IMPLEMENTATION OF STATE CHARTS IN STRUCTURED TEXT LANGUAGE

A MASTER'S THESIS
in
Electrical and Electrnics Engineering
Atilim University

Submitted by
SYED TAIMUR ALI SHAH

MARCH 2018

IMPLEMENTATION OF STATE CHARTS IN STRUCTURED TEXT LANGUAGE

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OF

ATILIM UNIVERSITY

BY

SYED TAIMUR ALI SHAH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

MARCH 2018

Approval of the Graduate School of Natural and Applied Sciences, Atılım University.

____

Prof. Dr. Ali Kara

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

____

Assos. Prof. Dr. Kemal Efe Eseller

Head of Department

This is to certify that we have read the thesis "IMPLEMENTATION OF STATE CHARTS IN STRUCTURED TEXT LANGUAGE" submitted by "SYED TAIMUR ALI SHAH" and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

____

Asst. Prof. Dr. Mehmet Efe Özbek

Supervisor

Examining Committee Members

Asst. Prof. Dr. Hakan Tora ______________________

Asst. Prof. Dr. Enver Çavuş ______________________

Asst. Prof. Dr. Mehmet Efe Özbek ______________________

Date: 16.03.2018

I declare and guarantee that all data, knowledge and information in this document has been obtained, processed and presented in accordance with academic rules and ethical conduct. Based on these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Syed Taimur Ali Shah

Signature:

ABSTRACT

IMPLEMENTATION OF STATE CHARTS IN STRUCTURED TEXT LANGUAGE

Syed Taimur Ali Shah

M.S. Electrical and Electronic Engineering Department

Supervisor: Assist. Prof. Dr. Mehmet Efe Özbek

March 2018, 73 pages

Structured text language is one the five languages of IEC 61131 standard and it is in a way similar to high level languages such as C, C++ etc. Structured text language is able to work alongside many other PLC programming languages so programs such as ladder logic programs can make use of a structured text subroutine.

Coding state charts in structured text on the other hand is a relatively newer area of research and hence there is almost very little material available regarding this topic. Extensive research has been done on coding state charts in C and C++ and even books have been published such as "State Charts in C and C++" by Samek. In this thesis state chart examples were selected and then converted into structured text code. The generated structured text code was also implemented wıth the help of Beckhoff TwinCAT 3.0 programming environment to check if the code generated was error free and ready for implementation.

A Simulink model for the aforementioned state chart was created and structured text code was generated for it using Simulink. At the end of the thesis, explanation of the manually generated structured text code and the Simulink generated structured text code and their comparisons have been provided.

Keywords: State Charts, Structured Text, Computer Keyboard Example, Time Bomb Game Example

ÖZ

DURUM MAKİNESİ DİYAGRAMLARININ YAPILANDIRILMIŞ METİN DİLİNDE UYGULANMASI

Syed Taimur Ali Shah

Yüksek Lisans

Elektrik ve Elektronik Mühendisliği Bölümü

Danışman: Yardımcı Doçent Dr. Mehmet Efe Özbek

Mart 2018, 73 sayfa

Yapılandırılmış Metin (Structured Text) dili, IEC 61131 standardının beş dilden biridir ve C, C ++ ve bunun gibi yüksek seviyeli dillere benzer özelliktedir. Yapılandırılmış metin dili, pek çok PLC programlama diliyle birlikte çalışabilir, böylece merdiven dili (ladder logic) programları gibi programlar bir yapılandırılmış metin alt yordamı kullanılabilirler.

Durum çizelgelerinin yapılandırılmış metinde kodlanması ise nispeten yeni bir araştırma alanıdır ve dolayısıyla literatürde bu konuyla ilgili çok az kaynak bulunmaktadır. C ve C++ dillerinde durum çizelgelerinin kodlanması üzerinde kapsamlı araştırmalar yapılmış ve hatta Samek tarafından yazılan “State Charts in C and C++” gibi kitaplar yayınlanmıştır. Bu projede, bazı durum diyagramı örenekleri seçilerek bunlar için yapılandırılmış metin kodu geliştirilmiştir. Beckhoff TwincCAT 3.0 programlama ortamında kod gerçeklinmiş ve üretilen kodun hatasız ve uygulama için hazır olup olmadığını kontrol edilmiştir.

Daha sonra, söz konusu durum şeması için bir Simulink modeli oluşturulmuş ve bunun için yapılandırılmış metin kodu üretilmiştir. Tezin sonunda, hem elle oluşturulan hem de Simulink kullanılarak oluşturulan yapısal metin kodları ile ilgili ayrı ayrı açıklamalar ve karşılaştırmalar verilmiştir.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and sincere appreciation to my supervisor Assistant Prof. Mehmet Efe Özbek. With his constant guidance and supervision I have been able to conduct a result oriented research and present my thesis report.

DEDICATION

I would like to dedicate my thesis report to my parents for constantly providing me support in all endeavors in life and sticking by me on my life decisions. I would also like to thank all my teachers for providing me with a sound knowledge base so I can apply that knowledge in my practical life. I would also like to support all my friends for their moral support.

GCHS

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION 1

1.1 Overview 1

1.2 Problem Description 1

1.3 Aim 1

CHAPTER 2. STATE CHARTS 2

2.1 States 2

2.2 Transition 2

2.3 Initial state 2

2.4 Final state 3

2.5 Fork 3

2.6 Join 3

CHAPTER 3. STRUCTURED TEXT LANGUAGE 4

3.1 IEC 61131 Languages 4
3.1.1 Sequential Function Chart (SFC) 4
3.1.2 Function Block Diagram (FBD) 4
3.1.3 Ladder Diagram Programming (LD) 5
3.1.4 Instruction List (IL) 5
3.1.5 Structured Text Language 6

3.2 Structured Programming Languages versus Unstructured Programming Languages 9
3.2.1 Key Differences 9

CHAPTER 4. SOFTWARE DEVELOPMENT ENVIRONMENTS 11

4.1 Beckhoff 11

4.2 Applications 11
4.2.1 MATLAB and Simulink 12
4.2.2 Visual Studio – Microsoft 12
4.2.3 TwinCAT 3 XAE – Beckhoff 12

4.3 User Defined Data Types 15
4.3.1 Structured data types 15
4.3.2 Enumerated data types 15
4.3.3 Sub-Range data types 15
4.3.4 Array Data types 15

4.4 Writing Code in TwinCAT 3.0 16

CHAPTER 5. DESCRIPTION OF THE STATE CHART 18

5.1 State Chart of Computer Keyboard Example 18

5.2 Structured Text code with Nested Switch Statement Implementation 19
5.2.1 TwinCAT Implementation 22

5.3 ST code with Object oriented state design 25
5.3.1 C++ Implementation of Keyboard Example using Object Oriented State Pattern... 26
5.3.2 Implementation of Object Oriented State Pattern in TwinCat Using ST 29

5.4 Time Bomb Game Example 36
5.4.1 Implementation of Object Oriented State Pattern in TwinCat Using ST 39

5.5 Simulink Model of Time bomb game 48
5.5.1 TwinCAT implementation of Simulink Generated code for time bomb game 49

CHAPTER 6. DISCUSSION OF THE RESULTS AND CONCLUSIONS 68

6.1 State Design Pattern (Object Oriented State Machine) - Manually Written Code 69

6.2 Nested States FSM (Switch/Case Statements) - Simulink Generated 70
6.2.1 Memory Usage of the Manually Generated Code 71
6.2.2 Memory Usage of the Simulink Generated Code 72

6.3 Comparison of the Results 72

LIST OF TABLES

TABLES

TABLE 3. 1 COMMON STRUCTURED TEXT OPERATORS 7
TABLE 3. 2 COMMON STRUCTURED TEXT INSTRUCTIONS 8
TABLE 3. 3 STRUCTURED VS UNSTRUCTURED 9
TABLE 3. 4 DETAILED COMPARISON BETWEEN STRUCTURE AND NON-STRUCTURED PROGRAMMING LANGUAGES 10

TABLE 4. 1 TWINCAT DATA TYPES 14

GCHS

LIST OF FIGURES

FIGURES

Figure 2. 1 State 2
Figure 2. 2 State with internal activities 2
Figure 2. 3 Transition 2
Figure 2. 4 Initial state 3
Figure 2. 5 Final state 3
Figure 2. 6 Fork 3
Figure 2. 7 Join 3

Figure 3. 1 SFC of Fermentation Process 4
Figure 3. 2 FBD of a simple I/O System 5
Figure 3. 3 LD for triggering a relay coil 5
Figure 3. 4 IL for a simple math’s problem 6

Figure 4. 1 TwinCAT Interface 13
Figure 4. 2 Pizza Oven 16
Figure 4. 3 Pizza Oven 16
Figure 4. 4 Pizza Oven 17

Figure 5. 1 State chart of a Computer Keyboard Example 18
Figure 5. 2 The MAIN of computer Keyboard Example 22
Figure 5. 3 Uppercase state 23
Figure 5. 4 Lowercase State 24
Figure 5. 5 Base and derived classes of software structure of the Keyboard Example 25
Figure 5. 6 Main class of software structure of the Keyboard Example 26
Figure 5. 7 Time Bomb Game State Chart 36
Figure 5. 8 Visualization of time bomb game example 47
Figure 5. 9 Simulink model of time bomb game 48
Figure 5. 10 TimBomb_Controller 49

Figure 6. 1Size of manually generated code 71
Figure 6. 2 Size of Simulink generated code 72

CHAPTER 1.
INTRODUCTION

This chapter will work as the building block for understanding the core of this thesis report including the use of structured text programming language for state charts, background, and the reason for carrying out this research and the target results.

1.1 Overview

Most of the research that has been carried out in recent years regarding state charts have been either done in C or C++. We are familiar with Finite State Machines (FSMs) which are used frequently in both programming and digital electronics. The applications of FSMs are widespread and some of them can be counted as industrial control or communication protocols. State machines have states which contain other states, hence making the state machines hierarchical state machines. These are depicted in the form of state charts. Little research has been carried out to convert hierarchical state machines (HSMs) directly into source code. One of the rare publishing carried out in this area of research is the book 'Practical state charts in C/C++' by Samek [7]. In his book he has coded state machine charts directly into C as well as C++.

There are also UML tools available which directly convert the state charts to their equivalent source code. But it is more beneficial if more people learn how to code state charts without the use of such tools as it would provide a more hands down understanding.

1.2 Problem Description

As discussed earlier knowing how to code state charts is of immense importance. Little research that has been carried out in this field is mostly restricted to programming state charts in C and C++. There is hardly any existing research about coding state machine charts into structured text (ST) language. Structured text language is a comparatively new language and is used intensively in latest programming environments for PLCs such as TwinCAT etc. So the idea for this thesis is to provide a solution for this specific quandary.

1.3 Aim

The aim of this thesis is to take a state chart from Miro Samek's [7] book and to write a code for this specific state chart in structured text language. Then a code is generated for that same state chart in C++. İmplementation of the structured text language code for the state chart in the programming environment TwinCAT 3.0 is also carried out. This thesis also includes the Simulink model for the state chart and the generated structured text code from this model. This thesis report also includes explanation of how both the codes work. The comparison of memory requirements and speed of the both the codes by using TwinCAT 3.0 were also carried out.

CHAPTER 2.
STATE CHARTS

State charts are the representation of the exact changes in state of a system, instead of the commands and processes which are the reasons for those changes. They show the behavior of the system to outside influences. A state chart represents the behavior of an object according to a chain of events in the system. To draw a state chart we need to specify the initial and final state of the related system. Then we take into consideration the other states that the system will transition into and we also have to keep in mind that some transitions will be nullified when the system is in a specific state.

2.1 States

The representation of the changes during the duration of an object is called state.

Figure 2. 1 State

Figure 2. 2 State with internal activities

2.2 Transition

The transition from one state to another in a system is represented by and arrow. The transition is labelled by its cause and its result. A self-transition of a state is show by the arrow pointing back towards the same state.

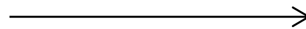


Figure 2. 3 Transition

2.3 Initial state

The initial state of an object is represented by a completely filled up circle.

Figure 2. 4 Initial state

2.4 Final state

The final state of an object is represented by a completely filled circle which is encircled by an empty circle.

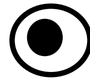


Figure 2. 5 Final state

2.5 Fork

When a transition splits into two concurrent transitions, it is called a fork.

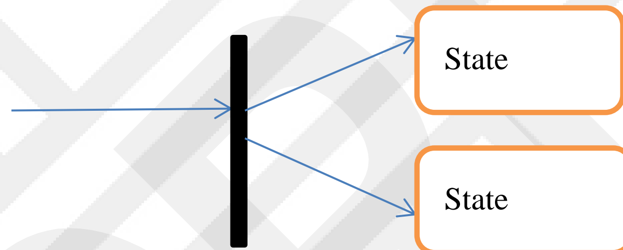


Figure 2. 6 Fork

2.6 Join

When two transitions are reduced back into one, it is called a join.

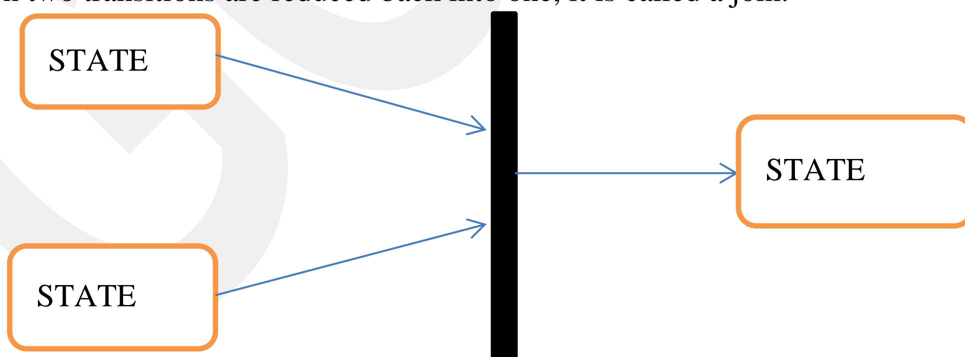


Figure 2. 7 Join

CHAPTER 3.
STRUCTURED TEXT LANGUAGE

In this chapter I will look into the five different languages from the IEC 61131-3 language standard with particular emphasis on Structured Text (ST) as I will be doing my project in Structured Text language.

3.1 IEC 61131 Languages

The IEC 61131-3 language standard has five distinct standard languages namely

- Sequential Function Chart (SFC)
- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Instruction List (IL)
- Structured Text (ST)

3.1.1 Sequential Function Chart (SFC)

SFC is similar to computer flowcharts as shown in the figure [Figure 3. 1 SFC of Fermentation Process] below. The action box which is the initial step is followed by transitions and more steps whereas the action box contains the written code in any language. When the transition happens one of the actions boxes deactivated whereas the next one is activated. There is a code for the transition step which determines if the transition can happen or not. The major advantage of this language is its user friendliness however it should be kept in mind that it is not suitable for all applications. The biggest downside of this language is that it cannot be converted to other IEC 61131-3 languages.

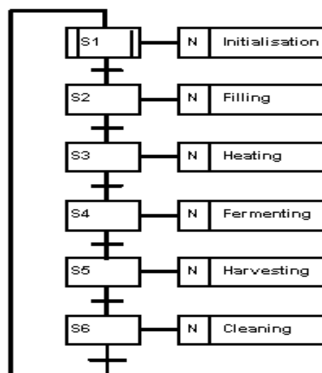


Figure 3. 1 SFC of Fermentation Process

3.1.2 Function Block Diagram (FBD)

FBD is a graphical language. It is used to code PLCs. FBD is a simplified language in which different blocks are wired together in a structured way and hence it is easy to grasp [Figure 3. 2 FBD of a simple I/O System. By following the path in FBD, it gets easier to understand the program. It suits program which have a simpler outlay but when it is about extensive programs with complex inputs and outputs and different functions then FBD becomes obsolete.

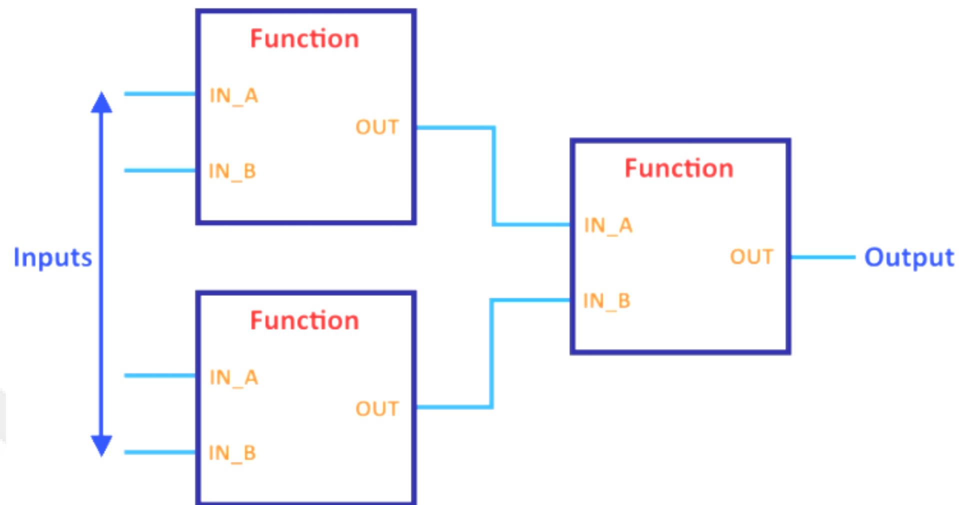


Figure 3. 2 FBD of a simple I/O System

3.1.3 Ladder Diagram Programming (LD)

LD was the first language used to program PLCs and it is the most easy and commonly used graphic programming language. It consists on inputs which have the ability to have a true or false value to activate outputs as shown in figure [Figure 3. 3 LD for triggering a relay coil.

