WIDE: Worksheet Integrated Development Environment for Arduino-based Embedded System Design

N. Ioannou, K. Tatas, A. Constantinides and C. Kyriacou,
Frederick University,
Nicosia, Cyprus
st020063@stud.frederick.ac.cy
{com.tk, com.ca , eng.kc}@.frederick.ac.cy

***Abstract*—This paper presents a novel programming approach for Arduino boards, utilizing tables within a spreadsheet environment. This method automatically generates over 80% of the code in a typical application, simplifying programming, facilitating debugging and presenting an easier learning curve, especially for non-electronics and computer science engineers. WIDE supports defining analog and digital pins, logical variables, state description, state flow, printing, menu, Infrared communication, Bluetooth, RS485 communication, and storing variables in EEPROM using tables that are easy to fill, modify and troubleshoot. The automatically generated code is collected into one worksheet and then copied to the Arduino IDE for compilation and downloading. This method has been successfully applied to various projects, allowing more time to be spent on problem-solving rather than the mechanics of code writing.**

***Keywords*—Microcontrollers, IDE, Arduino, WIDE, spreadsheet.**

I. Introduction

Embedded systems integrate hardware and software for operation in a physical environment and fulfill requirements for function, safety, durability, sustainability, user-friendliness, interfaces to other systems, regulations and standards [1]. They have long been used in aerospace, manufacturing, enterprise and consumer machines. More recently, they have also been used in smart homes, renewable energy systems, electric vehicles and the Internet of Things [2].

The complexity of programming embedded systems increases when they are used in demanding environments such as aerospace or self-driving cars. In less demanding environments such as agriculture, the complexity arises from the interaction with various sensors, actuators and communication with an IoT platform, as well as the need for a human-machine interface and the constant updating of designs due to the rapid advancement of technology.

As quoted in [1], "Unfortunately, embedded systems are hardly covered in the 2013 edition of the Computer Science Curriculum, as published by ACM and the IEEE Computer Society [10]. However, the growing number of applications results in the need for more education in this area." Embedded systems courses in 2005 [3], before the announcement of Arduino [4, 5], were aimed at electrical engineering and computer science students and were redesigned to cover both software and hardware in the same subject [6]. Arduino was launched in 2005 and was based on Hernando Barragan's master's thesis with the aim of enabling designers (artists) to “explore electronic art and tangible media” [7]. In 2024, almost twenty years after the introduction of Arduino, courses on embedded systems are taught in biology [8], in courses for students of all disciplines in universities [9], in physics departments [10], in MOOC platforms for anyone without any prerequisite [11], in secondary schools and the audience is practically everyone. The Arduino software is so popular that it was downloaded 39 million times last year alone [12].

Since a large part of Arduino's target audience are not electronics and computer science engineers, and since the development of embedded/IoT applications is also extending to other disciplines, work is constantly being done to simplify programming. Some development environments for Arduino include visual programming environments with function blocks and some other development environments with block instructions that generate code that can be used in the Arduino IDE.

Programming without code is considered by some to be easier and this is the approach of XOD [13]. XOD is a visual programming language for microcontrollers, including Arduino. XOD is a block module programming environment that resembles an electrical schematic. In XOD, functions are blocks that have connections for inputs, outputs and other parameters to which wires can be connected. Arguments and data are transferred from one block to another via wires. It is comparable to the ladder diagram and the function languages of the PLC. The disadvantage is that the programmer has to learn the function of the individual blocks and the type of connection. The program is not written anywhere, it is in the design and the programmer has to follow the wires to describe what the program does.

With the Arduino PLC IDE, Arduino can be programmed in IEC 61131-3 standard languages. PLC programming has definitely proven itself so that engineers without programming skills can also use it. It has similar disadvantages to the visual programming languages.

Then there are a number of textual programming languages with text encapsulated in graphical blocks such as MicroBlocks [14], Ardublockly [15] and Ottodiy Software [16], descendants of Blocky by Google, which encapsulate the instructions in blocks. Programming languages with block statements protect the programmer from typing errors and from mixing different types of variables. It is very easy for the programmer to create new functions, which also look

better and are more user-friendly than in pure text languages, as each parameter can be preceded by a text. The downside is that a lot of coding is required and the program is in the canvas, so the programmer has to zoom in and out and move left and right to find specific code.

In this paper, we present a CAD tool for programming Arduino using tables that automatically generate code and are easy to fill in, understand, modify and troubleshoot intuitively. The environment in which the programming is done is a spreadsheet in LibreOffice, which we call WIDE, an acronym for Worksheet Integrated Development Environment. In Wide, the programmer can develop a program with all the expected features such as modularity, human interface, retention of user variables, fail-safety, statistics and different communication modes, automating up to 80% of the code.

