Application Development with Microcontrollers

Dinesh Sharma
Joseph John
P.C. Pandey
Kushal Tuckley

Department of Electrical Engineering
Indian Institute of Technology, Bombay

September 10, 2023

Structure of a Traditional Program

- Traditional computer programs start with some initialization code which gives starting values to variables and configures devices as desired.
- The next task is computation of desired values. The program asks for inputs as and when it needs them and may read values from external devices which are connected to the system.
- When outputs are ready, it provides these in the form of displays, prints, plots or outputs to external devices.
- The circuit simulation program “ngspice” is an example of such a program.
- The computation algorithm decides when inputs will be taken, when output will be provided etc.
- When all the outputs have been provided, the program terminates.

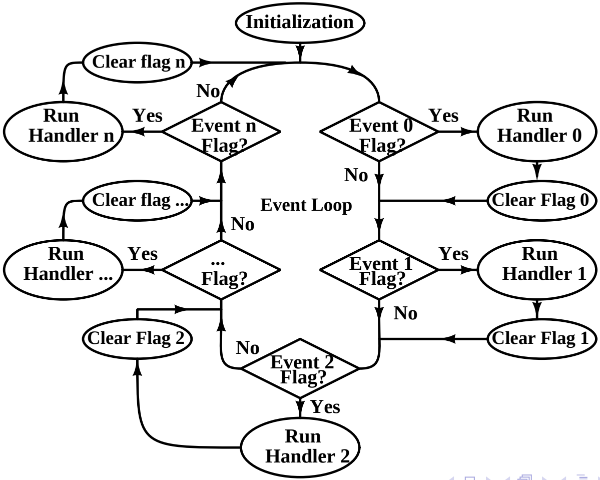
Structure of an Event Driven Program

The structure of an event driven microcomputer program is quite different.

- Microcomputers are typically used in embedded systems.
- These are applications where the presence of a microprocessor or a program is not apparent to the user.
  For example, a digital camera has a microcomputer which is running a program, but the user is not aware of this. The interface presented to the user is that of a camera.
- These applications are event driven. External events decide which portion of the program will run at any time.
- These programs never terminate ...
  What would happen to a camera if its program terminated?
- So the basic program structure is that of an initialization phase followed by an endless loop.

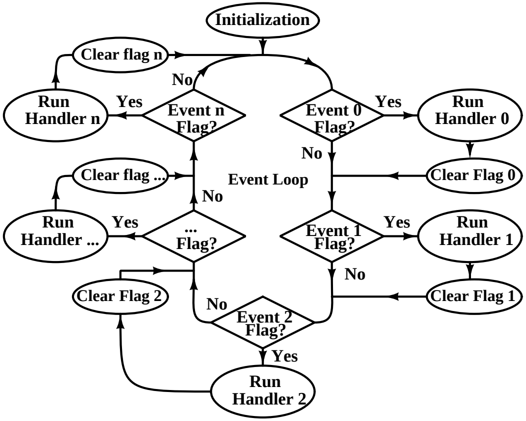
Structure of an Event Driven Program

An event-driven program enters an endless loop after the initialization phase.



Structure of an Event Driven Program

The program is aware of multiple events which might occur in the external world. It includes software (called an event handler) for every such event. This software should run whenever the event occurs.

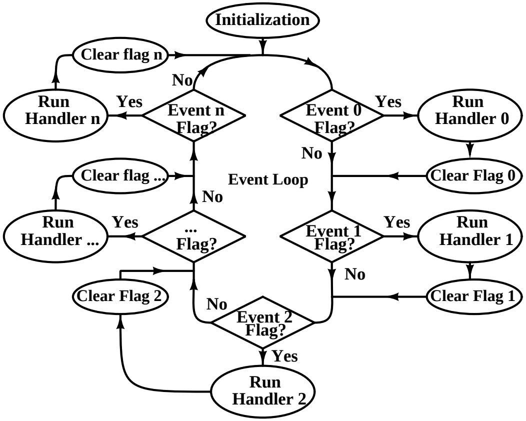


Each event is designed to set a “Flag” when it occurs.

(A flag is just a bit or a variable which is given a recognizable value when the event occurs).

