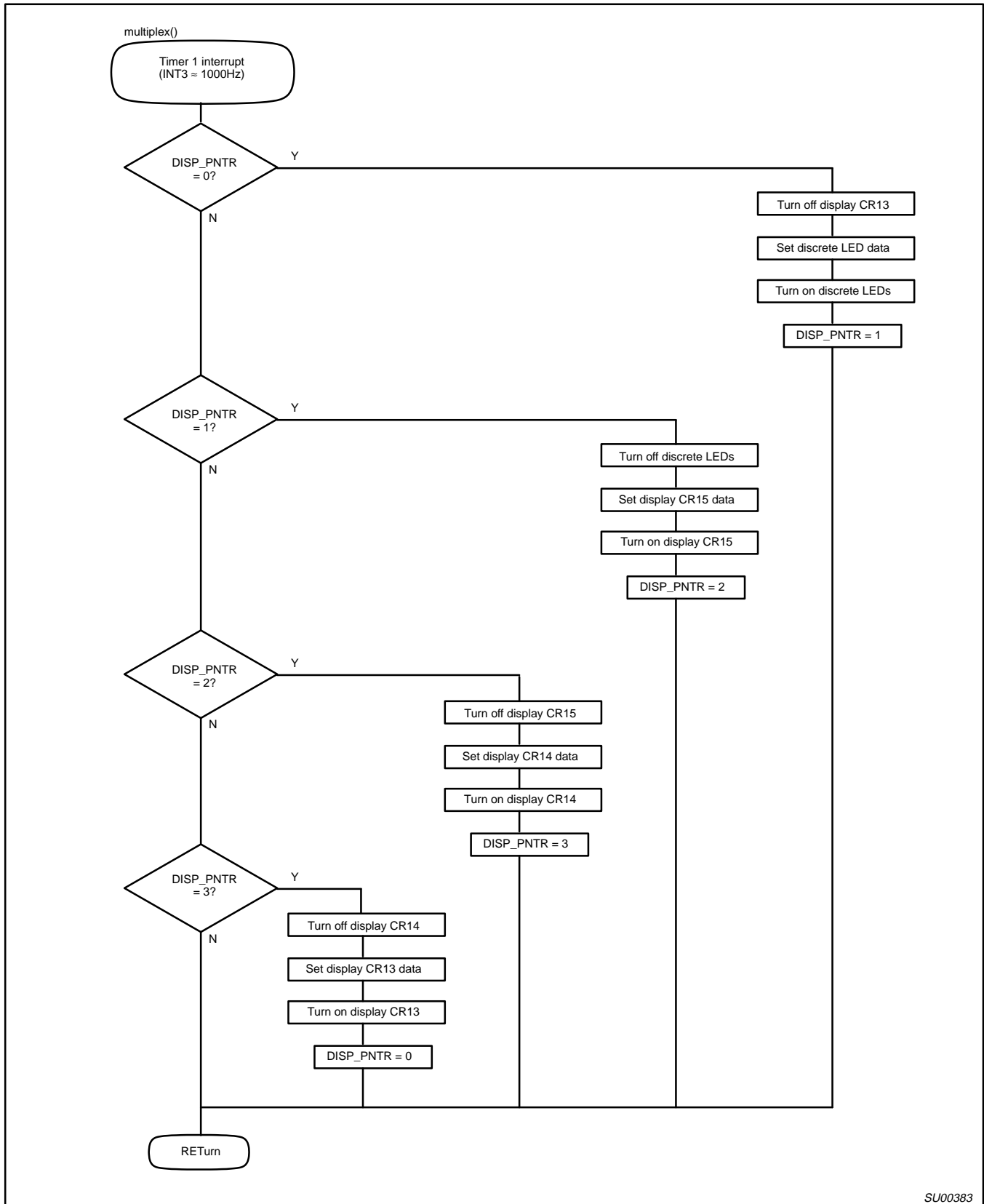


Figure 7. Program Flowchart (Continued)

SU00382

Airflow measurement using the 83/87C752 and "C"