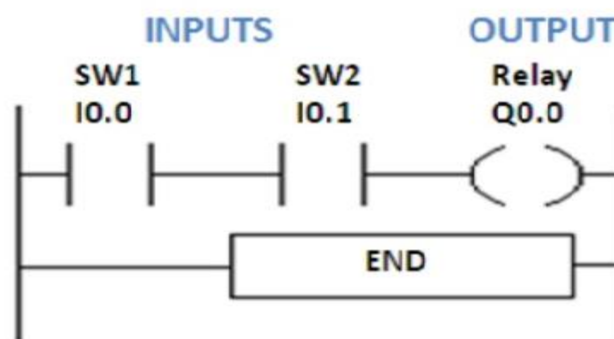


Figure 3. 3 LD for triggering a relay coil

3.1.4 Instruction List (IL)

IL is a low level step by step language and it is made up of extensive lines of coding and every line has a characteristic and specific operation [Figure 3. 4 IL for a simple math's problem. It is faster and consumes less space than visual languages such as LD etc. but it has its downside because it is harder to comprehend.

```
Label     LD     a1     (* result :=al *)
          ADD(   a2     (* delayed ADD, result :=a2 *)
          MUL(   a3     (* delayed MUL, result :=a3 *)
          SUB    a4     (* result :=a3-a4 *)
          )             (* execute delayed MUL, *)
                        (* resule :=a1+(a2*(a3-a4) *a5) *)
          ADD    a6     (* a1+(a2*(a3-a4)*a5)+a6 *)
          ST     res    (* store current result in res *)
```

Figure 3. 4 IL for a simple math's problem

3.1.5 Structured Text Language

ST is closely related to high level languages such as C and C++ etc. It consists of a set of pre-determined instructions which can be executed when required. ST is the best solution for the rising complexity in PLCs. As indicated by the name ST makes a complex program more structured by allowing us to add comments with every step and to give line spacing when we want to differentiate between different components of the program. As ST is non-graphical hence it is faster than visual languages such as LD, FBD etc. ST consists of expressions which are constructs that are showing a value after their computation. Expressions usually consist of operators and operand. Operands can have varying values such as a function call, another expression or either a constant or variable.

Shown next are the tables for common ST operators [Table 3. 2 Common Structured Text Instructions and instructions [Table 3. ***2*** Common Structured Text Instructions].

OPERATION	SYMBOL
Put in parentheses	(expression)
Function call	Function name (parameter list)
Exponentiation	EXPT
Negate Building of complements	- NOT
Multiply Divide Modulo	* / MOD
Add Subtract	+ -
Compare	<,>,<=,>=
Equal to Not Equal to	= <>
Boolean AND	AND
Boolean XOR	Boolean XOR
Boolean OR	OR

Table 3. 1 Common Structured Text Operators

Instruction	Example
Assignment	A:=B; CV := CV + 1; C:=SIN(X);
Calling a function block and use of the FB version	CMD_TMR(IN := %IX5, PT := 300);A:=CMD_TMR.Q;
RETURN	RETURN;
IF	IF D<0.0 THEN C:=A; ELSIF D=0.0 THEN C:=B; ELSE C:=D; END_IF;
CASE	CASE INT1 OF 1: BOOL1 := TRUE; 2: BOOL2 := TRUE; ELSE BOOL1 := FALSE; BOOL2 := FALSE; END_CASE;
FOR	FOR I:=1 TO 100 BY 2 DO IF ARR[I] = 70 THEN J:=I; EXIT; END_IF; END_FOR;
WHILE	WHILE J<= 100 AND ARR[J] <> 70 DO J:=J+2; END_WHILE;
REPEAT	REPEAT J:=J+2; UNTIL J= 101 OR ARR[J] = 70 END_REPEAT;
EXIT	EXIT;
Empty instruction	;

Table 3. 2 Common Structured Text Instructions

3.2 Structured Programming Languages versus Unstructured Programming Languages

Structured Language	Non structured language
Code can be separated into different components	Code cannot be separated into components
Creation of loops is possible	Creation of loops is not possible
Newer languages	Old languages
No strict field concept	Strict field concept
E.g. C++, OOP, ST, JAVA etc.	E.g. BASIC, FORTAN etc.

Table 3. 3 Structured vs Unstructured

3.2.1 Key Differences

One of the biggest differences between structured programming languages and unstructured programming languages is the fact that structured languages enable the coder to divide the code into smaller blocks and compartments. Unstructured coding is called spaghetti coding because there is no structure to the code and everything is jumbled up and is continuous.

Regarding programming, the biggest difference between unstructured and structured programming language is that structured programming language gives freedom to coder to code a program by comparting the program into smaller blocks or modules. Because of this reason a coder finds ease in working on one component of the program at a time. Due to this we can inspect the module individually before integrating it into the main program. So debugging and modifying the module becomes far less complicated while leaving the rest of the code as it is.

On the other hand, unstructured programming leads to writing a code in an unbroken chain and in a continuous manner without giving the option of dividing the code into smaller components. Due to this reason the complication factor increases as the whole code is a single unit. Alongside increase in complication, it also becomes really tough to debug or modify the code if some bug arises in the program. It is harder because if the coder encounters a bug he has to check the whole code instead of just on module

Additionally, unstructured programming languages allow only basic data types, such as numbers, strings and arrays (numbered sets of variables of the same type), which is not the case with structured programming languages. However, unstructured programming languages are often touted for providing freedom to programmers to program as they want. Structured programming languages often make extensive use of subroutines, block structures and for and while loops, as opposed to using simple tests and jumps such as the GOTO statement which could lead to “spaghetti code”, which unstructured programming languages do. Still, spaghetti code is highly

difficult to follow and to maintain, which is why many people don't prefer to use unstructured programming languages.

CHARACTERISTICS	Structured Programming Language	Unstructured Programming Language
Also known as	Modular Programming	Non-structured Programming
Subset of	Procedural Programming	None. It is the earliest programming paradigm
Purpose	To enforce a logical structure on the program being written to make it more efficient and easier to understand and modify	Just to code
Programming	Divides the program into smaller units or modules	The entire program must be coded in one continuous block
Precursor to	Object-oriented programming (OOP)	Structured programming, specifically procedural programming and then object-oriented programming
Code	Produces readable code	Producing hardly-readable ("spaghetti") code
For Projects	Usually considered a good approach for creating major projects	Sometimes considered a bad approach for creating major projects
Freedom	Has some limitations	Offers freedom to programmers to program as they want
Allowed data types	Structured languages allow a variety of data types	Non-structured languages allow only basic data types, such as numbers, strings and arrays (numbered sets of variables of the same type)
Modify and debug	Easy to modify and to debug	Very difficult to modify and to debug

Table 3. 4 Detailed comparison between Structure and non-Structured programming languages

CHAPTER 4.
SOFTWARE DEVELOPMENT ENVIRONMENTS

In this chapter I will provide an in-depth insight into the programming environment known as TwinCAT and then later I will write a structured text code and implement it in TwinCAT.

4.1 Beckhoff

Elektro Beckhoff GmbH was founded in 1953 in Germany, installations and retail sales. In 1980, Beckhoff was founded in Germany, Verl Automation as the parent company's financial unit. In 2005, Elektro Beckhoff GmbH was diversified three of their own companies:

1. Beckhoff Automation GmbH & Co. KG, specializing in PC-based automation technology.
2. Elektro Beckhoff GmbH, specializing in building technology and automation.
3. Beckhoff Technik and Design GmbH, whose specialization area is of the highest quality retail electronics for consumer electronics, light systems and home appliances.

Beckhoff delivers open automation systems based on PC-based control technology. The product range includes fieldbus components, hardware, industrial PCs and control panels as well as automation application software, as discussed in this thesis, Embedded TwinCAT 3 in Visual Studio.

4.2 Applications

The MATLAB and Simulation of The Math Works, Microsoft Visual, were used in this work Studio, Beckhoff TwinCAT 3. With these programs an operating chain is available to enable the Simulink model for TwinCAT 3 to be simulated and modified in real time without the Simulink program use.

4.2.1 MATLAB and Simulink

MATLAB is the commercial and maintained numerical computing software of The MathWorks. It can be used in itself and by the add-ons to calculate matrices, functions and data visualization, algorithm implementation, and user interface creation.

The earliest version of MATLAB is from the 1970's, when it started to develop student's aid, universities and other mathematical communities. MATLAB was commercialized in 1984, since then it has been under the auspices of The MathWorks. Today, MATLAB is used worldwide and is an important tool in many fields and in many schools.

Simulink is a MATLAB add-on that can be used to create simulations and process simulations. It can model and simulate dynamic processes, as well as create tables of measured process quantities.

4.2.2 Visual Studio – Microsoft

Visual Studio is a program development environment that can use multiple programming languages. The first Visual Studio was released in 1997. It was Microsoft's first of many programming language programming tools, including Visual C ++ and Visual Basic programming languages. Today it is used for Windows, web, and mobile applications.

4.2.3 TwinCAT 3 XAE – Beckhoff

TwinCAT 3 is completely embedded in Visual Studio and can use many features of visual Studio. The communication between the different parts of the program is "TwinCAT Transport layer - ADS "interface.

TwinCAT uses Visual studio shell and runs on 64 bit systems. A new project in TwinCAT resembles as shown in the figure [Figure 4. 1 TwinCAT Interface.

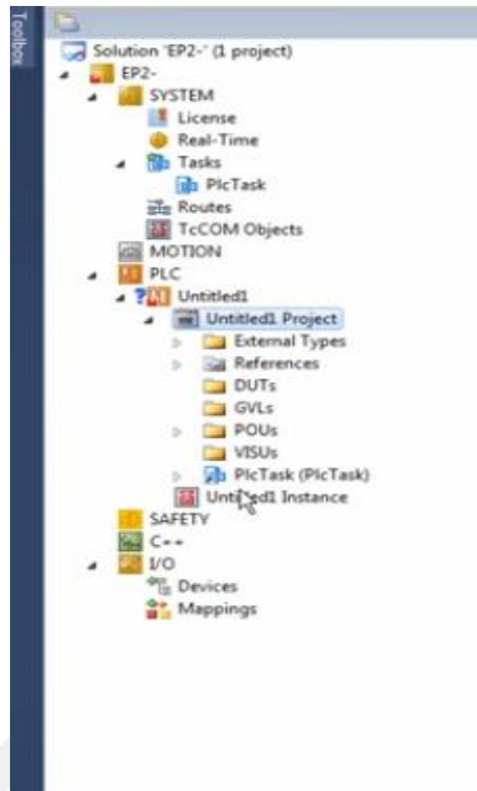


Figure 4. 1 TwinCAT Interface

- Under the **System** tab we have real timer tab in which we can change the time which will in turn determine how fast or slow our program will run.
- Then under **Tasks** we have real time tasks which contain cycle ticks which is a multiplier on our base time.
- **Routes** are basically links to your hardware elsewhere in your system, if we have a TCP/IP Ethernet connection up to our connected systems we can log in and do all our development on top of them and send modified code.
- All the servo systems go under the motions tab.
- Under the **PLC** tab we have subfolders and this is the main software area of the whole program. **References** are libraries so we get standard libraries with general functions. **DUTs** are data user types which are basically the data types defined by the user. **GVLs** are global variables and they are globally accessible for the programs. **POUs** are the programming organizational units and this is where the bulk of the code is situated. **VISUs** are visualizations and they can be used to mimic hardware.
- **Safety** is for the case that we have an actual safety system which is out of the scope at the moment.
- Under the **C++** tab we can program in C++ language but generally it is not required.
- **I/O** is where the program reaches out and actually pulls sensory data in and outputs to physical devices.

Standard Data Types	Standard Data Types	User Defined Data Types
BOOL	INT	Array
BYTE	UINT	POINTER
WORD	DINT	ENUM (enumerated data types)
DWORD	UDINT	STRUCT (structures)
SINT	LINT (64 bit integer, currently not supported by TwinCAT)	ALIAS (derived data types)
USINT	ULINT (Unsigned 64 bit integer, currently not supported by TwinCAT)	Sub-range data types
REAL		
LREAL		
STRING		
TIME		
TIME OF DAY		
DATE		
DATE AND TIME		

Table 4. 1 TwinCAT Data Types

BOOL is a really important data type which is a single bit on or off. These can be used to send flags or hard outputs, cylinder extensions or lights or such stuff. A typical input comes into the system as a Boolean. **BYTE** is another important data type which is basically a string of eight Boolean. **Word** is 16 bit where as **DWORD** is 32 bit. Most data types are defined in 32 bits. **INT** stores a number; **SINT** is short signed integer and is 8 bit but also stores positive or negative numbers. USINT, SINT, INT, UINT are not usually used in abundance. Instead **DINT** is used because we end up taking 32 bits worth of data anyway and as DINT is 32 bit so we rarely run out of data. If we are dealing with decimal numbers we go to **REAL** because it can store really large numbers but they are not precise. **LREAL** is long REAL and it is a 64 bit REAL but it is not used often. **STRING** data type is used to store ASCII string and it is 8 bit per ASCII character and is terminated by null.

4.3 User Defined Data Types

4.3.1 Structured data types

These are composite data types and are manufactured by putting elements between keywords 'STRUCT' and 'END_STRUCT'. It should also be kept in mind that there can be other structured data types within a structured data type.

```
TYPE RECTANGLE:
STRUCT
Top Left: Point;
Height: INT;
Width: INT;
END_STRUCT;
END_TYPE
```

4.3.2 Enumerated data types

When elements of enumeration are enclosed within parentheses they are called enumerated data types. For example:

```
TYPE Color:
(Grey, Pink, yellow);
END_TYPE
```

4.3.3 Sub-Range data types

When the range of other data types such as integers is limited, we construct a sub-range data type. This data type is denoted by the data type which is to be restricted and an upper and lower bound separated by dots and enclosed within parentheses.

```
TYPE Angle:
INT (-180 . . . +180);
END_TUPE
```

4.3.4 Array Data types

Construction of an Array data type is carried out by ARRAY which is then followed by a construct of indices and the keyword OF and the primary data type which will be enclosed in the array. For example