The program runs in an endless loop, checking if the flag for any event is set.

Structure of an Event Driven Program



- If any event flag is set, it runs the corresponding handler, clears the flag and re-joins the loop.
- If the flag is not set, it just goes and examines the next event flag and so on . . . in an endless loop.

A program with such a structure is said to be **event driven.**

Event Driven Programs with Arduino

- The software structure in the Arduino Integrated Development Environment (IDE) is optimized for event driven programs.
- The in-built main program (in C or C++) first calls a user supplied function called **setup( )**.
- All initialization code is to be put in this function by the user.
- The library included in the IDE provides many functions which make the user's task easy.

  For example, to make the digital pin 13 an output pin, one can just call the library function **pinMode** as:

  pinMode(13, OUTPUT);

  And to set up the serial communication to use 9600 baud as the data rate, one can call:

  Serial.begin(9600)

Digital and Analog I-O

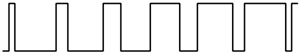
- After the user supplied function “**setup**” returns, the main program enters an endless loop.
- In each iteration of the loop, it calls another user supplied function called **loop( )**.
- The user can place all operational details relevant to continuous running of the application in this function.
- The library included in the IDE provides many functions which make it easy to input and output digital values.
- The micro-controller used in Arduino (AVR 328P) has an in-built 10 bit ADC with 6 input channels which can be used to input analog voltages. Library functions included in the IDE enable one to convert the analog input at any of the 6 channels to a 10 bit digital word.

Digital and Analog I-O

- The function **analogRead** provides the capability of reading the analog voltage on any of the analog input pins and converting it to a 10 bit digital value.
- **analogRead** accepts a channel number as its input and returns a 10 bit integer – which is the converted value from the ADC.
  For example,
  int sensorValue = analogRead(A0);
  declares the variable sensorValue to be a (16 bit) integer and places the 10 bit ADC value corresponding to the analog input on channel A0 in it.
- Channels A0 to A5 are available for ADC input.
- The library also has a mapping function which can map this 10 bit range onto a given range – say 0 to 100.

Digital and Analog I-O

AVR 328P does not have a built-in D to A converter. However, it can output a **Pulse Width Modulated (PWM)** digital stream on selected pins.



In Pulse Width Modulation, the ratio of durations of ON and OFF periods is varied.

- The ratio of ON time to total cycle time (ON duration + OFF duration) is called **Duty Cycle**.
- If this output is averaged, it will produce an analog value proportional to the duty cycle.
- Arduino IDE library provides functions for PWM output with the desired duty cycle.
- Explicit averaging may not be required for relatively slow devices such as motors. Also, human responses are slow enough so that the intensity of an LED driven by a PWM waveform will appear to be proportional to the duty cycle.

Digital and Analog I-O

Library function **analogWrite** permits writing PWM output to selected pins.

The function is somewhat inaccurately named – since the output is not really analog but a Pulse Width Modulated digital stream.

- The function takes the pin number and the duty cycle as its arguments. The duty cycle is provided as an integer between 0 and 255. (The actual duty cycle is the provided number/255).
- For example, **analogWrite(3, 128)** will output a PWM waveform on pin 3 with approximately equal ON and OFF times.
- A low pass filter can be connected to the pin to provide explicit averaging if required.
- Of course an external D to A converter can always be used if more accurate analog outputs are required.

Manifest Constants defined in Arduino Library

The Arduino library defines several symbolic names for constants. These are known as **manifest** constants (because they make the function of these constants clear).

Frequently used manifest constants are:

HIGH | **LOW :** represent the digital state of a pin.

INPUT | **OUTPUT** | **INPUT_PULLUP:** represent the mode in which a pin is to be used.
(**INPUT_PULLUP** is an input for which an internal pull up is enabled. This permits driving it with open collector outputs.)

LED_BUILTIN: is the pin number to which the on-board LED is connected. For Uno and Nano boards, this is pin number 13.

true | **false:** represent the logic value.

Library functions in Arduino Library: Digital I-O

The Arduino library provides many functions which permit one to use the functionality provided by the micro-controller.
We have seen many of these already.

