First Steps

The Getting Started for This product is not a stand-alone description. It is a part of the manual and can be called via "First Steps".
Safety Guidelines

This manual contains notices intended to ensure personal safety, as well as to protect the products and connected equipment against damage. These notices are highlighted by the symbols shown below and graded according to severity by the following texts:

![Danger]

**Danger**
indicates that death, severe personal injury or substantial property damage will result if proper precautions are not taken.

![Warning]

**Warning**
indicates that death, severe personal injury or substantial property damage can result if proper precautions are not taken.

![Caution]

**Caution**
indicates that minor personal injury can result if proper precautions are not taken.

**Caution**
indicates that property damage can result if proper precautions are not taken.

**Note**
draws your attention to particularly important information on the product, handling the product, or to a particular part of the documentation.

Qualified Personnel

Only qualified personnel should be allowed to install and work on this equipment. Qualified persons are defined as persons who are authorized to commission, to ground and to tag circuits, equipment, and systems in accordance with established safety practices and standards.

Use as intended

Note the following:

![Warning]

**Warning**
This device and its components may only be used for the applications described in the catalog or the technical description, and only in connection with devices or components from other manufacturers which have been approved or recommended by Siemens.

This product can only function correctly and safely if it is transported, stored, set up, and installed correctly, and operated and maintained as recommended.

Trademarks

**SIMATIC®, SIMATIC HMI® and SIMATIC NET®** are trademarks of Siemens AG.

Third parties using for their own purposes any other names in this document which refer to trademarks might infringe upon the rights of the trademark owners.
Designing an S7-SCL Program

Welcome to "Measured Value Acquisition" - A Sample Program for First-Time Users

What You Will Learn

The sample program for first-time users shows you how to use S7-SCL effectively. At first, you will probably have lots of questions, such as:

- How do I design a program written in S7-SCL?
- Which S7-SCL language functions are suitable for performing the task?
- What debugging functions are available?

These and other questions are answered in this section.

S7-SCL language Elements Used

The sample program introduces the following S7-SCL language functions:

- Structure and use of the various S7-SCL block types
- Block calls with parameter passing and evaluation
- Various input and output formats
- Programming with elementary data types and arrays
- Initializing variables
- Program structures and the use of branches and loops

Required Hardware

You can run the sample program on a SIMATIC S7-300 or SIMATIC S7-400 and you will need the following peripherals:

- One 16-channel input module
- One 16-channel output module

Debugging Functions

The program is constructed in so that you can test the program quickly using the switches on the input module and the displays on the output module. To run a thorough test, use the S7-SCL debugging functions.

You can also use all the other system functions provided by the STEP 7 Standard package.
Task

Overview

Measured values will be acquired by an input module and then sorted and processed by an S7-SCL program. The results will be displayed on an output module.

Acquire Measured Values

A measured value is set using the 8 input switches. This is then read into the measured value array in memory when an edge is detected at an input switch (see following diagram).

The range of the measured values is 0 to 255. One byte is therefore required for the input.

Processing Measured Values

The measured value array will be organized as a ring buffer with a maximum of eight entries.

When a signal is detected at the Sort switch, the values stored in the measured value array are arranged in ascending order. After that, the square root and the square of each number are calculated. One word is required for the processing functions.
Selectable Outputs

Only one value can ever be displayed on the output module. The following selections can therefore be made:

- Selection of an element from a list
- Selection of measured value, square root or square

The displayed value is selected as follows:

- Three switches are used to set a code that is copied if a signal is detected at a fourth switch, the Coding switch. From this, an address is calculated that is used to access the output.

- The same address identifies three values: the measured value, its square root and its square. To select one of these values, two selector switches are required.
Design of a Structured S7-SCL Program

Block Types

The task defined above is best solved using a **structured S7-SCL program**. This means using a modular design; in other words, the program is subdivided into a number of blocks, each responsible for a specific subtask. In S7-SCL, as with the other programming languages in STEP 7, you have the following block types available.