4.4 Writing Code in TwinCAT 3.0

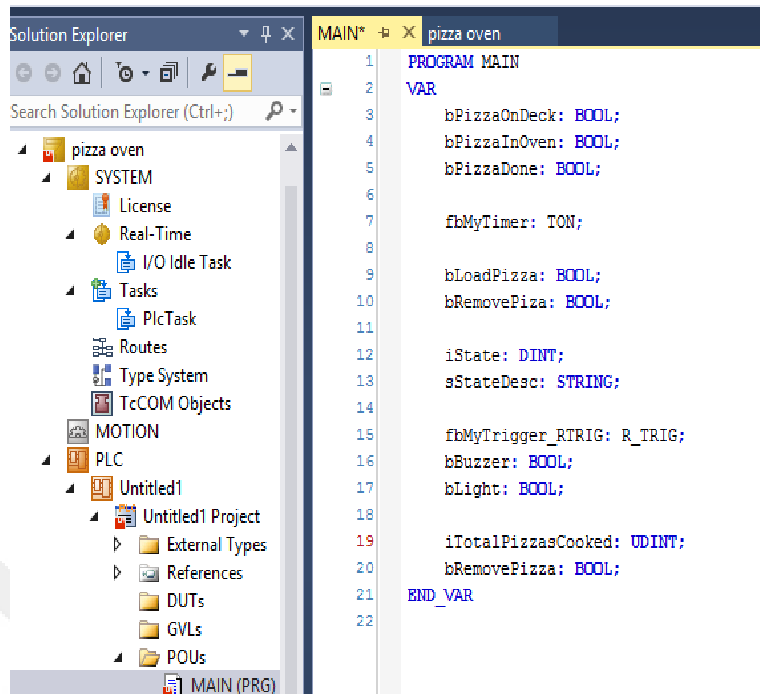


Figure 4. 2 Pizza Oven

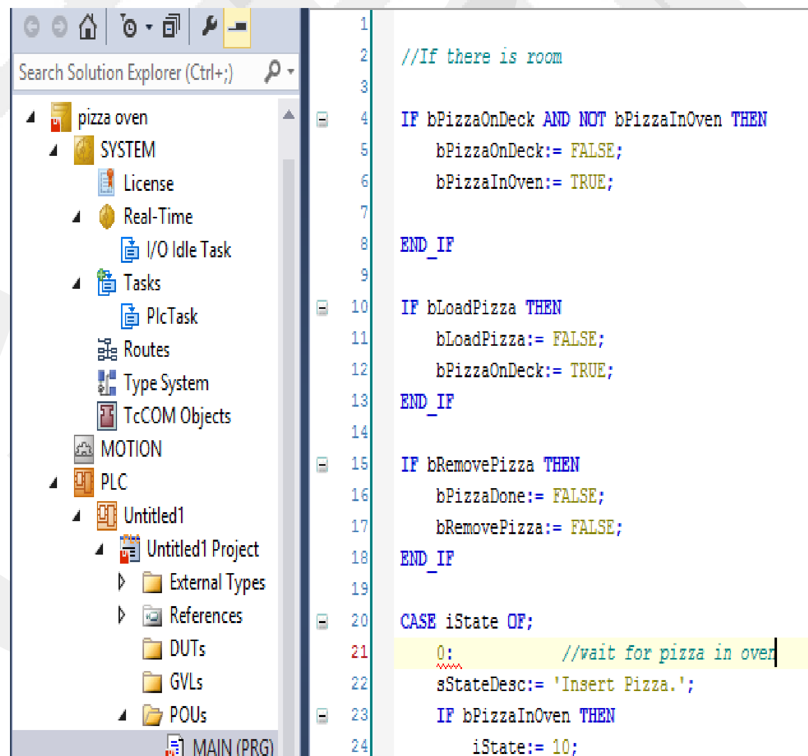


Figure 4. 3 Pizza Oven

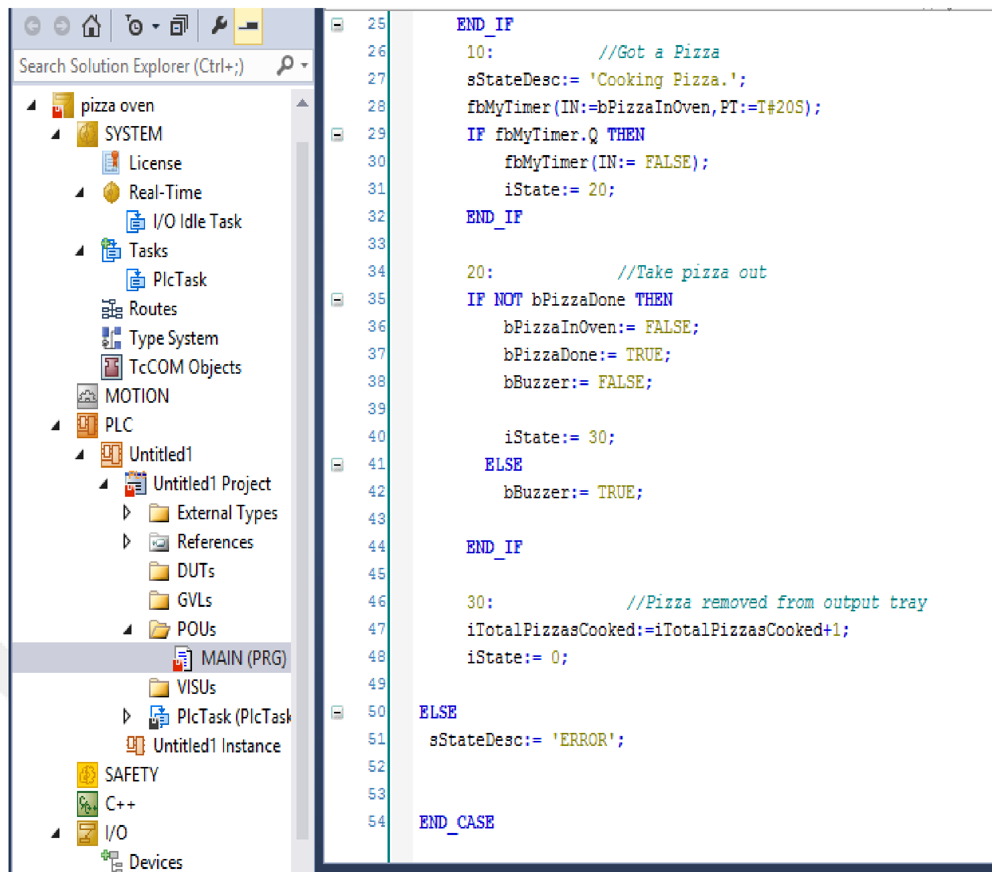


Figure 4. 4 Pizza Oven

The above figures Figure 4. 2 Pizza Oven], [Figure 4. 3 Pizza Oven and [Figure 4. 4 Pizza Oven] depict a general pizza wending oven. It is built by using timers, triggers, counter and such. The conveyor belt takes pizzas in and out according to the states. To understand the concept of Function blocks it should be understood that inputs, outputs and timers etc. are grouped separately instead of them being forming the same block. From the above example we can clearly observe that coding state machine charts in structured text will be really beneficial as with structured text we are able to divide the whole code into different function blocks and then we can work on these blocks separately. It provides a very clear and clean approach towards programming. If we encounter an error in the above code we can easily find out the block of the code where the problem is arising from and then we can go to that block and rectify the problem. Even if the code needs to be updated with changing requirements or circumstances, modification can easily be achieved because we know exactly the block where the modification is required. Or we can add or delete stages in a state chart simply by adding or removing function blocks. The whole point of me giving this example was to put light on the fact that coding state charts in structured text (ST) language is comparatively easier than other programming languages.

CHAPTER 5.
Description of the state chart

The state chart that I have chosen is the computer Keyboard Example state machine chart from Samek's [7] book titled Practical UML State charts in C/C++ second edition. In this chapter I will represent that state chart and present its C++ code as well as the corresponding code in structured text (ST) programming language. I will then implement the ST code in TwinCAT programming environment.

5.1 State Chart of Computer Keyboard Example

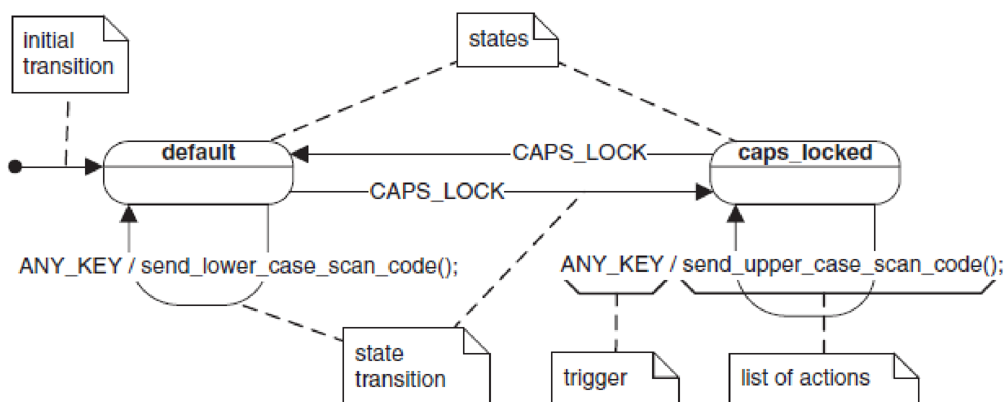


Figure 5. 1 State chart of a Computer Keyboard Example

In the figure [Figure 5. 1 State chart of a Computer Keyboard] above is state chart that I have chosen to base my thesis on. It is the state chart of a computer keyboard state machine. There are two states in this particular chart namely the default state and the caps locked state and they are depicted by the two rounded triangles. Arrows are used to represent configurations and they are accompanied by the events that trigger them and also the actions that are carried out due to these events. The start point of this state machine is depicted by the filled out circle at the beginning of the chart and the initial transition originates from it and goes to the default state which is the lower case alphabets. The transitions between the default and caps locked state are not labelled because they are not triggered by an event.

I will code this state machine chart in C++ as well as structured text (ST) programming language in the following sections.

5.2 Structured Text code with Nested Switch Statement Implementation

//Main Program

```
PROGRAM MAIN                                //DECLARATION
     VAR_INPUT
          caps_btn: BOOL;                   //CAPSLOCK button
          str_in: STRING;                   //input string
     END_VAR
     VAR_OUTPUT
          str_out: STRING: = '';            //output string
     END_VAR
     VAR
          State: DINT: = 0;                 //caps lock state
for state machine
          caps_btn_RTRIG: R_TRIG;           //Rising edge
Trigger for caps lock
          caps_locked: BOOL: = 0;           //CAPS LOCKED if
True
     END_VAR

     //IMPLEMENTATION
     caps_btn_RTRIG (CLK: =caps_btn);

     CASE state OF
          0:                                 //initial state
               state: = 10;
          10:                                //CAPS_OFF
               caps_locked:= 0;
               IF caps_btn_RTRIG.Q THEN
                    state := 20;
               END_IF
          20:                               //CAPS_ON
               caps_locked := 1;
               IF caps_btn_RTRIG.Q THEN
                    state := 10;
               END_IF
     END_CASE

     IF caps_locked THEN
          str_out := upperCase(str_in);
     ELSE
          str_out := lowerCase(str_in);
     END_IF

END_PROGRAM
```

```
//Lower Case Function
FUNCTION lowerCase : STRING              //DECLARATION
     VAR_INPUT
          str_in : STRING := '';         //input string
     END_VAR
     VAR
          new_str: STRING := '';         //private VAR
for string modification
          i: DINT;                       //counter
     END_VAR

     //IMPLEMENTATION
     FOR i:= 0 TO SIZEOF(str_in)-1 DO
          IF (str_in[i] >= 97 AND str_in[i] <= 122) THEN
               new_str[i] := str_in[i];
          ELSIF (str_in[i] >= 65 AND str_in[i] <= 97)
THEN
               //new_str := '';
               new_str[i] := str_in[i] + 32;
               //lowerCase := new_str;
          ELSE
               //new_str := '';
               new_str[i] := str_in[i];
               //lowerCase := new_str;
          END_IF
     END_FOR
     lowerCase := new_str;

END_FUNCTION

//Upper Case Function
FUNCTION upperCase : STRING           //DECLARATION
     VAR_INPUT
          str_in : STRING := '';      //input string
     END_VAR
     VAR
          new_str : STRING := '';     //private VAR for
string modification
          i: DINT;                    //counter
     END_VAR

     //IMPLEMENTATION
     FOR i:= 0 TO SIZEOF(str_in)-1 DO
```

```
            IF (str_in[i] >= 65 AND str_in[i] <= 90) THEN
                 new_str[i] := str_in[i];
            ELSIF (str_in[i] >= 97 AND str_in[i] <= 122)
THEN
                 //new_str := '';
                 new_str[i] := str_in[i] - 32;
                 //upperCase := new_str;
            ELSE
                 //new_str := '';
                 new_str[i] := str_in[i];
                 //upperCase := new_str;
            END_IF
       END_FOR
       upperCase := new_str;

END_FUNCTION
```

5.2.1 TwinCAT Implementation

In the main of the code is the basic structure in which we define the inputs, outputs and the variables. I then defined the three states which are the initial, caps off and caps on states. When the caps button is triggered the output string that we will get will show capital letters and when the machine is in caps off state then the output string will show us lowercase letters as are in the input string.

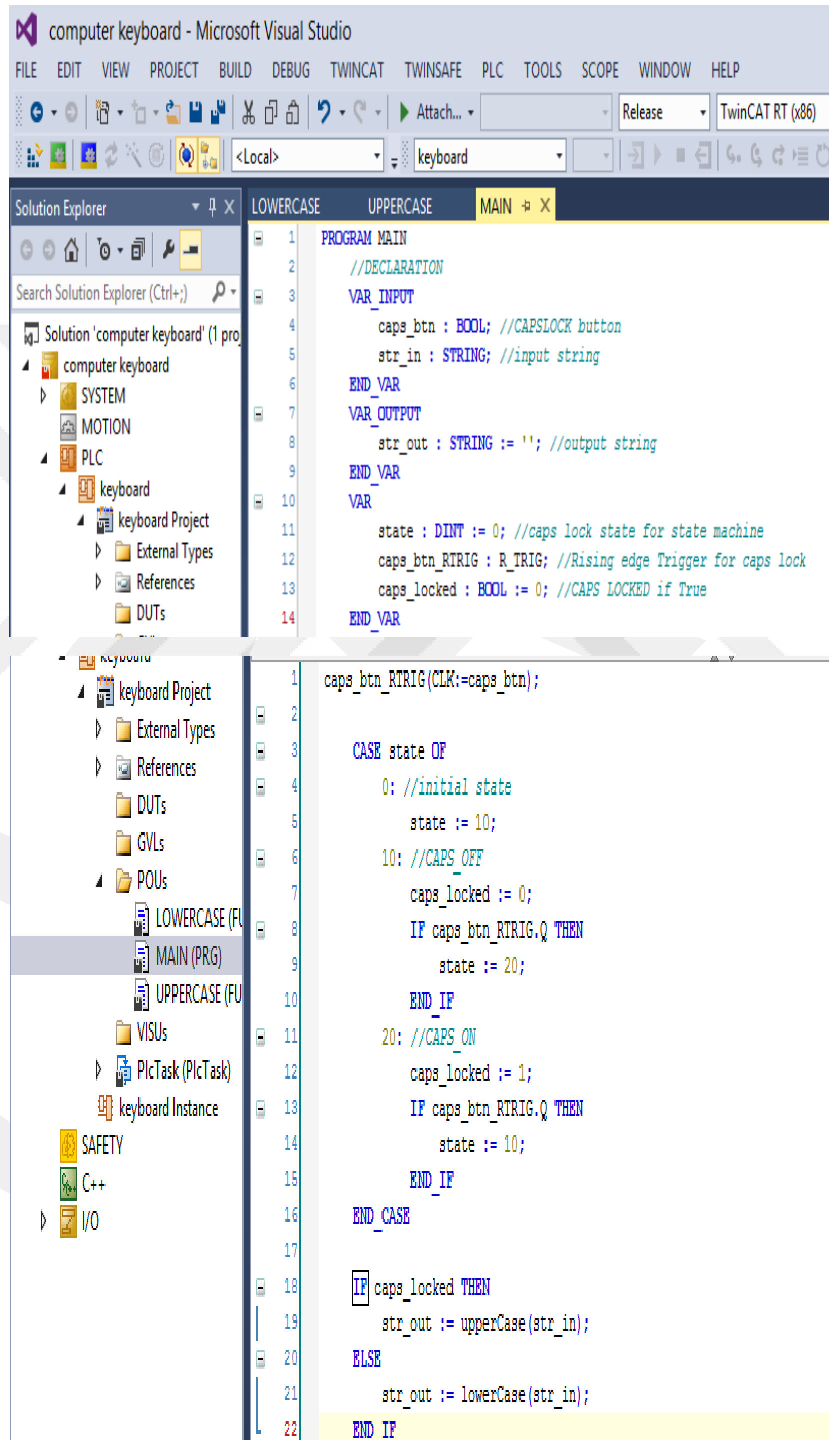


Figure 5. 2 The MAIN of computer Keyboard Example

I then created a programming organizational unit (POU) for the state when the caps lock button is triggered i.e. we get output as capital letters. Shown below [Figure 5. 3 Uppercase state] is the respective implementation in TwinCAT programing environment.

```
FUNCTION upperCase : STRING
    //DECLARATION
    VAR_INPUT
        str_in : STRING := ''; //input string
    END_VAR
    VAR
        new_str : STRING := ''; //private VAR for string modification
        i: DINT; //counter
    END_VAR

FOR i:= 0 TO SIZEOF(str_in)-1 DO
        IF (str_in[i] >= 65 AND str_in[i] <= 90) THEN
            new_str[i] := str_in[i];
        ELSIF (str_in[i] >= 97 AND str_in[i] <= 122) THEN
            //new_str := '';
            new_str[i] := str_in[i] - 32;
            //upperCase := new_str;
        ELSE
            //new_str := '';
            new_str[i] := str_in[i];
            //upperCase := new_str;
        END_IF
    END_FOR
    upperCase := new_str;
```

Figure 5. 3 Uppercase state

After creating a POU for the uppercase state I then created another POU for the Lowercase state and in this state the string is returned such that we get lowercase letters as shown below [Figure 5. 4 Lowercase State].

```
FUNCTION lowerCase : STRING
  //DECLARATION
  VAR_INPUT
    str_in : STRING := ''; //input string
  END_VAR
  VAR
    new_str: STRING := ''; //private VAR for string modification
    i: DINT; //counter
  END_VAR
```

```
FOR i:= 0 TO SIZEOF(str_in)-1 DO
    IF (str_in[i] >= 97 AND str_in[i] <= 122) THEN
      new_str[i] := str_in[i];
    ELSIF (str_in[i] >= 65 AND str_in[i] <= 97) THEN
      //new_str := '';
      new_str[i] := str_in[i] + 32;
      //lowerCase := new_str;
    ELSE
      //new_str := '';
      new_str[i] := str_in[i];
      //lowerCase := new_str;
    END_IF
  END_FOR
  lowerCase := new_str;
```