Digital I/O: **digitalRead(pin);** returns a **HIGH** or **LOW** value depending on the voltage level on the pin.

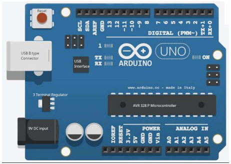
digitalWrite(pin, value); writes **HIGH** or **LOW** to a pin.

pinMode(type); sets the direction for I-O on a digital pin. The type can be **INPUT**, **INPUT_PULLUP** or **OUTPUT**.

If the type is **INPUT_PULLUP**, an internal pull up circuit is attached to the input pin. This is useful when the external driver is of open collector or open drain type.

Arduino Library Functions for Analog I-O

int analogRead(pin); returns the 10 bit ADC value corresponding to the analog voltage on the given pin.
void analogWrite(pin, PWMvalue); will output a PWM waveform on the given pin with the specified duty cycle (range 0 to 255).
void analogReference (type); configures the reference voltage used for A to D conversion. The argument “type” can be:



DEFAULT – to use the default analog reference of 5 volts (on 5V Arduino boards) or
INTERNAL – to use a built-in reference equal to 1.1 volts (= silicon band gap), or
EXTERNAL – to use the voltage applied to the AREF pin (0 to 5V only) as the reference.

This function must be called ***before*** analogRead to avoid shorting internal and external references.

Arduino Library Functions for Timing and Delay

Timing: **void delay(d_milli);** delays the execution of the following statements of the program by the given number of milliseconds.

void delayMicroseconds(d_micro) is similar to delay, but the argument is interpreted as microseconds.

int millis( ); returns the number of milliseconds which have elapsed since the start of the program.

int micros( ); returns the number of microseconds which have elapsed since the start of the program.

Some of these functions use the internal timers and interrupts for their operation. Others use software timing. Use of interrupts in the user program can interfere with their functioning.

Library functions in Arduino Library

Other I/O: **void tone(pin, frequency, [duration]);** A call to this function generates a square wave on the given pin with the specified frequency for the given duration. The third argument is optional. If duration is not given, the output will continue till a call is made to the function:
void noTone(pin);

Only one frequency can be generated at a time. Multiple frequencies cannot be generated on different pins using this function.

Minimum frequency which can be generated is 31 Hz. (This limitation comes from the maximum divider value which can be loaded in the timer/counter chip).

This function should not be used along with PWM output, because both use the same resources.

Interrupting the processor in Arduino

Interrupts in Arduino An interrupt stops the main program in order to run a specified function. The main program resumes from where it was stopped when the specified function returns.

A few library functions are used for managing interrupts. **void noInterrupts( )** disables interrupts from occurring, while **void interrupts( )** enables interrupts.

On Uno and Nano cards, external devices can interrupt the running program by sending a pulse on digital pin 2 or on digital pin 3.

Internal peripherals such as timers etc. also use interrupts.

Therefore care has to be taken while disabling interrupts – it may interfere with functions like serial communication.

Interrupting the processor in Arduino

Attaching an interrupt: Function **void attachInterrupt( )** is used for causing an interrupt when a signal of a specified type is seen on digital pins 2 or 3 in an UNO or Nano card.

The function takes 3 arguments. The first argument is the interrupt number, the second is the name of the function to be run when an interrupt occurs and the third argument specifies the kind of signal on the interrupt pin which will result in an interrupt.

The interrupt number should not be given directly, but should be determined from a call to the function **digitalPinToInterrupt(pin)**.

The returned value from this function should be given as the first argument to **attachInterrupt**.

Interrupting the processor in Arduino: function arguments

Attaching an interrupt: contd. The second argument is the name of the user supplied function which should be run after stopping the main program.

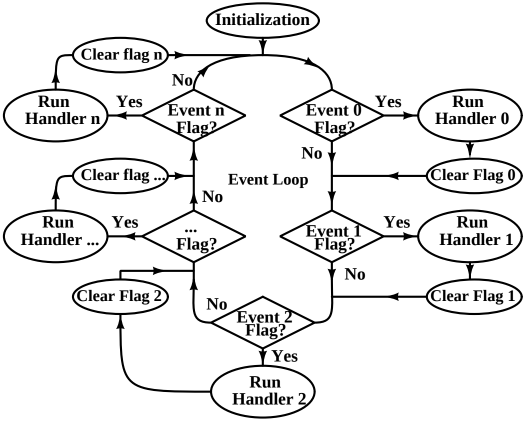
The last argument specifies the type of input on pin 2 or 3 which will result in an interrupt. This argument can be one of pre-defined manifest constants **LOW, CHANGE, RISING** or **FALLING**.

