ISSN 0280-5316
ISRN LUTFD2/TFRT--5749--SE

Code Generation from JGrafchart to ATMEL AVR

Ana Llorente

Department of Automatic Control
Lund Institute of Technology
January 2005

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
January 2005

Document Number
ISRNLUTFD2/TFRT--5749--SE

Author(s)
Ana Llorentes

Supervisor
Karl-Erik Årzén at Automatic Control in Lund

Sponsoring organization

Title and subtitle
Code Generation from JGrafchart to ATMEL AVR. (Kodgenerering från JGrafchart till ATMEL AVR).

Abstract

This master thesis has been development at the Department of Automatic Control in Lund University between October 2003 and February 2004 under the supervision of Karl-Erik Årzen.The topic of this master thesis has been to investigate and implement the code generation for an AVR Mega8 microcontroller from JGrafchart, program development by the Department of Automatic Control.This project concerns two items. The first is the code generation for a subset of JGrafchart. The main goal is to obtain a C program that can be compiled for an ATMEL AVR Mega8 processor using the cross-compiler avr-gcc. The obtained program will follow the same execution model as JGrafchart.

The second goal was to achieve a bidirectional communication between the host machine, a PC running JGrafchart, and the target machine, the AVR microcontroller. The on-line communication is necessary in order to provide animation of the execution in the target on the host and to provide user interface possibilities from the host to the target. To achieve the animation, a new execution model will be created in JGrafchart. Both, the code generation and the on-line communication have been development in Java language as part of JGrafchart.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
71

Recipient's notes

Security classification

The report may be ordered from the Department of Automatic Control or borrowed through:University Library, Box 3, SE-221 00 Lund, Sweden Fax +46 46 222 42 43

ACKNOWLEDGMENT

I would like to express my gratitude to Karl Erik Årzen for receiving under his supervision and for his help, always giving me a bit of his time to answer my questions.

Thanks to Bo Lincoln, PhD in the Department of Automatic Control, for answering all the questions about the AVR microcontroller. Without his help it would have been more difficult to understand this device. I would like also to thank Leif Andersson, Johan Åkesson and Anders Blomdell for his help during the development of this master thesis.

I want to thank the entire friends in Sparta for the great time we enjoy and to my parents and brother, Roberto, who always believed and supported me no matter what happened.

I want to dedicate this work to the people who help me in the difficult times during the development of this project, especially at the beginning. Thanks, Iván.

1. INTRODUCTION 3

1.1. Outlines of the report 4

2. JGRAFCHART 5

2.1. Introduction 5
2.2.1. Graphical Function Chart Elements 6
2.2.2. Types and Variables 10
2.2.3. Grafchart Textual Language 10

2.3. JGrafchart Graphical Editor 11

3. ATMEL AVR 13

3.1. General Features 13

3.2. AVR ATMega8 Pin Configuration 14

3.3. Pin Assignments for the AVR serial I/O Board 15

3.4. Interrupts and handlers 15
3.4.1. Overview 15
3.4.2. Periodic execution 17

4. CODE GENERATION 20

4.1. Code Generator 20
4.1.1. Steps and Initial Steps 20
4.1.2. Transitions 22
4.1.3. Variables 24
4.1.4. Input and Output 25

4.2. AVR Program 28
4.2.1. Execution Model 28
4.2.2. AVR Program Structure 30

4.3. Compiling the C Program 31

4.4. AVR Memory Constraints 31

5. COMMUNICATION JGRAFCHART – AVR 33

5.1. Communication Parameters 33

5.2. JGrafchart AVR Mode 33

5.3. Bidirectional Communication ... **35**
5.3.1. Establish communication ... 35
5.3.2. From AVR to JGrafchart ... 37
5.3.3. From JGrafchart to AVR ... 40

6. EXAMPLE ... **43**

6.1. JGrafchart Program ... **43**

6.2. AVR Program ... **44**

7. IMPROVEMENTS AND FUTURE DEVELOPMENT ... **49**

7.1. Improvements ... **49**

7.2. Future Development ... **49**

8. SUMMARY AND CONCLUSIONS ... **50**

9. REFERENCES ... **51**

APPENDIX I: REGISTER DESCRIPTION ... **52**

A) Analog to digital converter ... **52**

B) USART ... **55**

C) TIMER/COUNTER0 ... **60**

D) TIMER/COUNTER1 ... **61**

APPENDIX II: AVR IO PORTS ... **66**

1. INTRODUCTION

An embedded system is a large or small system that is built into a product, a piece of equipment or another computer system, and that performs some task useful to the product, equipment or system. It consists of a computer system that is programmed to perform a particular task. The embedded system is programmed to perform its task from the time it is powered up until it is shut down.

An embedded control system consists of a compact microchip integrated in an electronics package, which is built into a mechanical or electrical device for the purpose of controlling that device.

The use of embedded control systems is growing rapidly. Nowadays we can find advanced control systems in consumer products such as cars, aircrafts, washing machines and home stereo equipment. In our everyday life we more and more depend on computers for assistance. Embedded systems must function at a high degree autonomy, which puts strains on the software since it must be designed not only to handle the most common scenarios but also breakdowns and unexpected failures.

This thesis project concerns two goals. The first one is C code generation for a subset of JGrafchart, a Sequential Function Chart editor and run-time system developed by the Department of Automatic Control in Lund University. The code obtained will be compiled for an ATMEL AVR Mega8 processor. This task allows the user to program the AVR processor using a graphical language. The new features of JGrafchart will facilitate the programming of the device removing all the problems that involve the programming though a low level language. Any user, without any knowledge of the features or architecture of the AVR microcontroller is able to program the chip.

The second approach, relating to the communication between the host machine (a PC running JGrafchart) and the target machine (the AVR processor), provides the possibility to display the execution of the program, that is running in the AVR processor, in JGrafchart. Besides supporting animation, this new mode provides user interface possibilities from the host to the target.

This thesis provides a new feature for JGrafchart that could be consider as a tool to simplify the programming of an AVR microcontroller allowing the execution of models created in JGrafchart in the AVR processor and showing the state of the execution in the graphic. Both parts, the code generation and the communication, are implemented in Java as part of JGrafchart.

1.1. Outline of the report

- Chapter 2 presents JGrafchart giving a brief description of the elements, features and textual language for expressing step actions and transitions. A graphical editor overview is also enclosed.
- Chapter 3 presents the ATMEL AVR Mega8: general features and pin configuration. It also explains how the interrupts and handlers work in the microcontroller. Handler interrupts to control the periodic execution and handler interrupts to read from an analog input are also explained at the end of this chapter.
- Chapter 4 describes how the C code for AVR is generated from the graphical model in JGrafchart. It also includes the execution model and the structure of the generated program. The steps that have to be followed to compile and load the obtained program are detailed.
- Chapter 5 deals with the communication between JGrafchart and ATMEL AVR Mega8. It begins with the communication parameters and then explains the new execution model in JGrafchart created to allow the communication and how this communication takes place: establish the communication, data reception, data transmission, data format…
- Chapter 6 shows a simple example of a JGrafchart program and its corresponding commented generated C code.
- Chapter 7 mentions some possible improvements and further developments of the master thesis.
- Chapter 8 contains the summary.

The report ends with two appendixes. First an explanation of the used AVR registers and its configuration including the meaning of every bit. Appendix II describes how to control the AVR IO ports.

2. JGRAFCHART

2.1. Introduction

JGrafchart is a new Java-based version of Grafchart development by the Department of Automatic Control in the beginning of 2001.

Grafchart is the name of a toolbox for supervisory level sequence control and procedure handling that has been developed at the department since 1991. The original version of Grafchart was developed in G2 from Gensym Corporation. Using this platform Grafchart was used for batch recipe control, diagnosis of mode-changing processes, alarm filtering, implementation of operator decision support systems, and implementation of robot cells.

JGrafchart is a graphical programming language for sequential, procedural and state-transition oriented applications. It is based on ideas from:

- Grafcet/sequential function charts (SFC). A graphical language for representing the sequential behavior of control system. SFC is based on a state-transition formalism similar to what is used in state machines. It shows the states of a system and the transitions between the states.
- Statecharts. Extend the original state machine with properties such as states in hierarchical levels, sub-states executed in concurrent and independent way or events broadcasted among the Statechart diagram.
- Ordinary textual programming languages. The language is state transition oriented.

A JGrafchart program consists of a function chart that represents an activity flow. In a more formal way a Grafcet function chart is a bipartite directed graph consisting of steps and transitions. JGrafchart can be used for all types of discrete-event based applications as logical control, operating procedure management, recipe-based batch control and workflow modeling.

As well as the graphical programming language, JGrafchart is the name of the graphical object editor for this language. The editor consists of a graphical user-interface thought which the user creates, compiles, executes and stores function charts. Therefore, JGrafchart is based on implementation or simulation. The implementation is based in real-time and in connection with the external environment.

JGrafchart is implemented in Java 2 and Swing. It runs in every computer platform that supports this environment. JGrafchart also uses a number of external software components:

- JGo, a class package from Northwoods Corporation that supports the development of model-view-controller based graphical object editors.
- The JavaCC parser generator
- Sun's XML parsers, in order to uses XML as the file storage format.
- Sun's JavaHelp system for on-line help.

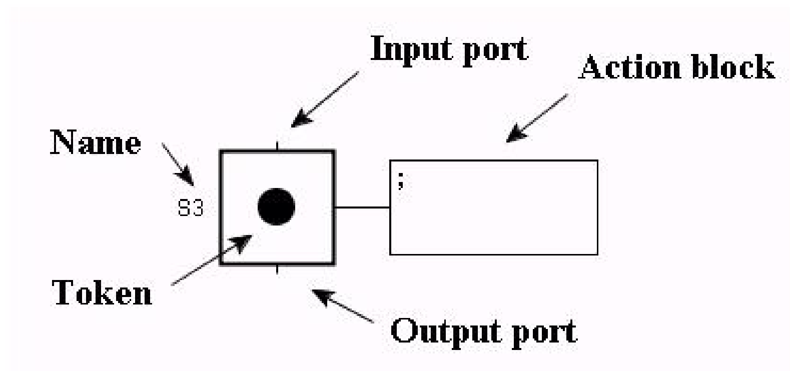
- XMLBlaster, a XML-based message-passing middleware. XMLBlaster is layered on top of Corba and/or XML-RPC. Using XMLBlaster JGrafchart can send and receive XML-structures.2.2. Grafchart Language

2.2.1. Graphical Function Chart Elements

JGrafchart supports the following language elements.

Steps

A step represents a state of the application. A step can be active or inactive. An active step is represented by a token.



The step has one input port and one output port. The input port can be connected to transitions, exception transitions, and parallel split objects. The output port can be connected to transitions and parallel join objects. Step actions are shown in the action block associated with the step.

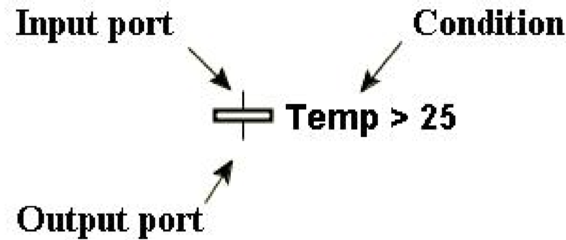
The actions are expressed in a textual action language and can be performed when the step is activated, deactivated, aborted or while the step is active.

There exists a special kind of steps, initial steps. These steps are activated when the execution of the application starts.

Transitions

A transition represents a condition for changing from one state to another. A transition is fired when all the immediately preceding steps connected to the input port of the transition are active and the condition is true.

When the transition fires it’s deactivates all the steps connected at the input port and activates all the steps connected at the output port.



Exception transitions are special transitions that have priority over "ordinary" transitions. This kind of transitions can be used only connected to certain types of steps.

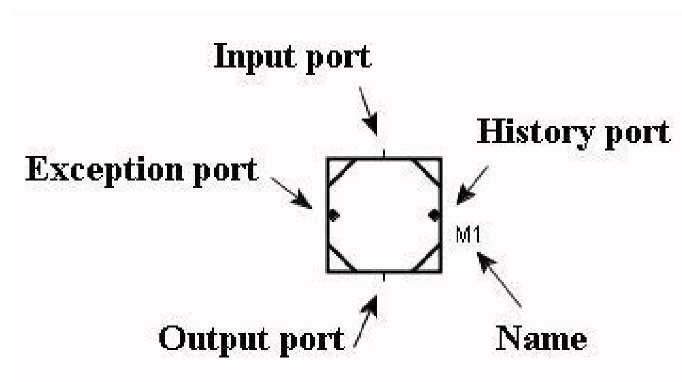
Parallel and Alternative paths

In JGrafchart it is possible to express parallel and alternative paths with parallel split and parallel join objects.