Figure 5. 4 Lowercase State

5.3 ST code with Object oriented state design

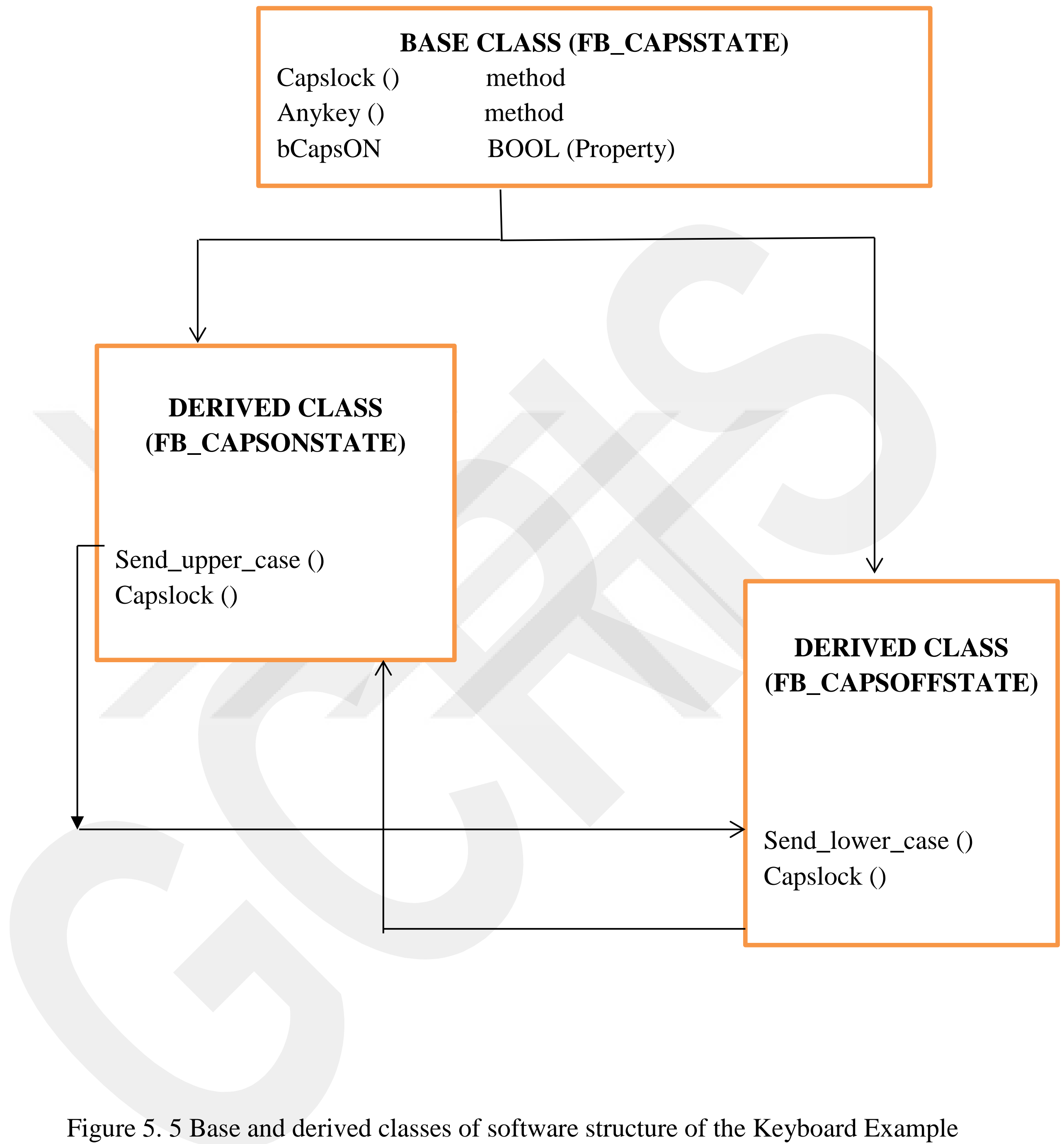
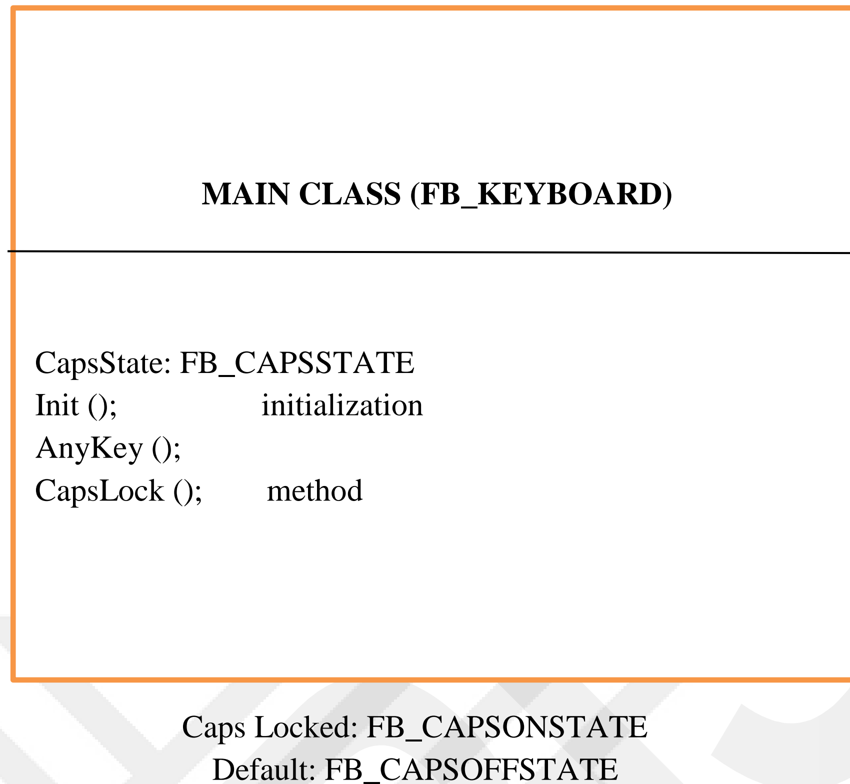


Figure 5. 5 Base and derived classes of software structure of the Keyboard Example



Caps Locked: FB_CAPSONSTATE
Default: FB_CAPSOFFSTATE

Figure 5. 6 Main class of software structure of the Keyboard Example

5.3.1 C++ Implementation of Keyboard Example using Object Oriented State Pattern

```
class Keyboard;
class CapsState {
public:
virtual void onCapsLock (Keyboard *) const {}
virtual void onAnyKey   (Keyboard *, char) const {}
virtual char send_upper_case_scan_code (char data) const
{return toupper(data);}
virtual char send_lower_case_scan_code (char data) const
{return tolower(data);}
};
class CapsONState : public CapsState {
public:
virtual void onCapsLock (Keyboard *context) const;
virtual void onAnyKey   (Keyboard *context, char
input_char) const;
virtual char send_upper_case_scan_code (char data) const;
};
class CapsOFFState : public CapsState {
public:
virtual void onCapsLock (Keyboard *context) const;
```

```
virtual void onAnyKey   (Keyboard *context, char
input_char) const;
virtual char send_lower_case_scan_code (char data) const;
};
class Keyboard {
public:
Keyboard (){};
void init();
void onCapsLock()                      {m_state-
>onCapsLock(this); }
void onAnyKey(char input_char)       {m_state-
>onAnyKey(this, input_char); }
char getChar()                       {return this-
>input_char;}

private:
void tran(CapsState const *target) {m_state = target; }
private:
CapsState const *m_state;
    char input_char;
    bool key_trig;
private:
    static CapsONState  const capslockedState;
    static CapsOFFState const defaultState;
    friend class CapsONState;
    friend class CapsOFFState;
};

//-------------------------------------------------------

CapsONState const Keyboard::capslockedState;
CapsOFFState const Keyboard::defaultState;

void Keyboard::init() {
    tran (&Keyboard::defaultState);
}

//-------------------------------------------------------
----------
void CapsONState::onCapsLock(Keyboard *context) const {
    context->tran(&Keyboard::defaultState);
}
```

```
void CapsONState::onAnyKey(Keyboard *context, char
input_char) const {
    context->input_char =
send_upper_case_scan_code(input_char);
}

char CapsONState::send_upper_case_scan_code(char data)
const
{
    return toupper(data);
}

//-----------------------------------------------------------
-----------

void CapsOFFState::onCapsLock(Keyboard *context) const
{
    context->tran(&Keyboard::capslockedState);
}

void CapsOFFState::onAnyKey(Keyboard *context, char
input_char) const {
    context->input_char =
send_lower_case_scan_code(input_char);
}

char CapsOFFState::send_lower_case_scan_code(char data)
const {
    return tolower(data);
}

//-----------------------------------------------------------
---------

//Testing

using namespace std;
static Keyboard keyboard1;
char get_char;

int main() {
    cout << "Type \"-\" to change Capslock State\n";

    keyboard1.init();
```

```
    for (;;)
    {
        cin >> get_char;
        if (get_char == '-')
         {
            cout << "Capslock state changed\n";
            keyboard1.onCapsLock();
         }
         else
         {
             keyboard1.onAnyKey(get_char);
             cout << keyboard1.getChar();
         }
         if(get_char == '\n'){cout << '\n';}
    }
    return 0;
}
```

5.3.2 Implementation of Object Oriented State Pattern in TwinCat Using ST

5.3.2.1 The Main Routine of the Program

```
PROGRAM MAIN
VAR_INPUT
     input_string : STRING; // Input string
     input_string_trig : BOOL; //Input string trigger

     BackSpace : BOOL := FALSE; //Backsapce Btn
     CapsBtn : BOOL := FALSE; //Capslock btn
     ClearText : BOOL := FALSE; //Clear buffer btn
END_VAR
VAR
     fbKeyboard1 : FB_Keyboard; //Keyboard instance
     output_string : STRING;
     CapsBtn_RTRIG : R_TRIG; //Capslock rising edge
trigger
     input_string_RTRIG : R_TRIG; //Rising edge trigger
for the input string
     BackSpace_RTRIG : R_TRIG; //Rising edge trigger for
the BackSapce
END_VAR
//Implementation
```

```
CapsBtn_RTRIG(CLK:=CapsBtn); //Rising edge input for
Capslock
input_string_RTRIG(CLK:=input_string_trig); //rising edge
input for Input string trig
BackSpace_RTRIG(CLK:=BackSpace);// Backspace btn rising
edge trigger

IF CapsBtn_RTRIG.Q THEN //Check the Capsbtn rising edge
trigger
     fbKeyboard1.Capslock();
END_IF

IF input_string_RTRIG.Q THEN //Check the input string
rising edge trigger
     fbKeyboard1.AnyKeyPress(input_string);
     output_string :=
CONCAT(output_string,fbKeyboard1.PrintString()); //Adding
the last output string to the new output string
END_IF

IF ClearText THEN
     fbKeyboard1.Clear();//Clearing the output string
     output_string := fbKeyboard1.PrintString();
END_IF

IF BackSpace_RTRIG.Q THEN //Check the backsapce rising
edge trigger
     output_string :=
fbKeyboard1.BackSpace(output_string); //removing last
charcter from the string
END_IF
```

// FUNCTION BLOCK CapsOFFState

```
FUNCTION_BLOCK PUBLIC FB_CapsOFFState EXTENDS
FB_CapsState

//This is the extension to the base class FB_CapsState
```

//FB_CapsOFFState.AnyKeyPress

```
METHOD PUBLIC AnyKeyPress
VAR_INPUT
     pContext : POINTER TO FB_Keyboard;
     str_in      : STRING;
END_VAR
```

// Implementation

```
//The AnyKeyPress Method handles any key press
pContext^.SaveString(SendLowerCaseCode(str_in)); //saving the input string to the to the StringOut var as lower case
//FB_CapsOFFState.CapsLock
METHOD PUBLIC Capslock
VAR_INPUT
     pContext : POINTER TO FB_Keyboard;
END_VAR
//Implementation
//CapsLock method handles the capslock event
pContext^.Transition(TRUE); //setting the bCapsIsON var to true
//FB_CapsOFFState.SendLowerCase
METHOD SendLowerCaseCode : STRING
VAR_INPUT
     data : STRING;
END_VAR
VAR
     i : DINT ;  //counter
     data_out : STRING := '';
END_VAR
//Implementation
//This method takes a string and converts it to lowercase ASCII code
FOR i:= 0 TO SIZEOF(data)-1 DO
     IF(data[i] >= 97 AND data[i] <= 122) THEN //Checking the ASCII code for a char if it between these values then its Lower Case
          data_out[i] := data[i];
     ELSIF (data[i] >= 65 AND data[i] <= 97) THEN ////Checking the ASCII code for a char if it between these values then its Upper Case
          data_out[i] := data[i] + 32;
     ELSE
          data_out[i] := data[i];
     END_IF
END_FOR

SendLowerCaseCode := data_out; //returning the string value
FB_CapsONState
FUNCTION_BLOCK PUBLIC FB_CapsONState EXTENDS FB_CapsState
```

FB_CapsONState is derived from the base function block FB_CapsState

```
//FB_CapsONState.AnyKeyPress
METHOD PUBLIC AnyKeyPress
VAR_INPUT
     pContext : POINTER TO FB_Keyboard;
     //pContext : FB_Keyboard;
     str_in      : STRING;
END_VAR
//Implementation
pContext^.SaveString(SendUpperCaseCode(str_in));//saving
the input string to the to the StringOut var as Upper
case
//FB_CapsONState.CapsLock
METHOD PUBLIC Capslock
VAR_INPUT
     pContext : POINTER TO FB_Keyboard;
END_VAR
//Implementation
pContext^.Transition(FALSE); //setting the bCapsIsOn to
False
//FB_CapsONState.SendUpperCase
METHOD SendUpperCaseCode : STRING
VAR_INPUT
     data : STRING;
END_VAR
VAR
     i : DINT ;  //counter
     data_out : STRING := '';
END_VAR
//Implementation
//This method turns lowercase character to Uppercase by
modifying the ASCII code of the character
FOR i:= 0 TO SIZEOF(data)-1 DO
     IF(data[i] >= 65 AND data[i] <= 90) THEN
////Checking the ASCII code for a char if it between
these values then its Upper Case
          data_out[i] := data[i];
     ELSIF (data[i] >= 97 AND data[i] <= 122) THEN
//Checking the ASCII code for a char if it between these
values then its Lower Case
          data_out[i] := data[i] - 32;
     ELSE
```

```
            data_out[i] := data[i];
        END_IF
END_FOR

SendUpperCaseCode := data_out; //returning the string
value
```

FB_CapsState

```
     FUNCTION_BLOCK PUBLIC FB_CapsState IMPLEMENTS
I_CapsState
```

This is our base state function block with which we derive state function blocks.

//FB_CapsState.AnyKeyPress

```
METHOD PUBLIC AnyKeyPress
VAR_INPUT
     pContext : POINTER TO FB_Keyboard;
     str_in      : STRING;
END_VAR
```

This is the base method which is being overwritten by the extension classes.

//FB_CapsState.CapsLock

```
METHOD PUBLIC Capslock
VAR_INPUT
     pContext : POINTER TO FB_Keyboard;
END_VAR
/Base Method which is overwritten by the extension
classes
```

//FB_Keyboard

```
FUNCTION_BLOCK FB_Keyboard
VAR
     refCapsState    : REFERENCE TO FB_CapsState;
     StringOut       : STRING; //Saved string goes in
this var
     //StringIn            : STRING;
     bCapsIsON       : BOOL := FALSE;
     iCapsState           : I_CapsState;
     //---------------------------------------
END_VAR
VAR_STAT
     capslocked_state : FB_CapsONState;
     default_state   : FB_CapsOFFState;
END_VAR
```

```
//The keyboard Function Block handles the Key Press Event
as well as the Capslock Event
//Assiging the Interface to a certain state
//FB_Keyboard.AnyKeyPress
METHOD PUBLIC AnyKeyPress
VAR_INPUT
     str_in : STRING;
END_VAR
//Implementation
StringIn := str_in;
IF bCapsIsON THEN
     capslocked_state.AnyKeyPress(THIS, str_in);
ELSE
     default_state.AnyKeyPress(THIS, str_in);
END_IF
//FB_Keyboard.BackSpace
METHOD BackSpace : STRING
VAR_INPUT
     str_in : STRING;
END_VAR
//Implementation
BackSpace := DELETE(str_in, 1, LEN(str_in));//removing
the last character from the string
//FB_Keyboard.CapsLock
METHOD PUBLIC Capslock
//Implementation
IF bCapsIsON THEN
     fbCapsONState.Capslock(THIS);
ELSE
     fbCapsOFFState.Capslock(THIS);
END_IF
//FB_Keyboard.Clear
METHOD Clear
//Implementation
// Clears the output string buffer
StringOut := '';


//FB_Keyboard.FB_init
METHOD FB_init : BOOL //BOOL is the return type of this
FB which is not needed becauee this fucntion doesn't have
any return value, its BOOL becuase its the Default
VAR_INPUT
```

```
     bInitRetains : BOOL; // if TRUE, the retain
variables are initialized (warm start / cold start)
     bInCopyCode : BOOL;  // if TRUE, the instance
afterwards gets moved into the copy code (online change)
END_VAR
```
//Implementation
```
//This initialization method handles the intial values of
the Keyboard function blocks when its Instantiated
Transition(FALSE);//setting the intial state to CAPS OFF
StringOut := ''; //clearing the string
```
//FB_Keyboard.PrintString
```
 METHOD PrintString : STRING
```
//Implementation
```
//This method returns the string saved in the Keyboard
Object;
PrintString := StringOut;
```
//FB_Keyboard.SaveString
```
METHOD SaveString
VAR_INPUT
     str_in : STRING;
END_VAR
```
//Implementation
```
//Saving the string to the Keyboard Object text buffer
StringOut := str_in;
```

//FB_Keyboard.Transition
```
METHOD PUBLIC Transition
VAR_INPUT
     bCapslocked : BOOL;
END_VAR
```
//Implementation
```
//Method for the CapsState Transitions
bCapsIsOn := bCapsLocked;
```

//Interface
```
Interface I_CapsState
```
//I_CapsState.AnyKeyPress
```
METHOD AnyKeyPress
VAR_INPUT
     pContext : POINTER TO FB_Keyboard;
     str_in      : STRING;
```

```
END_VAR
//I_CapsState.CapsLock
METHOD Capslock
VAR_INPUT
     pContext : POINTER TO FB_Keyboard;
END_VAR
```