LOW interrupts when the signal is ‘0’, **RISING** interrupts when the signal changes from ‘0’ to ‘1’, **FALLING** interrupts on a ‘1’ to ‘0’ transition and **CHANGE** interrupts when either of these transitions occurs.

Once the interrupt is recognized, further interrupts are disabled till the specified function has run.

Interrupting the processor in Arduino: Handler function

The function run after interrupting the main program should be short and the processor should remain in interrupted state for as small a time as possible.

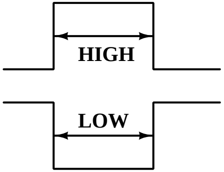


Rather than running a detailed computation as the interrupt function, one uses the interrupt function to just set an event flag.

The handling of the event then occurs in the main program in the event loop as outlined in the beginning of this lecture.

Pulse width Measurement

Pulse width measurement



pulseIn( );, **pulseInLong( );** – these functions are used for measuring the width of a high or a low pulse. These take two arguments - the pin number where the pulse will be applied and whether the HIGH of LOW duration is to be measured.

The function **pulseIn** is suitable for use with interrupts disabled. (The software timing used by this function stops during interrupts and so the returned value will be inaccurate if interrupts occur).

The function **pulseInLong( )** uses hardware timing with interrupts. So the interrupts must be enabled when this function is called.

Pulse width Measurement

pulseIn( ) is more useful for short pulses since we can afford to disallow interrupts for a short time.

As its name implies, **pulseInLong( )** is more suitable for long pulses. However, it may interfere with serial communications and delay functions.

The syntax for calling these functions is:
pulseIn(pin, HIGH); or **pulseIn(pin, LOW);** and **pulseInLong(pin, HIGH);** or **pulseInLong(pin, LOW);**.

pin is the pin number on which the pulse will arrive. The second argument decides whether the width of the **HIGH** part of the pulse will be measured or the **LOW** part.
These functions have an optional third argument which specifies the maximum time for which the function will wait for the pulse to end.
If not provided, this time is taken to be 1 second.

Other Libraries

- We have taken a brief tour of many of the functions provided by the built-in library of Arduino IDE.
- There are hardware cards which plug directly into the connectors of Arduino Uno. These are designed for specific functions – such as driving stepper motors, speed control for battery operated (BO) motors, communicating using bluetooth etc.
  These cards are called **shields** (because of the way they sit on the Arduino card).
- Libraries to support these shields are available. One can download these libraries and include them in the IDE.
- Since these libraries are mostly developed by hobbyists and placed in the public domain, documentation may be vague or non-existent.
  However, the source code for functions included in these libraries is available and one can figure out how to use these functions for projects.

Application Development with Arduino

Application development using Arduino boards involves:

1. Choice of appropriate additional hardware,
2. Development of required algorithms
3. Implementation of algorithms in software

Many of the hardware techniques such as negative feedback, hysteresis using Schmitt triggers etc. have their counterparts in software.

We'll review some of these briefly.

Application Development with Arduino

To illustrate some of the algorithms used for application development, we'll use temperature control of a water bath as an example.

We would like to keep the temperature of water in the bath as close as possible to a given temperature (called **set point**) which is higher than the room temperature (no cooling required) and which should be programmable.
Let us first look at the hardware required for this:

- We obviously need a heater. To control the temperature, we'll require some means of controlling power to the heater. This can be done through
  1. ON/OFF control through a relay, or
  2. Pulse Width control through thyristors driven from Arduino, or
  3. Voltage control through a programmable power source to the heater.
- Accordingly, we shall need drivers for relays/thyristors/DC source which will be driven using digital signals from Arduino.

Application Development: ON/OFF control

We also require hardware for measuring the actual temperature (using LM35 temperature sensor or a thermocouple), and means for adjusting the set point (through a potentiometer or dialing it in through a keyboard).