The first one is used to split up the execution in two parallel branches and the second one to merge together the execution in two parallel paths into a single path.

Macro Steps

Macro steps are steps that contain an internal function chart. A macro step represents a hierarchical step containing a substructure of steps and transitions on the subworkspace of the macro step.

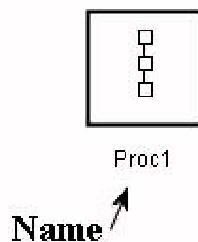


When the macro step is activated the enter step of the internal function chart is activated. When the execution in the internal function chart has reached the exit step, the transition after the macro step is enabled.

An exception transition is a special type of transition that may be connected to a macro step. The exception transition is connected to the exception port on the left-hand side of the macro step. An ordinary transition connected to a macro step does not become enabled until the execution of the macro step has reached the exit step. An exception transition, however, is enabled all the time while the macro step is active. When the transition is fired the execution inside the macro step is aborted and the step succeeding the exception transition becomes activated. Exception transitions have priority over ordinary transitions in cases where both are fireable. Macro steps have history. When an exception transition aborts the execution of a macro step, the execution state is saved. The macro step can be resumed in this saved state if a transition connected to the special history port of the macro step is fired. The history-input port is located on the right hand side of the macro step.

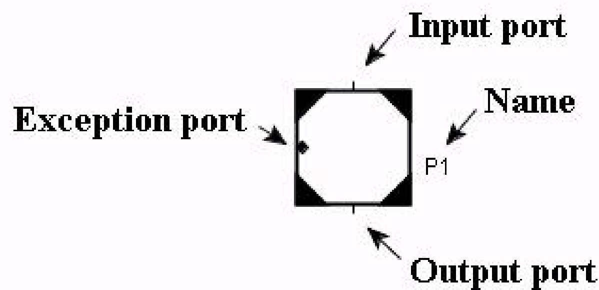
Procedures

Step-transition sequences that are used in several different contexts can be represented as procedures.

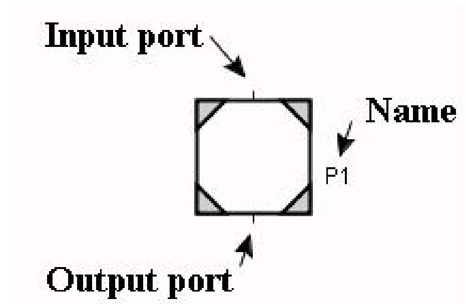


A procedure has a procedure body stored on the subworkspace of the procedure. The body begins with an enter step and ends with an exit step in the same way as for macro steps. Procedures are reentrant and may be recursive. Procedures may have parameters. All internal variables within the procedure can be used as parameters. Parameters can be assigned values both using call-by-value and call-by-reference. A procedure can be called in three different ways:

- Directly from the procedure object.
- Through a procedure step.
- Through a process step.



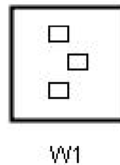
When a procedure is called from a procedure step, the transition(s) after the procedure step does not become enabled until the execution has reached the exit step of the procedure. Hence, this corresponds to an ordinary procedure call.



When a procedure is called from a process step the transition(s) after the procedure step is enabled immediately. Conceptually, the procedure call is spawned and executed in a separate execution thread. The same thing happens when the procedure is called (started) from an action.

Workspace Objects

Workspace objects are objects that contain a subworkspace. The subworkspace may contain arbitrary language elements.



Workspace objects can be used for three main purposes:

- As a way of structuring large applications.
- As a compound variable comparable to a C struct,
- As a way of representing objects that have attributes and methods

When a workspace object is used as a structuring element it is possible to programmatically enable and disable it. It is also possible to have a workspace object being executed at a longer scan-cycle than the rest of the application.

2.2.2. Types and Variables

JGrafchart supports four basic types of variables: integer, boolean, real and string. For each basic type there is a corresponding variable.

The variables are either input variables, output variables or internal variables. Input variables are represented in JGrafchart as Inputs. They are connected to an input channel and receive their values from the external environment. They can be connected to I/O cards, input sockets, etc. Inputs may be digital or analog. Output variables are represented in JGrafchart as Outputs. Output variables at are connected to a output channel and receive their values from JGrafchart and transmit these values to a external environment. They can be connected to I/O cards, output sockets, etc. Outputs may be digital or analog.

2.2.3. Grafchart Textual Language

JGrafchart contains also textual language elements. The textual format is used for expressing step actions and transition conditions.

A step action consists of two parts: an action qualifier and the action. The action qualifier decides when the action should be executed. Five types of qualifiers are available:

- Enter actions (Stored actions) are executed when a step becomes active or when a button is pressed.
- Exit actions are executed once immediately before the step is deactivated.
- Periodic actions are executed periodically while the step is active.
- Abort actions are executed once when a step is aborted due to the firing of an exception transition.
- Normal actions are used to associate the truth-value of a boolean variable or a digital output with the activation status of the corresponding step. The value becomes true when the step become active and becomes false when the step becomes deactivated. If the step becomes deactivated and activated in the same scan-cycle the boolean variable remains true.

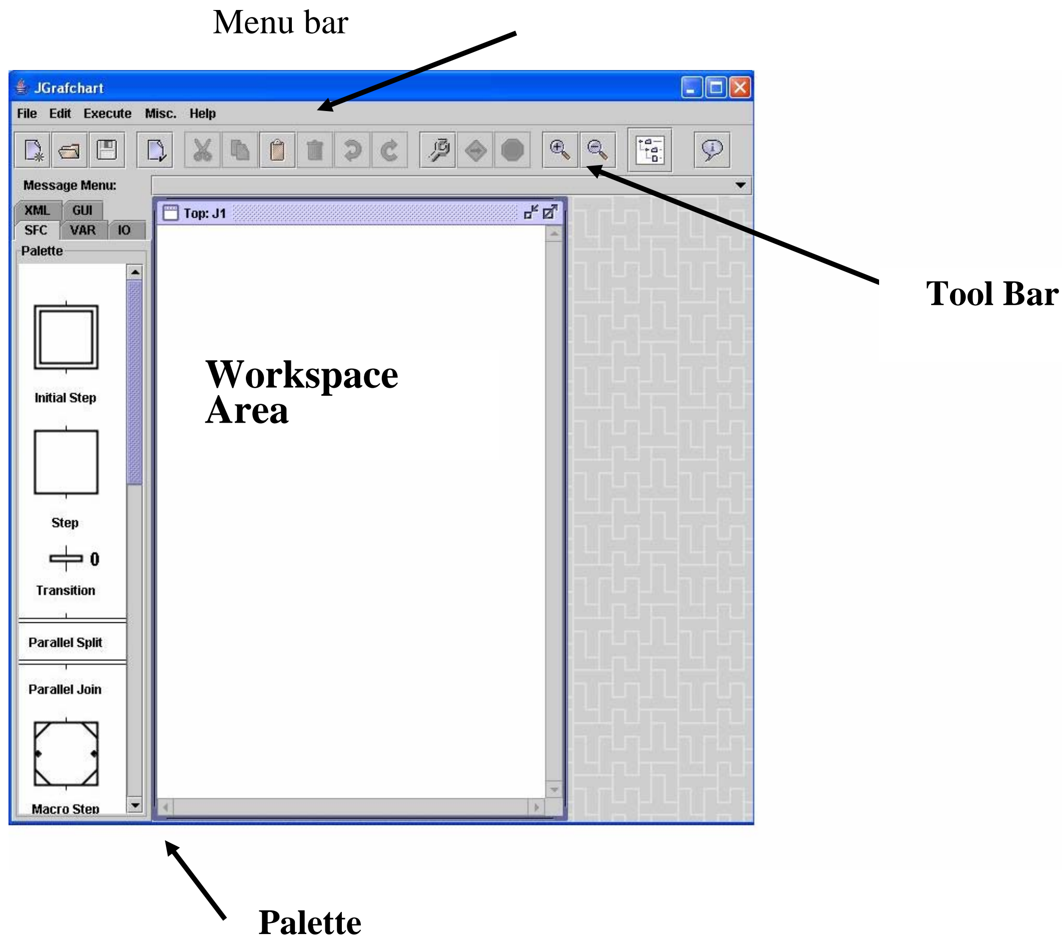
The “action” in the syntax definitions can be of two different types:

- Assignment: “variable expression” = “expression”
- Method call: “object expression”. “methodname”(arg1,…, argn)

The action language is also used to express transition conditions. A transition condition is represented by an expression, which is evaluated, as a boolean expression.

2.3. JGrafchart Graphical Editor

The JGrafchart editor consists of a graphical user-interface through which the user creates, compiles, executes, and stores function charts.



The editor consists of the following main parts:

- The menu bar contains pull-down menus containing menu choices by which the user controls the editor.
- The tool bar contains tool buttons that provide shortcuts to some of the menu choices available from the menus in the menu bar. The toolbar also contains a message menu. The message menu contains compilation error and warning messages. It is also possible for an application to write messages to the message menu and to clear the message menu. The message menu is implemented as a pull-down menu. It works as a stack where new messages are pushed on top of the stack and shown at the top of the menu. The toolbar can be removed and added using menu choices,

- The palette is a five-tabbed pane containing the different language elements in JGrafchart. The user creates an application by drag-and-drop from the palette into a workspace. The five tabs are named: SFC (the default palette), VAR, IO, XML, and GUI. The SFC palette contains the basic JGrafchart language elements, e.g, steps, transitions, procedures, etc. The VAR palette contains JGrafchart variables and lists. The IO menu contains input and output objects. The XML palette contains objects for communication using XML. The objects in this menu assume the availability of the CCOM infrastructure. This is not available in all distributions. The GUI palette contains graphical objects, e.g., texts, rectangles, icons, etc. It also contains plotters, browsers, and buttons. In addition to the drag-and-droppable objects the GUI palette also contains two mode buttons: the line-mode button and the spline-mode button. By clicking on these buttons the mouse-behavior changes from the standard select-object mode to a line-drawing mode. The palette can be removed and added.
- The workspace area is the area where the user application workspaces are shown. The workspaces are implemented Swing internal frames. They can be maximized/minimized, iconized, deleted, scrolled, and panned using standard window operations.

3. ATMEL AVR

The microcontroller chosen for the development in this master thesis is an ATMEL AVR MEGA8 processor. ATMEL´s AVR microcontrollers have RISC architecture, and supports internal oscillators, timers, UART, SPI, pull-up resistors, ADC, analog comparator, and watch-dog timers.

3.1. General Features

ATMEL AVR Mega8 is a low power CMOS 8-bit microcontroller based on the AVR RISC architecture. By executing powerful instructions in a single clock cycle, it achieves throughputs approaching 1MIPS per MHz, allowing the system designer to optimize power consumption versus processing speed.

The AVR has 32 general purpose registers. All the registers are directly connected to the Arithmetic Logic Unit (ALU).

The AVR Architecture has two main memory spaces:

- Program memory space. The program memory space contains 8KByte on-chip reprogrammable flash memory for program storage.
- Data memory space. The data memory space is consists of a 1 KByte SRAM.

In addition, AT Mega8 contains 512 bytes of data EEPROM memory that is organized as a separate data space.

AVR provides 23 general purpose I/O lines. These lines are divided in three 8-bit bidirectional I/O ports.

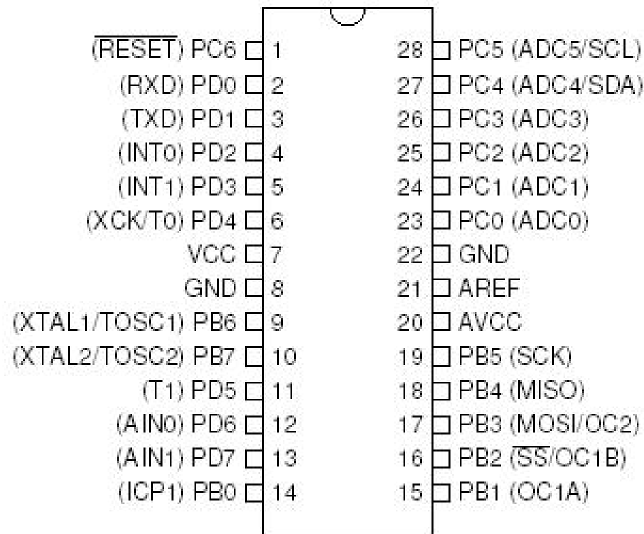
Other main characteristics of the AVR Mega8 are as follows: three flexible timers/counters with compare modes, internal and external interrupts, serial programmable USART, a byte oriented two-wire serial interface, a 6 channel ADC where four channel have 10-bit accuracy and two have 8-bit accuracy. It also includes a programmable watchdog timer with internal oscillator, a SPI serial port, and five software selectable power saving modes.