5.4 Time Bomb Game Example

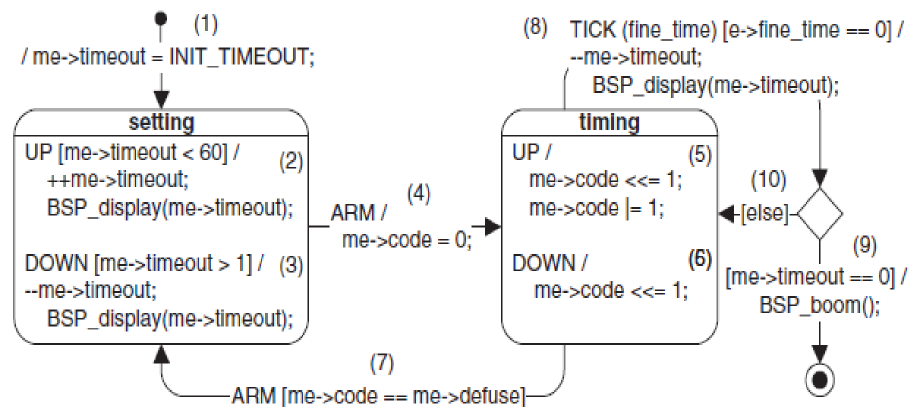


Figure 5. 7 Time Bomb Game State Chart

Following is the C++ implementation of the time-bomb state with the object-oriented State pattern from Samek's [7] book.

```
(1)          class Bomb3;            // context class,
forward declaration
(2)          class BombState {
              public:
(3)          virtual void onUP (Bomb3 *) const {}
              virtual void onDOWN(Bomb3 *) const {}
              virtual void onARM (Bomb3 *) const {}
(4)          virtual void onTICK(Bomb3 *, uint8_t) const
{}
                };
(5)         class SettingState : public BombState {
              public:
              virtual void onUP (Bomb3 *context) const;
              virtual void onDOWN(Bomb3 *context) const;
              virtual void onARM (Bomb3 *context) const;
              };
(6)         class TimingState : public BombState {
              public:
              virtual void onUP (Bomb3 *context) const;
```

```
            virtual void onDOWN(Bomb3 *context) const;
            virtual void onARM (Bomb3 *context) const;
            virtual void onTICK(Bomb3 *context, uint8_t
fine_time) const;
            };
(7)       class Bomb3 {
              public:
(8)         Bomb3(uint8_t defuse) : m_defuse(defuse) {}

(9)          void init(); //
the init() FSM interface

(10)        void onUP ()
{ m_state->onUP (this); }
              void onDOWN()
{ m_state->onDOWN(this); }
              void onARM ()
{ m_state->onARM (this); }
(11)        void onTICK(uint8_t fine_time)     { m_state-
>onTICK(this, fine_time); }
              private:
(12)       void tran(BombState const *target) { m_state =
target; }
              private:
(13)       BombState const *m_state;               //
the state variable
(14)       uint8_t m_timeout;
// number of seconds till explosion
             uint8_t m_code;
// currently entered code to disarm the bomb
             uint8_t m_defuse;
// secret defuse code to disarm the bomb
             private:
(15)      static SettingState const setting;
(16)      static TimingState const timing;
(17)      friend class SettingState;
(18)      friend class TimingState;
            };

//.........................................................
..........  // the initial value of the timeout
             #define INIT_TIMEOUT 10
(19)      SettingState const Bomb3::setting;
(20)       TimingState const Bomb3::timing;
```

```
            void Bomb3::init() {
(21)        m_timeout = INIT_TIMEOUT;
(22)         tran(&Bomb3::setting);
              }

//........................................................
..........
            void SettingState::onUP(Bomb3 *context)
const {
(23)         if (context->m_timeout < 60) {
              ++context->m_timeout;
              BSP_display(context->m_timeout);
                 }
             }
            void SettingState::onDOWN(Bomb3 *context)
const {
             if (context->m_timeout > 1) {
             --context->m_timeout;
             BSP_display(context->m_timeout);
                }
             }
             void SettingState::onARM(Bomb3 *context)
const {
             context->m_code = 0;
(24)        context->tran(&Bomb3::timing); // transition
to "timing"
             }

//........................................................
..........
            void TimingState::onUP(Bomb3 *context) const
{
            context->m_code <<= 1;
             context->m_code j= 1;
             }
            void TimingState::onDOWN(Bomb3 *context)
const {
            context->m_code <<= 1;
            }
           void TimingState::onARM(Bomb3 *context) const
{
           if (context->m_code == context->m_defuse) {
          context->tran(&Bomb3::setting); // transition
to "setting"
```

```
             }
           }
            void TimingState::onTICK(Bomb3 *context,
uint8_t fine_time) const {
(25)       if (fine_time == 0) {
             --context->m_timeout;
             BSP_display(context->m_timeout);
(26)       if (context->m_timeout == 0) {
             BSP_boom();
// destroy the bomb
               }
            }
          }
```

5.4.1 Implementation of Object Oriented State Pattern in TwinCat Using ST

5.4.1.1 The Main Routine of the Program

```
PROGRAM MAIN
VAR_INPUT
     bUP_btn   : BOOL; // Push Button Up
     bDOWN_btn : BOOL; // Push Button Down
     bARM_btn  : BOOL; // ARM Push Button
END_VAR
VAR
     fbBomb : FB_Bomb; // Bomb Instance of  the
class(Function Block) FB_Bomb
     intFine_Time : USINT; //Fine time for Tick Event/
every 10 ticks of fine time is 1 sec
     tTimer         : TON; //Timer

     //Trigs
     UP_RTRIG  : R_TRIG; //Rising edge trigger for button
UP
     DOWN_RTRIG: R_TRIG; //Rising edge trigger for button
DOWN
     ARM_RTRIG : R_TRIG; //Rising edge trigger for button
ARM
     InCode: BYTE; //The input defuse code, this need to
be set while the bomb is timing using UP and Down
buttons, for demostration only
END_VAR
VAR_OUTPUT
     TimeOut                  : USINT; // Bomb count down
END_VAR
//Implementation
```

```
InCode := fbBomb.prop_m_code;

fbBomb(DefuseCode := 4);

UP_RTRIG(CLK:=bUP_btn); //Assigning the Rising egde
trigger to the button
DOWN_RTRIG(CLK:=bDOWN_btn); //Assigning the Rising egde
trigger to the button
ARM_RTRIG(CLK:=bARM_btn); //Assigning the Rising egde
trigger to the button

TimeOut := fbBomb.prop_m_timeout; //Bomb coutdown


IF (fbBomb.prop_ARMED = TRUE) THEN //starting the timer
when the bomb is Armed
          tTimer(IN:=TRUE, PT:=T#100MS);
ELSE
          tTimer(IN:=FALSE, PT:=T#100MS); //stopping the
timer when the bomb is not Armed
END_IF

IF tTimer.Q THEN              // Checking the timer if
reached the Preset Time Value
     tTimer(IN:=FALSE, PT:=T#100MS);    //Reseting the
timer
     fbBomb.onTICK(intFineTime:= intFine_TIme); //setting
the Fine time to the bomb instance
     intFine_Time := intFine_Time + 1; //incrementing the
fine time
END_IF


IF ((intFine_Time + 1) = 10) THEN //Reseting the fine
time if it reaches 10, so it counts from 0 to 9 only
     intFine_Time := 0;
END_IF


IF UP_RTRIG.Q THEN //Checking for pushbutton UP press
     fbBomb.onUP(); //calling the onUP event
END_IF

IF DOWN_RTRIG.Q THEN //Checking for pushbutton DOWN press
```

```
     fbBomb.onDOWN(); //calling the onDOWN event
END_IF

IF ARM_RTRIG.Q THEN //Checking for pushbutton ARM press
     fbBomb.onARM(); //calling the onARM event
END_IF
```

// STATE MACHINE

//1 FB_Bomb

```
FUNCTION_BLOCK FB_Bomb
VAR_INPUT
     DefuseCode : BYTE; //Defuse code set at the start of
the instance
END_VAR
VAR
m_state   :  REFERENCE TO FB_BombState; // A base state
reference for polymorphism, for setting the subclass into
the base class identifier
m_timeout      : USINT; //bomb countdown
m_code         : BYTE; //input of defuse code which needs
to equal m_defuse for the bomb to stop
m_defuse  : BYTE; //defuse code which is needed to defuse
the bomb
bBOOM          : BOOL := FALSE; //Bomb Explodes if TRUE
bARMED         : BOOL := FALSE; //If True bomb
transitions to timeing state, if False bomb transitions
to setting state
iBombState     : I_BombState; //Interface
END_VAR
VAR_STAT
fbSetting : FB_SettingState;
fbTiming  : FB_TimingState;
END_VAR
```

//Implementation

```
prop_m_defuse := DefuseCode; //Assigning the defuse code
to the local variable
```

//2.FB_Bomb.FB_INIT

```
METHOD FB_init : BOOL
VAR_INPUT
     bInitRetains : BOOL; // if TRUE, the retain
variables are initialized (warm start / cold start)
     bInCopyCode : BOOL;  // if TRUE, the instance
afterwards gets moved into the copy code (online change)
END_VAR
```

//Implementation

```
//Initial Timeout in secs
m_timeout := 30;
//Initial State
tran(FALSE); //FALSE is setting state according to the bARMED var
```
//3.FB_Bomb.OnARM
```
METHOD onARM
```
//Implementation
```
m_state.onARM(THIS); //Calling the ARM method depending on which state we are in
```

//4.FB_Bomb.OnDOWN
```
METHOD onDOWN
```
//Implementation
```
m_state.onDOWN(THIS); //Calling the DOWN method depending on which state we are in
```

//5.FB_Bomb.onTICK
```
METHOD onTICK
VAR_INPUT
     intFineTime : USINT;
END_VAR
```
//Implementation
```
m_state.onTICK(THIS, intFineTime); //Calling the TICK method depending on which state we are in
```

//6.FB_Bomb.onUP
```
METHOD onUP
```
//Implementation
```
m_state.onUP(THIS); //Calling the UP method depending on which state we are in
```
//PROPERTIES

//1.FB_Bomb.propARMED
```
PROPERTY prop_ARMED : BOOL  //PROPERTY to get/set the bARMED local var using the FB instance
```
//FB_Bomb.propARMED.GET:
```
prop_ARMED := bARMED;
```
//FB_Bomb.propARMED.SET:
```
bARMED := prop_ARMED;
```

//2.FB_Bomb.propBOOM
```
PROPERTY prop_BOOM : BOOL //PROPERTY to get/set the bBOOM local var using the FB instance
```

```
//FB_Bomb.propBOOM.GET:
prop_BOOM := bBOOM;
//FB_Bomb.propBOOM.SET:
bBOOM := prop_BOOM;
//3.FB_Bomb.prop_m_code
PROPERTY prop_m_code : BYTE //PROPERTY to get/set the
m_code local var using the FB instance
//FB_Bomb.prop_m_code.GET:
prop_m_code := m_code;
//FB_Bomb.prop_m_code.SET:
m_code := prop_m_code;

//4.FB_Bomb.prop_m_difuse
PROPERTY prop_m_defuse : BYTE //PROPERTY to get/set the
m_defuse local var using the FB instance
//F_Bomb.prop_m_difuse.GET:
prop_m_defuse := m_defuse;
//F_Bomb.prop_m_difuse.SET:
m_defuse := prop_m_defuse;

//5.FB_Bomb.prop_m_timeout
PROPERTY prop_m_timeout : BYTE //PROPERTY to get/set the
m_timeout local var using the FB instance
//FB_Bomb.prop_m_timeout.GET:
prop_m_timeout := m_timeout;
//FB_Bomb.prop_m_timeout.SET:
m_timeout := prop_m_timeout
//FB.Bomb.Transition
METHOD tran
VAR_INPUT
     bARMED_Local : BOOL; //Transition value
END_VAR
//Implementation:
//State Transition Method
IF bARMED_Local THEN
m_state REF= fbTiming; //Transition to the Timing state
when bARMED_local is TURE
prop_ARMED := TRUE; //setting the bARMED var
ELSE
m_state REF= fbSetting; //Transition to the Timing state
when bARMED_local is FALSE
prop_ARMED := FALSE; //setting the bARMED var
END_IF
```

```
//FB_BombState
FUNCTION_BLOCK FB_BombState IMPLEMENTS I_BombState
//Base State

//1.FB_bombState.onARM
METHOD onARM
VAR_INPUT
pfbBomb : POINTER TO FB_Bomb;
END_VAR
//2.FB_bombState.onDOWN
METHOD onDOWN
VAR_INPUT
pfbBOMB : POINTER TO FB_Bomb;
END_VAR
//3.FB_BombState.onTICK
METHOD onTICK
VAR_INPUT
pfbBomb : POINTER TO FB_Bomb;
intFineTime : USINT;
END_VAR
//4.FB_BombState.onUP
METHOD onUP
VAR_INPUT
     pfbBomb : POINTER TO FB_Bomb;
END_VAR
VAR
     i : DINT := 0;
END_VAR
//Implemenatation
i := i + 1;
//FB_SettingState
FUNCTION_BLOCK FB_SettingState EXTENDS FB_BombState
//1.FB_SettingState.onARM
METHOD onARM
VAR_INPUT
pfbBomb : POINTER TO FB_Bomb;
END_VAR
//Implementation
pfbBomb^.prop_m_code := 0;//reseting the input defuse
code
pfbBomb^.tran(TRUE); // transition to the Timing state
pfbBomb^.prop_BOOM := FALSE; //setting the bBOOM var
//2.FB_SettingState.onDOWN
METHOD onDOWN
```

```
VAR_INPUT
pfbBomb : POINTER TO FB_Bomb;
END_VAR
//Implementation
IF (pfbBomb^.prop_m_timeout > 1) THEN //Guard condtion
for the timeout, so it can't go below 1 while setting it
     pfbBomb^.prop_m_timeout := pfbBomb^.prop_m_timeout -
1; //decrementing the time out on PushButton Down press
END_IF
//3.FB_SettingState.onUP
METHOD onUP
VAR_INPUT
pfbBomb : POINTER TO FB_Bomb;
END_VAR
//Implementation
IF (pfbBomb^.prop_m_timeout < 60) THEN ////Guard condtion
for the timeout, so it can't go over 60 while setting it
     pfbBomb^.prop_m_timeout := pfbBomb^.prop_m_timeout +
1; //incrementing the timeout on PushButton UP press
END_IF

//FB_TimingState
FUNCTION_BLOCK FB_TimingState EXTENDS FB_BombState
//1.FB_TimingState.onARM
METHOD onARM
VAR_INPUT
pfbBomb : POINTER TO FB_Bomb;
END_VAR
//Implementation
IF (pfbBomb^.prop_m_code = pfbBomb^.prop_m_defuse) THEN
//Checking the input defuse code to disarm the bomb
     pfbBomb^.tran(FALSE); //Transistion to Setting state
if the above condtion is TRUE, Bomb Disarmed
END_IF
//2.FB_TimingState.onDOWN
METHOD onDOWN
VAR_INPUT
pfbBOMB : POINTER TO FB_Bomb;
END_VAR
//Implementation
pfbBomb^.prop_m_code := SHL(pfbBomb^.prop_m_code, 1);
//Inserting 0 when pushbutton DOWN is pressed by shifting
m_code to the left
//3.FB_TimingState.onTICK
```

```
METHOD onTICK
VAR_INPUT
pfbBomb : POINTER TO FB_Bomb;
intFineTime : USINT;
END_VAR
```
//Implementation
```
IF (intFineTime = 0) THEN //Checking the fine time, this
reaches 0 every 1 sec
     pfbBomb^.prop_m_timeout := pfbBomb^.prop_m_timeout -
1; //Decrementing the countdown of the bomb by 1
     IF (pfbBomb^.prop_m_timeout = 0) THEN //Checking if
the coundown reaches 0
          pfbBomb^.prop_BOOM := TRUE; //Bomb Explode
          pfbBomb^.tran(FALSE); //Transition to the
setting state
     END_IF
END_IF
```

//4.FB_TimingState.onUP
```
METHOD onUP
VAR_INPUT
pfbBomb : POINTER TO FB_Bomb;
END_VAR
```
//Implementation
```
pfbBomb^.prop_m_code := SHL(pfbBomb^.prop_m_code,
1);//Shifting m_code to the left by 1 bit
pfbBomb^.prop_m_code := pfbBomb^.prop_m_code OR 1;//
Biwise OR, adding 1 to the least significant bit;
```