AN429

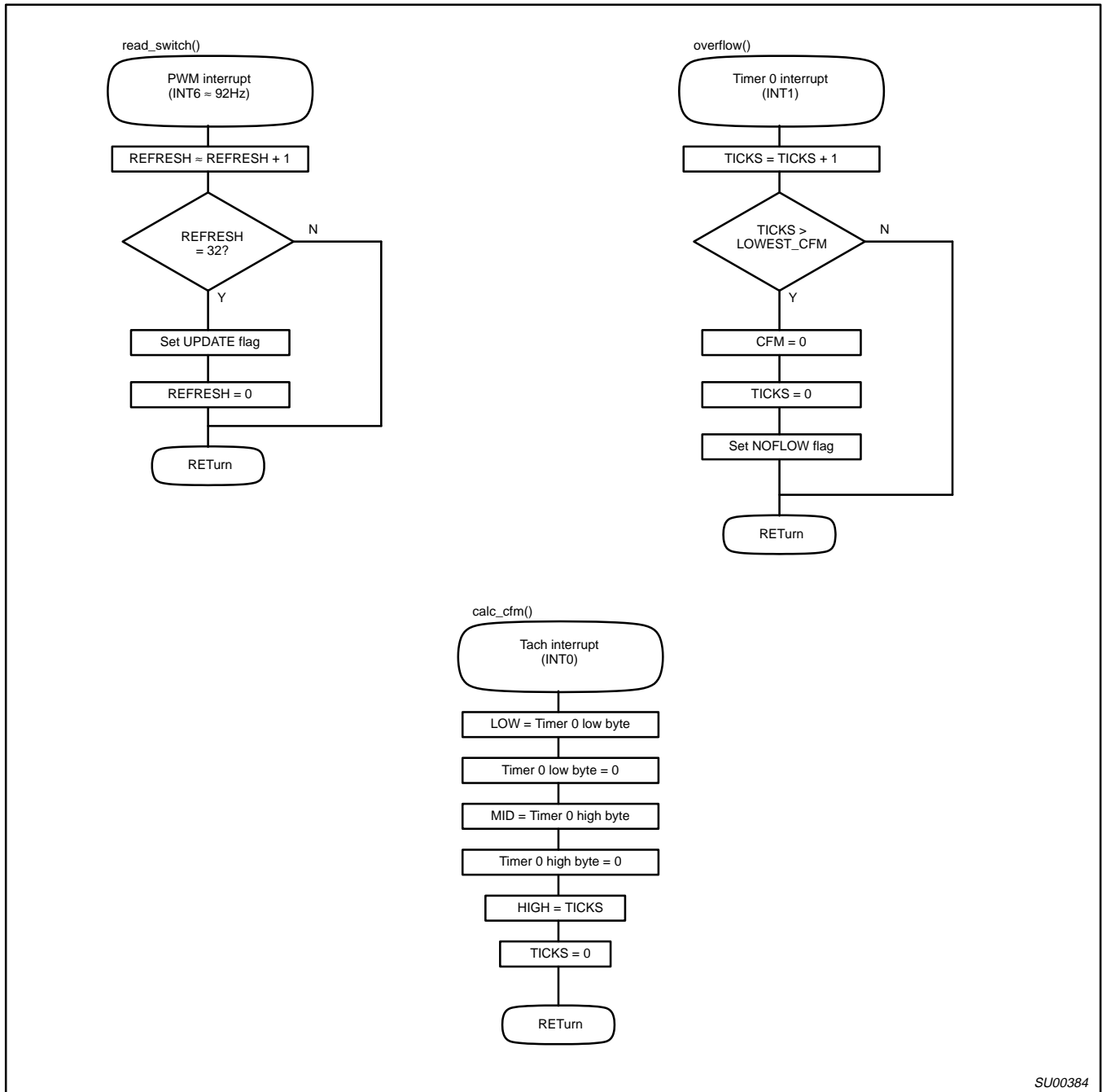


SU00383

Figure 7. Program Flowchart (Continued)