<table>
<thead>
<tr>
<th>STEP 7-Blocks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OB</td>
<td>Organization blocks form the interface between the S7 CPU operating system and the user program. The organization blocks specify the sequence in which the blocks of the user program are executed.</td>
</tr>
<tr>
<td>FB</td>
<td>Function blocks are logic blocks with static data. Since an FB has a &quot;memory&quot;, it is possible to access its parameters (for example, outputs) at any point in the user program.</td>
</tr>
<tr>
<td>FC</td>
<td>Functions are logic blocks that do not have memory. Since they do not have memory, the calculated values must be processed further immediately after the function is called.</td>
</tr>
<tr>
<td>DB</td>
<td>Data blocks are data areas in which the user data are stored. There are shared data blocks that can be accessed by all logic blocks and there are instance data blocks that are assigned to a specific FB call.</td>
</tr>
<tr>
<td>UDT</td>
<td>User-defined data types are structured data types you can create yourself as required and then use as often as you wish. A user-defined data type is useful for generating a number of data blocks with the same structure. UDTs are handled as if they were blocks.</td>
</tr>
</tbody>
</table>
Arrangement of Blocks in S7-SCL Source Files

An S7-SCL program consists of one or more S7-SCL source files. A source file can contain a single block or a complete program consisting of various blocks.
Defining the Subtasks

Subtasks

The subtasks are shown in the figure below. The rectangular shaded areas represent the blocks. The arrangement of the logic blocks from left to right is also the order in which they are called.
### Selecting and Assigning the Available Block Types

The individual blocks were selected according to the following criteria:

<table>
<thead>
<tr>
<th>Function</th>
<th>Block Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>User programs can only be started in an OB. Since the measured values will be acquired cyclically, an OB for a cyclic call (OB1) is required. Part of the program - data input and data output - is programmed in the OB.</td>
<td>⇒ &quot;Cycle&quot; OB</td>
</tr>
<tr>
<td>The subtask &quot;acquire measured values&quot; requires a block with a memory; in other words, a function block (FB), since certain local block data (for example, the ring buffer) must be retained from one program cycle to the next. The location for storing data (memory) is the instance data block ACQUIRE_DATA. The same FB can also handle the address and select output subtask, since the data is available here.</td>
<td>⇒ &quot;Acquire&quot; FB</td>
</tr>
<tr>
<td>When selecting the type of block for the subtasks sort measured values and calculate results, remember that you need an output buffer containing the calculated results &quot;square root&quot; and &quot;square&quot; for each measured value. The only suitable block type is therefore an FB. Since this FB is called by an FB higher up in the call hierarchy, it does not require its own DB. Its instance data can be stored in the instance data block of the calling FB.</td>
<td>⇒ &quot;Evaluate&quot; FB</td>
</tr>
<tr>
<td>A function (FC) is best suited for the subtasks calculate square root and square since the result can be returned as a function value. Moreover, no data used in the calculation needs to be retained for more than one program cycle. The standard S7-SCL function SQRT can be used to calculate the square root. A special function SQUARE will be created to calculate the square and this will also check that the value is within the permitted range.</td>
<td>⇒ &quot;SQRT&quot; FC (square root) and &quot;Square&quot; FC</td>
</tr>
</tbody>
</table>
Defining the Interfaces Between Blocks

Overview

The interface of a block is formed by parameters that can be accessed by other blocks.

Parameters declared in the blocks are placeholders that have a value only when the block is actually used (called). These placeholders are known as formal parameters and the values assigned to them when the block is called are referred to as the actual parameters. When a block is called, input data is passed to it as actual parameters. After the program returns to the calling block, the output data is available for further processing. A function (FC) can pass on its result as a function value.

Block parameters can be subdivided into the categories shown below:

<table>
<thead>
<tr>
<th>Block Parameter</th>
<th>Explanation</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input parameters</td>
<td>Input parameters accept the actual input values when the block is called. They are read-only.</td>
<td>VAR_INPUT</td>
</tr>
<tr>
<td>Output parameters</td>
<td>Output parameters transfer the current output values to the calling block. Data can be written to and read from them.</td>
<td>VAR_OUTPUT</td>
</tr>
<tr>
<td>In/out parameters</td>
<td>In/out parameters accept the actual value of a variable when the block is called, process the value, and write the result back to the original variable.</td>
<td>VAR_IN_OUT</td>
</tr>
</tbody>
</table>

Cycle OB

The **Cycle OB** has no formal parameters itself. It calls the **Acquire FB** and passes the measured value and the control data for its formal parameters to it.