5.4.1.2 Visualization of time bomb game example

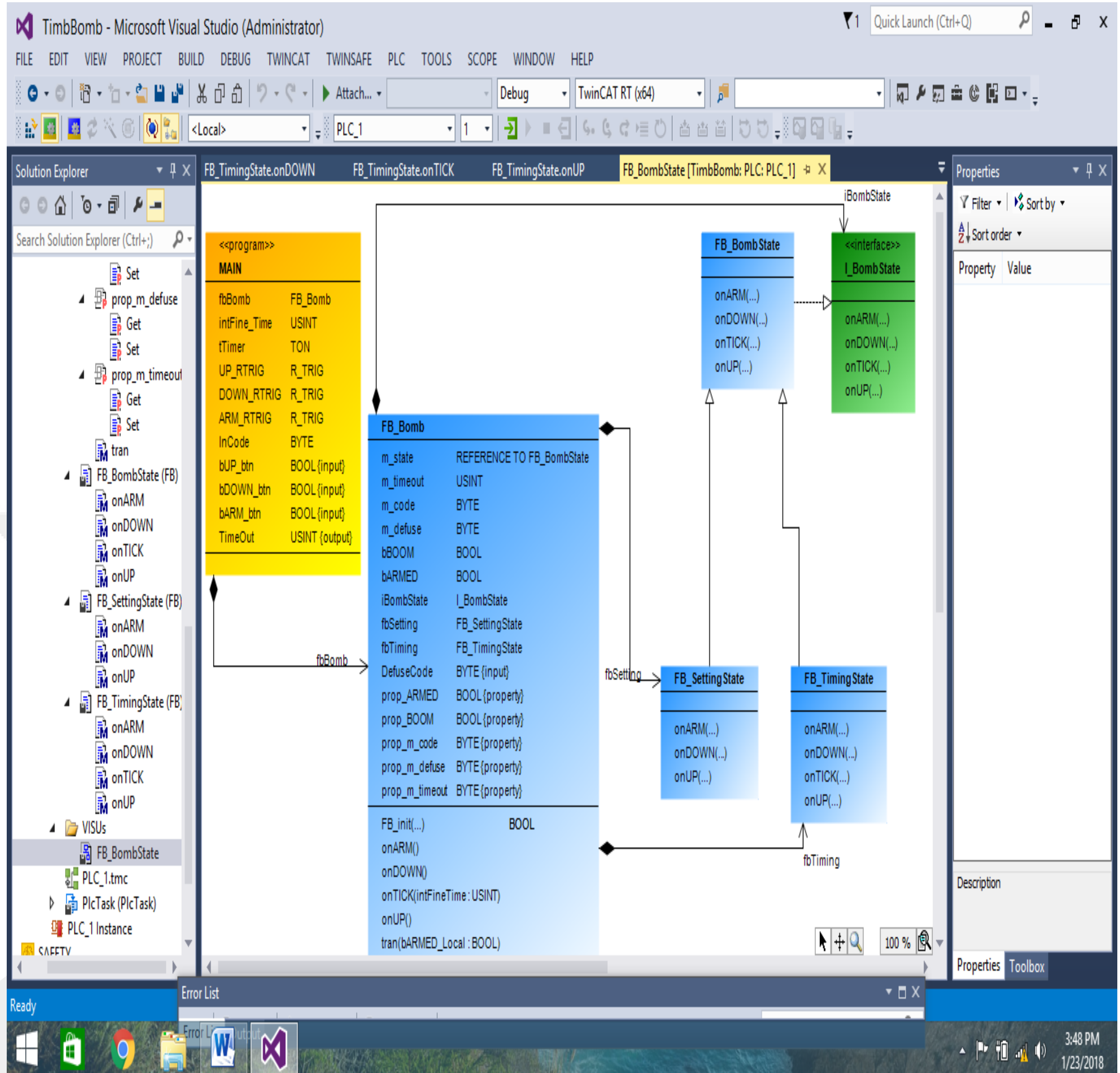


Figure 5. 8 Visualization of time bomb game example

5.5 Simulink Model of Time bomb game

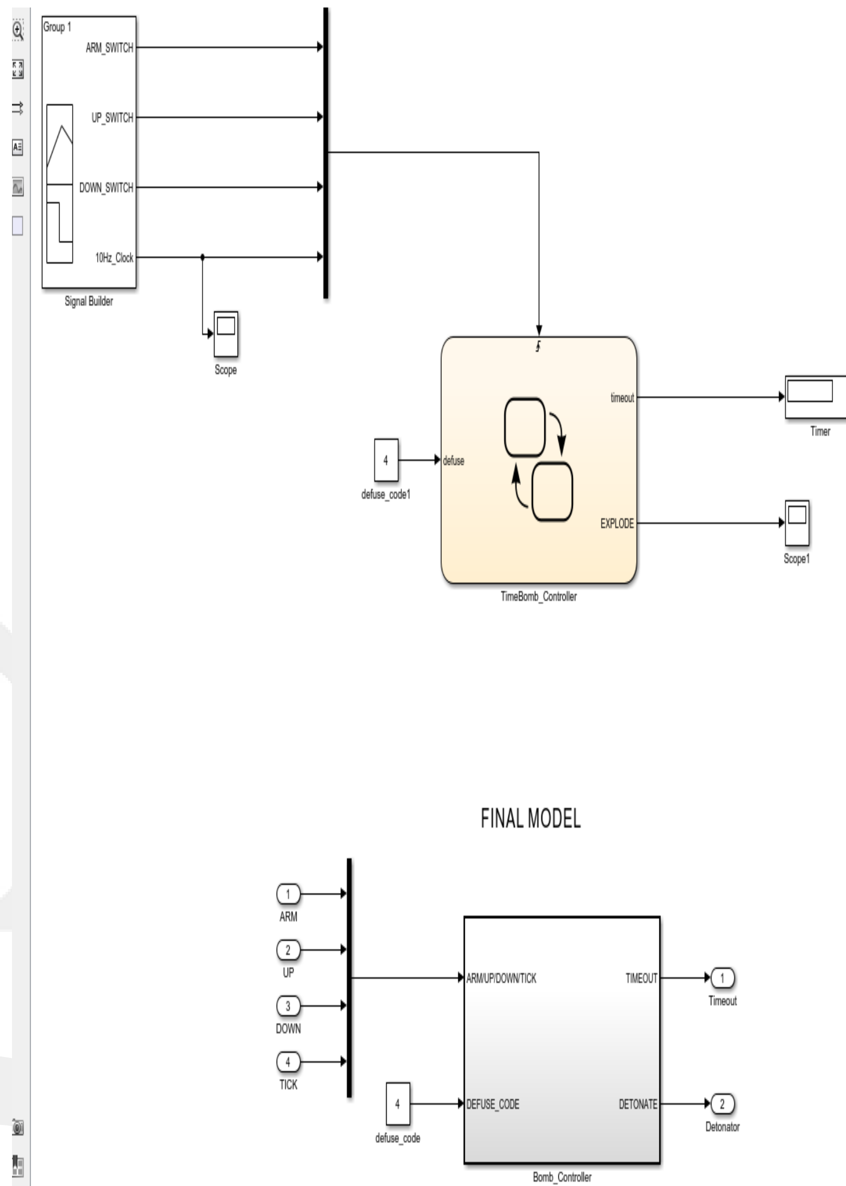


Figure 5. 9 Simulink model of time bomb game

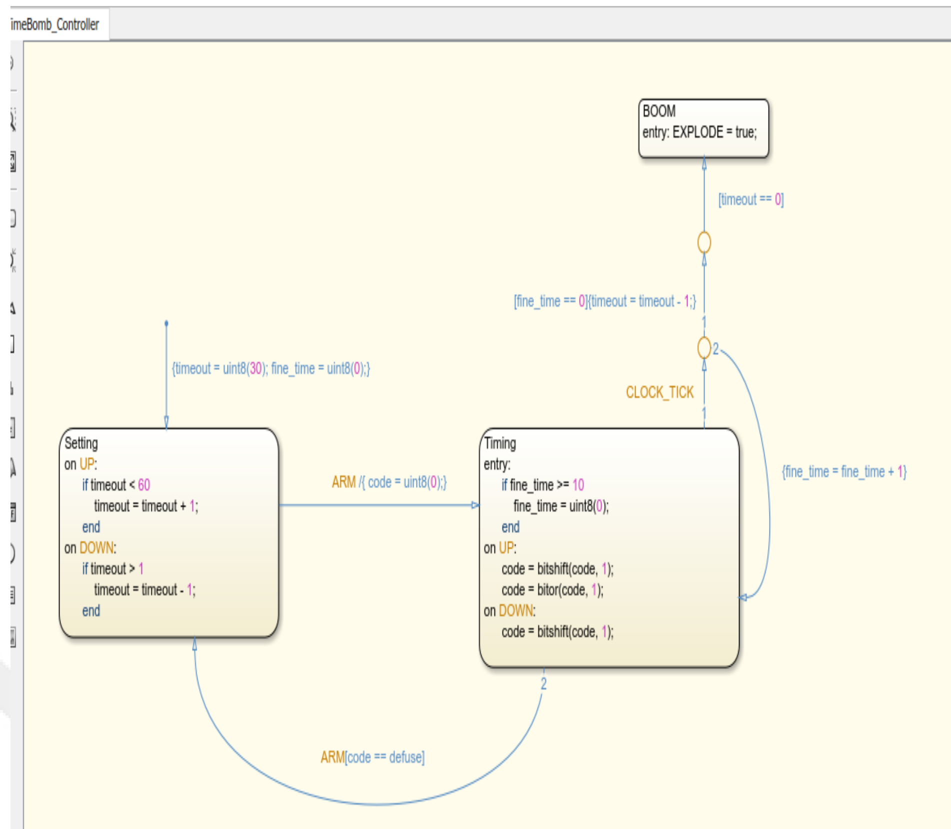


Figure 5. 10 TimBomb_Controller

5.5.1 TwinCAT implementation of Simulink Generated code for time bomb game

Main of Code

```
//PROGRAM MAIN
VAR_INPUT
	bARM		: BOOL; // ARM Button
	bUP		: BOOL; // UP Button
	bDOWN		: BOOL; // DOWN Button
END_VAR
VAR
	arBombInterface : ARRAY[0..3] OF LREAL; // Input
Martix
	fbBomb	: Bomb_Controller;  // Bomb Function Block
	refTICK	: REFERENCE TO LREAL ; //Reference to the
Tick
	tTimer	: TON; //TICK Timer
	init : SINT := 0; //0 for Initialization 1 for
Operation
END_VAR
```

```
VAR_OUTPUT
     iTimeout: USINT; //Time Out
     bBOOM     : BOOL; // Detonator
END_VAR
```

//IMPLEMENTATION

```
//Since the generated code uses a matrix for inputs with
type LREAL, we need to convert the BOOL inputs from BOOL
to real
arBombInterface[0] := BOOL_TO_REAL(bARM);//ARM bool to
real, first item in the matrix
arBombInterface[1] := BOOL_TO_REAL(bUP); //UP bool to
real, second item in the martix
arBombInterface[2] := BOOL_TO_REAL(bDOWN);// DOWN bool to
real, third item in the matrix

fbBomb(ssMethodType:= init, ARMUPDOWNTICK:=
arBombInterface, DEFUSE_CODE:= 4);// Instantiating the
Bomb_Controler Function block

IF init = 0 THEN // Initialization of the function block
     init := 1;
END_IF

iTimeout := fbBomb.TIMEOUT; //time out of the bomb
bBOOM := fbBomb.DETONATE; //Detonator

refTICK REF= arBombInterface[3];//Reference for the TICK
event
refTick := 0;

tTimer(IN:=TRUE, PT:=T#100MS); // Enabling the TICK timer

IF tTimer.Q THEN // TICK timer, once every 100MS
     refTICK := 1;
     tTimer(IN:=FALSE);
END_IF
```