Airflow measurement using the 83/87C752 and "C"

AN429



SU00384

Figure 7. Program Flowchart (Continued)

Airflow measurement using the 83/87C752 and "C"

AN429

```

/*
    this program measures the air flow through a rotary flowmeter
    and displays the calculated cfm. the output of the flowmeter
    tachometer is a small duty cycle pulse train with period
    which is proportional to the flow. the flow is compensated
    for changes in pressure and temperature to maintain
    calibration. if the flow exceeds an adjustable setpoint
    it energizes a 2 form c relay for user application.
*/

/*
    these pragmas specify compiler command line options
*/
#pragma CODE                /* generate code                */
#pragma SYMBOLS              /* and symbols          */
#pragma PL (60)              /* 60 lines per page   */
#pragma PW (120)             /* 120 cols per page   */
#pragma OT (3)               /*                       */
#pragma ROM (SMALL)          /* single-chip mode    */

/*
    include the 8XC752-specific definitions and
    the standard i/o library.
*/
#include <reg752.h>
#include <stdio.h>

/*
    define symbolic names for program constants
*/
#define ZERO_K                2730        /* 0 degrees centigrade in 1/10 kelvin */
#define ONE_TENTH_CFM         4444444L    /* 1/10 cfm in microseconds           */
#define STD_TEMP              2980        /* 25 degrees centigrade in 1/10 kelvin */
#define STD_ATM               147         /* one atmosphere in 1/10 psi          */
#define LOWEST_CFM            0x40        /* maximum period from meter 0x400000 */
#define START_ADC0            0x28        /* commands to start appropriate      */
#define START_ADC1            0x29        /* a/d channel conversion cycle       */
#define START_ADC2            0x2a        /*                                     */
#define START_ADC3            0x2b        /*                                     */
#define START_ADC4            0x2c        /*                                     */
#define ADCI                  0x10        /* a/d converter status flags         */
#define ADCS                  0x08        /*                                     */
#define FREERUN_I             0x10        /*                                     */
#define SEG_A                 0x01        /* P3 position for display segment 'a' */
#define CFM                   0x01        /* P3 position for 'cfm' led           */
#define SEG_B                 0x02        /* P3 position for display segment 'b' */
#define DEGREES               0x02        /* P3 position for 'degrees' led       */
#define SEG_C                 0x04        /* P3 position for display segment 'c' */
#define PSI                   0x04        /* P3 position for 'psi' led           */
#define SEG_D                 0x08        /* P3 position for display segment 'd' */
#define SETPOINT              0x08        /* P3 position for 'setpoint' led      */
#define SEG_E                 0x10        /* P3 position for display segment 'e' */
#define SEG_F                 0x20        /* P3 position for display segment 'f' */
#define SEG_G                 0x40        /* P3 position for display segment 'g' */
#define SEG_DP                0x80        /* P3 position for display decimal pt. */

typedef unsigned char byte; /* byte data type is unsigned 8-bit */
typedef unsigned int word; /* word data type is unsigned 16-bit */
typedef unsigned long l_word; /* l_word data type is unsigned 32-bit */

#define TRUE 1 /* define logical true / false */
#define FALSE 0 /* values for bit variables */

```

Airflow measurement using the 83/87C752 and "C"

