13th International Research/Expert Conference
”Trends in the Development of Machinery and Associated Technology”
TMT 2009, Hammamet, Tunisia, 16-21 October 2009

MICROCONTROLLER BASED METHODOLOGY FOR PART TRACKING IN INDUSTRIAL AUTOMATION

Coskun Kazma[1], Fatih Ileri[2], H. Fatih Ugurdag[1]
[1]Computer Eng. Dept., [2]Electrical Eng. Dept., Bahcesehir University, Istanbul, Turkey

ABSTRACT

The goal of this work was to design a microcontroller based sorting system. Besides solving the particular problem, a new methodology has also been devised, which simplifies automation tasks. This methodology requires a separate algorithm thread for each station. Subsequent threads are tied to each other through a FIFO. As a byproduct, a generic class library have been developed.
Keywords: class library, industrial automation, infrared sensor, microcontroller, PIC, sorting.

1. INTRODUCTION

Automation systems that perform sorting [1] are widely used. Shipping, automotive, food processing industries require such sorting. Mostly, PLCs are used to control such systems. However, we have used a PIC microcontroller (MCU) instead of a PLC in our sorting system. Use of an MCU allowed us to write automation algorithms using a high-level programming language. Changes in the high-level specification of a system can be finalized much faster if programming is done using high-level abstraction. Use of a PIC instead of PLC introduces cost savings in some automation problems. (Note that in some costly PLC systems, it is possible to program in high-level languages besides ladder diagrams [2].) Development of generic class libraries as well as use of a separate thread for each station have introduced significant advantages. MCU algorithms of stations run in parallel and communicate through FIFOs. Besides, class libraries here can be used with any number of stations.

2. OUR AUTOMATION PROBLEM

Our initial goal was to put together a prototype machine vision system for quality control on a conveyor [3]. It was later decided to focus on assembly line control and monitoring aspects and a simple mechanism based on infrared sensors was put in place of a vision system. Although the system now functions as a quality control system, it will be later used as the first stage of a sorting system. The objective of our conveyor system (Fig. 1) is to discard parts less than a certain height out of the conveyor by a reject arm. Infrared detectors are used to sense height (Sharp 2D120XF74). They transmit infrared light, then sense reflected light, and convert the intensity of received light to voltage. The voltage output of the sensor is very low when there is no object. The PIC MCU samples in the sensor's voltage output, then compares it to a threshold, and in effect serves as a presence sensor.
Station A in Fig. 1 is composed of 2 sensors, and associated logic senses the height of incoming part. Sensor 2 (the lower sensor of Station A) flags a new part. If Sensor 1 triggers together with Sensor 2, it is understood that the part has sufficient height. If Sensor 1 does not trigger, it then means that the part is short.
When a short part reaches Sensor 3 (Station B), the reject arm next to Station C needs to open. When the same part arrives at Station C, the reject arm then has to close and push the part out of the conveyor line. The reason why we use this two-stage motion is that the reject arm motor is relatively slow compared to the speed the conveyor belt moves.
Since parts are categorized (with respected to their height) at a single point (Station A), part tracking is particularly crucial. Basic idea in part tracking is as follows. The part that arrives first/ nth at Station A is also the first/ nth part to arrive at Station B as well as Station C. Flow of data is required between algorithm threads of stations so that the stations after Station A can pull the short/tall information from

a parts list created by Station A. Another point is that algorithms of the stations must run in parallel. That is because by the time a part that goes through Station A reaches the reject arm, new parts may arrive at Station A or may even go beyond Station A. If that is the case, the algorithms of stations B and C should not keep the MCU busy to an extent where it does not service Station A frequent enough. If that is not possible, algorithms should be preemptable.

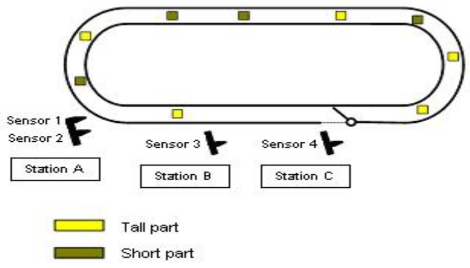


Figure 1. Our conveyor system.