// Bomb_Controller

```
FUNCTION_BLOCK Bomb_Controller
VAR_INPUT
     ssMethodType: SINT;
     ARMUPDOWNTICK: ARRAY[0..3] OF LREAL;
     DEFUSE_CODE: USINT;
END_VAR
```

```
VAR_OUTPUT
     TIMEOUT: USINT;
     DETONATE: BOOL;
END_VAR
VAR
     callChartStep: DINT;
     sf_internal_predicateOutput: BOOL;
     outState: USINT;
     tempOutEvent_idx_0: DINT;
     tempOutEvent_idx_1: DINT;
     tempOutEvent_idx_2: DINT;
     rtb_inputevents_idx_3: SINT;
     qY: UDINT;
     guard1: BOOL := FALSE;
     guard2: BOOL := FALSE;
     guard3: BOOL := FALSE;
     guard4: BOOL := FALSE;
     i0_ZCFCN_d_RISING: ZCFCN_d_RISING;
     code: USINT;
     fine_time: USINT;
     sfEvent: DINT;
     is_c1_Bomb_Controller: USINT;
     c_TimeBomb_Controller1_Trig: ARRAY[0..3] OF USINT :=
[3, 3, 3, 3];
END_VAR
```

```
//IMPLEMENTATION
CASE ssMethodType OF
0:
(* SystemInitialize for Outport: '<Root>/TIMEOUT'
incorporates:
*  SystemInitialize for Chart:
'<S1>/TimeBomb_Controller1' *)
(* Entry: Bomb_Controller/TimeBomb_Controller1 *)
(* Entry Internal: Bomb_Controller/TimeBomb_Controller1
*)
(* Transition: '<S2>:2' *)
(* '<S2>:2:1' timeout = uint8(5); *)
TIMEOUT := 30;
(* SystemInitialize for Chart:
'<S1>/TimeBomb_Controller1' *)
(* '<S2>:2:1' fine_time = uint8(0); *)
```

```
fine_time := 0;
is_c1_Bomb_Controller := 2;
1:
(* Chart: '<S1>/TimeBomb_Controller1' incorporates:
*  TriggerPort: '<S2>/input events' *)
(* Inport: '<Root>/ARM//UP//DOWN//TICK' *)
i0_ZCFCN_d_RISING(u0 := c_TimeBomb_Controller1_Trig[0],
u1 := ARMUPDOWNTICK[0]);
callChartStep := i0_ZCFCN_d_RISING.y0;
outState := i0_ZCFCN_d_RISING.y1;
c_TimeBomb_Controller1_Trig[0] := outState;
sf_internal_predicateOutput := callChartStep <> 0;
tempOutEvent_idx_0 := callChartStep;
(* Inport: '<Root>/ARM//UP//DOWN//TICK' *)
i0_ZCFCN_d_RISING(u0 := c_TimeBomb_Controller1_Trig[1],
u1 := ARMUPDOWNTICK[1]);
callChartStep := i0_ZCFCN_d_RISING.y0;
outState := i0_ZCFCN_d_RISING.y1;
c_TimeBomb_Controller1_Trig[1] := outState;
sf_internal_predicateOutput :=
sf_internal_predicateOutput OR (callChartStep <> 0);
tempOutEvent_idx_1 := callChartStep;
(* Inport: '<Root>/ARM//UP//DOWN//TICK' *)
i0_ZCFCN_d_RISING(u0 := c_TimeBomb_Controller1_Trig[2],
u1 := ARMUPDOWNTICK[2]);
callChartStep := i0_ZCFCN_d_RISING.y0;
outState := i0_ZCFCN_d_RISING.y1;
c_TimeBomb_Controller1_Trig[2] := outState;
sf_internal_predicateOutput :=
sf_internal_predicateOutput OR (callChartStep <> 0);
tempOutEvent_idx_2 := callChartStep;
(* Inport: '<Root>/ARM//UP//DOWN//TICK' *)
i0_ZCFCN_d_RISING(u0 := c_TimeBomb_Controller1_Trig[3],
u1 := ARMUPDOWNTICK[3]);
callChartStep := i0_ZCFCN_d_RISING.y0;
outState := i0_ZCFCN_d_RISING.y1;
c_TimeBomb_Controller1_Trig[3] := outState;
sf_internal_predicateOutput :=
sf_internal_predicateOutput OR (callChartStep <> 0);
IF sf_internal_predicateOutput THEN
rtb_inputevents_idx_3 := DINT_TO_SINT(callChartStep);
(* Gateway: Bomb_Controller/TimeBomb_Controller1 *)
callChartStep := 0;
IF DINT_TO_SINT(tempOutEvent_idx_0) = 1 THEN
```

```
callChartStep := 1;
sfEvent := 0;
END_IF;
IF callChartStep = 1 THEN
(* During: Bomb_Controller/TimeBomb_Controller1 *)
guard4 := FALSE;
CASE is_c1_Bomb_Controller OF
1:
;
;
2:
(* During 'Setting': '<S2>:1' *)
(* '<S2>:4:1' sf_internal_predicateOutput = ... *)
(* '<S2>:4:1' ARM; *)
IF sfEvent = 0 THEN
(* Transition: '<S2>:4' *)
(* '<S2>:4:1' code = uint8(0); *)
code := 0;
is_c1_Bomb_Controller := 3;
(* Entry 'Timing': '<S2>:3' *)
(* '<S2>:3:1' if fine_time >= 10 *)
IF fine_time >= 10 THEN
(* '<S2>:3:1' fine_time = uint8(0); *)
fine_time := 0;
END_IF;
ELSE
(* '<S2>:1:1' sf_internal_predicateOutput = ... *)
(* '<S2>:1:1' UP; *)
IF (sfEvent = 1) AND (TIMEOUT < 60) THEN
(* '<S2>:1:1' if timeout < 60 *)
(* '<S2>:1:1' timeout = timeout + 1; *)
callChartStep := UDINT_TO_DINT(USINT_TO_UDINT(TIMEOUT) +
1);
IF DINT_TO_UDINT(callChartStep) > 255 THEN
callChartStep := 255;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := DINT_TO_USINT(callChartStep);
END_IF;
(* '<S2>:1:1' sf_internal_predicateOutput = ... *)
(* '<S2>:1:1' DOWN; *)
IF (sfEvent = 2) AND (TIMEOUT > 1) THEN
(* '<S2>:1:4' if timeout > 1 *)
(* '<S2>:1:4' timeout = timeout - 1; *)
```

```
qY := USINT_TO_UDINT(TIMEOUT) - 1;
IF qY > USINT_TO_UDINT(TIMEOUT) THEN
qY := 0;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := UDINT_TO_USINT(qY);
END_IF;
END_IF;
ELSE
(* During 'Timing': '<S2>:3' *)
(* '<S2>:33:1' sf_internal_predicateOutput = ... *)
(* '<S2>:33:1' CLOCK_TICK; *)
IF sfEvent = 3 THEN
(* Transition: '<S2>:33' *)
(* '<S2>:59:1' sf_internal_predicateOutput = ... *)
(* '<S2>:59:1' fine_time == 0; *)
IF fine_time = 0 THEN
(* Transition: '<S2>:59' *)
(* '<S2>:59:1' timeout = timeout - 1; *)
qY := USINT_TO_UDINT(TIMEOUT) - 1;
IF qY > USINT_TO_UDINT(TIMEOUT) THEN
qY := 0;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := UDINT_TO_USINT(qY);
(* '<S2>:38:1' sf_internal_predicateOutput = ... *)
(* '<S2>:38:1' timeout == 0; *)
IF TIMEOUT = 0 THEN
(* Transition: '<S2>:38' *)
is_c1_Bomb_Controller := 1;
(* Outport: '<Root>/DETONATE' *)
(* Entry 'BOOM': '<S2>:37' *)
(* '<S2>:37:1' EXPLODE = true; *)
DETONATE := TRUE;
ELSE
guard4 := TRUE;
END_IF;
ELSE
guard4 := TRUE;
END_IF;
ELSE
(* Inport: '<Root>/DEFUSE_CODE' *)
(* '<S2>:30:1' sf_internal_predicateOutput = ... *)
(* '<S2>:30:1' (ARM) &amp;&amp; (code == defuse); *)
```

```
IF (sfEvent = 0) AND (code = DEFUSE_CODE) THEN
sf_internal_predicateOutput := TRUE;
ELSE
sf_internal_predicateOutput := FALSE;
END_IF;
IF sf_internal_predicateOutput THEN
(* Transition: '<S2>:30' *)
is_c1_Bomb_Controller := 2;
ELSE
(* '<S2>:3:1' sf_internal_predicateOutput = ... *)
(* '<S2>:3:1' UP; *)
IF sfEvent = 1 THEN
(* '<S2>:3:3' code = bitshift(code, 1); *)
code := code * 2;
(* '<S2>:3:4' code = bitor(code, 1); *)
code := BYTE_TO_USINT(USINT_TO_BYTE(code) OR 16#1);
END_IF;
(* '<S2>:3:3' sf_internal_predicateOutput = ... *)
(* '<S2>:3:3' DOWN; *)
IF sfEvent = 2 THEN
(* '<S2>:3:4' code = bitshift(code, 1); *)
code := code * 2;
END_IF;
END_IF;
END_IF;
END_CASE;
IF guard4 THEN
(* Transition: '<S2>:36' *)
(* '<S2>:36:1' fine_time = fine_time + 1 *)
callChartStep := UDINT_TO_DINT(USINT_TO_UDINT(fine_time)
+ 1);
IF DINT_TO_UDINT(callChartStep) > 255 THEN
callChartStep := 255;
END_IF;
fine_time := DINT_TO_USINT(callChartStep);
is_c1_Bomb_Controller := 3;
(* Entry 'Timing': '<S2>:3' *)
(* '<S2>:3:1' if fine_time >= 10 *)
IF fine_time >= 10 THEN
(* '<S2>:3:1' fine_time = uint8(0); *)
fine_time := 0;
END_IF;
END_IF;
END_IF;
```

```
callChartStep := 0;
IF DINT_TO_SINT(tempOutEvent_idx_1) = 1 THEN
callChartStep := 1;
sfEvent := 1;
END_IF;
IF callChartStep = 1 THEN
(* During: Bomb_Controller/TimeBomb_Controller1 *)
guard3 := FALSE;
CASE is_c1_Bomb_Controller OF
1:
;
;
2:
(* During 'Setting': '<S2>:1' *)
(* '<S2>:4:1' sf_internal_predicateOutput = ... *)
(* '<S2>:4:1' ARM; *)
IF sfEvent = 0 THEN
(* Transition: '<S2>:4' *)
(* '<S2>:4:1' code = uint8(0); *)
code := 0;
is_c1_Bomb_Controller := 3;
(* Entry 'Timing': '<S2>:3' *)
(* '<S2>:3:1' if fine_time >= 10 *)
IF fine_time >= 10 THEN
(* '<S2>:3:1' fine_time = uint8(0); *)
fine_time := 0;
END_IF;
ELSE
(* '<S2>:1:1' sf_internal_predicateOutput = ... *)
(* '<S2>:1:1' UP; *)
IF (sfEvent = 1) AND (TIMEOUT < 60) THEN
(* '<S2>:1:1' if timeout < 60 *)
(* '<S2>:1:1' timeout = timeout + 1; *)
callChartStep := UDINT_TO_DINT(USINT_TO_UDINT(TIMEOUT) +
1);
IF DINT_TO_UDINT(callChartStep) > 255 THEN
callChartStep := 255;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := DINT_TO_USINT(callChartStep);
END_IF;
(* '<S2>:1:1' sf_internal_predicateOutput = ... *)
(* '<S2>:1:1' DOWN; *)
IF (sfEvent = 2) AND (TIMEOUT > 1) THEN
```

```
(* '<S2>:1:4' if timeout > 1 *)
(* '<S2>:1:4' timeout = timeout - 1; *)
qY := USINT_TO_UDINT(TIMEOUT) - 1;
IF qY > USINT_TO_UDINT(TIMEOUT) THEN
qY := 0;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := UDINT_TO_USINT(qY);
END_IF;
END_IF;
ELSE
(* During 'Timing': '<S2>:3' *)
(* '<S2>:33:1' sf_internal_predicateOutput = ... *)
(* '<S2>:33:1' CLOCK_TICK; *)
IF sfEvent = 3 THEN
(* Transition: '<S2>:33' *)
(* '<S2>:59:1' sf_internal_predicateOutput = ... *)
(* '<S2>:59:1' fine_time == 0; *)
IF fine_time = 0 THEN
(* Transition: '<S2>:59' *)
(* '<S2>:59:1' timeout = timeout - 1; *)
qY := USINT_TO_UDINT(TIMEOUT) - 1;
IF qY > USINT_TO_UDINT(TIMEOUT) THEN
qY := 0;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := UDINT_TO_USINT(qY);
(* '<S2>:38:1' sf_internal_predicateOutput = ... *)
(* '<S2>:38:1' timeout == 0; *)
IF TIMEOUT = 0 THEN
(* Transition: '<S2>:38' *)
is_c1_Bomb_Controller := 1;
(* Outport: '<Root>/DETONATE' *)
(* Entry 'BOOM': '<S2>:37' *)
(* '<S2>:37:1' EXPLODE = true; *)
DETONATE := TRUE;
ELSE
guard3 := TRUE;
END_IF;
ELSE
guard3 := TRUE;
END_IF;
ELSE
(* Inport: '<Root>/DEFUSE_CODE' *)
```

```
(* '<S2>:30:1' sf_internal_predicateOutput = ... *)
(* '<S2>:30:1' (ARM) &amp;&amp; (code == defuse); *)
IF (sfEvent = 0) AND (code = DEFUSE_CODE) THEN
sf_internal_predicateOutput := TRUE;
ELSE
sf_internal_predicateOutput := FALSE;
END_IF

IF sf_internal_predicateOutput THEN
(* Transition: '<S2>:30' *)
is_c1_Bomb_Controller := 2;
ELSE
(* '<S2>:3:1' sf_internal_predicateOutput = ... *)
(* '<S2>:3:1' UP; *)
IF sfEvent = 1 THEN
(* '<S2>:3:3' code = bitshift(code, 1); *)
code := code * 2;
(* '<S2>:3:4' code = bitor(code, 1); *)
code := BYTE_TO_USINT(USINT_TO_BYTE(code) OR 16#1);
END_IF;
(* '<S2>:3:3' sf_internal_predicateOutput = ... *)
(* '<S2>:3:3' DOWN; *)
IF sfEvent = 2 THEN
(* '<S2>:3:4' code = bitshift(code, 1); *)
code := code * 2;
END_IF;
END_IF;
END_IF;
END_CASE;
IF guard3 THEN
(* Transition: '<S2>:36' *)
(* '<S2>:36:1' fine_time = fine_time + 1 *)
callChartStep := UDINT_TO_DINT(USINT_TO_UDINT(fine_time)
+ 1);
IF DINT_TO_UDINT(callChartStep) > 255 THEN
callChartStep := 255;
END_IF;
fine_time := DINT_TO_USINT(callChartStep);
is_c1_Bomb_Controller := 3;
(* Entry 'Timing': '<S2>:3' *)
(* '<S2>:3:1' if fine_time >= 10 *)
IF fine_time >= 10 THEN
(* '<S2>:3:1' fine_time = uint8(0); *)
fine_time := 0;
```

```
END_IF;
END_IF;
END_IF;
callChartStep := 0;
IF DINT_TO_SINT(tempOutEvent_idx_2) = 1 THEN
callChartStep := 1;
sfEvent := 2;
END_IF;
IF callChartStep = 1 THEN
(* During: Bomb_Controller/TimeBomb_Controller1 *)
guard2 := FALSE;
CASE is_c1_Bomb_Controller OF
1:
;
;
2:
(* During 'Setting': '<S2>:1' *)
(* '<S2>:4:1' sf_internal_predicateOutput = ... *)
(* '<S2>:4:1' ARM; *)
IF sfEvent = 0 THEN
(* Transition: '<S2>:4' *)
(* '<S2>:4:1' code = uint8(0); *)
code := 0;
is_c1_Bomb_Controller := 3;
(* Entry 'Timing': '<S2>:3' *)
(* '<S2>:3:1' if fine_time >= 10 *)
IF fine_time >= 10 THEN
(* '<S2>:3:1' fine_time = uint8(0); *)
fine_time := 0;
END_IF;
ELSE
(* '<S2>:1:1' sf_internal_predicateOutput = ... *)
(* '<S2>:1:1' UP; *)
IF (sfEvent = 1) AND (TIMEOUT < 60) THEN
(* '<S2>:1:1' if timeout < 60 *)
(* '<S2>:1:1' timeout = timeout + 1; *)
callChartStep := UDINT_TO_DINT(USINT_TO_UDINT(TIMEOUT) +
1);
IF DINT_TO_UDINT(callChartStep) > 255 THEN
callChartStep := 255;
END_IF;

(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := DINT_TO_USINT(callChartStep);
```

```
END_IF;
(* '<S2>:1:1' sf_internal_predicateOutput = ... *)
(* '<S2>:1:1' DOWN; *)
IF (sfEvent = 2) AND (TIMEOUT > 1) THEN
(* '<S2>:1:4' if timeout > 1 *)
(* '<S2>:1:4' timeout = timeout - 1; *)
qY := USINT_TO_UDINT(TIMEOUT) - 1;
IF qY > USINT_TO_UDINT(TIMEOUT) THEN
qY := 0;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := UDINT_TO_USINT(qY);
END_IF;
END_IF;
ELSE
(* During 'Timing': '<S2>:3' *)
(* '<S2>:33:1' sf_internal_predicateOutput = ... *)
(* '<S2>:33:1' CLOCK_TICK; *)
IF sfEvent = 3 THEN
(* Transition: '<S2>:33' *)
(* '<S2>:59:1' sf_internal_predicateOutput = ... *)
(* '<S2>:59:1' fine_time == 0; *)
IF fine_time = 0 THEN
(* Transition: '<S2>:59' *)
(* '<S2>:59:1' timeout = timeout - 1; *)
qY := USINT_TO_UDINT(TIMEOUT) - 1;
IF qY > USINT_TO_UDINT(TIMEOUT) THEN
qY := 0;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := UDINT_TO_USINT(qY);
(* '<S2>:38:1' sf_internal_predicateOutput = ... *)
(* '<S2>:38:1' timeout == 0; *)
IF TIMEOUT = 0 THEN
(* Transition: '<S2>:38' *)
is_c1_Bomb_Controller := 1;
(* Outport: '<Root>/DETONATE' *)
(* Entry 'BOOM': '<S2>:37' *)
(* '<S2>:37:1' EXPLODE = true; *)
DETONATE := TRUE;
ELSE
guard2 := TRUE;
END_IF;
ELSE
```

```
guard2 := TRUE;
END_IF;
ELSE
(* Inport: '<Root>/DEFUSE_CODE' *)
(* '<S2>:30:1' sf_internal_predicateOutput = ... *)
(* '<S2>:30:1' (ARM) &amp;&amp; (code == defuse); *)
IF (sfEvent = 0) AND (code = DEFUSE_CODE) THEN
sf_internal_predicateOutput := TRUE;
ELSE
sf_internal_predicateOutput := FALSE;
END_IF;
IF sf_internal_predicateOutput THEN
(* Transition: '<S2>:30' *)
is_c1_Bomb_Controller := 2;
ELSE
(* '<S2>:3:1' sf_internal_predicateOutput = ... *)
(* '<S2>:3:1' UP; *)
IF sfEvent = 1 THEN
(* '<S2>:3:3' code = bitshift(code, 1); *)
code := code * 2;
(* '<S2>:3:4' code = bitor(code, 1); *)
code := BYTE_TO_USINT(USINT_TO_BYTE(code) OR 16#1);
END_IF;
(* '<S2>:3:3' sf_internal_predicateOutput = ... *)
(* '<S2>:3:3' DOWN; *)
IF sfEvent = 2 THEN
(* '<S2>:3:4' code = bitshift(code, 1); *)
code := code * 2;
END_IF;
END_IF;
END_IF;
END_CASE;
IF guard2 THEN
(* Transition: '<S2>:36' *)
(* '<S2>:36:1' fine_time = fine_time + 1 *)
callChartStep := UDINT_TO_DINT(USINT_TO_UDINT(fine_time)
+ 1);
IF DINT_TO_UDINT(callChartStep) > 255 THEN
callChartStep := 255;
END_IF;
fine_time := DINT_TO_USINT(callChartStep);
is_c1_Bomb_Controller := 3;
(* Entry 'Timing': '<S2>:3' *)
(* '<S2>:3:1' if fine_time >= 10 *)
```

```
IF fine_time >= 10 THEN
(* '<S2>:3:1' fine_time = uint8(0); *)
fine_time := 0;
END_IF;
END_IF;
END_IF;
callChartStep := 0;
IF rtb_inputevents_idx_3 = 1 THEN
callChartStep := 1;
sfEvent := 3;
END_IF;
IF callChartStep = 1 THEN
(* During: Bomb_Controller/TimeBomb_Controller1 *)
guard1 := FALSE;
CASE is_c1_Bomb_Controller OF
1:
;
;
2:
(* During 'Setting': '<S2>:1' *)
(* '<S2>:4:1' sf_internal_predicateOutput = ... *)
(* '<S2>:4:1' ARM; *)
IF sfEvent = 0 THEN
(* Transition: '<S2>:4' *)
(* '<S2>:4:1' code = uint8(0); *)
code := 0;
is_c1_Bomb_Controller := 3;
(* Entry 'Timing': '<S2>:3' *)
(* '<S2>:3:1' if fine_time >= 10 *)
IF fine_time >= 10 THEN
(* '<S2>:3:1' fine_time = uint8(0); *)
fine_time := 0;
END_IF;
ELSE
(* '<S2>:1:1' sf_internal_predicateOutput = ... *)
(* '<S2>:1:1' UP; *)
IF (sfEvent = 1) AND (TIMEOUT < 60) THEN
(* '<S2>:1:1' if timeout < 60 *)
(* '<S2>:1:1' timeout = timeout + 1; *)
callChartStep := UDINT_TO_DINT(USINT_TO_UDINT(TIMEOUT) +
1);
IF DINT_TO_UDINT(callChartStep) > 255 THEN
callChartStep := 255;
END_IF;
```

```
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := DINT_TO_USINT(callChartStep);
END_IF;
(* '<S2>:1:1' sf_internal_predicateOutput = ... *)
(* '<S2>:1:1' DOWN; *)
IF (sfEvent = 2) AND (TIMEOUT > 1) THEN
(* '<S2>:1:4' if timeout > 1 *)
(* '<S2>:1:4' timeout = timeout - 1; *)
qY := USINT_TO_UDINT(TIMEOUT) - 1;
IF qY > USINT_TO_UDINT(TIMEOUT) THEN
qY := 0;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := UDINT_TO_USINT(qY);
END_IF;
END_IF;
ELSE
(* During 'Timing': '<S2>:3' *)
(* '<S2>:33:1' sf_internal_predicateOutput = ... *)
(* '<S2>:33:1' CLOCK_TICK; *)

IF sfEvent = 3 THEN
(* Transition: '<S2>:33' *)
(* '<S2>:59:1' sf_internal_predicateOutput = ... *)
(* '<S2>:59:1' fine_time == 0; *)
IF fine_time = 0 THEN
(* Transition: '<S2>:59' *)
(* '<S2>:59:1' timeout = timeout - 1; *)
qY := USINT_TO_UDINT(TIMEOUT) - 1;
IF qY > USINT_TO_UDINT(TIMEOUT) THEN
qY := 0;
END_IF;
(* Outport: '<Root>/TIMEOUT' *)
TIMEOUT := UDINT_TO_USINT(qY);
(* '<S2>:38:1' sf_internal_predicateOutput = ... *)
(* '<S2>:38:1' timeout == 0; *)
IF TIMEOUT = 0 THEN
(* Transition: '<S2>:38' *)
is_c1_Bomb_Controller := 1;
(* Outport: '<Root>/DETONATE' *)
(* Entry 'BOOM': '<S2>:37' *)
(* '<S2>:37:1' EXPLODE = true; *)
DETONATE := TRUE;
ELSE
```

```
guard1 := TRUE;
END_IF;
ELSE
guard1 := TRUE;
END_IF;
ELSE
(* Inport: '<Root>/DEFUSE_CODE' *)
(* '<S2>:30:1' sf_internal_predicateOutput = ... *)
(* '<S2>:30:1' (ARM) &amp;&amp; (code == defuse); *)
IF (sfEvent = 0) AND (code = DEFUSE_CODE) THEN
sf_internal_predicateOutput := TRUE;
ELSE
sf_internal_predicateOutput := FALSE;
END_IF;
IF sf_internal_predicateOutput THEN
(* Transition: '<S2>:30' *)
is_c1_Bomb_Controller := 2;
ELSE
(* '<S2>:3:1' sf_internal_predicateOutput = ... *)
(* '<S2>:3:1' UP; *)
IF sfEvent = 1 THEN
(* '<S2>:3:3' code = bitshift(code, 1); *)
code := code * 2;
(* '<S2>:3:4' code = bitor(code, 1); *)
code := BYTE_TO_USINT(USINT_TO_BYTE(code) OR 16#1);
END_IF;
(* '<S2>:3:3' sf_internal_predicateOutput = ... *)
(* '<S2>:3:3' DOWN; *)
IF sfEvent = 2 THEN
(* '<S2>:3:4' code = bitshift(code, 1); *)
code := code * 2;
END_IF;
END_IF;
END_IF;
END_CASE;
IF guard1 THEN
(* Transition: '<S2>:36' *)
(* '<S2>:36:1' fine_time = fine_time + 1 *)
callChartStep := UDINT_TO_DINT(USINT_TO_UDINT(fine_time)
+ 1);
IF DINT_TO_UDINT(callChartStep) > 255 THEN
callChartStep := 255;
END_IF;
fine_time := DINT_TO_USINT(callChartStep);
```

```
is_c1_Bomb_Controller := 3;
(* Entry 'Timing': '<S2>:3' *)
(* '<S2>:3:1' if fine_time >= 10 *)
IF fine_time >= 10 THEN
(* '<S2>:3:1' fine_time = uint8(0); *)
fine_time := 0;
END_IF;

END_IF;

END_IF;

END_IF;
END_CASE
```

