Chapter 14 OOP

Introduction

What is OOP? OOP is short for Object Oriented Programming and implies an object is the focus of the program. This chapter discusses the use of objects in PLC programming and their use for making programs that are more readable. Software engineering in general has looked for programming tools that allow more robust and reliable computer programs.

Earlier programming discussions of “top-down” computer programming and “structured programming” have evolved into the use of objects that encapsulate an idea and stand alone in the program as that evolved concept. The overall idea of any programming endeavour has been to break the problem down into component parts and solve each part as it pertains to the whole and then combine the parts into a unified overall program. While the idea of a data structure is important, the focus of a top-down program or structured program would focus on the structure or flow rather than the task.

OOP

The idea of pluggable entities or “lego” modules implies that modules can be created that can be plugged into one another and perform as a whole. This entails the idea of information that may be needed only inside the “lego” program and thus hidden from the outside world, in short, the idea of encapsulation. This leads to the idea of objects and object oriented programming or OOP. In OOP, the data is protected since it can be manipulated only inside the shell or protected program called the object. The logic is protected since it is only executed inside the object and the details of the program are known inside the object as to how the program and data interact.

OOP protects the data. It is easier to write in a general sense since the module can block out other programming considerations and the data can be stored inside the OOP as opposed to elsewhere. Data can, obviously, be stored anywhere the programmer wants but the protection of the OOP gives a security that wasn’t available in earlier programming methods. And the programmer and the end user can focus on the object. This gives a great advantage to maintaining the program since the focus is placed where it was originally intended, on the object itself. And if some part changes, the programmer is reminded of the entity that is being affected and that changes should be studied as a whole for the object and not for just a part of the object. In the case of maintainability of control programs for the factory floor, the diagnostic part should be modified when the control portion is modified. If encapsulated, the programmer is reminded of both together since both should be encapsulated inside the object.

The programming language C refers to an object as a class. Other languages have similar names. In Siemens S7-Basic, the object is referred to as a Function (FC) or Function Block (FB). Allen-Bradley’s RSLogix 5000 also has introduced a function with version 18 called the AOI or add-on instruction. While it will not be featured in this chapter, its use is similar to the function or function block described here.

To program an FC or FB, first identify the object or objects involved. Identify the messages or...
signals the object needs to respond to and the outputs that result from these messages or signals. The FC or FB, while considered a class, may also be considered a template for the program. And the idea of FB’s or FC’s calling other FB’s or FC’s is a powerful concept and creates the idea of sub classes or sub-sub classes. The FC or FB can also be re-used again and again in the same or other programs for the same or different clients. This gives the idea of a class or OOP a huge advantage over conventional programming as we build programs over the years or from job to job.

Arguments for the benefits of reusability include:

- Reliability
- Efficiency of programming effort
- Saving time in program development
- Decreased maintenance effort
- Resulting cost savings
- Consistency of programs

Some additional terms of OOP include:

- Encapsulation – combining of programs and data to manipulate outcome in an object
- Inheritance – building a hierarch of objects, each with inheritance of the parent object
- Polymorphism – allowing one set of actions to share an object with another set of actions

FCs can be locked by the creator. This helps to preserve and protect the code and can actually help to simplify the overall program by firmly defining a functionality that is unchanged from one instance of code to another. STEP7 enables the user to create a storage location for custom functions called a Library. Several frequently used standardized and system functions are provided to the user in several libraries included with STEP7. The user can create a custom library and add items as needed, supporting programming standardization across projects.

The S7 architecture also supports the structuring of user-defined data storage locations, called data blocks, and reusable data templates called PLC Data Types.

The S7-1200 and 1500 controllers use programming elements that comply with IEC 61131-3 standard. At the core of the programming structure are code and data containers, known collectively as “Blocks”. The programmable logic controller provides various types of blocks in which the user program and the related data can be stored. Depending on the requirements of the process, the program can be structured in different blocks.

**Organization blocks**: (OB’s) form the interface between the operating system and the user program. The entire program can be stored in OB1 that is cyclically called by the operating system (linear program) or the program can be divided and stored in several blocks (structured program).

**Function**: (FC’s) contains a partial functionality of the program. It is possible to program functions so that they can be assigned parameters. As a result, functions are also suited for
programming recurring, complex partial functionalities such as calculations. System functions (SFC) are parameter-assignable functions integrated in the CPU’s operating system. Both their number and their functionality are fixed. More information can be found in the Online Help.

**Function Block**: (FB’s) offer the same possibilities as functions. In addition, function blocks have their own memory area in the form of instance data blocks. As a result, function blocks are suited for programming frequently recurring, complex functionalities such as closed-loop control tasks. System function blocks (SFB) are parameter-assignable functions integrated in the CPU’s operating system. Both their number and their functionality are fixed.

