Implementation of Sequential Function Charts with microcontrollers

N. A. Ivanescu, Th. Borangiu, S. Brotac and A. Dogar
University Politehnica, Bucharest, Romania

Abstract — The paper describes an efficient method for process control using microcontrollers, based on logic control diagrams, that are commonly used by PLCs. The chosen software solution allows even quasi-simultaneous interpretation of several Sequential Function Charts (SFC). The C compiler program CodeVision AVR was used for implementing this control solution on ATMELE microcontrollers from the ATMEGA family. The experimental results are described for the programming of a CNC machine using an ATMEGA microcontroller based circuit.

I. INTRODUCTION

It is well-known that the programmable logical controllers (PLC), used for controlling industrial processes, use standard programming languages like Sequential Function Chart (SFC), Ladder Diagram (LD), Function Blocks (FB) and so on. But in practice, there are many devices and systems that require a controller, their operation is rather complicated, but they don’t have a large number of I/O signals. Other systems contain signals that vary rapidly and therefore cannot be controlled by classical PLC’s, being known the fact that the PLCs have a relatively slow execution cycle (tens or even hundreds of ms). In most ofthese situations, using a PLC is inefficient or even impossible, instead a microcontroller (in fact a PLC from the hardware point of view, but at a reduced scale) can handle the control process and at the same time reduces the cost of the controlling system.

Over the time, some research was done on implementing PLCs programming languages on microcontrollers, but in general the found solutions were based on literally languages (like Instruction List) or Ladder Diagram. For several types of microcontrollers there are even some software packages (commercial) that allow developing of Ladder Diagram type logic programs, the machine code being automatically generated for the microcontroller.

Our goal was to develop a basic source code (in C language) for ATMELE microcontrollers that interpret one or more logic diagrams of SFC type. We consider that the logic diagrams (with states, transitions and actions) represent the most natural way to design a process controller.

II. SFC CHARTS AND PLC CYCLE

The logic diagrams are used as the main “way” for developing of control applications by the majority of the logic programming packages (Isagraf, Codesys). A simple example SFC diagram is shown in Fig. 1:

![SFC Diagram](image-url)

Figure. 1 Example of an SFC chart

The main elements of the SFC language, common to all programming environments are:

1. **Step**: is an element that corresponds to a stable status of the controller, is identified by a unique number and can be active for certain periods of time.
2. **Transition**: is an element that indicates the possibility of diagram’s evolution from one step to another. Every transition is associated with a logical condition.
3. **Actions associated to steps**: when a step is active, some actions could be performed, like: starting a motor, incrementing a counter, stopping a timer and so on.
4. **Divergence**: is an element that describes different variants of transition from one step, according to more transition conditions associated to that step.
5. **Convergence**: describes simultaneous activation of several sequences of steps, following a common condition coming true.
The dynamics of such a diagram is well known, if a step is active its actions are executed and when one condition of transition becomes true, this step is deactivated and the next step (or steps) in the diagram is activated.

In order that a microcontroller interprets such diagrams, first of all it must emulate a PLC execution cycle. Any PLC has the following execution cycle (simplified):

```
Void execution_cycle (void){
    input_aquisition();
    SFC_execution();
    output_update();
}
```

![Figure 2. PLC execution cycle](image)

During stage 1, “Input acquisition”, electrical values of the inputs are converted in numerical values and stored in special registers, dedicated for input signals.

In stage 2, “Program execution” the PLC identifies all the active steps and executes their associated actions, eventually modifying the registers where the numerical output values are stored. Also all the conditions associated with the active steps are analyzed and in case one is found true, the following step is considered to become active in the next PLC cycle.

Finally the PLC enters the 3’rd stage, updating all electrical values of the outputs by converting their numerical values previously stored in dedicated registers. The period of time that one cycle lasts depends on several factors such as:

- PLC processor speed
- Number and type of I/O signals
- Complexity of the user program

### III. EMULATING A PLC CYCLE ON A MICROCONTROLLER

The first element that must be implemented when trying to emulate a PLC is to create a system timer. Normally middle range microcontrollers have 2-3 timer circuits on 8 or 16 bits or even more. The idea is to create a function that can count rather large period of time until it overflow. This goal can be achieved by using 32 bit timer (normally scaled at maximum for a longer time counting) or, if 32 bit timer is not available, timer can be “extend” by incrementing a 16 bit variable at every timer overflow interrupt obtaining similar results with a hardware 32 bit timer.