The target group of the WIDE presented here is the entire Arduino community. The goal is to develop an environment in which most of the program is written in tables and the code is generated automatically. Filling in tables to configure a device or program a system is used in many industrial and commercial products as it is easier for the user as many details are hidden from him.

The rest of the article is organized as follows: Section II is the general description of the Arduino IDE spreadsheet preprocessor WIDE, Section III describes in detail how to set the input and output ports, Section IV describes how to implement the finite state machine model using three tables, Section V describes how to set the variables, Section VI describes how to write part of the program in free text instead of tables, Section VII describes how to download the program to the microcontroller, and Section VIII summarizes the results and discusses future work.

II. General description of WIDE (Worksheet Integrated Development Environment)

In this paper, we present the WIDE programming environment, a preprocessor for the Arduino IDE. The program is developed in different worksheets, and all program segments from the various worksheets are copied to a worksheet titled "AllProgram," which contains the entire program code. The program is then copied and pasted into the Arduino IDE for compilation and downloading to the board.

The programmer fills in tables in all but two worksheets, and the code is generated automatically. The programmer writes free text in worksheets labeled "functions" and "config" where they write user-defined functions.

WIDE uses the finite state machine paradigm for generating code. Finite state machines are a model of computation commonly used in embedded systems, that models the system as a set of states and transitions between them. WIDE generates code automatically if the programmer fills in three tables, one for the state description and two for the sequence of states.

The most important categories of worksheets are for:

- the definition of the input and output pins and the associated variables (pinsAnalog, pinsDigital),
- setting the states and the flow between the states (stateDescr, stateFlow, Emergency (flow for emergency states)),
- the setting of logical variables (variablesLogical),
- writing free text as in the Arduino IDE (functions, config),
- provision of a human-machine interface (print, Menu, LCD_I2C_2x16)),
- provision of communication (serialRead, I2C, IRread),
- setting the main functions of an Arduino program (setup, loop).

III. Worksheets for setting the input and output pins and the associated variables

The first phase in the development of a microcontroller project is the ideation phase where the problem and the solutions are formulated. Actions and measurements need to be performed for the project, with actions linked to the output pins of the microcontroller and measurements linked to the input pins. Instead of making notes on a piece of paper, the programmer can open the WIDE spreadsheet and start filling in tables with the setting information for the pins, having their notes in spreadsheet format (as shown in Fig. 1).

WIDE has two worksheets, two tables, one for setting analog signals and one for setting digital signals, where digital signals are inputs as well as outputs and PWM (Pulse Width Modulation = analog output signals).

Arduino pins	Shield pins	Comments	Names in program	Input
A0	Current of Motor A	Motor current of motor A 0 to 1025 corresponds to 0 to 3A , 1 is 3ma	CurrentA	INPUT
A1	Current of MotorB	same for motor B		INPUT
A2	connector	key key1 >900 value close to 5volt, no key then voltage is 0 volt and value is 0 to 7)	Keypad	INPUT

Arduino pins	Shield pins	Enegized state (used to create boolean variables from the analog inputs)	Type (boolean, int)	Create variable of pevious value (1 or empty)	Activate pull up resistor (1 or empty)
A0	Current of Motor A		int	1	
A1	Current of MotorB		int		
A2	connector	>500	boolean	1	

Fig. 1. Setting up analog pins (one table shown in two pict.)

To set up the analog signals, the programmer opens the “pinsAnalog” worksheet shown in Fig. 1. In the “ "Arduino pins" column, the name of the pin used in the “pinMode(An, INPUT)” command is entered. On the Arduino UNO, the analog pins are called “An,” while on other boards, they are simply named with a number. If a shield is connected to the board, the pin of the shield connected to the board’s pin is described in the “Shield pins” column. The “Comments” column contains all useful information about the pin, such as its function, the range of physical properties, and the associated electrical properties.

The "Names in program" column is the name actually used in the program. In the ideation phase, some programmers make up names and then use similar names in the actual program, which can lead to confusion. In WIDE, the name entered in the table during the ideation phase is exactly the same as in the program.

The "Energized state" column is used to create Boolean variables from analog values, as shown in line 3 for input pin "A2". This is useful if a signal is to be considered LOW (0 volts) at a value other than 0 volts. The column "Create variable of previous value" creates a variable with the value of the previous cycle of the program so that the rise and fall

of signals can be detected. The column "Activate pull-up resistor" is used to activate the pull-up resistor and is mainly used for inputs that are not connected so that they do not receive stray signals that would lead to random values.

After entering the data for the analog signals, the programmer enters the names of the digital pins that serve as input or output. All analog signals that were not used as analog can be used as digital in the AVR and other board families and can be programmed as digital in WIDE.

In addition, further input and output pins can be labeled as the project is refined.