**Data Blocks**: (DB’s) are data areas of the user program in which user data is managed in a structured manner.

**Permissible Operations**: You can use the entire operation set in all blocks (FB, FC and OB). We will now start to explore these basic “program structuring elements” of S7 beginning with the FC, or Function. A Function is defined by the IEC 61131 standard as a code container that does not retain internal values from one scan to the next. Functions in the S7 PLC behave in this fashion, and act as a container for user developed program code. A function may have a set of local variables defined for use within the function. Typically, when “called” in the main program, a function will have new “values” (or actual parameters) loaded into the local variable (called formal parameters) for use during execution of the function. Once the “results” are calculated and function execution finishes, the resulting “output value(s)” get returned to the main program.

Before you can create the program for the parameter-assignable FC, you have to define the formal parameters in the declaration table. It is up to the programmer to select the declaration type for each formal parameter.

- The ‘IN’ declaration type should be assigned only to declaration types that will be read for instructions in the subroutine.
- Use the ‘OUT’ declaration type for parameters that will be written to within the function.
- Use the “IN_OUT” for formal parameters that need to have a reading access and a writing access, such as a bit passed into the block that is used in the block for an edge operation.
- TEMP variables are intended to be used for holding interim calculation values or other values that are only required when the block is executing. TEMP variables exist in the local data stack while the block is executing and are overwritten when the block exits. The TEMP variables are - even though they are listed under "Interface" - not components of the block interface, since they do not become visible when the block is called and that no actual parameters have to be passed for the declared TEMP variables in the calling block.

The interface of a block forms the IN, OUT, and IN_OUT parameters. The RETURN parameter is a defined, additional OUT parameter that has a specific name according to IEC 61131-3. This parameter only exists in FCs in the interface. The declared formal parameters of a block are its interface to the "outside" meaning they are "visible" or relevant to other blocks that call this block. If the interface of a block is changed by deleting or adding formal parameters later on, then the calls have to be updated.
When an FC is added to a project, the FC is accessible via the Project Browser. When the FC is to be executed must be determined. This is defined by which OB in which the FC is to be called. For example, if the FC is to be executed every scan, it is placed in OB1. To call an FC in OB1, drag and drop the FC from the project browser onto a network.

**Blocks Types**

This is primarily a Siemens concept although Allen-Bradley has also introduced the idea with their function blocks in later versions of RSLogix 5000.

**Global DBs**

A data block (DB) is a data area in the user program containing user data. Global data blocks store data that can be used by all other blocks. The structure of the global data blocks is user defined.

**Several Types of Blocks in STEP 7 Basic**

Interaction between the operating system and the various block types is pictured below:
Operating System

These interrupt OBs can be triggered by high-speed counters and input channels.

Diagnostic

If a diagnostics-capable module in which the diagnostic error interrupt is enabled detects an error, this module triggers a diagnostic error interrupt. There is only one OB with the fixed number 82.

Time Error

The operating system calls the time error interrupt, OB80, if one of the following events occurs:
1. The cyclic program exceeds the maximum cycle time
2. The called OB is currently being executed
3. An overflow has occurred in an interrupt OB queue
4. Interrupt loss due to high interrupt load

Same as for OB above

For Time-delay interrupts, the operating system starts the corresponding "time-delay interrupt" OBs after a user-defined delay time. The delay time starts running after the call of the instruction SRT, DINT. Together with the "cyclic interrupts" there is a limit to four of these OB types.

For Cyclic Interrupts, these interrupt OBs serve to start program code execution in periodic time intervals independently of the cyclic program execution. Together with the "time-delay interrupts" there is a limit to four of these OB types.
We will begin with a Function and start the project at the end of the chapter. Remember the directions from Ch. 8 – no instructions other than contacts and coils could be used to add two 16-bit numbers. The following show an 8-bit version of the same problem:

For the application, we need a FC rather than an FB since there is nothing to remember from function to function. Something comes in. Something goes out. Nothing is set in the function we need to remember. Choose Add new block, then Function and we will choose LAD (Ladder) because of the need for only contacts and coils:
Notice at left, we are now in the Function “Block_1” and we have a new network to start.

These variables are the ones inside the Function and are, in our case, Bool. We give pseudonames to these variables and begin to program the program inside the Function:

- Input – InBitWd1
  InBitWd2
- Output – SumBit
- In/Out – Carry
As can be seen later, the Input, Output and InOut variables are the ones that are visible inside the Function when used in OB1.