AN429

```

/*
    define look-up table of possible seven segment display
    characters. the table consists of 11 elements corresponding
    to the 10 digits ('0'-'9') and error symbol ('E') that can be
    displayed. Each element is defined by ANDing (|) the bit
    mask for each segment (SEG_A - SEG_G) comprising the
    character. the table contents need to be inverted before
    use to be compatible with U2 (udn2585a). for example,
    '~segments[3]' specifies the segment mask to display '3'.
*/
code byte segments [ ] =
{
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F |          /* 0 */
    SEG_B | SEG_C |          /* 1 */
    SEG_A | SEG_B |          /* 2 */
    SEG_A | SEG_B | SEG_C | SEG_D |          /* 3 */
    SEG_A |          SEG_C | SEG_D |          SEG_F | SEG_G | /* 4 */
    SEG_A |          SEG_C | SEG_D |          SEG_E | SEG_F | SEG_G | /* 5 */
    SEG_A |          SEG_C | SEG_D |          SEG_E | SEG_F | SEG_G | /* 6 */
    SEG_A | SEG_B | SEG_C |          /* 7 */
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G | /* 8 */
    SEG_A | SEG_B | SEG_C | SEG_D |          SEG_F | SEG_G | /* 9 */
    SEG_A |          SEG_D | SEG_E | SEG_F | SEG_G |          /* E */
};

/*
    define the '752 special function bits which control i/o lines.
    note that i/o line (and constant) names are capitalized
sbit RELAY      = 0x96; /* active hi to turn on setpoint relay
sbit STROBE_0   = 0x80; /* active hi to enable display status led's
sbit STROBE_1   = 0x81; /* active hi to enable display cr15 (tenths)
sbit STROBE_2   = 0x82; /* active hi to enable display cr14 (ones)
sbit NO_FLOW    = 0x83; /* flag set when no flow detected
sbit STROBE_3   = 0x84; /* active hi to enable display cr13 (tens)
sbit SEL_0      = 0x93; /* active low pushbutton inputs used to
sbit SEL_1      = 0x94; /* select the display mode
sbit INTR       = 0x95; /*
sbit UPDATE     = 0x97; /* flag set when time to update display

/*
    define memory variables. note memory variable names are lower case
data word cfm; /* gas flow in tenths of a cfm
data word setpoint; /* relay setpoint in tenths of a cfm
data word degree_c /* temperature in tenths centigrade
data word l_word corr; /* intermediate calculation value
data word psi; /* pressure in tenths of a psi
data byte display0; /* variables to hold values for the
data byte display1; /* displays during refresh.
data byte display2; /* display0=status LEDs, display1=CR15,
data byte display3; /* display2=CR14, display3=CR13
data byte disp_pntr; /* pointer to next display to enable
data byte refresh; /* counter determines display updates
data byte high; /* bits 16 - 23 of flow period
data byte middle; /* bits 8 - 15 of flow period
data byte low; /* bits 0 - 7 of flow period
data byte ticks; /* incremented by timer overflow

/*
    the program consists of four interrupt handlers (multiplex,
    read_switch, overflow, calc_cfm) and a main program.
    multiplex - refresh the seven-segment and discrete status LEDs
    read_switch - signal periodic pushbutton sampling and display update
    overflow - accumulate high order bits of time between tach pulses
    calc_cfm - accumulate low order bits of time between tach pulses
    main - calc airflow, control relay, sample pushbuttons, update display

/*
/*
    multiplex -
    use the free-running I timer to multiplex the seven-segment and
    discrete leds at approx. 1000 hz.
*/

```

Airflow measurement using the 83/87C752 and "C"

AN429

```

void multiplex () interrupt 3
{
    switch (disp_ptr)
    {
        case 0x00:
            STROBE_3 = FALSE; /* turn off display cr13 */
            P3 = 0xff; /* turn off all segments */
            P3 = display0; /* load segments for led's */
            STROBE_0 = TRUE; /* turn on status led's */
            disp_ptr = 1; /* increment ptr to display */
            break;
        case 0x01:
            STROBE_0 = FALSE; /* turn off status led's */
            P3 = 0xff; /* turn off all segments */
            P3 = display1; /* load segments for tenths */
            STROBE_1 = TRUE; /* turn on display cr15 */
            disp_ptr = 2; /* increment ptr to display */
            break;
        case 0x02:
            STROBE_1 = FALSE; /* turn off display cr15 */
            P3 = 0xff; /* turn off all segments */
            P3 = display2; /* load segments for units */
            STROBE_2 = TRUE; /* turn on display cr14 */
            disp_ptr = 3; /* increment ptr to display */
            break;
        case 0x03:
            STROBE_2 = FALSE; /* turn off display cr14 */
            P3 = 0xff; /* turn off all segments */
            P3 = display3; /* load segments for tens */
            STROBE_3 = TRUE; /* turn on display cr13 */
            disp_ptr = 0; /* increment ptr to display */
    }
}

/*
    read_switch -
    use the free running pwm prescaler to generate
    interrupts at 92 hz. every 32nd interrupt set
    the UPDATE flag which causes main () to sample
    the pushbuttons and update the led displays.
*/

void read_switch () interrupt 6
{
    if (refresh++ == 32)
    {
        UPDATE = TRUE;
        refresh = 0;
    }
}

/*
    overflow -
    whenever time0 overflows (from 0xffff to 0x0000)
    increment the variable 'ticks' which accumulates the
    highest order (16 - 23) bits of the gas flow period
    in microseconds. if the variable 'ticks' is greater
    than the period corresponding to a flow of < 0.1 cfm
    then set the NO_FLOW flag which causes main () to
    display '00.0'
*/

void overflow () interrupt 1
{
    if (++ticks > LOWEST_CFM)
    {
        cfm = 0;
        ticks = 0;
        NO_FLOW = TRUE;
    }
}