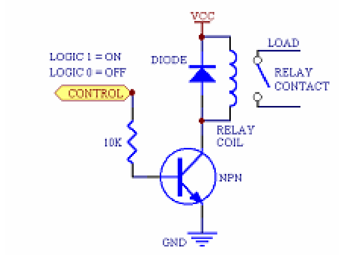


Figure 2. The relay driver circuit.

The MCU on the system control board is a PIC 16F877A. The PIC receives an input signal from the sensors through its analog ports and drives the reject arm through a driver circuit (Fig. 2), which it controls through its digital output pins. PIC 16F877A MCU produces a 5V and low current output signal (TTL level) at its digital I/O pins. The MCU triggers a relay to provide 12V to the reject arm's motor (retrofitted from a windshield wiper) for it to open. It then triggers another relay by outputting a logic 1 on a different port. This second relay feeds -12V to the reject arm's motor, which moves it in the opposite direction and closes it.

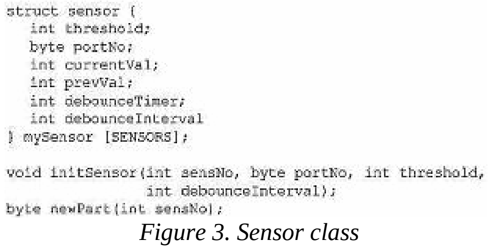
3. METHODOLOGY

Methodology has 2 parts: (i) Concurrent execution of algorithms of different stations. That is why we call them algorithm “threads.” (ii) Usage of our specifically developed generic library classes. Sensors are placed at 3 different locations on the conveyor (see Fig. 1). Each location is a separate station. The main program's task is to run station programs concurrently – without one blocking another.
Implementing such system with an FPGA is easy [4 because approach used in FPGA design is based on integration of modules running concurrently. Such designs are done by using hardware description languages such as Verilog and VHDL. An algorithm thread in our application would translate to an “always” block in Verilog and “process” block in VHDL. Nevertheless, FPGA costs more as a component as well as its board design and manufacture compared to PIC based solutions. Also, it’s harder to find engineers who can design using Verilog/VHDL. Another advantage of PIC is its embedded analog to digital converters, which allow them to be easily coupled with analog sensors.
To run multiple concurrent processes on a processor in a non-blocking fashion, we need an OS with preemptive multi-tasking. That requires the use of a microprocessor rather than an MCU plus external RAM and flash memory, combination of which costs a whole lot more money than a PIC MCU based board. Another alternative to using a full-scale OS on a microprocessor is to write our own simple timer-interrupt driven OS (i.e., firmware).
The question at this point is: “Is there a simple way to concurrently (parallel and non-blocking) run multiple stations' algorithm threads on an MCU without a firmware or OS?” Each station's algorithm requires a separate infinite loop since each station does its assigned job over and over again. These loops cannot be placed in the main program one after another because in that case only the first infinite loop executes and the CPU never gets to the other loops and rest of the program. A simple solution can be summarized as follows. All stations' loop bodies are put in a single loop one after another. That is, we put in a single loop first the loop body of Station A, then loop body of Station B, and so on. However, if one station's code takes too long to complete, then other stations will go blind (i.e., stop sampling their respective sensors), or in other words, they will be blocked. This is possible especially when a station has to wait for a certain amount of time between subtasks.
In most C code development platforms/libraries, such wait functions are implemented by *sleep(wait_time) or delay(wait_time)* functions. If there's no underlying OS, these functions are implemented by making the CPU idle (NOP instruction). In our case, we used a function called