The Flash Program memory can be reprogrammed through an SPI serial interface, by a conventional non-volatile memory programmer, or by an on-chip boot program running on the AVR core.

By combining an 8-bit RISC CPU with a self-programmable flash on a monolithic chip, the ATMEL Mega8 is a powerful microcontroller that provides a cost effective solution for many embedded control applications.

The Atmel AVR comes with a full suite of programming and system development tools, including C compilers, macro assemblers, program debugger/simulators, in-circuit emulators, and evaluation kits.

3.2. AVR ATMega8 Pin Configuration



Pin Description

- VCC : Digital supply voltage
- GND : Ground
- PORT B: an 8-bit bi-directional I/O port. PB6 and PB7 can be used to perform others functions as input and output of the inverting oscillator amplifier or input for the asynchronous timer/counter2.
- PORT C: a 7-bit bi-directional I/O port. PC6 can be used as reset input.
- PORT D: an 8-bit bi-directional I/O port.
-
- PORT B, PORT C and PORT D are also used for various special features of the ATmega8.
- AVCC : Supply voltage pin for the A/D converter, PORT C (0..3) and ADC (7..6)
- AREF : Analog reference pin for the A/D Converter

3.3. Pin Assignments for the AVR serial I/O Board

The pin assignments in the board used by the Department of Automatic Control are the following:

PortB	XTL	XTL	do2	do1	d05	ao1	ao0	do0
	PB7	**PB6**	**PB5**	**PB4**	**PB3**	**PB2**	**PB1**	**PB0**

PortC			do4	do3	ai1	ai0	ai3	ai2
			PC5	**PC4**	**PC3**	**PC2**	**PC1**	**PC0**

PortD	di5	di4	di3	di2	di1	di0	TXD	RXD
	PD7	**PD6**	**PD5**	**PD4**	**PD3**	**PD2**	**PD1**	**PD0**

As can be seen in the table above this pin configuration allows:

- 6 digital inputs
- 6 digital outputs
- 4 analog inputs
- 2 analog outputs.

The number that identifies the input or the output is associated with the channel in the graphic representation of inputs and outputs in JGrafchart. The code generation assumes the pin assignment above. If this configuration is changed it is necessary to rewrite parts of the code generation to associate the correct pin with the correct IO.

3.4. Interrupts and handlers

This part describes the interrupts and their handlers in AVR. First an overall description of the AVR interrupt vector is given. The following parts deal with the interrupts and handlers used in the AVR C program to ensure periodic execution, read the analog inputs and the serial communication.

3.4.1. Overview

The AVR provides several interrupt sources. These interrupts and the reset vector have a separate program vector in the program memory space. All the interrupts are assigned individual enable bits which must be written a logic one together with the global interrupt enable bit in the status register in order to enable the interrupt.

The complete list of interrupts is shown in the following figure. This list also determines the priority level of each interrupt so RESET has the higher priority.

VECTOR NUMBER	SOURCE	DEFINITION
1	RESET	External Pin, Power-on Reset and Watchdog reset
2	INT0	External Interrupt Request 0
3	INT1	External Interrupt Request 1
4	TIMER2 COMP	Timer/Counter2 Compare Match
5	TIMER2 OVF	Timer/Counter2 Overflow
6	TIMER1 CAPT	Timer/Counter1 Capture Event
7	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	TIMER1 COMPB	Timer/Counter1 Compare Match A
9	TIMER1 OVF	Timer/Counter1 Overflow
10	TIMER0 OVF	Timer/Counter0 Overflow
11	SPI, STC	Serial Transfer Complete
12	USART, RXC	USART, Receive Complete
13	USART, UDRE	USART Data Register Empty
14	USART, TXC	USART, Transmission Complete
15	ADC	ADC Conversion Complete
16	EE_RDY	EEPROM Ready
17	ANA_COMP	Analog Comparator
18	TWI	Two-wire serial interface
19	SPM_RDY	Store Program Memory Ready

The lowest addresses in the Program memory space are by default defined as the Reset and Interrupt vector. Both can be moved to the start of the boot Flash section.

When an interrupt occurs all the interrupts are disabled. However it is possible to enable nested interrupts. This means that all enabled interrupts can interrupt the current interrupt routine. When AVR exits from an interrupt, it will always return to the main program and execute more instruction before any pending interrupt is served.

AVR provides two instructions to enable and disable interrupts: SEI and CLI .

3.4.2. Periodic execution

In order to ensure periodical execution of the AVR program with the correct rate a timer is needed a timer and an interrupt handler for this timer. The timer chosen for this purpose is TIMER/COUNTER0 that is a general purpose, single channel, 8 bit Time/Counter module.

The interrupts of this timer are enabled through the Timer Mask Interrupt (TIMSK) and the interrupt request signal is visible in the Timer Interrupt Flag Register (TIFR).

The counter direction of this timer is always up and the counter simply overruns when it passes its maximum 8-bit value (MAX=0xFF) and restarts from the bottom (0x00). During normal operation the Timer/Counter Overflow interrupt is executed each time the counter becomes zero.

With all of this it is possible to ensure periodic execution of the code by just setting a bit to indicate to the main program that it is time to execute a new scan cycle:

```
while (1){
        while (not bit is set){
        //busy wait
        }
reset bit;
}
```

This bit is set by the interrupt handler when the number of times this handler has been executed corresponds to the number of times indicated by the parameter Thread Sleep Interval given by JGrafchart.

```
    SIGNAL(SIG_OVERFLOW0){
        static int i=0;
        int times=ThreadSleepInterval*7.2;
        i++;
        if (i==times){
                bit=true;
                i=0;
        }
}
```

Here, *times* is determined by the JGrafchart parameter and the number of interrupts per millisecond. This value is calculated through the clock frequency, in this microcontroller 14,7456 Mhz, and the clock source. For this purpose the configuration selected for the clock is preescaler 8. With these values the number of interrupts is 7,2 per ms.

3.4.3. Analog Input

The Atmega8 features a 10-bit successive approximation Analog Digital Converter (ADC). The ADC is connected to an 8-channel analog multiplexer which allows eight voltage inputs.

The analog input channel is selected by writing to the MUX bits in ADMUX. The ADC is enabled by setting the ADC Enable bit ADEN in ADCSRA. The result of the conversion is presented in the ADC Data Registers, ADCH and ADCL. For single ended conversion the result is:

$$ADC = \frac{Vin * 1024}{Vref}$$

where VIN is the voltage on the selected input and VREF the selected voltage reference, in this case 2,56V.

The ADC has its own interrupt, called SIG_ADC, which can be triggered when a conversion completes. This interrupt is used to read the analog inputs. For this purpose, at the beginning of the code the ADC is configured as:

```
ADCMUX=0xC0;
ADCSRA=0xEF;
```

The first line indicates the Voltage Reference Internal 2.56V with external capacitor at AREF pin, ADC conversion in the ADC Data Register (left adjust) and selects the first MUX channel in order to connect the ADC0 analog input to the ADC. The second line configures the ADC to enable the conversion, start the conversion, enable the ADC interrupt and the operating mode Free Running. In this mode the ADC samples and updates the Data Registers continuously.

As was mentioned before the analog inputs are read through the interrupt handler of SIG_ADC. In this handler the values of the Data Registers are read and depending on which MUX channel that is connected, the value is assigned to the variable which has the corresponding channel in JGrafchart. So, if a new analog input is read and the MUX channel selected in this moment is channel 0 (ADC0) the variable updated will be the variable that has channel 2 in JGrafchart.

```
SIGNAL(SIG_ADC){
        static unsigned channel=0;
        short readed;
        readed = ADCL | (ADCH<<8);
        if (channel==2){
                AIn=readed;
        }
        if (channel==1){
                AIn2=readed;
        }
```

```
        channel = (channel + 1 )%4;
        ADMUX = (0xC0) | (channel);
        ADCSRA=0xEF;
}
```

Above is an example of an interrupt handler to read the analog inputs. AIn and AIn2 are integer variables that represent two analog input of JGrafchart whose channels are respectively 0 and 3. Each time the interrupt handler is executed the MUX channel is incremented module 4 in order to pull in all the possible inputs.

4. CODE GENERATION

The first item of this master thesis is automatic C-code generation for a subset of JGrafchart. The goal is to generate a C program and cross-compile it for an ATMEL AVR Mega8 processor. The C code obtained is compiled with the cross-compiler avr-gcc using the avr-libc standard library.

The code generation is integrated within JGrafchart. As a consequence the function in charge of the generation is implemented in Java. The code obtained follows the same execution model as JGrafchart.

4.1. Code Generator

As mentioned before, the code generation is written as a part of JGrafchart. The main function responsible for it is named *codeGenerationAction()* and is contained in the class Editor. This method is responsible for writing the generated code into a file. First, all the code that is common to any AVR program such as type definitions, interrupt handlers, device configuration, etc is generated after this, all the elements in the JGrafchart program are traversed and identified. Once the elements are identified, the proper code is generated. The followings points explain the code that has to be generated for each element. The code will be written in the result file in the correct order at the end of the method *codeGenerationAction()*.

4.1.1. Steps and Initial Steps

Before generating any code for a step or a initial step we have to decide how to represent this element in the AVR C program. Both, steps and initial steps, are represented by a structured variable:

```
struct step{
        int x;
        int newx;
        int t;
}
```

This structure contains three attributes to control the execution: the first one, x, determines if the step is active in the current scan cycle. Newx is true if the step should be active in the next scan cycle. Finally, t contains the number of scan cycles since the step last was activate.

So, each step will be declared as a struct containing these fields and identified by the name that the step has in JGrafchart. Not all steps have a name in a JGrafchart model. In this case a unique name for the step is automatically created in order to identify this structure in the C program. In addition to being declared before use, each step must be initialized. An initial step has to be initialized in such a way that it is active, at the beginning of the execution. Therefore, initial steps should have both, x and newx, as true.

```
struct step S0={1,1,0};
struct step S1={0,0,0};
```

This is an example of declaration and initialization of two steps. The first one, S0, is an initial step.

The state and the timer of the steps have to be updated during the execution. These actions are performed at the end of each cycle. The state of every step is updated by assigning to the attribute x the value contained in the attribute newx:

```
S0.x = S0.newx;
S1.x = S1.newx;
```

The time for every step is also updated at the end of the cycle. If a step is activated in the current cycle, the value of its timer is incremented by one. In other cases the value for the timer is set to zero:

```
S0.t = (S0.t + S0.x) * S0.x;
S1.t = (S1.t + S1.x) * S1.x;
```

Whenever code is generated for a step, the functions to generate the code for periodic and normal actions are called. Both, normal and periodic actions are generated in the same way. First, the condition is written and then the action is extracted from JGrafchart, and put into the code of this condition. When doing this, all the actions of the step are parsed looking for the correct action qualifier: “N” for normal actions and “P” for periodic actions.

The periodic actions are executed when the step is active in the current scan cycle and will remain active. So, the code generated for these actions is:

```
if (S0.x) && (S0.newx){
        //Periodic actions for step S0
}
```

Normal actions are used to associate the truth-value of a boolean variable or a digital output with the activation status of the corresponding step. The value becomes true when the step become active and becomes false when the step becomes deactivated. If the step becomes deactivated and activated in the same scan-cycle the boolean variable remains true. This behavior is obtained by checking the value of the attributes x and newx:

```
if ((S0.x)&&!(S0.newx)){
        Variable or digital output = false;
}
else{
        if (S0.newx){
                Variable or digital output = true;
        }
}
```

During the code generation for initial steps, in addition to generating the code for periodic and normal actions in the same way as forthe other steps, the enter actions for this type of steps have to be generated. These actions are located at the beginning of the C program immediately before the loop that controls the periodic execution. These actions will be executed only when the program begins the execution. The enter actions for an initial step will be generated in the way explained in the following chapter.

The last action performed for a step is to update the vector used to send information from AVR to JGrafchart. The value of the attribute x is written in the proper position of the vector.