Acquire FB

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Data Type</th>
<th>Declaration Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>measval_in</td>
<td>INT</td>
<td>VAR_INPUT</td>
<td>Measured value</td>
</tr>
<tr>
<td>newval</td>
<td>BOOL</td>
<td>VAR_INPUT</td>
<td>Switch for entering measured value in ring buffer</td>
</tr>
<tr>
<td>resort</td>
<td>BOOL</td>
<td>VAR_INPUT</td>
<td>Switch for sorting and evaluating measured data</td>
</tr>
<tr>
<td>funct_sel</td>
<td>BOOL</td>
<td>VAR_INPUT</td>
<td>Selector switch for square root or square</td>
</tr>
<tr>
<td>selection</td>
<td>WORD</td>
<td>VAR_INPUT</td>
<td>Code for selecting output value</td>
</tr>
<tr>
<td>newsel</td>
<td>BOOL</td>
<td>VAR_INPUT</td>
<td>Switch for reading in code</td>
</tr>
<tr>
<td>result_out</td>
<td>DWORD</td>
<td>VAR_OUTPUT</td>
<td>Output of calculated result</td>
</tr>
<tr>
<td>measval_out</td>
<td>DWORD</td>
<td>VAR_OUTPUT</td>
<td>Output of measured value</td>
</tr>
</tbody>
</table>
Evaluate

The **ACQUIRE FB** calls the **EVALUATE FB**. The data they share is the measured value array that require sorting. This array is therefore declared as an in/out parameter. A structured array is created as an output parameter for the calculated results Square Root and Square. The following table shows the formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Declaration Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sortbuffer</td>
<td>ARRAY[..] OF REAL</td>
<td>VAR_IN_OUT</td>
<td>Measured value array, corresponds to ring buffer</td>
</tr>
<tr>
<td>calcbuffer</td>
<td>ARRAY[..] OF STRUCT</td>
<td>VAR_OUTPUT</td>
<td>Array for results: Structure with &quot;square root&quot; and &quot;square&quot; components of type INT</td>
</tr>
</tbody>
</table>

SQRT and Square

These functions are called by **EVALUATE**. They require an input value (argument) and return their results as a function value.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Declaration Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>REAL</td>
<td>VAR_INPUT</td>
<td>Input for SQRT</td>
</tr>
<tr>
<td>SQRT</td>
<td>REAL</td>
<td>Function value</td>
<td>Square root of input value</td>
</tr>
<tr>
<td>value</td>
<td>INT</td>
<td>VAR_INPUT</td>
<td>Input for SQUARE</td>
</tr>
<tr>
<td>SQUARE</td>
<td>INT</td>
<td>Function value</td>
<td>Square of input value</td>
</tr>
</tbody>
</table>
Defining the Input/Output Interface

The figure below shows the input/output interface. Note that when input/output is in bytes, the lower-order byte is at the top and the higher-order byte is at the bottom. If input/output is in words, on the other hand, the opposite is true.
Defining the Order of the Blocks in the Source File

When arranging the order of the blocks in the S7-SCL source file, remember that a block must exist before you use it; in other words, before it is called by another block. This means that the blocks must be arranged in the S7-SCL source file as shown below:

```
FC SQUARE
FB EVAL
FB ACQ
OB CYCLE
```