delay_ms(wait_time_in_ms) for stations B and C to drive the reject arm in forward and reverse directions for predetermined amounts of time. To address this problem, we have used an approach that is one of the basic principles of digital logic design, which is called register transfer level (RTL) design. In this approach, the designer takes a cross-section of the whole program in a single clock cycle. In the hardware design world, a clock period is defined as the time interval between two consecutive rising edges of a clock signal. In Our application, a clock cycle can be thought of as the time period between two consecutive sampling points of a sensor.
Our proposed method is still about putting in a single loop the codes for stations A, B and C one after another. However, the code for each station is not anymore the loop body of the code we would have normally written for that particular station if it had executed as a separate OS process. Instead the code we insert for a station in that single loop is the “one cycle execution trace” for that station, which is obtained with the above mentioned RTL design approach.
In our sorting system, we make the consecutive station algorithm threads with FIFOs. With this mechanism, data flow is performed without any data loss or any timing errors.
The *main()* function of the system software calls the station algorithms in a single loop. In the loop body of the main function, first Station A detects a new incoming part and then with the help of its two sensors it decides whether the part is defective (short) or not (tall). Once a decision is made, Station A writes the result into the FIFO between Station B and itself (*myFifo[0]*). The decision process of Station A is one of the most important steps of the sorting operation. The defective or not decision made as a result of this step is conveyed to the other stations in the system through FIFOs. Hence, it is very critical that a reliable decision is made by Station A.
Station B, just like Station A, first detects if there is new part going through, then it pops a part data from *myFifo[0]* to see if the part going through is defective or not. If the popped data tells Station B that the part is defective, Station B then opens the reject arm to a certain angle. To do that, voltage (+12V) should be supplied to the motor driving the reject arm for a certain period of time. Our PIC code measures time by, what we call, a timer variable. As soon as the information of a defective part is pulled from the FIFO, the timer variable starts decrementing. When the variable reaches zero, voltage supply to the motor is dropped to zero, and as a result, the motor stops. To summarize, instead of making an atomic call to a sleep function to wait for a certain amount of time until we stop the motor, we execute a few instructions (hence a non-blocking operation) to decrement the timer variable when it is non-zero. So a count-down (a sleep operation) is triggered by setting the timer variable to a non-zero value.
Another operation that is performed by Station B is to pop the defective or not flag for the part going through from *myFifo[0]* and push it into FIFO *myFifo[1]* between Station C and itself. Station C is the last station of our sorting system. When Station C detects a new part, Station C (just like Station B) pops the defective flag from *myFifo[1]* (the FIFO between Station B and C). If the popped data tells that the part is defective, Station C closes the reject arm by applying the reverse of the voltage (-12V) applied by Station B to open the arm. For this operation, we keep another timer variable, whose initial value is proportional to the wait time and has been adjusted carefully. Station C sets a PIC output to 1 before the timer variable is triggered, and as a result, the reject arm starts to close. When the variable reaches zero, Station C resets the PIC output and hence the voltage to the reject motor, and the arm stops.

4. CLASS LIBRARY

Proposed methodology has 3 classes: (i) sensor, (ii) FIFO, and (iii) timer classes. Note that since PIC's compiler doesn't support C++ (object-oriented code), we tried to achieve the same class behavior with C structs and functions. In any case, C++ requires more memory than C and runs slower.



```
struct sensor {
   int threshold;
   byte portNo;
   int currentVal;
   int prevVal;
   int debounceTimer;
   int debounceInterval
} mySensor [SENSORS];

void initSensor(int sensNo, byte portNo, int threshold,
                int debounceInterval);
byte newPart(int sensNo);
```

Figure 3. Sensor class

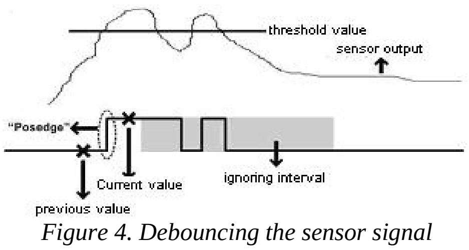
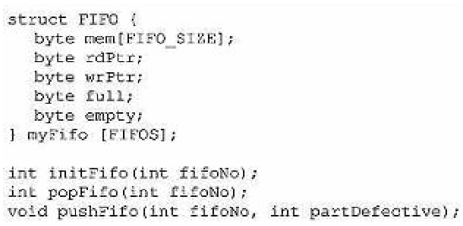