#### A. Timer functions

In our experiments we implemented the second solution because we used an Atmel ATMEGA processor having only 16 bit timer. The period of time maximum measured by this system timer is more than 12 days when the oscillator has 3.68 Mhz frequency.

The main functions regarding time created inside the project are:

- void system_timer (void) : increments the extension variable every time the hardware 16 bit timer overflows.
- void reset_timer (void): reset the 16 bit timer registers and also the extension variable.
- void set_timer (word time_value): give the extension variable + timer registers the value of parameter time_value.
- word get_timer (void): returns the current combined value of extension variable + timer registers as a 32 bit value.
- word timer_to_usec (word period): returns the number of microseconds corresponding to the numerical value of parameter period.

Other time functions were developed, like Timer On/Off Delay, but their number is limited by the number of timer comparing interrupts (usually not more than 3 compare interrupts exists for one timer circuit) which are based on.

 Normally these functions can be used to determine time related events appearing in the controlled process. One big advantage of a microcontroller versus a PLC is the time resolution; very small periods of time can be measured (like microseconds). This is efficient in applications where input signals vary very fast or when important events must be identified in very short time.

#### B. Processing of I/O signals

In order to emulate the stages belonging to a PLC cycle the microcontroller has to execute infinitely a high-level function similar to the one above, that simply call another 3 functions that implements the 3 stages discussed in the previous chapter.

```
Void execution_cycle (void){
    input_aquisition();
    SFC_execution();
    output_update();
}
```

Let’s discuss the functions one by one. In microcontrollers digital inputs are acquired automatically at every clock pulse and stored in special I/O registers that are not located in user RAM. But working with these special registers is difficult so inside the input_aquisition() function the value of these I/O registered are transferred in dedicated variables from RAM memory. One bit is dedicated for a digital signal.

Regarding the analogue signals, usually microcontrollers have a small number of analogue lines (8 or 16) and the conversation from voltage to numerical values is done by a simple procedure but it takes a certain amount of time for every channel. Depending on the analogue-digital converter resolution, 1 or even 2 RAM memory locations (can be declared as variables in C program) must be dedicated for storing numerical values from analogue signals. So the input_aquisition() function contains code that store the values of analogue lines in these locations.
From logical type of view the `input_aquisition()` function has the following structure:

```c
void input_aquisition (void) {
    - transfer value of input registers in associated memory locations
    - for i:=1 to total_number_of_analogue_signals do
      associated_variables_for_analogue_lines = converted_value_from_analogue_line_i
}
```

where the significance of the used terms were previously described.

Regarding the output update, the idea is that usually midrange microcontrollers have only digital outputs (in fact digital lines are often bidirectional) and miss analogue outputs. So we concentrated on digital outputs only. Although digital outputs can be accessed directly to change their values, this approach is not advisable because in some applications output values could be modified in more than one step (eventually due to bugs in program or misbehavior of the system) and it’s not good to effectively change the value of the outputs during interpreting the SFC charts. In fact several memory locations (variables) are dedicated to temporarily store the output values, during the SFC execution stage. These values are transferred directly to output registers, when `output_update()` is executed. Practically `output_update()` function looks like below (from logical point of view):

```c
void output_update(void){
    for:=1 to total_number_of_output_registers do
        output_register = associated_variables_for_outputs
}
```

C. SFC interpretation and execution

The most complex part of the PLC cycle emulation is the SFC interpretation. In general more than one SFC chart can be interpreted and executed in this project. Also with a simple trick we considered that SFC’s including convergence (parallelism) elements could be virtually split in separate charts, issue discussed later in the article. An example of creating virtual SFC from a normal SFC with convergence elements is shown in Fig. 3.

The basic ideas used in this project to accomplish SFC diagrams execution are described below.