[Diagram showing the order of blocks and calls]

```
Defining Symbols

Using symbolic names for module addresses and blocks makes your program easier to follow. Before you can use these symbols, you must enter them in the symbol table.

The figure below shows the symbol table of the sample program. It describes the symbolic names that you declare in the symbol table so that the source file can be compiled free of errors:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding</td>
<td>IW 0</td>
<td>WORD</td>
</tr>
<tr>
<td>Coding switch</td>
<td>I 0.7</td>
<td>BCOL</td>
</tr>
<tr>
<td>CYCLE</td>
<td>OB 1</td>
<td>OB 1</td>
</tr>
<tr>
<td>Entry</td>
<td>IB 1</td>
<td>BYTE</td>
</tr>
<tr>
<td>EVALUATE</td>
<td>FB 20</td>
<td>FB 20</td>
</tr>
<tr>
<td>Function switch</td>
<td>I 0.2</td>
<td>BCOL</td>
</tr>
<tr>
<td>Input 0.0</td>
<td>I 0.0</td>
<td>BCOL</td>
</tr>
<tr>
<td>Output</td>
<td>QW 4</td>
<td>INT</td>
</tr>
<tr>
<td>Output switch</td>
<td>I 0.3</td>
<td>BCOL</td>
</tr>
<tr>
<td>ACQUIRE</td>
<td>FB 10</td>
<td>FB 10</td>
</tr>
<tr>
<td>ACQUIRE_DATA</td>
<td>OB 10</td>
<td>FB 10</td>
</tr>
<tr>
<td>Sorting switch</td>
<td>I 0.1</td>
<td>BCOL</td>
</tr>
<tr>
<td>SQUARE</td>
<td>FC 41</td>
<td>FC 41</td>
</tr>
</tbody>
</table>
Creating the SQUARE Function

Statement Section of the SQUARE Function

Statement Section

The program first checks whether the input value exceeds the limit at which the result would be outside the numeric range. If it does, the maximum value for an integer is inserted. Otherwise, the square calculation is performed. The result is passed on as a function value.

FUNCTION SQUARE : INT
(*-----------------------------------------------------
This function returns as its function value the square of the input value or if there is overflow, the maximum value that can be represented as an integer.
-----------------------------------------------------*)
VAR_INPUT
   value : INT;
END_VAR
BEGIN
IF value <= 181 THEN
   SQUARE := value * value; //Calculation of function value
ELSE
   SQUARE := 32_767; // If overflow, set maximum value
END_IF;
END_FUNCTION
Creating the EVALUATE function block

Flow Chart for EVALUATE

The figure shows the algorithm in the form of a flow chart:
Declaration Section of FB EVALUATE

Structure of the Declaration Section
The declaration section of this block consists of the following subsections:

- Constants: between `CONST` and `END_CONST`.
- In/out parameters between `VAR_IN_OUT` and `END_VAR`.
- Output parameters: between `VAR_OUTPUT` and `END_VAR`.
- Temporary variables: between `VAR_TEMP` and `END_VAR`.

```plaintext
CONST
  LIMIT := 7;
END_CONST

VAR_IN_OUT
  sortbuffer : ARRAY[0..LIMIT] OF INT;
END_VAR

VAR_OUTPUT
  calcbuffer : ARRAY[0..LIMIT] OF
    STRUCT
      squareroot : INT;
      square      : INT;
    END_STRUCT;
END_VAR

VAR_TEMP
  swap        : BOOL;
  index, aux  : INT;
  valr, resultr: REAL;
END_VAR
```
Statement Section of FB EVALUATE

Program Sequence

The in/out parameter "sortbuffer" is linked to the ring buffer "measvals" so that the original contents of the buffer are overwritten by the sorted measured values. The new array "calcbuffer" is created as an output parameter for the calculated results. Its elements are structured so that they contain the square root and the square of each measured value.

The figure below shows you the relationship between the arrays.

![Diagram showing the relationship between measured values, sort buffer, and calculate buffer.]

This interface shows the heart of the data exchange for processing the measured values. The data is stored in the instance data block ACQUIRE_DATA since a local instance for FB EVALUATE was created in the calling FB ACQUIRE.

Statement Section of EVALUATE

First, the measured values in the ring buffer are sorted and then the calculations are made.

- **Sort algorithm**
  The permanent exchange of values method is used to sort the measured value buffer. This means that consecutive values are compared and their order reversed until the final order is obtained throughout. The buffer used is the in/out parameter "sortbuffer".

- **Starting the calculation**
  Once sorting is completed, a loop is executed in which the functions SQUARE for squaring and SQRT for extracting the square root are called. Their results are stored in the structured array "calcbuffer".
Statement Section of EVALUATE

The statement section of the logic block is as follows:

```
BEGIN
(******************************************************************************
 Part 1 Sorting : According to the "bubble sort" method: Swap pairs of values until the measured value buffer is sorted. ******************************************************************************)
REPEAT
  swap := FALSE;
  FOR index := LIMIT TO 1 BY -1 DO
    IF sortbuffer[index-1] > sortbuffer[index]
      THEN  aux := sortbuffer[index];
      sortbuffer[index] := sortbuffer[index-1];
      sortbuffer[index-1] := aux;
      swap := TRUE;
    END_IF;
  END_FOR;
UNTIL NOT swap
END_REPEAT;
(******************************************************************************
 Part 2 Calculation : Square root with standard function SQRT and squaring with the SQUARE function. **************************************************************************)
FOR index := 0 TO LIMIT BY 1 DO
  valr := INT_TO_REAL(sortbuffer[index]);
  resultr := SQRT(valr);
  calcbuffer[index].squeroot := REAL_TO_INT(resultr);
  calcbuffer[index].square := SQUARE(sortbuffer[index]);
END_FOR;
END_FUNCTION_BLOCK
```
Creating the function block ACQUIRE

Flow Chart for ACQUIRE

The following figure shows the algorithm in the form of a flow chart:

- **Start**
  - newval changed?
    - yes: Copy measured value to cyclic buffer, recalculate index
    - no: no change
  - resort changed?
    - yes: ANALYZE
    - no: no change

- **Copy calculated results to results array**
  - new code changed?
    - yes: Analyze code and calculate output address
    - no: Analyze code and calculate output address

- **TRUE**
  - function choice?
    - TRUE: Load square root result
    - FALSE: Load square result

- **Load measured value**

  - End
Declaration Section of FB ACQUIRE

Structure of the Declaration Section

The declaration section in this block consists of the subsections:

- **Constants:** between `CONST` and `END_CONST`.
- **Input parameters:** between `VAR_INPUT` and `END_VAR`.
- **Output parameters:** between `VAR_OUTPUT` and `END_VAR`.
- **Static variables:** between `VAR` and `END_VAR`.

This also includes declaration of the local instance for the `EVALUATE` block.

```
CONST
    LIMIT := 7;
    QUANTITY := LIMIT + 1;
END_CONST

VAR_INPUT
    measval_in : INT ;   // New measured value
    newval     : BOOL;   // Measured value in
"measvals" ring buffer
        resort : BOOL;   // Sort measured values
        funct_sel: BOOL; // Select calculation square
"root/square
        newsel : BOOL;  // Take output address
        selection : WORD; // Output address
END_VAR

VAR_OUTPUT
    result_out : INT;    // Calculated value
    measval_out : INT;   // Corresponding measured value
END_VAR