4.1.2. Transitions

Similar to steps, transitions are represented as a variable structure. In this case, this struct only contains one field that determines if the transition is marked for fired. The value of this attribute is true if the condition evaluates to true and all the preceding steps are activated.

```
struct transition{
        int markedforfired;
}
```

Whenever a transition is detected, the first task is to declare it. Transitions, in JGrafchart, do not have any identification. Therefore, it is necessary to generate a name in order to identify them. This unique name is formed by the prefix TR and a number. In addition to being declared, the transition must be initialized:

```
struct transition TR0 = {0};
struct transition TR1 = {0};
```

Before generating the code in charge of evaluating the conditions and mark for fired a transition, it is necessary to build two lists containing the succeeding and preceding steps of this transition. So, the method *compileStructure()* of the class *GCTransition* is called. This method looks for the steps connected to the transition and adds them to an array.

When the lists of succeeding and preceding steps are built, it is time to generate the code that evaluates the conditions that are necessary to fire a transition. Besides the transition condition, it is necessary to check all the preceding steps in order to know if they are active. It means the attribute x must be true. If the two requirements evaluate true the transition have to be marked for firing.

```
if (all the preceding steps are active && transition condition = true) {
        transition.markedforfired=true;
}
```

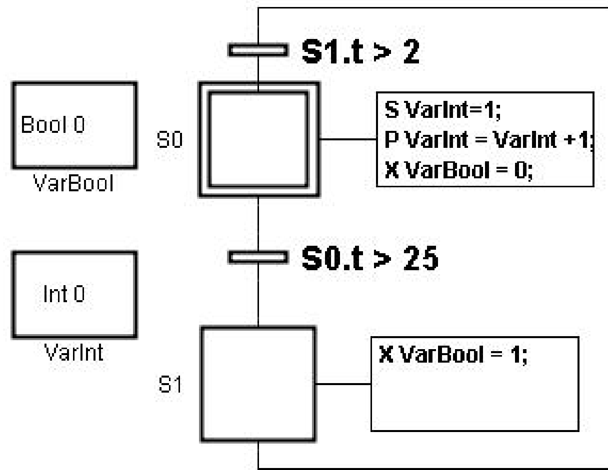


Figure 4.1

The code generated for the example showed in Figure 4.1 will be:

```
if (S0.x && (S0.t>25){
        TR0.markedforfired=true;
}
if (S1.x && (S1.t>2)){
        TR1.markedforfired=true;
}
```

The next task for a transition is to generate the code for a transition that has been marked for firing. The first step of this is to generate the condition to check if the attribute markedforfired is true. When a transition complies with this condition, the following actions have to be generated:

- The code to set the attribute *markedforfire*d to false.
- The attribute newx of all preceding steps have to be set to false and the exit actions of these steps will be generated.

  All the steps belonging to the structure *precedingSteps* generated by the method *compileStructure* are traversed setting their attribute newx to false and parsing their actions looking for the actions with qualifier "X".
- The attribute newx of all succeeding steps will be set true and all their enter actions will be written in the C file.

  All the steps belonging to the structure *succedingSteps* generated by the method *compileStructure* are traversed setting their attribute newx to true and parsing their actions looking for the actions with qualifier "S".

Continuing with the example showed in Figure 4.1, the code generated corresponding to these actions will be:

```
If (TR0.markedforfired){
        TR0.markedforfired = false;
        S0.newx = false;
        VarBool=0;          <- Exit action of step S0
        S1.newx = true;
}

if (TR1.markedforfired){
        TR1.markedforfired = false;
        S1.newx = false;
        VarBool = 1;        <- Exit action of preceding step S1
        S0.newx = true;
        VarInt  = 1;        <- Enter action of succeding step S0
}
```

S0 is the initial step. As was commented in the previous chapter, the enter actions for this kind of step are generated also when the the code for the initial step is generated.

4.1.3. Variables

Concerning variables only code for integer and boolean variables will be generated. This feature is due to the limitations of the AVR microcontroller. So, if the type of a variable is different from integer or boolean a warning window informs the user about this.

During the code generation, when the element detected is identified as an internal variable, the first action to perform is to check the type of the variable: normal or AVR variables. The AVR variables will be explained with more detail in the chapter relating to communication between AVR and JGrafchart. Basically, the main difference between these variables is that the AVR variables will be not sent from AVR to JGrafchart. Then, it is necessary to distinguish the two types of variables in order to add the variable to the proper structure. The way to differentiate the variables is the name: AVR variables are named with the prefix “AVR” follow by the name of the variable.

The vector containing the normal variables will be used by the method in charge of reading the values of the variables from the serial port and animating the graphic, whereas the second structure will be consulted during the generation of the interrupt handler to read the values of the serial port from the AVR program.

Another difference is that for normal variables it is necessary to generate the code to update the vector that contains the value of the variables. This vector will be used to send the variables at the beginning of the cycle.

if (AVRVariable)
Add the variable to AVRReferences2.
else{
Add the variable to AVRReferences.
Update the vector writing the value of the variable.
}
Declare and initialize the variable

Both, AVR variables and normal variables have to be declared and initialized. JGrafchart allows to have variables with the same name. Whenever it occurs, the code generator must detect it and warn to the user about it with a window error reporting that there are two variables with the same name. The code generator is stopped until this error is corrected. Besides, is checked that the name of the variable do not match up with a reserved word in the C language ("int", "volatile", "union",…)

If there is no problem with the names, the variables are declared and initialized. Both, integer and boolean variables are declared as *volatile int*. They are declared as volatile since the variables will be used or updated in several parts of the program. The variables are initialized taking the initial value from JGrafchart. If they have no initial value defined, they are declared with default initial value zero. It means false for the boolean variables.

4.1.4. Input and Output

JGrafchart allows three types of input and output: digital, analog and a special IO in which the top-level workspace can act as the client in a TCP socket connection. This communication is called Socket IO and supports four basic data types. However, only code for analog and digital IO will be generated. In JGrafchart, digital IO has two versions: ordinary and inverse logic.

Both input variables and output variables will be represented in the C program as variables. Before generating the code to declare and to initialize these variables, three verifications have to be done:

- In JGrafchart the name of the elements can be the same, as the variables. So, it is necessary to check for repeated names and give advice about this to the user. This is performed in the same way as for the variables.
- The AVR microcontroller allows a limited number of inputs and outputs (6 digital outputs, 4 analog inputs,) so, during the code generation; every time an input or output is detected it is necessary to check if the maximum number of this type of elements has been exceeded. This event is communicated to the user through a window containing a message.
- It is necessary to check that two or more IO variables of the same type do not have the same channel (e.g. two digital input reading from the channel 1)

When these verifications have been checked, it is time to declare and initialize the variables. All are declared as *int* and only the analog input will be volatile. This is due to the analog input will be read through an interrupt handler and then these variables will be used in several parts of the C program. Both analog input and analog output in JGrafchart are real variables with double value, however in the AVR program they will be represented as integers which take values from 0 to 1024 for input voltages between -10 V and 10 V.

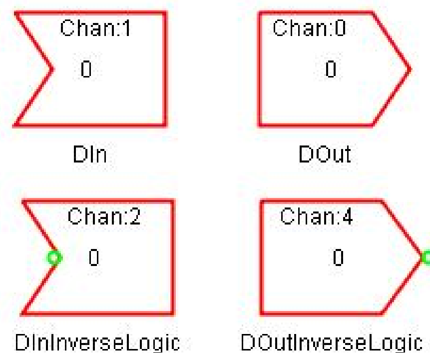
Inputs are initialized with the values taken from JGrafchart whereas the outputs are initialized with the value zero since these elements cannot be changed by the user.

Next, the code corresponding to the reading and the writing will be explained in more detail. In both cases the range of the channels must be checked. If the channel for an input or an output taken from JGrafchart is not in the range of channels allowed the user receives a message containing the range allowed.

Digital Input and Digital Output

AVR allows 6 digital outputs and 6 digital inputs, so the range of channels allowed is from 0 to 5 in both cases. As mentioned before there are two possibilities for digital IO: ordinary and inverse logic.

The functions that generate the code for these purposes take the channel from JGrafchart and then they identify if the digital IO is ordinary or inverse logic. Using the channel the method is able to assign the IO variable with the correct pin following the pin assignments showed in the Figure 3.



The code generated for reading and writing these digital inputs and outputs is showed below. The code to read will be at the beginning of the cycle and the code to write at the end.

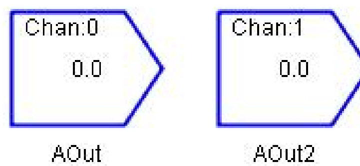
```
DIn= !(PIND & BV(PIND3));
DInInverseLogic= (PIND & BV(PIND4));

PORTB= (PORTB & (~(1<<0)) | (DOut ? (1<<0):0));
PORTC= (PORTC & (~(1<<5)) | (DOutInverseLogic ? 0:(1<<5)));
```

Analog Output

The number of analog outputs that is supported by the AVR is two, so the range of channels is from 0 to 1. The analog output writing is performed using the TIMER/COUNTER1. With a suitable configuration of it, it is possible to give the output in the OCR1A/B registers. It uses AVR´s Timer1 in mode 10bits Fast PWM:

TCCR1A=0xA3;
TCCR1B=0x09;

As the generation for digital IO, the function in change of generating this code takes the value of the channel and assigns the value of the variable to the proper OCR1A/B register.

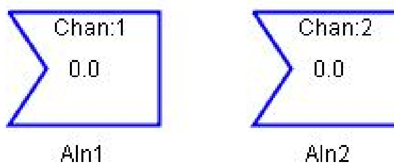


The code to write the value to an analog output is showed below. This code will be at the end of the main program:

OCR1A = AOut;
OCR1B = AOut2;

Analog Input

The AVR board supports four analog inputs with a possible range of channels from 0 to 3. The analog inputs are read through the interrupt handler of the signal SIG_ADC. So, the code generated by the function will be located in this interrupt handler. This code assigns the data to the proper input variable according to the channel.



The interrupt handler generated to read from these analog inputs will be the following. Only the part corresponding to the checking of the channel and assigning the read value to the proper variable is generated by the method called when a analog input is detected. The other code is generated at the beginning of the main function.

```
SIGNAL(SIG_ADC){
        static unsigned channel=0;
        short readed;
        readed = ADCL | (ADCH<<8);
        if (channel==0){
                AIn1=readed; //ADC3
        }
        if (channel==1){
                AIn2=readed; //ADC0
        }
        channel = (channel + 1 )%4;
        ADMUX = (0xC0) | (channel);
        ADCSRA=0xEF;
}
```

4.2. AVR Program

4.2.1. Execution Model

The execution model follows by the C program generated is the same as the model used by JGrafchart. So, the AVR main program that executes a scan cycle, executes the same actions in the same order as the execution performed by the function *executeOnce* belonging to the class *GCDocument* in JGrafchart.

In order to ensure periodic execution, it uses a timer and an interrupt handler associated with this timer. This handler will indicate to the main program when it is time to execute a new scan cycle. Then, at the beginning of the cycle, the execution thread have to sleep until the start of the next scan cycle. The period of the execution is determined by the JGrafchart parameter "Thread Sleep Interval" that can be set in the dialog "Workspace Properties". This value is the period of the execution thread associated with the top-level workspace. This field is taken by the code generator and declared as a constant in the C code.

After the wait, the code to execute one scan cycle begins. In every scan the following operations are performed:

- Read from inputs. However, only digital inputs are read in this point of the execution. Analog inputs are a special case and they are read through the interrupt handler commented in the previous chapter.
- All the transitions are traversed. The state of all preceding steps (*State.x*) and the transition condition are checked. If the preceding steps are all active and the condition evaluates to true the attribute of this transition (*Transition.markedforfired*) is set. It means the transition is marked for firing.

- Again, the transitions are traversed. In this case, evaluating for each transition, if it is marked for firing (*markedforfired=true)*. If this condition is satisfied, the following actions will be performance:

  1. The state of the transition will be set to false.

  2. For each preceding step, the newx attribute is set to false and all the exit actions of these steps are executed.

  3. For each succeeding step, the newx attribute is set to true and all the enter actions of these steps are executed

- After this, all the steps that contain periodic actions, are traversed evaluating the attributes x and newx in order to test if the step is active in the current scan cycle and will remain so also in the next scan cycle. In this case, periodics actions of the step are executed.
- Steps that contain normal actions are evaluated in order to determinate if the activation status of the step has changed since the last scan. If the step becomes active then the boolean variable or the digital output is set to true. The value of the boolean variable or the digital output is set to false if the step becomes deactivate.
- The attribute t, that counts the number of cycles since the step last was activate, is updated for all the steps.
- The state of each step is updated by assigning newx to x.
- Outputs are written. Both, digital and analog outputs are written.