To determine which steps are active at every cycle an array of `function pointers` are declared and used in the following manner:

- The number of pointers in the array is equal to the number of separate SFC diagrams (independent or extracted from convergence elements) and every pointer is used to store the address of the function associated to the current active step of the associated SFC diagram.
- Initially the function pointers point to the initial steps of the correspondent charts. Some of the pointers associated with non-independent charts can have NULL value because charts are not real independent SFC and it is possible that some time they do not have any active step.
- During SFC execution, the pointers get the values of the addresses of the functions associated with the current active steps.
- The general `execution_cycle` function will test the pointers values and will execute only the functions pointed by non-NILL pointers. In this way the sequences of steps included in a parallelism element are not executed when they have no active step.

Every “step - associated actions” combination is implemented as a C function. Inside such a function the user must insert the correct code to realize the following objectives:

1) execution of “pulse” type associated actions only in the first execution cycle in which the function is called. In order to achieve this, every “step - associated actions” function must be associated with a logical variable (named `first_cycle_step_i`) that must be true only at the first cycle when function is executed. Otherwise it should be false.
2) Execution of “non-pulse” type associated actions in every cycle in which the function is executed.
3) Test of all the associated transition conditions. If any of it is found true, the pointer associated with the respective SFC is updated with the address of the function associated with the next step in the SFC diagram.
4) Update of step status. For every step of every SFC a logical variable is declared, containing the status (active or inactive) of the step. This feature is particularly important at the exit of a parallelism element, the conditions being that all the final steps to be active.

The analogy step – C function is shown in Fig. 4.
To conclude this part of the presentation, the simplified structure of a function associated with a step is presented below (in pseudo code):

```c
void step_i (void) {
    if first_cycle_step_i = true then{
        execute_pulse_actions;
        step_i_status = TRUE;
    }
    execute_non_pulse_actions;
    for j:=1 to total_number_of_transition_conditions do
        if condition_j = true then
            update_correspondent_function_pointer;
            if parallelism_element_follows then
                update_function_pointers_for_virtual_SFC
                step_i_status = FALSE;
                jump_out_of_the_function;
    }
}
```

You can notice here that if one transition condition is found true, the other ones are ignored. This is due to the fact that normally, in an SFC diagram, where divergences elements exist, the transition conditions exclude each other. Still, if conditions do not exclude each other, they have to be assigned with priority levels. In this case the step associated function must be modified a little to analyze all the transition conditions and to validate only the one with the highest priority.

A special situation appears when the active step is the final step of a parallel sequence (assimilated with a virtual SFC). In this case the transition condition is that all the other final steps, from the in case of a parallelism element following the step, all the pointers to the beginning steps (assimilated with initial steps of virtual SFC) of all parallel sequences must be updated. They will become different from NILL and in the next execution cycle the virtual SFC’s will be executed.

D. Management of multiple diagrams

Now that interpretation and execution of active steps are clarified, let’s concentrate about managing of multiple SFC diagrams execution. This job is accomplished by SFC_execution() function. The project was developed having in mind to be an open system and to easily add more diagrams to be interpreted and executed. The logical structure of this function is the following:

```c
void SFC_execution(void){
    i = 1;
    do
        if function_pointer(i) != NILL then
            execute_function(function_pointer(i))
        i++;
    while i<=total_number_of_SFC_diagrams
}
```

Where:
- `function_pointer(i)` is the i element of the function pointer array.
- `execute_function` is a simple function that practically produces a jump in the program at the address indicated by `function_pointer(i)`
- `total_number_of_SFC_diagrams` is a variable equal to the number of the SFC diagrams managed by this program. This variable can be easily updated when an SFC diagram is added to the program.

From the structure presented above results that SFC_execution function tests the status of all real or virtual SFC’s and if their associated pointer is not NILL program will jump to the function associated with the current active step.

SFC diagrams are processed one by one in the order they were introduced, but taking into account that the input and output signals values remain constant during the execution stage, this can be considered as a parallel processing of all the diagrams.

Until here we discussed only about the cycling part of the microcontroller’s program, by infinitely executing execution_cycle function. The basic skeleton of the program also contains an initialization part. All SFC diagrams that are introduced in this management programs must have an initial step, initially active or not (discussed previously). All SFC diagrams that have active initial steps must be declared inside a special function in order to initialize its associated function pointer. A specific function called SFC_Init was created having the following logical structure:

```c
void SFC_Init (void){
    Function_pointer(1) =
        address_of_initial_function_of_SFC_1;
    Function_pointer(2) =
        address_of_initial_function_of_SFC_2;
    ......................
    Function_pointer(n) =
        address_of_initial_function_of_SFC_n;
}
```

In this function, by simply specifying the name of the function associated to an initial step, the correspondent pointer obtains the initial value, thus the SFC_execution function will correctly execute the function associated with the initial step. For virtual SFC (that don’t have an active initial step) there is no need to introduce the NILL value cause this value is given by the microcontroller implicitly.