VAR
    measvals            : ARRAY[0..LIMIT] OF INT := 8(0);
    resultbuffer  : ARRAY[0..LIMIT] OF
STRUCT
    squareroot          : INT;
    square              : INT;
END_STRUCT;
    pointer            : INT := 0;
    oldval             : BOOL := TRUE;
    oldsort             : BOOL := TRUE;
    oldsel             : BOOL := TRUE;
    address             : INT := 0; // Converted
output address
    outvalues_instance: EVALUATE;  // Define local instance
END_VAR
```
Static Variables

The FB block type was chosen because some data needs to be retained from one program cycle to the next. These are the static variables declared in the declaration subsection "VAR, END_VAR".

Static variables are local variables whose values are retained throughout the processing of every block. They are used to save values of a function block and are stored in the instance data block.

Initializing Variables

Note the initialization values that are entered in the variables when the block is initialized (after being downloaded to the CPU). The local instance for the EVALUATE FB is also declared in the declaration subsection "VAR, END_VAR". This name is used subsequently for calling and accessing the output parameters. The shared instance ACQUIRE_DATA is used to store the data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Initialization Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>measvals</td>
<td>ARRAY [..] OF INT</td>
<td>8(0)</td>
<td>Ring buffer for measured values</td>
</tr>
<tr>
<td>resultbuffer</td>
<td>ARRAY [..] OF STRUCT</td>
<td>-</td>
<td>Array for structures with the components &quot;square root&quot; and &quot;square&quot; of the type INT</td>
</tr>
<tr>
<td>index</td>
<td>INT</td>
<td>0</td>
<td>Index for ring buffer identifying location for next measured value</td>
</tr>
<tr>
<td>oldval</td>
<td>BOOL</td>
<td>FALSE</td>
<td>Previous value for reading in measured value using &quot;newval&quot;</td>
</tr>
<tr>
<td>oldsort</td>
<td>BOOL</td>
<td>FALSE</td>
<td>Previous value for sorting using &quot;resort&quot;</td>
</tr>
<tr>
<td>oldsel</td>
<td>BOOL</td>
<td>FALSE</td>
<td>Previous value for reading in code using &quot;newset&quot;</td>
</tr>
<tr>
<td>address</td>
<td>INT</td>
<td>0</td>
<td>Address for output of measured value or result</td>
</tr>
<tr>
<td>eval_instance</td>
<td>Local instance</td>
<td>-</td>
<td>Local instance for the EVALUATE FB</td>
</tr>
</tbody>
</table>
Statement Section of FB ACQUIRE

Structure of the Statement Section

The statement section of ACQUIRE is divided into three subsections:

- Acquire measured values:
  If the input parameter "newval" is different from the "oldval", a new measured value is read into the ring buffer.

- Start sorting and calculation
  Sorting and calculation are started by calling the EVALUATE function block when the input parameter "resort" has changed compared with "oldsort".

- Evaluating the coding and preparing output data
  The coding is read word by word. According to SIMATIC conventions, this means that the upper group of switches (byte 0) contains the higher-order eight bits of the input word and the lower group of switches (byte 1) the lower-order bits. The figure below shows the location of the coding switches.

Calculating the Address

The figure below shows how the address is calculated: Bits 12 to 14 of input word IW0 contain the coding that is read in when an edge is detected at the coding switch (bit 15). The "address" is obtained by shifting right using the standard function SHR and masking the relevant bits using an AND mask.

This address is used to write the array elements (calculated result and corresponding measured value) to the output parameters. Whether square root or square is output depends on "funct_sel".

An edge at the coding switch is detected because "newsel" is different from "oldsel".
Statement Section

The statement section of the logic block is shown below:

```
BEGIN
(* ***********************************************************
Part 1 : Acquiring measured values. If "newval" changes, the
measured value is entered. The MOD operation is used to
implement a ring buffer for measured values.
***********************************************************)
IF newval <> oldval THEN
  pointer := pointer MOD QUANTITY;
  measvals[pointer] := measval_in;
  pointer := pointer + 1;
END_IF;
oldval := newval;
(* ***********************************************************
Part 2 : Start sorting and calculation
if "resort" changes, start sorting the
ring buffer and run calculations with the
measured values. Results are stored in a new array called
"calcbuffer".
***********************************************************)
IF resort <> oldsort THEN
  pointer := 0;               //Reset ring buffer pointer
  eval_instance(sortbuffer := measvals); //Call EVALUATE
END_IF;
oldsort := resort;
resultbuffer := eval_instance.calcbuffer; //Square and square
root
(* ***********************************************************
Part 3 : Evaluate coding and prepare output: If
"newsel" changes, the coding for addressing the array element
for output is recalculated: The relevant bits of "selection"
are masked and converted to integer. Depending on the setting
of
the "funct_sel" switch, "squareroot" or "square" is selected
for output.
***********************************************************)
IF newsel <> oldsel THEN
  address := WORD_TO_INT(SHR(IN := selection, N := 12) AND
16#0007);
END_IF;
oldsel := newsel;
IF funct_sel THEN
  result_out := resultbuffer[address].square;
ELSE
  result_out := resultbuffer[address].squareroot;
END_IF;
measval_out := measvals[address]; //Measured value display
END_FUNCTION_BLOCK
```
Creating the CYCLE Organization Block

Tasks of the CYCLE OB

An OB1 was chosen because it is called cyclically. It performs the following tasks for the program:

- Calls and supplies the ACQUIRE function block with input and control data.
- Reads in the data returned by the ACQUIRE function block.
- Outputs the values to the display

At the beginning of the declaration section, there is the 20-byte temporary data array "system data".
Program Code of the CYCLE OB