4.2.2. AVR Program Structure

Include header files and constants declaration
Declaration types and variables

- Definition of struct variables
- Declaration of steps and transitions
- Declaration of internal variables
- Declaration of IO variables

Interrupt handlers and function

- SIG_ADC Interrupt handler – Read analog inputs
- SIG_OVERFLOW0 Interrupt handler – Periodic execution
- SIG_UART_RECV Interrupt handler – Receive from serial port
- Function send (char c) – Send data to serial port

Main program

```
Configuration of ADC, USART and Timers
Configuration Output Ports
Declare communication vectors
Enter actions of initial steps

while(){
        If (is time of a new scan cycle){
                If (is time to send)
                        Send steps status and variables value

                Read inputs
                Evaluate condition transitions
                Fire transitions
                Periodic actions
                Normal actions
                Update time of steps
                Update state steps
                Write outputs
                Update communication vectors
        }
}
```

4.3. Compiling the C Program

Once the C program has been generated, it is time to compile it. The cross compiler avr-gcc is used. A cross-compiler is a compiler that runs on one machine and produces object code for another machine. In our case, the compiler runs on a PC and generates code that will be executed on another platform, the AVR microcontroller.

AVR-GCC C complier is made available through the GNU project under the GNU public license. A C library function implemented in the avr-libc standard library is available for the ATMEL AVR microcontroller family. This standard library provides, among others, the definition of symbols for the interrupt vectors, routines to handler interrupts, IO register set and their respective bit values as specified in the Atmel documentation.

The necessary header files are included in the first lines of the C program:

```
#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
```

The first one includes the IO register definitions for the AVR microcontroller used and several macros for IO access. Signal.h and interrupt.h contain the symbols for the interrupt vectors that are stored at the beginning of the flash memory and functions to enable and disable interrupts.

When the program has been compiled properly using the avr-gcc compiler, it has to be uploaded to the microcontroller and executed it in the target platform.

During the development of this project the compilation and uploading have been performed by a makefile. In this file the compiler used, the device (ATMega8) and all the commands to compile and load the program on the AVR chip are specified.

4.4. AVR Memory Constraints

The AVR architecture has two main memory spaces: the data memory and the program memory space. In addition, the ATMega8 features an EEPROM Memory for data storage. For program storage, the ATMega8 contains 8k bytes of Flash memory and 1Mb of RAM Memory for data storage. As we can see the memory is small. In order to test the capacities of these memories, several tests have been executed.

The first test will be created a simple program only with steps and transitions. The AVR microcontroller only supports a program with 32 steps and transitions. These elements are represented in the C program as variable struct. As it was mentioned before, steps and transitions are represented as structs containing three and one *int* fields, respectively. Each *int value* uses two bytes in memory, so only 256 bytes of the data memory space are used.

The second test will check the number of variables supported by the procesor. This simple task will be performed creating a JGrafchart program that only contains variables. The type of the variables does not play any role, since both, integers and booleans, will be represented as *int* in the C program. In this case the microcontroller allows around 300 variables. So, about 600 bytes of the data memory space will be used.

If the results of the data memory occupation are examined, it is easy to realize that the memory constraints are not given by the size of the data memory. In both tests, the number of necessary bytes to store these elements is less than the memory capacity. So, we can reach the conclusion that the program memory is the cause of the constraints.

When the number of elements or the number of variables grows, the size of the program also grows. Therefore, due to the fact that the size of the memory is really small (only 8k bytes), this memory will be full before reaching the limit of the data memory.

Finally, one can conclude that regardless of the memory that causes the constraint, the number of steps, transitions and variables is limited. AVR allows 32 steps, 32 transitions and about 300 variables, to be precise.

5. COMMUNICATION JGRAFCHART – AVR

The second goal of this master thesis was the bidirectional communication between the host machine, a PC running JGrafchart, and the target machine, the AVR microcontroller. The on-line communication is necessary in order to provide animation of the execution in the target on the host and to provide user interface possibilities from the host to the target. RS-232 is used for the communication.

5.1. Communication Parameters

The RS-232 interface is used to communicate the host machine and the target machine. The PC and the AVR board are joined through a standard null-modem cable connected to the serial port.

In order to achieve a correct communication it is necessary to initialize the serial port in the correct way. Parameters as baud rate, data bit format, number of stop bits or the type of parity generation and parity check have to be configured in the same way in both devices, the PC and the AVR. It means that the configuration has to be the same in the Java and at AVR code side. The values chosen for these settings are:

- Baud Rate: 38400 bauds.
- Character size for data bit format: 8 bits.
- Parity mode: disable.
- Number of stop bits: 1 stop bit.

During the communication, it is often necessary to send multibyte numbers, integers to be precise. This is the reason why we have to choose the order of the bytes that are going to be sent. There are two possibilities: little endian and big endian. Little endian means that the low order byte of the number is sent before the high order byte and this it's the way chosen to send the integers.

5.2. JGrafchart AVR Mode

As mentioned before, the communication between the devices provides the way to display the execution that occurs in the AVR microcontroller on JGrafchart. To achieve this goal, a new execution model in JGrafchart is created. This new mode, called AVR Mode, allows the user to watch the execution of the model which he has loaded before in the microcontroller, in JGrafchart. It is also possible to send down variable values from the PC to the AVR.

The AVR Mode provides the possibility of configuring the rate that AVR uses to send the status of the program to JGrafchart. This integer parameter named Send Rate is configurable in the option "Workspace Properties", see Figure 5.1. If the value of Send Rate is 1, the microcontroller sends the information every cycle, if 2, each 2 cycles and so on.

The animation provided by this mode includes both the steps and the variables. In the first case, a token moves around in the graph showing the state in which the execution is in each moment. Concerning the variables, two possibilities exist. The normal case is that the value of a variable is displayed in the corresponding variable object in JGrafchart during the execution of the program. However, it is also possible to define special AVR-variables. For AVR-variables the current values are not sent periodically from AVR to JGrafchart. Instead, JGrafchart sends an update message to the AVR whenever one of these variables receives a new value in JGrafchart, e.g., through a step action in some other workspace or by the user through a button. In this way it is possible to control the execution in the AVR from JGrafchart.

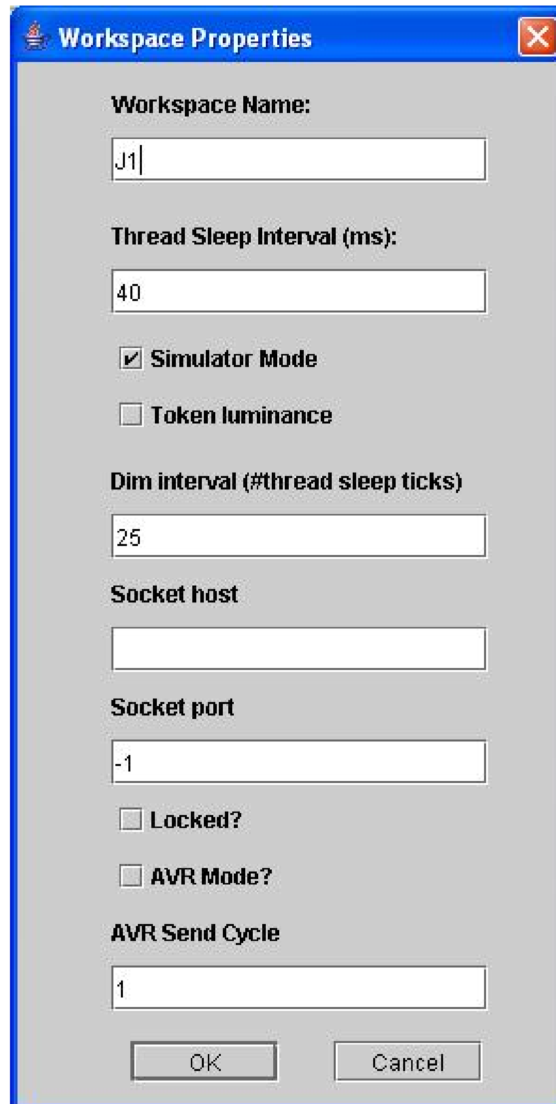


Figure 5.1

5.3. Bidirectional Communication

The communication between the host and the target machine consists of two different types or directions. In one direction, from AVR to JGrafchart, the sent information is used by the Java program to animate the execution of the code down in AVR. In the other direction it's possible to send down variables values from JGrafchart to AVR. Figure 5.3 shows the communication between the devices and the information sends in both directions.

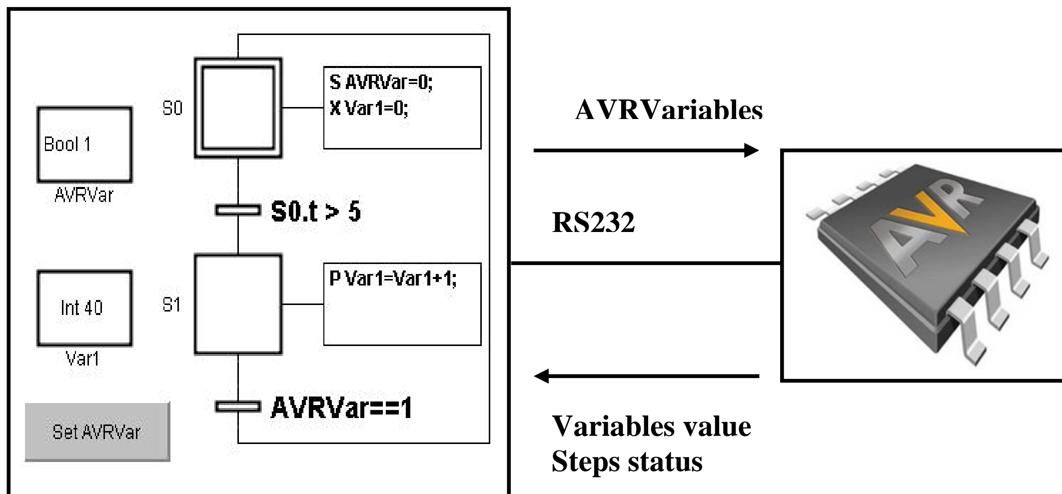


Figure 5.3

To establish the communication, the AVR microcontroller provides an Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) that allows the serial communication between the AVR and an external device. In the same way Java provides the Java Communication API that contains support for RS232 serial ports.

5.3.1. Establish communication

The first step in the communication is to establish all the communication parameters in both the AVR code and the Java program (JGrafchart)

USART Initialization

The USART has to be initialized before any communication. The initialization process consists of setting the baud rate, setting the frame format, and enabling the transmitter or the receiver.

The Universal Synchronous and Asynchronous serial Receiver and Transmitter accepts all 30 combinations of the following as valid frame format: serial frames with 5, 6, 7, 8 and 9 data bits, 1 or 2 stop bits, 1 start bit and no, even or odd parity bit. These parameters of the communication are set through the register USART Control and Register C (UCSRC):

```
UCSRC= 0x86;
```

Writing this value in this register, the communication parameters asynchronous operation mode, no parity bit, 1 stop bit and a character size of the frame of 8-bits are configured.

The configuration of the baud rate is generated by using the USART Baud Rate Registers (UBRRL and UBRRHs) settings. With two parameters, the Oscillator Frequency and the value of the bit U2X, it is possible to select the baud rate by writing the UBRRL register with the proper value. For a clock frequency of 14.7456 MHz and the double USART transmission speed disabled (U2X=0), the UBRR registers have to be written in the following way to set the baud rate as 38400 bauds.

```
UBRRH = 0x00;
UBRRL = 23;
```

JAVA Communication API

The Java Communication API is centered on the class *CommPort.* This class describes the methods for controlling I/O that are common to different kinds of communication ports. In addition, this class provides general methods for receiving and sending data from and to the communication port. The class *SerialPort* is a subclass of *CommPort* that includes methods for low-level control of serial port. Another Java Class is used, *CommPortIdentifier*. This class is the central class for controlling access to the communication port.

First, the application uses methods in CommPortIdentifier to negotiate with the driver to discover which communication ports are available and the select a port for opening. It then uses methods in the CommPort and SerialPort classes to communicate through the port.