To conclude this part of the article, the skeleton program developed until this point allows the user to simply initialize the pointers and to develop the associated step-functions according to the SFC diagrams necessary to be executed and according to the logical structures previously discussed. Further a practical application was implemented using the ideas debated above.

IV. PROGRAMMING A CNC MACHINE

In order to experiment the algorithms and the methods conceived, our team developed a full C program for an ATMEGA 8515 Atmel processor, part of an electrical circuit that was sending specific programs to a Computer
Numerically Controlled (CNC) machine. The functions that the microcontroller had to accomplish were:
- serially reception of data from another device
- reading and writing from/to an external eeprom memory connected by SPI interface to the processor
- sending data to the CNC machine using digital lines according to a specific protocol that the machine accepts
- displaying different type of information on a LCD display connected to it
- managing a “keyboard” consisting in 16 keys and interpreting events from it

We determined here 4 automatons that should had to run simultaneously (there are some constraints in system functioning though, not explained here):
1) Eeprom manager
2) CNC programming manager
3) LCD manager
4) Keyboard manager

Because the serially reception of data was too simple to add it as an automaton, it was implemented in the serial interrupt procedure.

So the initializing function looks like:

```c
void SFC_Init (void)
{
    pointer(1)= address_of_EEprom_manager_init_step;
    pointer(2)= address_of_cnc_manager_init_step;
    pointer(3)= address_of_lcd_manager_init_step;
    pointer(4)= address_of_kb_manager_init_step;
}
```

Their SFC diagrams were developed, so the total number of SFC diagrams is 4. E.g., for managing information display on LCD the following SFC diagram was drawing out (simplified):

![SFC Diagram](image)

Figure 5. SFC for controlling the LCD display

As it's shown in Fig. 5, SFC consists of 5 steps, each with different actions to be executed. For all of these steps 5 functions were developed in the C program, in accordance with the method presented in the previous chapter. To be noticed that in Step 4 the processor receives characters one by one and all at a time, due to the principle that the time in which one step is executed must be minimized. Otherwise, cycle time increases very much, possibly negatively influencing the whole control process (inputs values missing, other diagrams misinterpreted and so on).

The SFC execution function can be simplified and looks like this:

```c
void SFC_execution(void)
{
    i = 1;
    do
        execute_function (function_pointer(i))
    i++;
    while i<=4;
}
```

NIL value testing was eliminated because there are no virtual SFC's in this application. Similar to LCD automaton, the other automatons are designed and implemented. The results of the whole process were very good. All 4 SFC diagrams worked perfectly together. It was extremely difficult to imagine a non-logical approach to this process.

V. CONCLUSIONS AND FUTURE WORK

The methods and the basic programs developed until now and explained in this paper facilitate the usage of a microcontroller for controlling different devices or even industrial systems. Anyway there are some limitations that must taking into account when starting to develop a program of this type:
- reduced number of inputs/outputs
- limited memory dimension, the necessary amount of memory must be well calculated before thinking of developing the program
- possibility of efficient design of additional circuits, commonly necessary when microcontrollers are used, because of low cost and low performance circuits inside them (compared with a PLC)

But some advantages must be also mentioned, like very short execution cycle, very low cost of the processor.

Also some software designing tips should be taken in consideration, e.g. all step-associated functions must be designed to be executed in the shortest time possible, to avoid the situation that one automaton looses important events or the systems fails to process variations of inputs values.

The future work will be concentrated in two directions:
1) Development of actual project by adding different features like: exit actions (executed when a step is deactivated), step timeout (if a step execution lasts longer that it is established it is automatically deactivated), step minimum time and others.
2) Development of another approach of interpreting SFC charts, similar to the algorithm used to create Ladder Diagrams from SFC.

REFERENCES