Using these pseudo-variables, we enter the program inside the Function:
After building the logic, right click on Block_1 in the project tree and compile the function:

Now, we are ready to incorporate the FC in the main program OB. We first add the words we want to add together:

After the words to be added are defined, we start programming in OB1. First, we add the logic for bit 0. This is a half-adder and we need only build this logic once. Then we drag the block Block_1 from the left and add it to the logic. We finish the process by adding the word.bit addresses for bit 1 to this function. We now are ready to add the bits 2-15 with function block for each and we are complete. This is left as an exercise.
A review of the Variables inside the FC and FBs defined by Siemens:

Before you can create the program for the parameter-assignable block, you have to define the block parameters in the Interface table. The block interface allows local tags to be created and managed.

The tags are subdivided into two groups shown by the table below:

| Block parameters that form the block interface when it is called in the program |
|-------------------------|-----------------|--------------------------|
| Type                    | Section         | Function                          | Available in                                               |
| Input parameters        | Input           | Parameters whose values are read by the block | Functions, function blocks and some types of organization blocks |
| Output parameters       | Output          | Parameters whose values are written by the block | Functions and function blocks                              |
| InOut Parameters        | InOut           | Parameters whose values are read by the block when it is called, and whose values are written again by the block after execution | Functions and function blocks                              |

Local data that are used for storage of intermediate results:

| Type | Section | Function | Available in |
|------|---------|----------|--------------|--------------|

Ch 14 OOP
<table>
<thead>
<tr>
<th>Temporary local data</th>
<th>Temp</th>
<th>Tags that are used to store temporary intermediate results. Temporary local data are retained for only one cycle</th>
<th>Functions, function blocks and organization blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static local data</td>
<td>Static</td>
<td>Tags that are used for storage of static intermediate results in the instance data block. Static data is retained until overwritten, which may be after several cycles</td>
<td>Function blocks only</td>
</tr>
</tbody>
</table>

A review of when Temporary Tags are active:

Creating an FB Block

“FB – Function block Code blocks that store their values permanently in instance data blocks, so that they remain available even after the block has been executed.

All In-, Out-, InOut- parameters are stored in the instance DB – the instance DB is the memory of the FB.

Definition Function blocks are code blocks that store their values permanently in instance data blocks, so that they remain available even after the block has been executed. They save their input, output and in/out parameters permanently in the instance data blocks. Consequently, the parameters are still available after the block execution. Therefore they are also referred to as blocks "with memory".

Block Interface The block interface for an FB looks similar to that of an FC. There are two groups of Block interface tags:
1. Block parameters that form the block interface when it is called in the program.
   - Input, Output, and In/Out parameters are a part of this group

2. Local data that are used for storage of intermediate results
   - Temporary local data and Static local data are part of this group

**Static Local Data**
An instance DB is used to save static local data. These static local data can only be used in the FB, in whose block interface table they are declared. When the block is exited, they are retained.

**Parameters**
When the function block is called, the values of the actual parameters are stored in the instance data block.
If no actual parameter is assigned to an interface parameter in a block call, then the last value stored in the instance DB for this parameter is used in the program execution.
You can specify different actual parameters with every FB call. When the function block is exited, the data in the data block is retained. To keep the data unique for each instance of a call it is required to assign a different instance DB each time a call instruction to an FB is written in code.

You can program parameter-assignable blocks for frequently recurring program code. This has the following advantages:

1. The program only has to be created once, which significantly reduces programming time.

2. The block is only stored in the user memory once, which significantly reduces the amount of memory used.

3. The FB can be called as often as you like, each time with a different address assignment. For this, the interface parameters (input, output, or in/out parameters) are supplied with different actual operands every time called.

**Multi-instance data block**

**Definition**
Multi-instances enable a called function block to store its data in the instance data block of the calling function block. This allows you to concentrate the instance data in one instance data block and thus make better use of the number of instance data blocks available.”
So, where do most programmers write the majority of their code when using Siemens? That is a good question but one that should consider the use of FB’s as the main area for large control programs. Why? We have the ability to use static global variables in this area and also the ability to have all the variable types including arrays present. In the classroom environment, it is not necessary to consider this because programs here probably are rather small. When they grow, however, use the FB as a primary area for large programming efforts.

Later, in the Lab Text, we will see what the Festo Programs give in the way of FB’s. They are very large and complex.
Lab 14.1

Revisit the Binary Addition/Binary Subtraction lab from chapter 8 to subtract one 16 bit word from another and put the 16 bit result in a third word using a function and using the Siemens TIA software.