Figure 4. Debouncing the sensor signal

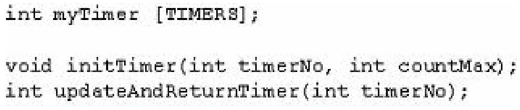
Fig. 3 shows the sensor class with its data fields and functions. Sensors produce a signal above a certain level when a part goes by. However, it is very likely that the signal bounces (i.e., has glitches as in Fig. 4). This signal has to be processed to decide if a new part has passed through (one part, more parts, or none). The first step for this process is to check whether the sensor's signal is above a predefined threshold. If the signal goes above the threshold, the signal has to be ignored for a while so that we do not think there are multiple parts going through (due to bouncing of the signal.) This operation is called "debouncing" (Fig. 4). This technique is borrowed from the hardware design world. When a button/switch on a board is sampled by a chip, the logic should be robust and should not be confused by the bouncing signal produced by the bouncing button. A button, being a mechanical device, may cause unwanted effects such as contacts bouncing on each other. Debouncing requires storing the previous value of the signal so that it can be compared with the current value to detect a rising edge (*posedge*) of the signal. So both values are added to the sensor class.
In sensor class, *threshold* holds the threshold value, *portNo* holds the port through which the sensor communicates with PIC, *currentVal* holds the most recent sample of the sensor, *prevVal* holds the previous sample, *debounceTimer* holds the value of the debounce timer variable, *debounceInterval* holds the length of the interval the signal is ignored, hence the start value of *debounceTimer*.
Function *initSensor()* of sensor class takes the sensor number as argument and initializes the fields of the sensor object for the particular sensor it is called for. Another function of this class is *newPart()*, which moves the current value of the sensor to the previous value. The function then stores the newly sampled value in *currentVal* field. After updating the current and previous values, *newPart()* decides whether a new part went by. If *prevVal* is 0 and *currentVal* is 1, then there is a posedge and hence a new part. After that, the signal debounce timer is started. The possibility of false alarm is prevented by the help of this process.



```
struct FIFO {
   byte mem[FIFO_SIZE];
   byte rdPtr;
   byte wrPtr;
   byte full;
   byte empty;
} myFifo [FIFOS];

int initFifo(int fifoNo);
int popFifo(int fifoNo);
void pushFifo(int fifoNo, int partDefective);
```

Figure 5. FIFO class



```
int myTimer [TIMERS];

void initTimer(int timerNo, int countMax);
int updateAndReturnTimer(int timerNo);
```

Figure 6. Timer class

The fields and functions of the FIFO class are shown on Fig. 5. The array that holds the contents of the FIFO is called *mem[]*. We have implemented the FIFO as a circular buffer. As a result of this, we do not have to shift the contents of the FIFO during push or pop. This saves CPU cycles. The variable *rdPtr* points to the oldest element of the array, and hence we read (i.e., pop) from that location. The variable *wrPtr* points to the tail of the circular buffer, and that is where we write when a new data comes in. The *empty* flag is set when the FIFO is completely empty, and the *full* flag is set when it is completely full. Function *initFifo()* initializes the fields of the class. Through *popFifo()* function, the element at index *rdPtr* is read and deleted, and then *rdPtr* is incremented. On the other hand, *pushFifo()* function writes the defective flag for a part to the end of the FIFO (i.e., where *wrPtr* points), and then increments *wrPtr.* The last class of our methodology is the timer class (Fig. 6). This class is a simple integer. With the function *initTimer()* timer is set to an upper value as if it's a timer watch. On the other hand, function *updateAndReturnTimer(),* decrements the value of the timer if non-zero and then returns its value. This function serves as the preemptable version of *sleep()* function. When the timer value reaches zero, then the sleep period ends.

5. REFERENCES

[1] Sanchiz, J.M., Sanchez, J.S.: An integral automation of industrial fruit and vegetable sorting by machine vision, 8th IEEE International Conference on Emerging Technologies and Factory Automation, Antibes-Juan les Pins, France, vol. 2, pp. 541-546, 2001.
[2] John , K.-H., Tiegelkamp, M.: IEC 61131-3: Programming Industrial Automation Systems, Springer, 2001.
[3] Ugurdag, H.F., Sena, M.: Machine Vision for Quality Improvement and Cost Cutting in Manufacturing, International Conference on Trends in the Development of Machinery and Associated Technology TMT '05, Antalya, Turkey, pp. 1149-1152, 2005.
[4] Monmasson , E., Cirstea, M.N.: FPGA Design Methodology for Industrial Control Systems—A Review, IEEE Trans. on Industrial Electronics, vol. 54, no. 4, pp. 1824-1842, 2007.