```

Airflow measurement using the 83/87C752 and "C"

AN429

```

/*
    calc_cfm -
    an external interrupt (int0) generated by a tach
    pulse from the flowmeter transfers the current value
    of timer0 into variables 'low' and 'middle', and then
    resets the timers. the 'ticks' variable described
    above is also copied to variable 'high', and then
    reset to zero. the NO_FLOW flag is cleared to
    enable display by main () of the calculated cfm.
*/

void calc_cfm () interrupt 0
{
    low = TL0;
    TL0 = 0;
    middle = TH0;
    TH0 = 0;
    high = ticks;
    ticks = 0;
    NO_FLOW = FALSE;
}

/*
    main -
    after initializing pins and variables, enter a continuous loop to...
    - calculate the airflow based on the tach, temp and pressure inputs.
    - compare the airflow to the setpoint input, and control the relay.
    - if the UPDATE flag is set (by the read_switch interrupt handler),
      sample the pushbuttons and update the display data.
*/

void main ()
{
    RELAY      = 0;          /* initialize output pins          */
    INTR       = 1;
    UPDATE     = 1;
    STROBE_0   = 0;
    STROBE_1   = 0;
    STROBE_2   = 0;
    STROBE_3   = 0;
    NO_FLOW    = 0;
    I2CFG      = FREERUN_I; /* enable I timer to run, no i2c   */
    RTL        = 0;        /* timer 0 period 0x10000 u_seconds */
    RTH        = 0;
    PWMP       = 255;      /* pwm timer interrupt at 923 hz    */
    TR         = 1;        /* enable timer 0                   */
    IT0        = 1;        /* INT0 is edge active              */
    ticks      = 0;        /* initialize variables              */
    cfm        = 0;
    low        = 0;
    middle     = 0;
    high       = 0;
    degree_c   = 250;      /* 25.0 tenths degrees c           */
    psi        = 147;      /* 14.7 tenths psi                  */
    corr       = 0;
    refresh    = 0;
    disp_pntr  = 0;
    IE         = 0xab;     /* enable interrupts                */

    /*
    main execution loop, executes forever.
    */

    while(1)
    {

```


Airflow measurement using the 83/87C752 and "C"