Let us first consider the simplest option – that of ON/OFF control.

- We can measure the temperature at regular intervals and compare it with the set point. If the actual temperature is higher than the set point, we turn the relay OFF, if it is lower, we turn it ON.
- When the temperature is close to the set point, a small amount of heating takes it above the set point which turns the relay OFF. However, then the temperature quickly drops below the set point, which turns the relay ON. This causes the relay to “chatter”.

How to stop the relay from chattering due to this frequent switching?

Application Development: ON/OFF control

To prevent the relay from chattering, we can use a small amount of hysteresis in the decision to turn the relay ON or OFF.

- Instead of comparing the temperature with a single set point, we use a “high limit” and a “low limit” on either side of the set point.
- We turn the relay ON only if the temperature is below the “low limit”. We turn if OFF only if the temperature is above the “high limit”.
- We now have a trade off – if the high and low limit are too close to the set point, the relay will turn ON and OFF fequently.
- If these limits are set far from the set point, the temperature will ramp between these two limits and the worst case error between the actual temperature and the set point will be high.

You would have noticed this kind of control in many electric irons.

Application Development: Proportional Control

A smoother way to control the temperature of the bath would be to apply power to the heater proportional to the error in temperature –

- The farther away we are below the set point, higher is the power applied.
- However as we approach the set point, the error becomes less and lower power will be applied.
- If we are at or above the set point, no power will be applied.

This results in more accurate control of temperature.

However, the power applied when we reach the set point is zero!
So the bath will always start cooling down due to heat losses as soon as it reaches the set point.

Proportionate control will always settle at non-zero error!
(A work around is to calculate the error from a point just above the set point, so that the power is non zero at the set point).

Application Development: Integral Control

We don't want to reduce the power to zero when we reach the set point. We want the heater to apply *constant* power, which keeps the bath at the set point.

- What operation gives a constant result when its argument reaches zero? The integral, of course!
- What we should do is to integrate the error and apply power proportional to this integral. Now when the error becomes zero, a constant power will be maintained which will keep the bath at the set temperature.
- If we overshoot the set point, error will become negative and the integral will reduce in value. With lower applied power, we'll come back to the set point and keep the power at this lower value.
- Since the proportional term is zero at the set point any way, we can use a value for applied power which is the weighted sum of proportional term and integral term. This is known as Proportional-Integral or P-I control.

Application Development: PI Control

The integration of error should be taken over how much time?

- Temperature error which was there long ago may be less relevant for determining the power to be applied at the current instant.
- Therefore we integrate the error over the recent history. Contribution due to error from long ago needs to be dropped out from the integral. This is known as the **reset rate**.
- Proportionality constants for the P and I contributions, as well as the reset rate have to be tuned for the specific system being controlled.

PI control works reasonably well for keeping the bath temperature constant. However, it is slow to react to sudden changes in the error. To take care of this, we add a third term – the differential term.

Application Development: PID Control

- PI control is very slow to react to sudden changes in error.
- For example, if we change the set point, suddenly there is a large error from the current temperature. PI control will be sluggish to react to it.
- To take care of sudden changes in the error, we need a term which will be proportional to the *rate of change* of the error. This can be provided by a term proportional to the differential of the error.
- When we include the differential term, the control strategy is called Proportional-Integral-Differential – or PID control.

PID control is widely used for controlling various process parameters.

How do we implement it in a micro-controller based system?

Implementing PID Control

- We measure the temperature at regular intervals, compute the error and store these values in an array.
- Every time we compute the current value of error, we add it to a moving sum and subtract the error which had been added “n” steps before. (“n” is the reset rate).
- We also compute the difference between the current and previous errors and use it as the differential term.
- We compute a weighted sum of the current error, current value of the moving sum and the differential term by multiplying each of these with their proportionality constants and adding them.
- We output this weighted sum to the external hardware which will apply power proportional to this value.

This algorithm is not restricted to temperature control. It can be applied to any parameter which has to be kept at a set value.
For example, it could be used for a line follower robot where we control the orientation to keep the robot centered on the line.

That is all folks!