To create the connection a new private class called *SerialPortReader* is created in the class *GCDocument.* The interface of this class is the following:

```
private class SerialPortReader{
        CommPortIdentifier portId;
        Enumeration portList;
        InputStream  is;
        OutputStream os;
        SerialPort serialPort;

        public SerialPortReader();
}
```

The steps to make the connection are performed in the class constructor *public SerialPortReader():*

1. Choose a port. It is necessary to know what ports are available on the computer. The method *CommPortIdentifiers.getPortIdentifiers( )* returns a ports list.

2. All the ports are traversed looking for the proper port. The name and the type of the port have to be checked. For this purpose methods of the class *CommPortIdentifiers* are used:

   portId.getPortType () == CommPortIdentifier.PORT_SERIAL
   portId.getName ().equals ("/dev/ttyS0")

3. Open the port.

   serialPort = (SerialPort) portId.open ("Reader", 2000)

4. Set the serial communication parameters to their proper values.

   serialPort.setSerialPortParams(38400, SerialPort.DATABITS_8, SerialPort.STOPBITS_1, SerialPort.PARITY_NONE)

5. Construct the objects to read and write to the serial port by calling the methods of the *SerialPort* object:

   is = serialPort.getInputStream();
   os = serialPort.getOutputStream();

With these actions, the program is ready to read and write using the serial port /dev/ttS0 with the communications parameters mentioned in Section 5.1. When the communication has finished, it is important to remember to close the communication using the method *CommPort.close().*

5.3.2. From AVR to JGrafchart

The main goal of the information sent in this direction of the communication is to animate the execution of the code down in AVR. The animation consists of updating the values of the variables and the step status, a token that moves around in the JGrafchart graph.

The AVR microcontroller sends the step status and variables at the beginning of the execution cycle. It is important to mention that it is not necessary to send any identification for steps and variables because the order in which this information is sent is always known. During the code generation, the steps and the variables are stored in two vector structures in the same order that they will be sent by the AVR code. Then, using these vectors and knowing the number of steps and variables that are going to be sent it is not necessary to include the variable or step identifiers in the information being sent, and still JGrafchart will perform the animation correctly.

Data Format

- Steps. For each step it's only necessary to send one bit indicating if the step is active or inactive. The minimum size sent is a byte, 8 bits, so a byte is sent also when the number of steps is less than 8. For example, for a JGrafchart program with 11 steps 2 bytes are sent:

Byte 0

S7.x	S6.x	S5.x	S4.x	S3.x	S2.x	S1.x	S0.x

Byte 1

					S10.x	S9.x	S8.x

- Variables. All the variables, integer or boolean, are stored as 2 bytes. So each variable to be sent is represented as 2 bytes. As mentioned before, the order of these bytes is little endian; we send the low byte before the high byte.

AVR Data Transmission

The AVR program sends the step status and variables values at the beginning of each cycle as indicated by the sendRate parameter in the AVR Mode using the AVR USART Transmitter. The USART Transmitter is enabled by setting the *Transmit Enable* (TXEN) bit in the UCSRB Register.

The transmit function used is based on polling of the *Data Register Empty* (UDRE) flag in the UCSRA Register:

```
void send(char data){
        while ((UCSRA & 0x20)==0){}
        UDR=data;
        while ((UCSRA & 0x20)==0){}
}
```

The function simply waits for the transmit buffer to be empty by checking the UDRE Flag, before loading it with new data to be transmitted. The function sends each time a *char*. It means that only one byte is sent each time the function is called.

When it is time to send the step status and variables values, the function is called once a byte has to be sent. Both the variables and the steps are stored in a vector, the steps in a vector of *char* (one byte per position) and the variables in a vector of *int* (2 bytes per position). These vectors are updated at the end of the cycle and are sent at the beginning of the next cycle in the following way:

- Vector declaration

```
char steps[100];
int variables[100];
```

- Sending vectors

```
if(it's time to send){
        for (int h=0;h<size(steps);h++)
                send(steps(h));
        for (int k=0;k<size(variables);k++){
                send((char)variables[k]&&0xFF);   //Low byte
                send((char)(variables[k]>>8)& ~( ~0<<8)); //High byte
        }
}
```

JGrafchart Data Reception

JGrafchart receives the step status and the variables values and updates the graphics. It displays a token in the step or steps that are active and displays the value of the variables.

The communication is implemented through a Java Thread. The Thread API requires a method called *run()* to do the body of the work for the thread and calls the method of the thread *start()* to start it running independently. In JGrafchart a new Thread subclass called SerialPortReaderThread is created. The interface of this class is:

```
public class SerialPortReaderThread extends Thread{
        InputStream is;
        OutputStream os;
        SerialPort pt;

        public SerialPortReaderThread(InputStream input, OutputStream
output, SerialPort sp);
        public void run();
        public void sendVariable(int id, int value);
}
```

The data reception is implemented in the method *run()* of the new subclass. This method receives the bytes in the *InputStream is* from the class *SerialPortReader*, interprets these values and updates the graph. As mentioned before, it is not necessary with any identification of the steps or variables. This is possible due to the information updated by the part of code generation. The number of steps, number of variables and order in which these values will be sent by AVR are provided during the code generation. This function updates these parameters and then these values are available during the data reception.

Once the identification of one received value is completed, the methods provided by the classes GCStep, IntegerVariable and BooleanVariable are used to place the token in the correct step and set the variable value to an integer or boolean.

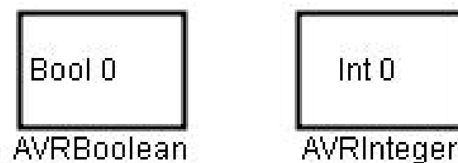
When the AVR Mode is activated and begins the execution in JGrafchart, the method *start()* of the SerialPortReaderThread is executed and the Java Virtual Machine calls the *run ()* method of this class.

```
if (AVRMode) {
     SerialPortReader p = new SerialPortReader();
     serialReader = new SerialPortReaderThread(p.is,p.os,p.serialPort);
    serialReader.start();
}
```

At the moment the application begins the execution in AVR Mode, the program creates a new object of the class *SerialPortReader* to open the connection with the serial port and waits for new data.

5.3.3. From JGrafchart to AVR

JGrafchart sends of the values of AVR-variables to the AVR. The AVR program receives the values to update these variables. AVR-variables are represented with the prefix *AVR* preceding the name of the variable:



JGrafchart sends these variables to AVR when they change their value. This value can be changed by the user through a button, a step action or using the GUI options. When the microcontroller receives these variables, it does not send the value in the other directions. As a result of this property the AVR variables are not showed in the JGrafchart animation.

These variables can be sent in any moment during the execution of the program and without any preestablished order. The modification of variables is totally asynchronous. So, it is necessary to send identification before any variable in order to identify which variable is being modified by JGrafchart and then, which variable have to be updated by the AVR program. The identification for both integer and boolean variables is an integer value.

Data Format

The necessity of identifying the variables to be sent implies that the number of bytes in the communication increases. For each variable, it is necessary to send three bytes, the first one to identify the variable in the target machine and the following two contain the new value of this variable:

AVRVariable (ID)	Low Byte	High Byte

The low order byte of the variable value is sent before the high order byte so the byte order is little endian.

JGrafchart Data Transmission

JGrafchart have to send the new AVR variable value each time the variable changes its value During the code generation, each time an AVR variable is detected, the reference to this variable is stored in a vector. The position of the variable in this vector is used to generate the identification for it. This id number is the parameter given to the function *sendVariable.*

The function *sendVariable (int id, int value)* of the class *SerialPortReaderThread* is the function responsible for writing the data to the serial port. The method is called every time JGrafchart detects a change in an AVR variable. There are two methods, one per type of variable, that are executed:

- For integer variables, the method *setStoredIntAction( )* belongs to the class *IntegerVariable*
- *setStoredBooleanAction()* if the AVR variable is a boolean variable.

These methods are responsible for calling the *sendVariable(int id, int value)* function with the proper parameters, the index in the references vector to identify the variable and the new value of this variable.

AVR Data Reception

AVR receives the AVR variables and updates their values. For this purpose, the AVR USART Receiver is used. It is enabled by setting the Receive Enable (RXEN) bit in the UCSRB Register. Besides, the USART Receiver has one flag that indicates the Receiver state. The Receive Complete (RXC) Flag indicates if there are unread data present in the receive buffer. When the Receive Complete Interrupt Enable (RXCIE) in UCSRB is set, the USART Receive Complete Interrupt will be executed as long as the RXC Flag is set. This interrupt and its associated handler will be used to receive the characters from the serial port; the AVR receives an interrupt every time a new byte is read from the serial port. The interrupt handler is responsible for waiting for the three bytes

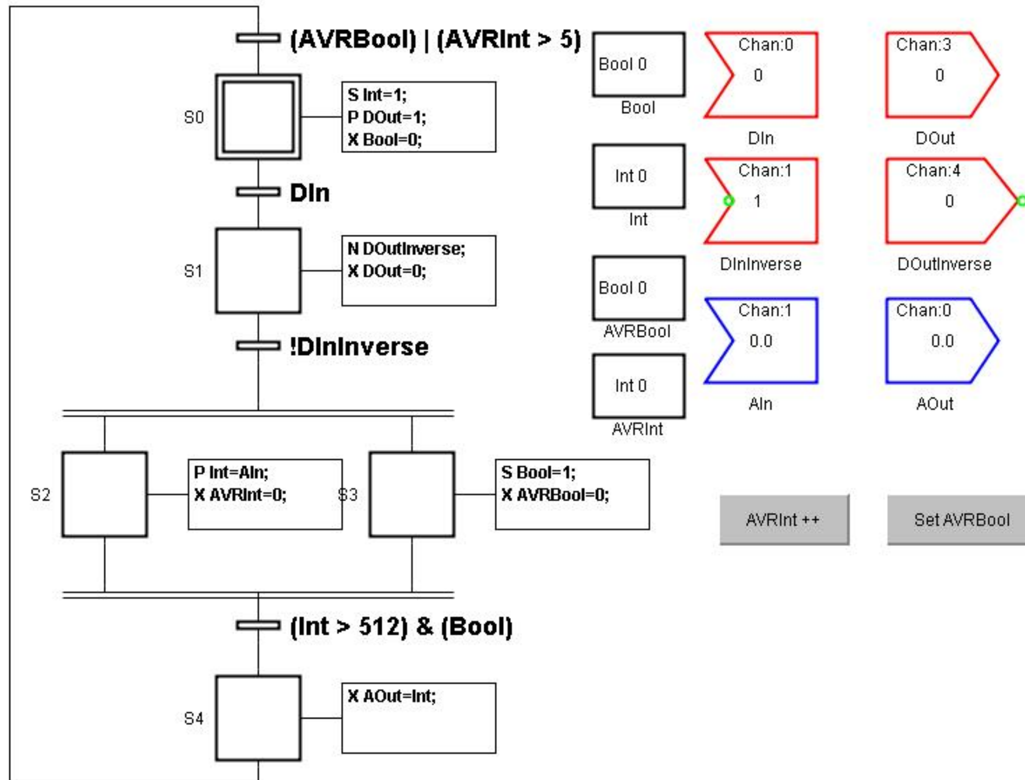
corresponding to the identification and the value, for identifying the received variable and for updating the variable value.

```
SIGNAL(SIG_UART_RECV){
        static int cc=0;
        static char variablesJG[3];
        cc++;
        variablesJG[cc-1]=UDR;
        if (cc==3){
                cc=0;
                int id=(int)variablesJG[0];
                if(id==0)
                        AVRInt= ((int)variablesJG[1]&0xFF)+
                (((int)variablesJG[2])<<8);
                if(id==1)
                        AVRBool=
                ((int)variablesJG[1]&0xFF)+(((int)variablesJG[2])<<8);
        }
}
```

This code shows an example of the SIG_UART_RECV interrupt handler included in an AVR program that has two AVR variables, one named *AVRInt* and one named *AVRBool*. Before the identification, it waits until three bytes have been received. Then, it checks the first byte of the data frame and updates the variable associated with this id number. After that, the code waits for another three byte frame.

6. EXAMPLE

6.1. JGrafchart Program



This is a simple example that includes all the elements in the code generation. All the types of actions qualifiers, one instance of each IO variable and all the possible variables are present. The AVR variables are modified by an action button. The integer AVR variable are modified through the button called AVRInt ++ and the action associated with it increments the value of the variable by one, each time the button is pushed. The other AVR variable, AVRBool, is also modified by a button. In this case the action associated with the button sets the boolean variable to true. As was mentioned in the chapter describing the communication, the value of these variables is not showed in the graphics during the execution. The reason for this is that these values are not sent from the microcontroller to JGrafchart.