// ZCFCN_d_RISING

```
//FUNCTION_BLOCK ZCFCN_d_RISING
VAR_INPUT
     u0: USINT;
     u1: LREAL;
END_VAR
VAR_OUTPUT
     y0: DINT;
     y1: USINT;
END_VAR
VAR
     rising: BOOL;
     guard1: BOOL;
     guard2: BOOL;
     guard3: BOOL;
     guard4: BOOL;
     guard5: BOOL;
     guard6: BOOL;
END_VAR
```

//IMPLEMENTATION

```
guard6 := FALSE;
guard5 := FALSE;
guard4 := FALSE;
guard3 := FALSE;
guard2 := FALSE;
guard1 := FALSE;
IF u1 > 0.0 THEN
    y1 := 1;
ELSIF u1 < 0.0 THEN
    y1 := 2;
```

```
ELSE
    y1 := 0;
END_IF;
IF u0 = 2 THEN
    IF y1 = 0 THEN
        guard6 := TRUE;
    ELSIF y1 = 1 THEN
        guard6 := TRUE;
    ELSE
        guard5 := TRUE;
    END_IF;
ELSE
    guard5 := TRUE;
END_IF;
IF guard6 THEN
    guard4 := TRUE;
END_IF;
IF guard5 THEN
    IF u0 = 0 THEN
        IF y1 = 1 THEN
            guard4 := TRUE;
        ELSE
            guard3 := TRUE;
        END_IF;
    ELSE
        guard3 := TRUE;
    END_IF;
END_IF;
IF guard4 THEN
    guard2 := TRUE;
END_IF;
IF guard3 THEN
    IF u0 = 101 THEN
        IF y1 = 1 THEN
            guard2 := TRUE;
        ELSE
            guard1 := TRUE;
        END_IF;
    ELSE
        guard1 := TRUE;
    END_IF;
END_IF;
IF guard2 THEN
    rising := TRUE;
```

```
END_IF;
IF guard1 THEN
    rising := FALSE;
END_IF;
IF rising THEN
    y0 := 1;
ELSE
    y0 := 0;
END_IF;
IF (u0 = 2) AND (y1 = 0) THEN
    y1 := 100;
END_IF;
```

CHAPTER 6.
DISCUSSION OF THE RESULTS AND CONCLUSIONS

In Samek's approach I noticed that he has used the feature of typecasting a lot available in C++ with the help of which variables are converted from one data type to another data type. Some of the inbuilt typecast functions available in C++ are shown in the following table [Table 6. 1 Inbuilt typecast functions in C++].

Typecast	Function
Itoa()	Long to string
itoa()	Integer to string
atol()	String to long
atoi()	String to integer
atof()	String to float

Table 6. 1 Inbuilt typecast functions in C++

Samek tried to overcome the restraints which are put by C++ for conversion of function pointer types by applying typecasting. Miro's approach led to a lot of state handlers being created which were useful mostly just once in the code.

Samek's approach also carries out the implementation of transition actions before the implementation of entry and exit functions which contradicts the UML rule stating that entry and exit actions should be carried out before transition actions.

The current state in Miro Samek's implementation is accompanied by function pointers that mirror the state handler functions of that particular state, by doing so we notice that the state handler functions also have to execute entry and exit actions. To achieve this Samek had to use reserved event codes, and the transitions then lead to a lot of function calls through function pointers.

To overcome these hurdles Heinzmann [5] put forward another approach for coding state charts in C++. He made use of the class templates in C++ to represent states. One of the biggest drawbacks to this approach is that not all compilers are compatible for a template hence by using templates in a code it makes the code less compatible as well.

When templates are used in code, the compilers have to generate separate code for every template type so this causes the code to increase exponentially (code bloating). The use of templates in complex systems tends to increase the build time as well.

It is also hard to debug a code which includes templates because usually the compiler replaces the template thus making it really hard for the debugger to locate the error in

the code. If a change is supposed to be made then the whole project has to be rebuilt as templates are located in headers.

By following object oriented state pattern approach using structured text in TwinCAT programming environment makes coding a state machine chart more structured and easy to understand. We follow a structured approach in this manner by following steps such as

1. Constructing the structure of the software
2. Implementation of the concept of software
3. Instantiation of the elements of the program
4. Allocation of function block instances to an interface instance
5. Use of a function block instance via an interface instance
6. Consolidation of submodules to subsystems

Object oriented approach using structured text is beneficial in a lot of aspects regarding programming state machine charts and some of them I have listed below.

Defining an interface ensures that the blocks have access to the attached program elements. The functions in the program are then generalized and their implementation is available through assigning a base class. Function blocks are derived from the base class and these function blocks can further be extended into subclasses with certain variables mandatory for their functionality.

Function blocks integrate the interface, which results in the creation of methods and properties in the interface of the base class. By doing so, we have elements in function blocks and their subclasses which are required by the base class. By using EXTENDS we make the function block a derivative of the base class hence the methods, properties and variables of the base class can be accessed.

Function blocks are instantiated. The interface instance refers function block instances and also the variables can be changed and modified.

6.1 State Design Pattern (Object Oriented State Machine) - Manually Written Code

The state machine for the Time Bomb Game in this technique is implemented using Object Oriented style, where state transition happens by Polymorphism, which will be explained later. First the Base Class which in our case FB_BombState is created with all the Methods for its subclasses, in this class no implementation of method is provided just the declarations, the implementation will be introduced in the subclass derived from this class. So to make our Time Bomb Game functional we are going to need two states a Timing State and a Setting State. The Setting State is created by sub classing the Base Class (FB_BombState) into a child class which we will call

(FB_SettingState) This calls contains all the relevant methods from the base class, with implementation according to the state the bomb is in currently, such methods are (onUP, onARM, onDOWN), hence we don't need to implement the (onTICK) method in this subclass because there is actually no time ticking while setting the bomb. The Timing State is created using similar manners, we make a subclass from the base class, the new class is called (FB_TimingState), as mentioned before, this class is going to have the same methods of the base class but, the implementation of these methods are going to be different from the one in the Setting State class, even though they carry the same name, so that the same function can do different job according the time bomb state. For example, in Timing State the UP button enters the code 1 for disarming purposes, while the same UP button in setting state is going to increase the Timer by 1 sec, same with the other functions. Now let's see how the state transition happens (Polymorphism). This happens when we create the Object from the Base Class (FB_BombState) then we assign the subclass object to the base class object m_state, the base class object which is of type FB_BombState. fbTiming the subclass objects of type FB_TimingState. We can assign fbTiming to the m_state variable by reference even though they have different types, this is called polymorphism m_state REF= fbTiming here we are morphing one class into the other, same thing goes for the SettingState.

6.2 Nested States FSM (Switch/Case Statements) - Simulink Generated

This Time Bomb game state machine is made using Matlab/Simulink, the Simulink generated code is using the Nested States technique, where a couple of Case statements are used with the help of IF statements to make the states. The generated code takes the shape of 2 actual Function Blocks, one of which is the state machine which is called Bomb_Controller, the other one is the ZCFCN_d_RISING, and the latter is there to manage the Rising Edge triggers for the machine. Now let's take a look at the state machine code, we can see there are a couple of case statements, the first one has 2 states, "0" which is the initial state of the bomb, this state is used to initialize the bomb variables, this state should only be entered at the start of the program then transitioning to the second state which is "1", this state has a couple of nested states within itself which are the TimingState and the SettingState, these states are controller using IF statements. Initially we are going to the Setting State this happens automatically because it is specified in the "0" state we talked about earlier "initialization". The Simulink generated Bomb_Controller has 3 inputs (ssMethodType, ARMUPDOWNTICK, DEFUSE_CODE): ssMethodType: this input needs to be 0 at the start of the program then transition to 1 as explained above (0 for initialization, 1 for operation) ARMUPDOWNTICK: this is an array for the inputs ARM, UP, DOWN, TICK respectively. DEFUSE_CODE: The defuse code for the bomb.

Now a comparison will be carried out regarding the memory requirements and speeds of the respective codes for the Time Bomb Game.

6.2.1 Memory Usage of the Manually Generated Code

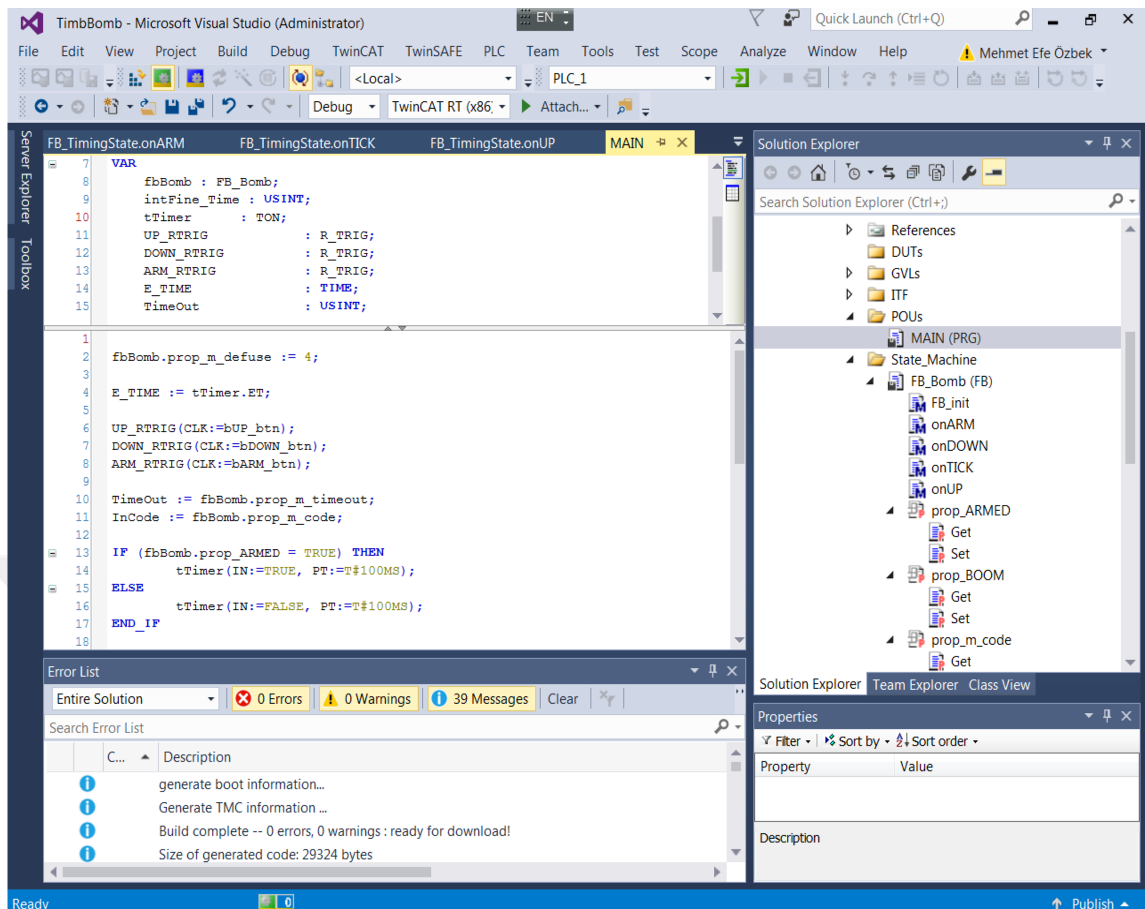


Figure 6. 1Size of manually generated code

Shown in [Figure 6. 1Size of manually generated code] is the manually generated code being tested in TwinCAT to find out its memory requirements? When the code was implemented in TwinCAT, it was found to be error free and also the size of the code was displayed. The size of manually generated code for the time bomb game was found out to be 29324 bytes.

6.2.2 Memory Usage of the Simulink Generated Code

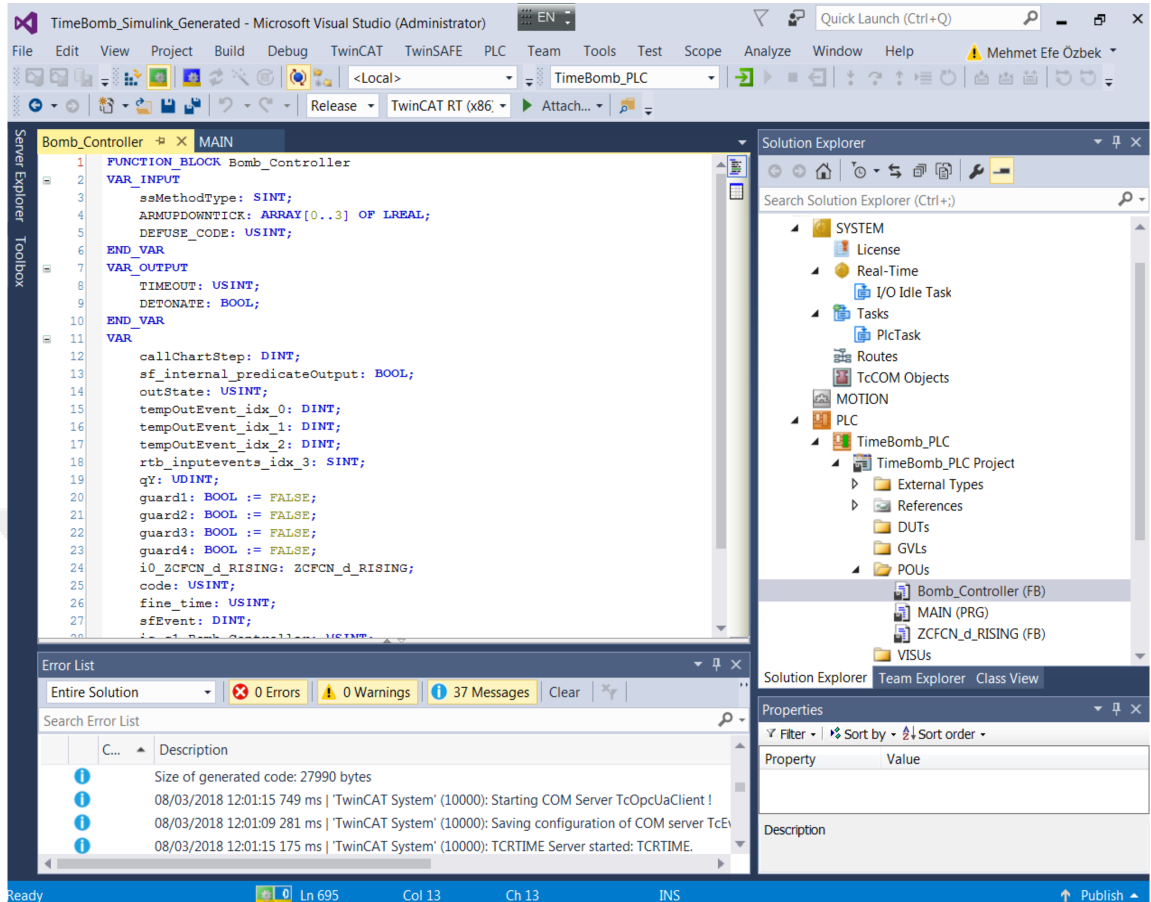


Figure 6. 2 Size of Simulink generated code

Similarly the Simulink generated code was implemented in TwinCAT to find out its memory requirements and also its function ability. It was also error free and the size of Simulink generated code was found out to be 27990 bytes.

6.3 Comparison of the Results

We can see that the simulink generated code takes up less memory then the manually genrated code whereas the speed of manually generated code varies from 8.8 µs to 13.2 µs while that of simulink generated code varies from 17.1 µs to 21 µs. From this data the conclusion drawn is that the manually generated code takes up more memory than the Simulink generated code while the Simulink generated code runs slower than the manually generated code.

References

1. C++ Templates and Classes and its Advantages, d. (2018). C++ TEMPLATES AND CLASSES AND ITS ADVANTAGES, DISADVANTAGES. [online] BrainKart. Available at: http://www.brainkart.com/article/C---Templates-and-Classes-and-its-Advantages--disadvantages_10012/ [Accessed 25 Dec. 2017].
2. Citeseerx.ist.psu.edu. (2018). Cite a Website - Cite This For Me. [online] Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.2016&rep=rep1&type=pdf [Accessed 4 Oct. 2017].
3. Infosys.beckhoff.com. (2018). Beckhoff Information System - English. [online] Available at: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplccontrol/html/TcPlcCtrl_Languages%20ST.htm&id= [Accessed 6 Aug. 2017].
4. Lee, D. and Yannakakis, M. (1994). Testing finite-state machines: state identification and verification. IEEE Transactions on Computers, 43(3), pp.306-320.
5. Members, A. (2018). ACCU :: Yet Another Hierarchical State Machine. [online] Accu.org. Available at: https://accu.org/index.php/journals/252 [Accessed 19 Jan. 2018].
6. PLC Academy. (2018). Structured Text Tutorial For PLC Programmers. [online] Available at: http://www.plcacademy.com/structured-text-tutorial/ [Accessed 10 Oct. 2017].
7. Samek, M. (2009). Practical UML statecharts in C/C++. Amsterdam: Elsevier/Newnes, pp.56-128.
8. Smartdraw.com. (2018). State charts - Everything to Know about State Charts. [online] Available at: https://www.smartdraw.com/state-diagram/#whatisStateDiagram [Accessed 7 Sep. 2017].
9. Systems, D. (2018). PLC Program State Diagram : CASE Structures in Structured Text - Drives & Systems. [online] Drives & Systems. Available at: http://www.drivesandsystems.com/plc-program-sequences/ [Accessed 19 Nov. 2017].