AN429

```

*/
    calculate base cfm rate - first create long word representing
    flow rate period in microseconds. then subtract the time
    overhead in servicing the routine 'calc_cfm'. then divide the
    period into the period for 1/10 cfm, to get flow rate in 1/10
    cfm resolution.

*/
    corr = high * 0x10000L;
    corr += (middle * 0x100L);
    corr += low;
    corr -= CORRECTION;
    corr = ONE_TENTH_CFM / corr;

/*
    read temperature - measure output from the LM35 sensor,
    scaled by the AMP-02. the scaling results in a range
    of 0 to 51.0 degrees centigrade, in 0.2 degree steps.

*/
    ADCON = START_ADC1;
    while (ADCON & ADCS) ;
    degree_c = ADAT;
    degree_c *= 2;

*/
    compensate cfm rate for temperature - convert temperature
    into degrees kelvin, then divide it into the measured flow
    rate multiplied by the calibration temperature of the flow-
    meter in degrees kelvin. (nominal 25 degrees centigrade)

*/
    corr *= STD_TEMP;
    corr /= (ZERO_K + degree_c);

*/
    read pressure - measure output of the KP100A pressure trans-
    ducer, scaled by the AMP_02. the scaling results in a range
    of 0 to 25.5 psi, in 1/10 psi steps.

*/
    ADCON = START_ADC0;
    while (ADCON & ADCS) ;
    psi = ADAT;

*/
    compensate cfm rate for pressure - multiply measured pres-
    sure and the calculated flow rate, and then divide it by
    the standard atmospheric pressure at sea-level. (nominal
    14.7 psi)

    corr *= psi;
    corr /= STD_ATM;
    cfm = corr;

*/
    read setpoint pot to obtain setpoint in the range of
    0 - 25.5 cfm in 1/10 cfm steps.

*/
    ADCON = START_ADC2;
    while (ADCON & ADCS) ;
    setpoint = ADAT;

*/

```

Airflow measurement using the 83/87C752 and "C"

AN429

```

test if cfm rate greater or equal to the
setpoint, and if so then energize relay
*/

    if (setpoint > cfm)
        RELAY = 0;
    else
        RELAY = 1;

*/

test if UPDATE flag has been set, and if so reset flag.
*/

    if (UPDATE)
    {
        UPDATE = 0;
    }

*/

then test is the NO_FLOW flag has been set. if so then
display '00.0' cfm
*/

    if (NO_FLOW)
    {
        display0 = ~CFM;
        display1 = ~segments[0];
        display2 = ~(segments[0] | SEG_DP);
        display3 = ~segments[0];
    }

*/

if the NO_FLOW flag was not set then read the display
select pushbuttons, and display the appropriate data.
*/

    else if (SEL_0)
    {
        if (SEL_1)
        {

*/

if no pushbutton is depressed then the default display is
the flow rate in cfm. if the flowrate is greater than
or equal to 30 cfm then display the overrange message
'EEE', otherwise display the flow in 'XX.X' format.
*/

                                if (cfm <= 300)
                                {
                                    display0 = ~CFM;
                                    display1 = ~segments[cfm % 10];
                                    cfm /= 10;
                                    display2 = !(segments[cfm % 10]);
                                    cfm /= 10;
                                    display3 = ~segments [cfm % 10];
                                }

                                else
                                {
                                    display0 = ~CFM;
                                    display1 = ~segments[10];
                                    display2 = ~segments[10];
                                    display3 = ~segments[10];
                                }
                            }

```

Airflow measurement using the 83/87C752 and “C”

AN429

```
*/
    if the temp pushbutton (SW1) is pressed then display the air temperature.
*/

        else
        {
            display0 = ~DEGREES;
            display1 = ~segments[degree_c % 10];
            degree_c /= 10;
            display2 = ~(segments[degree_c % 10] | SEG_DP);
            degree_c /= 10;
            display3 = ~segments[degree_c % 10];
        }
    }
}
else
{

*/
    if the psi pushbutton (SW2) is pressed then display the air pressure.
*/

        if(SEL_1)
        {
            display0 = ~PSI;
            display1 = ~segments[psi % 10];
            psi /= 10;
            display2 = ~(segments[psi % 10] | SEG_DP) ;
            psi /= 10;
            display3 = ~segments[psi % 10] ;
        }

*/
    if the setpoint pushbutton (SW3) is pressed then display the setpoint.
*/

        else
        {
            display0 = ~SETPOINT;
            display1 = ~segments[setpoint % 10] ;
            setpoint /= 10;
            display2 = ~(segments[setpoint % 10] | SEG_DP ;
            setpoint /= 10;
            display3 = ~segments[setpoint % 10] ;
        }
    }
}
}
}
```