```plaintext
ORGANIZATION_BLOCK CYCLE

CYCLE is like an OB1, i.e. it is called cyclically by the S7 system.
Part 1: Function block call and transfer of the input values Part 2: Reading in of the output values and output with output switchover

VAR_TEMP
  systemdata : ARRAY[0..20] OF BYTE; // Area for OB1
END_VAR
BEGIN
(* Part 1: ***************************************************)
ACQUIRE_ACQUIRE_DATA(
  measval_in := WORD_TO_INT(input),
  newval := "Input 0.0", //Input switch as signal identifier
  resort := Sort_switch,
  funct_sel := Function_switch,
  newsel := Coding_switch,
  selection := Coding);

(* Part 2: **************************************************)
IF Output_switch THEN
//Output changeover
  Output := ACQUIRE_DATA.result_out; //Square root or square
ELSE
  Output := ACQUIRE_DATA.measval_out; //Measured value
END_IF;
END_ORGANIZATION_BLOCK
```

Data Type Conversion

The measured value is applied to the input as a BYTE data type. It must be converted to the INT data type. You will need to convert it from WORD to INT (the prior conversion from BYTE to WORD is made implicitly by the compiler). The output on the other hand requires no conversion, since this was declared as INT in the symbol table.
Test Data

Requirements

To perform the test, you require an input module with address 0 and an output module with address 4.

Before performing the test, set all eight switches in the upper group to the left ("0") and all eight switches in the lower group to the right ("1").

Reload the blocks on the CPU, since the initial values of the variables must also be tested.

Test Procedure

Run the test as described in the table.

<table>
<thead>
<tr>
<th>Test</th>
<th>Action</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Set the code to &quot;111&quot; (I0.4, I0.5 and I0.6) and enter this with the coding switch (I0.7).</td>
<td>All outputs on the output module (lower-order byte) are activated and the LEDs light up.</td>
</tr>
<tr>
<td>2</td>
<td>Display the corresponding square root by setting the output switch (I0.3) to &quot;1&quot;.</td>
<td>The LEDs on the output module indicate the binary number &quot;10000&quot; (=16).</td>
</tr>
<tr>
<td>3</td>
<td>Display the corresponding square by setting the function switch (I0.2) to &quot;1&quot;.</td>
<td>15 LEDs on the output module light up. This indicates an overflow since the result of 255 x 255 is too high for the integer range.</td>
</tr>
<tr>
<td>4a</td>
<td>Reset the output switch (I0.3) back to &quot;0&quot;.</td>
<td>The measured value is displayed again. All LEDs on the outputs of the lower-order output byte are set.</td>
</tr>
<tr>
<td>4b</td>
<td>Set the value 3 (binary &quot;11&quot;) as the new measured value at the input.</td>
<td>The output does not change at this stage.</td>
</tr>
<tr>
<td>5a</td>
<td>Monitor reading in of the measured value: Set the code to &quot;000&quot; and enter it with coding switch (I0.7) so that you can later watch the value input.</td>
<td>The output module shows 0; i.e none of the LEDs lights up.</td>
</tr>
<tr>
<td>5b</td>
<td>Switch over the input switch &quot;Input 0.0&quot; (I0.0). This reads in the value set in test stage 4.</td>
<td>The output displays measured value 3, binary &quot;11&quot;.</td>
</tr>
<tr>
<td>6</td>
<td>Start sorting and calculation by switching over the sort switch (I0.1).</td>
<td>The output again indicates 0 since the sorting process has moved the measured value to a higher position in the array.</td>