Concerning the analog IO, there are two analog IO variables: one for input and one for output. During the execution of step S2, the value from the channel 1 is being pulled, when the value is greater than the value indicated in the transition, the transition is fired and the step S4 is activated. Immediately before this step is deactivated, the same value that was received from the input is put in the output.

6.2. AVR Program

```
#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
```

Header files avr-libc standard library

```
#define true 1;
#define false 0;

const double h = 40;
const double sendRate=1;
```

Execution and communication parameters taken from JGrafchart

```
struct step{
        int x, newx, t;
};

struct transition{
        int markedforfired;
};
```

Definition of struct variables for steps and transitions

```
struct step S0={1,1,0};
struct step S2={0,0,0};
struct step S3={0,0,0};
struct step S1={0,0,0};
struct step S4={0,0,0};

struct transition TR0={0};
struct transition TR1={0};
struct transition TR2={0};
struct transition TR3={0};
```

Declaration and initialization of steps and transitions

```
volatile int bit=true;

volatile int Bool = false;
volatile int Int = 0;
volatile int AVRBool = false;
volatile int AVRInt = 0;
```

Declaration and initialization of variables. Both, AVR variables and normal variables

```
int DInInverse = true;
int DOut = false;
int DOutInverse = false;
int DIn = false;
volatile int AIn=0;
int AOut=0;
```

Declaration and initialization of IO variables. Both, input variables and output variables.

```
SIGNAL(SIG_ADC){
        static unsigned channel=0;
        short readed;
        readed = ADCL | (ADCH<<8);

         if (channel==3){
                AIn=readed; //ADC3
        }
        channel = (channel + 1 )%4;
         ADMUX = (0xC0) | (channel);
        ADCSRA=0xEF;
 }
```

SIG_ADC Interrupt handler. Read analog input AIn, connected to pin PC3 (ADC3)

```
SIGNAL(SIG_OVERFLOW0){
        static int i=0;
        int a=h*7.2;

        i++;
        if (i==a){
                bit=true;
                i=0;
        }
}
```

SIG_OVERFLOW0 Interrupt handler. Periodic execution

```
SIGNAL(SIG_UART_RECV){
        static int cc=0;
        static char variablesJG[3];

        cc++;
        variablesJG[cc-1]=UDR;
         if (cc==3){
           cc=0;
           int id=(int)variablesJG[0];
           if(id==0)
               AVRBool= ((int)variablesJG[1]&0xFF)+
               ((int)variablesJG[2])<<8);
           if(id==1)
               AVRInt= ((int)variablesJG[1]&0xFF)+
               (((int)variablesJG[2])<<8);
        }
}
```

SIG_UART_RECV Interrupt handler. Receive the values for the AVR variables from the serial port.

```
void send(char c){
        while ((UCSRA & 0x20)==0){}
        UDR=c;
        while ((UCSRA & 0x20)==0){}
}
```

Function send (char c) Send a byte to the serial port

```
int main() {

        char steps[32];
        int variables[350];
        int ii=0;

         Int=1;        <----------      Enter action Inicial Step

        ADMUX=0xC0;         {  Configuration Analog
        ADCSRA=0xEF;        {  Digital Converter

        UCSRA=0x00;
        UCSRB=0x98;
        UCSRC=0x86;         {  USART configuration
        UBRRH=0x00;
        UBRRL=23;

        DDRB=0x3F;               {  Configuration Output
        DDRC=0x30;               {  Ports

        TCCR0=0x02;
        TCNT0=0x00;
         TIMSK=TIMSK | (1<<0);   {  TIMER0 and TIMER1
                                 {  Configuration
        TCCR1A=0xA3;
        TCCR1B=0x09;

        sei();
         while(1){
                while(!bit) {
                        ; //Busy wait
                 }                          Code to send steps
                bit=false;                  status and
                                            variables value
                ii++;
                if(ii==sendRate){
                        int h;
                        for(h=0;h<1;h++){
                                send(steps[h]);
                        }
                        int y;
                        for(y=0;y<2;y++){
                                send((char)variables[y]&0xFF);
                                send((char)(variables[y]>>8)& ~(~0<<8));
                        }
                        ii=0;
                }
```

```
DInInverse= (PIND & BV(PIND3));      Read digital
DIn= !(PIND & BV(PIND2));            inputs Din
                                     (PD2) and
                                     DInInverse
                                     (PD3)
 if (S0.x && (DIn)) {
         TR0.markedforfired = true;
 }
 if (S1.x && (!DInInverse)) {
         TR1.markedforfired = true;
}
 if (S2.x && S3.x && ((Int > 512) & (Bool))) {
         TR2.markedforfired = true;
 }
if (S4.x && ((AVRBool) | (AVRInt > 5))) {
         TR3.markedforfired = true;
}

 if (TR0.markedforfired ) {
         TR0.markedforfired=false;
         S0.newx = false;
         Bool=0;
         S1.newx = true;
}
 if (TR1.markedforfired ) {
         TR1.markedforfired=false;
         S1.newx = false;
         DOut=0;
         S2.newx = true;
         S3.newx = true;
         Bool=1;
 }
 if (TR2.markedforfired ) {
         TR2.markedforfired=false;
         S2.newx = false;
         AVRInt=0;
         S3.newx = false;
         AVRBool=0;
          S4.newx = true;
}
if (TR3.markedforfired ) {
         TR3.markedforfired=false;
         S4.newx = false;
         AOut=Int;
          S0.newx = true;
         Int=1;
 }
```

```
        if ((S0.x) && (S0.newx)) {
                DOut=1;
        }
        if ((S2.x) && (S2.newx)) {
                Int=AIn;
        }
```
Periodic actions of steps S0 and S2

```
        if ((S1.x) && !(S1.newx)) {
                DOutInverse=false;
        }
        else {
                if (S1.newx){
                        DOutInverse=true;
                }
        }
```
Normal action of step S1

```
        S0.t = (S0.t + S0.x) * S0.x;
        S2.t = (S2.t + S2.x) * S2.x;
        S3.t = (S3.t + S3.x) * S3.x;
        S1.t = (S1.t + S1.x) * S1.x;
        S4.t = (S4.t + S4.x) * S4.x;

        S0.x = S0.newx;
        S2.x = S2.newx;
        S3.x = S3.newx;
        S1.x = S1.newx;
        S4.x = S4.newx;
```
Update state steps: time and status

Write outputs. Digital: DOut (PC4) and DOutInverse(PC5). Analog: AOut(PB1)

```
        PORTC= (PORTC & (~(1<<4)) | (DOut ? (1<<4):0));
        PORTC= (PORTC & (~(1<<5)) | (DOutInverse ? 0:(1<<5)));
        OCR1A = AOut;

        int i;
        for (i=0;i<1;i++)
                steps[i]=0;

        variables[0]=Bool;
        variables[1]=Int;

        steps[0] |=(S0.x<<0);
        steps[0] |=(S2.x<<1);
        steps[0] |=(S3.x<<2);
        steps[0] |=(S1.x<<3);
        steps[0] |=(S4.x<<4);
    }
}
```
Update the communication vectors: variables and steps

7. IMPROVEMENTS AND FUTURE DEVELOPMENT

7.1. Improvements

- Code optimization. As was mentioned in the chapter 4.4 the AVR memory is really small. The memory space which causes the main problem is the program memory space. So, a proper code optimization can achieve a smaller program and then make it possible to execute programs containing more.
- Once the C program has been generated, the user has to compile and load the C program using a makefile. It could be possible to add an option in JGrafchart that compiles and load automatically the obtained program. The compilation errors or loading problems could be shown to the user if they exist.
- The animation of the execution is showed in the JGrafchart program. This animation includes steps and variables value. However, the transitions and the IO variables are not included in the animation.
- Generate code for actions method call. This is only possible if there exists an equivalent function in the avr-libc.

7.2. Future Development

- The code generator only generates code for the JGrafchart basic elements. Code for macro steps could be implemented.
- In the present function, the generated code includes all the code relating to communication. Two versions of C code generation could be created: one without the communication code and other with the complete code. The first one that not includes all the code relative to the communication could be used to program the AVR exclusively.
- Add the possibility of generating code for more devices. Create a new Java class with the functions responsible of the code generation. So, in this way it is easier to replace the created class with other that generated the code for other device.

8. SUMMARY AND CONCLUSIONS

The development performed as a part of this master thesis adds to JGrafchart an extra feature: programming an ATMEL AVR microcontroller though a graphical language and the possibility to show the execution that takes place in the processor in JGrafchart. It also provides user interface possibilities to modify variables.

The result of the work is a C program that can be compiled and then downloaded to the microcontroller using the avr-gcc compiler. So, all the difficulties relating to low-level programming are removed. The program obtained runs in the AVR following the same execution model as JGrafchart.

A new execution model is added to JGrafchart. In this model the state of the program and the variables value can be followed directly in the graphical model, during the execution in the device.

Not all the features of JGrafchart can be transfer to the AVR. So, the programs developed in JGrafchart have to match the capacities that the microcontroller provides.

9. REFERENCES

[1] Automatic Control Department Homepage. Lund Institute of Technology. University

http://www.control.lth.se

[2] ATMEL Homepage: Datasheets and documentation about the ATMega8.

http://www.atmel.com

[3] Rich Nesswold, *A GNU Development Environment for the AVR Microcontroller,* 2002.

[4] John Morton. *AVR: an Introduction Course*. First Edition. Newnes Publisher. September 2002. IBSN: 0-7506-5635-2

[5] Claus Kuhnel. *AVR RISC Microcontroller Handbook*. Newnes Publisher. July 1998. ISBN: 0-7506-9963-9

[6] Ian Darwing, *Java Cookbook.* Second Edition. O'Reilly Press. August 2001. ISBN: 0-596-00170-3

[7] Elliotte Rusty Harold. *Java I/O*. First Edition. O' Really Press. March 1999. ISBN: 1-56592-485-1

[8] John Zukowski. *Programacion en Java2*. First Edition. Anaya Multimedia. December 1999. IBSN: 84-415-0948-4

[9] Java API Reference.

http://java.sun.com

[10] Java Communication API Reference Documentation

http://java.sun.com/products/javacomm/

APPENDIX I: REGISTER DESCRIPTION

A) Analog to digital converter

After the conversion is complete (ADIF is high), the conversion result can be found in the ADC Result Registers (ADCL, ADCH). For single ended conversion, the result is:

$$ADC = \frac{Vin * 1024}{Vref}$$

where VIN is the voltage on the selected input and VREF the selected voltage reference, in this case 2,56V.

ADC Multiplexer Selection Register – ADMUX

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7:6 – REFS1:0: Reference Selection Bits

These bits select the voltage reference for the ADC, as shown in Table I. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Table I

- Bit 5 – ADLAR: ADC Left Adjust Result

The ADLAR bit affects the presentation of the ADC conversion result in the ADC Data Register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC Data Register immediately, regardless of any ongoing conversions.

- Bits 3:0 – MUX3:0: Analog Channel Selection Bits

The value of these bits selects which analog inputs are connected to the ADC. See Table II for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	
1001	
1010	
1011	
1100	
1101	
1110	1.23V (V_{BG})
1111	0V (GND)

Table II

ADC Control and Status Register A – ADCSRA

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 – ADEN: ADC Enable

Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- Bit 6 – ADSC: ADC Start Conversion

In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This

first conversion performs initialization of the ADC. ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

- Bit 5 – ADFR: ADC Free Running Select

When this bit is set (one) the ADC operates in Free Running mode. In this mode, the ADC samples and updates the Data Registers continuously. Clearing this bit (zero) will terminate Free Running mode.

- Bit 4 – ADIF: ADC Interrupt Flag

This bit is set when an ADC conversion completes and the Data Registers are updated. The ADC Conversion Complete Interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt Handling Vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a Read-Modify-Write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.

- Bit 3 – ADIE: ADC Interrupt Enable

When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.

- Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits

These bits determine the division factor between the XTAL frequency and the input clock to the ADC.

The ADC Data Register – ADCL and ADCH

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

When an ADC conversion is complete, the result is found in these two registers.

When ADCL is read, the ADC Data Register is not updated until ADCH is read. Consequently, if the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH.

The ADLAR bit in ADMUX and the MUXn bits in ADMUX affect the way the result is read from the registers. If ADLAR is set, the result is left adjusted. If ADLAR is cleared (default), the result is right adjusted.