Remember:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 carry 1
1 + 0 + carry = 0 carry 1
1 + 1 + carry = 1 carry 1

These are the rules for binary addition.

To see binary addition at work:

<table>
<thead>
<tr>
<th>Carry</th>
<th>1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number 1</td>
<td>0 1 0 0 1 1 0 1 1 0 0</td>
</tr>
<tr>
<td>+ Number 2</td>
<td>0 1 0 1 1 0 1 1 0 1 0</td>
</tr>
<tr>
<td>Results</td>
<td>1 0 1 0 1 0 0 0 1 1 0</td>
</tr>
</tbody>
</table>

Binary addition may take place in ladder logic. Instructions are provided to carry out this function (ADD), but it is worthwhile to examine the process of binary addition using ladder logic.

Since Bit 0 does not have a carry_in, half-adder logic may be employed but only for this bit. It can be seen that half-adder logic is simpler than full-add logic by comparing Fig. 8-35 (Half-Adder) to Fig. 8-36 (Full Adder).
Accessing Bits in Words (Siemens)

Examples

In the PLC tag table, “DW” is a declared tag of type DWORD. The examples show bit, byte, and word slice access:

<table>
<thead>
<tr>
<th>Access</th>
<th>LAD</th>
<th>FBD</th>
<th>SCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit access</td>
<td>“DW”.b1</td>
<td></td>
<td>IF “DW”.b1 THEN END_IF;</td>
</tr>
<tr>
<td>Byte access</td>
<td>“DW”.b2</td>
<td>“Byte”</td>
<td>IF “DW”.b2 = “DW”.b3 THEN END_IF;</td>
</tr>
<tr>
<td>Word access</td>
<td></td>
<td></td>
<td>out:= “DW”.w0 AND “DW”.w1;</td>
</tr>
</tbody>
</table>

Accessing a tag with an AT overlay

The AT tag overlay allows you to access an already-declared tag of a standard access block with an overlaid declaration of a different data type. You can, for example, address the individual bits of a tag of a Byte, Word, or DWord data type with an Array of Bool. To overlay a parameter, declare an additional parameter directly after the parameter that is to be overlaid and select the data type “AT”. The editor creates the overlay, and you can then choose the data type, struct, or array that you wish to use for the overlay.
Example

This example shows the input parameters of a standard-access FB. The byte tag B1 is overlaid with an array of Booleans:

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AT[0]</td>
<td>Boar</td>
</tr>
<tr>
<td></td>
<td>AT[1]</td>
<td>Boar</td>
</tr>
<tr>
<td></td>
<td>AT[2]</td>
<td>Boar</td>
</tr>
<tr>
<td></td>
<td>AT[3]</td>
<td>Boar</td>
</tr>
<tr>
<td></td>
<td>AT[4]</td>
<td>Boar</td>
</tr>
<tr>
<td></td>
<td>AT[5]</td>
<td>Boar</td>
</tr>
<tr>
<td></td>
<td>AT[6]</td>
<td>Boar</td>
</tr>
<tr>
<td></td>
<td>AT[7]</td>
<td>Boar</td>
</tr>
</tbody>
</table>

Table 4- 8 Overlay of a byte with a Boolean array

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Another example is a DWord tag overlaid with a Struct:

<table>
<thead>
<tr>
<th></th>
<th>DW1</th>
<th>DWord</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AT[1]</td>
<td>Struct</td>
</tr>
<tr>
<td></td>
<td>s1</td>
<td>Word</td>
</tr>
<tr>
<td></td>
<td>s2</td>
<td>Byte</td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td>Byte</td>
</tr>
</tbody>
</table>

The overlay types can be addressed directly in the program logic:

<table>
<thead>
<tr>
<th>LAD</th>
<th>FBD</th>
<th>SCL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>IF #AT[1] THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF (#DW1_Struc.s1 = #W16#0000C) THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END_IF;</td>
</tr>
<tr>
<td></td>
<td>MOVE</td>
<td>out1 := #DW1_Struc.s2;</td>
</tr>
<tr>
<td></td>
<td>EN</td>
<td>ENO</td>
</tr>
<tr>
<td></td>
<td>EN</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>EN</td>
<td>ENO</td>
</tr>
</tbody>
</table>

Rules

- Overlaying of tags is only possible in FB and FC blocks with standard access.
- You can overlay parameters for all block types and all declaration sections.
- An overlaid parameter can be used like any other block parameter.
- You cannot overlay parameters of type VARIANT.
- The size of the overlaying parameter must be less than or equal to the size of the overlaid parameter.
- The overlaying variable must be declared immediately after the variable that it overlays and identified with the keyword “AT”.