</tr>
<tr>
<td>7</td>
<td>Display the measured value after sorting: Set the code &quot;110&quot; (I0.6 = 1, I0.5 = 1, I0.4 = 0 of IB0; corresponds to bit 14, bit 13 and bit 12 of IW0) and read it in by switching over the coding switch.</td>
<td>The output now indicates the measured value &quot;11&quot; again since it is the second highest value in the array.</td>
</tr>
<tr>
<td>8a</td>
<td>Display the corresponding results as follows: Switching over the output switch (I0.3) displays the square of the measured value from the 7th step.</td>
<td>The output value 9 (binary &quot;1001&quot;) is displayed.</td>
</tr>
<tr>
<td>8b</td>
<td>Switch over the function switch (I0.2) to obtain the square root.</td>
<td>The output value 2 (binary &quot;10&quot;) is displayed.</td>
</tr>
</tbody>
</table>
Additional Test

The following tables describe the switches on the input module and the examples for square and square root. These descriptions will help you to define your own tests:

- Input is made using switches. You can control the program with the top eight switches and you can set the measured value with the bottom 8 switches.
- Output is indicated by LEDs. The top group displays the higher-order output byte, the bottom group the lower-order byte.

<table>
<thead>
<tr>
<th>Switch</th>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel 0</td>
<td>Enter switch</td>
<td>Switch over to read in measured value</td>
</tr>
<tr>
<td>Channel 1</td>
<td>Sort switch</td>
<td>Switch over to start sorting/calculation</td>
</tr>
<tr>
<td>Channel 2</td>
<td>Function switch</td>
<td>Switch left (&quot;0&quot;): Square root, Switch right (&quot;1&quot;): Square</td>
</tr>
<tr>
<td>Channel 3</td>
<td>Output switch</td>
<td>Switch left (&quot;0&quot;): Measured value, Switch right (&quot;1&quot;): Result</td>
</tr>
<tr>
<td>Channel 4</td>
<td>Code</td>
<td>Output address bit 0</td>
</tr>
<tr>
<td>Channel 5</td>
<td>Code</td>
<td>Output address bit 1</td>
</tr>
<tr>
<td>Channel 6</td>
<td>Code</td>
<td>Output address bit 2</td>
</tr>
<tr>
<td>Channel 7</td>
<td>Code switch</td>
<td>Switch over to enter code</td>
</tr>
</tbody>
</table>

The following table contains eight examples of measured values that have already been sorted.

You can enter the values in any order. Set the bit combination for each value and transfer this value by operating the input switch. Once all values have been entered, start sorting and calculation by changing over the sort switch. You can then view the sorted values or the results (square root or square).

<table>
<thead>
<tr>
<th>Measured Value</th>
<th>Square Root</th>
<th>Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0001 = 1</td>
<td>0, 0000 0001 = 1</td>
<td>0000 0000, 0000 0001 = 1</td>
</tr>
<tr>
<td>0000 0011 = 3</td>
<td>0, 0000 0010 = 2</td>
<td>0000 0000, 0000 1001 = 9</td>
</tr>
<tr>
<td>0000 0111 = 7</td>
<td>0, 0000 0111 = 3</td>
<td>0000 0000, 0011 0001 = 49</td>
</tr>
<tr>
<td>0000 1111 = 15</td>
<td>0, 0000 0100 = 4</td>
<td>0000 0000, 1110 0001 = 225</td>
</tr>
<tr>
<td>0001 1111 = 31</td>
<td>0, 0000 0110 = 6</td>
<td>0000 0011, 1100 0001 = 961</td>
</tr>
<tr>
<td>0011 1111 = 63</td>
<td>0, 0000 1000 = 8</td>
<td>0000 1111, 1000 0001 = 3969</td>
</tr>
<tr>
<td>0111 1111 = 127</td>
<td>0, 0000 1011 = 11</td>
<td>0011 1111, 0000 0001 = 16129</td>
</tr>
<tr>
<td>1111 1111 = 255</td>
<td>0, 0001 0000 = 16</td>
<td>0111 1111, 1111 1111 = Overflow!</td>
</tr>
</tbody>
</table>