B) USART

USART I/O Data Register – UDR

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDR (Read)
	TXB[7:0]								UDR (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDR. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDR Register location. Reading the UDR Register location will return the contents of the Receive Data Buffer Register (RXB).

For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the Transmitter and set to zero by the Receiver. The transmit buffer can only be written when the UDRE Flag in the UCSRA Register is set. Data written to UDR when the UDRE Flag is not set, will be ignored by the USART Transmitter. When data is written to the transmit buffer, and the Transmitter is enabled, the Transmitter will load the data into the Transmit Shift Register when the Shift Register is empty. Then the data will be serially transmitted on the TxD pin.

The receive buffer consists of a two level FIFO. The FIFO will change its state whenever the receive buffer is accessed. Due to this behavior of the receive buffer, do not use Read-Modify-Write instructions (SBI and CBI) on this location. Be careful when using bit test instructions (SBIC and SBIS), since these also will change the state of the FIFO.

USART Control and Status Register A – UCSRA

Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

- Bit 7 – RXC: USART Receive Complete

This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e. does not contain any unread data). If the Receiver is disabled, the receive buffer will be flushed and consequently the RXC bit will become zero. The RXC Flag can be used to generate a Receive Complete interrupt (see description of the RXCIE bit).

- Bit 6 – TXC: USART Transmit Complete

This flag bit is set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (UDR). The TXC Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXC Flag can generate a Transmit Complete interrupt (see description of the TXCIE bit).

- Bit 5 – UDRE: USART Data Register Empty

The UDRE Flag indicates if the transmit buffer (UDR) is ready to receive new data. If UDRE is one, the buffer is empty, and therefore ready to be written. The UDRE Flag can generate a Data Register Empty interrupt (see description of the UDRIE bit). UDRE is set after a reset to indicate that the Transmitter is ready.

- Bit 4 – FE: Frame Error

This bit is set if the next character in the receive buffer had a Frame Error when received (i.e., when the first stop bit of the next character in the receive buffer is zero). This bit is valid until the receive buffer (UDR) is read. The FE bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRA.

- Bit 3 – DOR: Data OverRun

This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the Receive Shift Register, and a new start bit is detected. This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

- Bit 2 – PE: Parity Error

This bit is set if the next character in the receive buffer had a Parity Error when received and the parity checking was enabled at that point (UPM1 = 1). This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

- Bit 1 – U2X: Double the USART transmission speed

This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

- Bit 0 – MPCM: Multi-processor Communication Mode

This bit enables the Multi-processor Communication mode. When the MPCM bit is written to one, all the incoming frames received by the USART Receiver that do not contain address information will be ignored. The Transmitter is unaffected by the MPCM setting.

USART Control and Status Register B – UCSRB

Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 – RXCIE: RX Complete Interrupt Enable

Writing this bit to one enables interrupt on the RXC Flag. A USART Receive Complete interrupt will be generated only if the RXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the RXC bit in UCSRA is set.

- Bit 6 – TXCIE: TX Complete Interrupt Enable

Writing this bit to one enables interrupt on the TXC Flag. A USART Transmit Complete interrupt will be generated only if the TXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXC bit in UCSRA is set.

- Bit 5 – UDRIE: USART Data Register Empty Interrupt Enable

Writing this bit to one enables interrupt on the UDRE Flag. A Data Register Empty interrupt will be generated only if the UDRIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDRE bit in UCSRA is set.

- Bit 4 – RXEN: Receiver Enable

Writing this bit to one enables the USART Receiver. The Receiver will override normal port operation for the RxD pin when enabled. Disabling the Receiver will flush the receive buffer invalidating the FE, DOR and PE Flags.

- Bit 3 – TXEN: Transmitter Enable

Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxD pin when enabled. The disabling of the

Transmitter (writing TXEN to zero) will not become effective until ongoing and pending transmissions are completed (i.e., when the Transmit Shift Register and Transmit Buffer Register do not contain data to be transmitted). When disabled, the Transmitter will no longer override the TxD port.

- Bit 2 – UCSZ2: Character Size

The UCSZ2 bits combined with the UCSZ1:0 bit in UCSRC sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use.

- Bit 1 – RXB8: Receive Data Bit 8

RXB8 is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from UDR.

- Bit 0 – TXB8: Transmit Data Bit 8

TXB8 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to UDR.

USART Control and Status Register C – UCSRC

Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

- Bit 7 – URSEL: Register Select

This bit selects between accessing the UCSRC or the UBRRH Register. It is read as one when reading UCSRC. The URSEL must be one when writing the UCSRC.

- Bit 6 – UMSEL: USART Mode Select

This bit selects between Asynchronous and Synchronous mode of operation (0 means Asynchronous and 1 Synchronous)

- Bit 5:4 – UPM1:0: Parity Mode

These bits enable and set type of Parity Generation and Check. If enabled, the Transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The Receiver will generate a parity value for the incoming data and compare it to the UPM0 setting. If a mismatch is detected, the PE Flag in UCSRA will be set.

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

- Bit 3 – USBS: Stop Bit Select

This bit selects the number of stop bits to be inserted by the transmitter. The Receiver ignores this setting (USBS = 0 means 1 stop bit, USBS=1 2 bits)

- Bit 2:1 – UCSZ1:0: Character Size

The UCSZ1:0 bits combined with the UCSZ2 bit in UCSRB sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use.

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

- Bit 0 – UCPOL: Clock Polarity

This bit is used for Synchronous mode only. Write this bit to zero when Asynchronous mode is used. The UCPOL bit sets the relationship between data output change and data input sample, and the synchronous clock (XCK).

USART Baud Rate Registers – UBRRL and UBRRHs

Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

- Bit 15 – URSEL: Register Select

This bit selects between accessing the UBRRH or the UCSRC Register. It is read as zero when reading UBRRH. The URSEL must be zero when writing the UBRRH.

- Bit 14:12 – Reserved Bits

These bits are reserved for future use. For compatibility with future devices, these bit must be written to zero when UBRRH is written.

- Bit 11:0 – UBRR11:0: USART Baud Rate Register

This is a 12-bit register which contains the USART baud rate. The UBRRH contains the four most significant bits, and the UBRRL contains the eight least significant bits of the USART baud rate. Ongoing transmissions by the Transmitter and Receiver will be corrupted if the baud rate is changed. Writing UBRRL will trigger an immediate update of the baud rate prescaler

C) TIMER/COUNTER0

Timer/Counter Control Register – TCCR0

Bit	7	6	5	4	3	2	1	0	
	–	–	–	–	–	CS02	CS01	CS00	TCCR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 2:0 – CS02:0: Clock Select

The three clock select bits select the clock source to be used by the Timer/Counter.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Timer/Counter Register – TCNT0

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter.

Timer/Counter Interrupt Mask Register – TIMSK

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable

When the TOIE0 bit is written to one, and the I-bit in the Status Register is set (one), the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, i.e., when the TOV0 bit is set in the Timer/Counter Interrupt Flag Register – TIFR.

D) TIMER/COUNTER1

Timer/Counter 1 Control Register A – TCCR1A

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7:6 – COM1A1:0: Compare Output Mode for channel A
- Bit 5:4 – COM1B1:0: Compare Output Mode for channel B

The COM1A1:0 and COM1B1:0 control the Output Compare Pins (OC1A and OC1B respectively) behavior. If one or both of the COM1A1:0 bits are written to one, the OC1A output overrides the normal port functionality of the I/O pin it is connected to. If one or both of the COM1B1:0 bit are written to one, the OC1B output overrides the normal port functionality of the I/O pin it is connected to. However, note that the *Data Direction Register* (DDR) bit corresponding to the OC1A or OC1B pin must be set in order to enable the output driver. When the OC1A or OC1B is connected to the pin, the function of the COM1x1:0 bits is dependent of the WGM13:0 bits setting. Table III shows the COM1x1:0 bit functionality when the WGM13:0 bits are set to a normal or a CTC mode (non-PWM).

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	Toggle OC1A/OC1B on Compare Match
1	0	Clear OC1A/OC1B on Compare Match (Set output to low level)
1	1	Set OC1A/OC1B on Compare Match (Set output to high level)

Table III

Table VI shows the COM1x1:0 bit functionality when the WGM13:0 bits are set to the fast PWM mode.

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at TOP
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at TOP

Table VI

Table V shows the COM1x1:0 bit functionality when the WGM13:0 bits are set to the phase correct or the phase and frequency correct, PWM mode.

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 9 or 14: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match when up-counting. Set OC1A/OC1B on Compare Match when downcounting.
1	1	Set OC1A/OC1B on Compare Match when up-counting. Clear OC1A/OC1B on Compare Match when downcounting.

Table V

- Bit 3 – FOC1A: Force Output Compare for channel A
- Bit 2 – FOC1B: Force Output Compare for channel B

The FOC1A/FOC1B bits are only active when the WGM13:0 bits specifies a non-PWM mode. However, for ensuring compatibility with future devices, these bits must be set to zero when TCCR1A is written when operating in a PWM mode. When writing a logical one to the FOC1A/FOC1B bit, an immediate Compare Match is forced on the waveform generation unit. The OC1A/OC1B output is changed according to its COM1x1:0 bits setting. Note that the FOC1A/FOC1B bits are implemented as strobes. Therefore it is the value present in the COM1x1:0 bits that determine the effect of the forced compare.

A FOC1A/FOC1B strobe will not generate any interrupt nor will it clear the timer in Clear Timer on Compare Match (CTC) mode using OCR1A as TOP. The FOC1A/FOC1B bits are always read as zero.

- Bit 1:0 – WGM11:0: Waveform Generation Mode

Combined with the WGM13:2 bits found in the TCCR1B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value and what type of waveform generation to be used, see Table VI. Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare Match (CTC) mode, and three types of Pulse Width Modulation (PWM) modes.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation[1]	TOP	Update of OCR1x	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

Table VI

Timer/Counter 1 Control Register B – TCCR1B

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 – ICNC1: Input Capture Noise Canceler

Setting this bit (to one) activates the Input Capture Noise Canceler. When the noise canceller is activated, the input from the Input Capture Pin (ICP1) is filtered. The filter function requires four successive equal valued samples of the ICP1 pin for changing its output. The Input Capture is therefore delayed by four Oscillator cycles when the noise canceler is enabled.

- Bit 6 – ICES1: Input Capture Edge Select

This bit selects which edge on the Input Capture Pin (ICP1) that is used to trigger a capture event. When the ICES1 bit is written to zero, a falling (negative) edge is used as trigger, and when the ICES1 bit is written to one, a rising (positive) edge will trigger the capture. When a capture is triggered according to the ICES1 setting, the counter value is copied into the Input Capture Register (ICR1). The event will also set the Input Capture Flag (ICF1), and this can be used to cause an Input Capture Interrupt, if this interrupt is enabled.

When the ICR1 is used as TOP value (see description of the WGM13:0 bits located in the TCCR1A and the TCCR1B Register), the ICP1 is disconnected and consequently the Input Capture function is disabled.

- Bit 5 – Reserved Bit

This bit is reserved for future use. For ensuring compatibility with future devices, this bit must be written to zero when TCCR1B is written.

- Bit 4:3 – WGM13:2: Waveform Generation Mode

See TCCR1A Register description.

- Bit 2:0 – CS12:0: Clock Select

CS12	CS11	CS10	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Output Compare Register 1 A – OCR1AH and OCR1AL

Bit	7	6	5	4	3	2	1	0	
	OCR1A[15:8]								OCR1AH
	OCR1A[7:0]								OCR1AL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Output Compare Register 1 B – OCR1BH and OCR1BL

Bit	7	6	5	4	3	2	1	0	
	OCR1B[15:8]								OCR1BH
	OCR1B[7:0]								OCR1BL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Output Compare Registers contain a 16-bit value that is continuously compared with the counter value (TCNT1). A match can be used to generate an Output Compare Interrupt, or to generate a waveform output on the OC1x pin. The Output Compare Registers are 16-bit in size. To ensure that both the high and Low bytes are written simultaneously when the CPU writes to these registers, the access is performed using an 8-bit temporary High byte Register (TEMP). This temporary register is shared by all the other 16-bit registers.

APPENDIX II: AVR IO PORTS

Each port pin consists of 3 Register bits: DDxn, PORTxn, and PINxn. The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configures as an input pin.

If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin. The port pins are tri-stated when a reset condition becomes active, even if no clocks are running.

If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero).

The following table summarizes the control signals for the pin value.

DDxn	PORTxn	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if external pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)