Complete the lab using a function instead of coding each network as separate logic.
Binary Subtraction:

To perform binary subtraction, the easiest method is to find the 2’s complement of the second number and then add the two numbers together.

The best method of finding the 2’s complement requires the use of a memory bit. The rule requires that bits from the original number be copied to the 2’s complement number starting at the right-most bit. The rule applies until a “1” is encountered. The first “1” is copied but a memory bit is set after which the bits are “flipped”. Try this rule. It works and may be employed using ladder logic and a Latch bit to quickly find the 2’s complement of a number. The logic for finding the 2’s complement of a number in ladder logic is begun in Fig. 8-37. Again, logic must be added to complete the function using rungs similar to rungs 4 and 5 of this figure but using bits 2 through 15.

Again, code the logic using a function.

Lab 14.2

Implement the following:

**Linking PLC UDT Tags to HMI Faceplates and Pop-ups**


*Jason Mayes*

Lab 14.3 **Repeat Lab 13.2C1 using an FB and UDT – Whack-a-Mole**

**Lab 13.2C** Add a table of results including whether the player hit the light while the light was on and how long the response was delayed from when the light first turned on. Results for each hit are to be saved sequentially in the table.

**Lab 13.2C1** Implement 13.2C above with a UDT output table. Save sequential hit data for later display or analysis.
Problems

1. Three types of parameters for interface of a function are:
   
   a
   
   b
   
   c

2. Local data is of two types. They are:
   
   a
   
   b

3. Data blocks are either Single ____ or multi _____. What is the deciding factor which to use?

4. List some program blocks that are standard.

5. Describe a function or function block that would have the title “Engine” and have two types of engines that could be called – Diesel or Gasoline.

6. What is the A-B process for adding a function?

For the next three programs, only use skeletal statements but enough to get the idea:

7. In the Kitchen:
   In the kitchen are several needs for automation including cooking breakfast. In the breakfast shelf are several kinds of cereal including oatmeal, cream-of-wheat and grits. Each requires the microwave and a cooking time. Each requires an amount to be weighed on a scale. When the weight is achieved, the bowl is placed in the microwave for a time period. Write a program using FC’s or FB’s to achieve cooking of the breakfast cereal.

8. Three numbers are to be added and the result displayed. Use a FC or FB to accomplish this.

9. In OB1, there is a FC1 accessed that **subtracts 1** from a number. Build the function block SUB1 to complete the operation. Show all **tables** and **logic**:
10. In OB1, there is a FC1 accessed that does the following:

```
FC1
"INV"
“S3” In_Num
“B0” In_Bit
Out_Num “X3”
```

Build the function block “INV” to complete the operation. Show all tables and logic:

11. In Siemens’ OB1, there is a FC1 accessed that does the following:

```
FC1
"INV"
“S3” In_Num
“S3By” In_Byt
Out_Num “X3”
```

Build the function block “INV” to complete the operation. Show all tables and logic inside the Function (FC):

Additional useful information:
12. In Siemens’ OB1, there is a FC1 accessed that does the following:

```
FC1
“MIX”
“S1” — In_Num_1
“S2” — In_Num_2
“S3” — Bit_Num
Out_Num — “X3”
```

Bit_Num determines how many bits are moved from In_Num_1. Remaining bits are moved from In_Num_2.
For example, if Bit_Num=7, the number of bits moved from In_Num_1 is 7 as shown at left.

Build the function “MIX” to complete the operation. Show all tables and logic inside the Function (FC1):

[YouTube Video](http://www.youtube.com/watch?v=aUILkF4al30&feature=relmfu)

Siemens SIMATIC S7-1200 Part 2 - Re-Usable Libraries
See how easy it is to implement reusable Libraries in Step 7 Basic Software eliminating time consuming coding of repeat functions. This is part two of a four part series showcasing the time and cost saving benefits of the new S7-1200 and its Step 7 Basic development software. For more information see: [http://www.usa.siemens.com/s7-1200](http://www.usa.siemens.com/s7-1200)

[YouTube Video](http://www.youtube.com/watch?v=L2NLcAQhiSg&feature=relmfu)

Siemens SIMATIC S7-1200 Part 4 - Project-wide Cross Referencing Made Easy
See how easy it is to troubleshoot the complete Controller and HMI software project together for both SIMATIC Basic HMI panels and S7-1200 Controllers. This is part four of a four part series showcasing the time and cost saving benefits of the new S7-1200 and its Step 7 Basic development software. For more information see: [http://www.usa.siemens.com/s7-1200](http://www.usa.siemens.com/s7-1200)

This work is licensed under a Creative Commons Attribution 4.0 International License.