The WIDE programming environment simplifies the process of setting up input and output pins and associated variables, allowing programmers to organize their projects more efficiently. The use of a spreadsheet-based preprocessor provides a structured approach to programming, making it easier for inexperienced programmers to develop microcontroller projects.

IV. Worksheets for Defining State Descriptions and Sequences (State Transitions)

With the finite state machine model, a complicated task is broken down into a series of smaller subtasks, each of which is executed in its own state. States are like self-contained microworlds that act on a discrete small task; actions take place in the state or during the transition to the state. Only one state is active at a time. A state stops executing its commands and control is transferred to another state when a trigger, Boolean variable or Boolean expression, or combination of triggers becomes active. Before entering a new state, a configuration takes place to prepare the environment of the new state that is about to become active.

The finite state machine model encourages programmers to focus on specifying each individual state and its transitions, which is typically easier to debug than the entire program. By breaking down the program into smaller, more manageable states, the programmer can concentrate on each state's specific functionality and ensure that it operates correctly. Additionally, the finite state machine model encourages programmers to consider all possible tasks and connections between them [17], promoting a more comprehensive and systematic approach to programming. This approach can help identify potential issues early on and ensure that the program functions as intended. As a result, the finite state machine model is a powerful tool for designing and implementing complex systems with multiple states and tasks.

The programmer can start by enumerating the states in the "stateDecr" tab shown in Fig. 2. For example, to control a garage door, the programmer can think of the different states that the door goes through: StOpened, StClosing, StClosed, StOpening as shown in Fig. 2, where St in front of the words stands for State.

STATE DESCRIPTION			LoopOfOperation	LoopOfChangeState
Value of state variable	State name	State description	Set variables or add actions or functions during state, use ";" after each action	Set varaiables or add actions or functions before entering state, use ";" after each action
10	StOpened	Opened (at open position)		ConfigOpened10(); printLCD216(0);
20	StClosing	closing (moving to close position)	operateMoving();	ConfigRunCCW();
30	StClosed	closed (at close position)		ConfigClosed30();printLCD216(0);
40	StOpening	opening (moving to open)	operateMoving();	ConfigRunCW();

Fig. 2. State description worksheet

A number defining the state is selected in the "Value of state variable" column, with numbers selected in ascending values in steps of 10 so that later, when a state needs to be divided into smaller states, a number in the same region is available. There is an associated name for each state number, and a description in words that is not translated into code. The "LoopOfOperation" column is then filled with the actions (outputs) that are executed during the active state. The next column "LoopOfChangeState" is filled with all actions (outputs) that are executed during the transition to the state.

State number	State name	Triger	New state number	New state name
10	StOpened	CombinedControlB	20	StClosing
20	StClosing	TimeToDecelerateOn	25	StClosingStop
30	StClosed	CombinedControlB	40	StOpening
40	StOpening	TimeToDecelerateOn	45	StOpeningStop

Fig. 3. State flow, transitions from state to state

The "stateFlow" worksheet shown in Fig. 3 records all transitions in a table format. The number of the state is entered in the "State number" column and the name of the state automatically appears in the "State name" column. A Boolean variable or a Boolean expression is entered in the "Trigger" column which, if true, controls the transition to the state in the "New state number" column. In the last column, the programmer adds notes to describe the transition in more detail.

State number	State name	Trigger	New state number	New State name
0	Anystate	watchDogHiCurrentOn	100	StTripByTimer
0	Anystate	watchDogMotorRunOn	110	StTripByHiCurrent

Fig. 4. Emergency states and transitions to them

In microcontroller systems and machines in general, there are emergencies that must be reacted to immediately, e.g. by an emergency stop switch. In WIDE, an emergency situation generates a trigger and transfers control to a state that handles the emergency situation, regardless of which state the program is currently in. The "Emergency" worksheet shown in Fig. 4 is the transition flow for the emergency states, which is the same as the "stateFlow" worksheet, but a zero is entered in the first column "State number", which stands for any state. When the corresponding trigger is activated, the control is transferred to the state in the "New state number" column.

The finite state machine model used in WIDE supports both a Moore and Mealy [17] type finite state machine model. Microcontrollers have analog values, and therefore, the models also have analog values, making the model a finite state machine with data paths [17]. According to Harel's work, the emergency states in WIDE are extensions of the state machine model to support a hierarchical structure [17]. This hierarchy allows the programmer to combine multiple states into a new hierarchical state, with all states except the emergency states being combined into a new hierarchical state.

The WIDE programming environment simplifies the process of defining state descriptions and sequences, allowing programmers to organize their projects more efficiently. The use of a spreadsheet-based preprocessor provides a structured approach to programming, making it easier for inexperienced programmers to develop microcontroller projects.

V. Worksheets for Defining Variables

In visual programming languages, the function blocks are connected with wires, and similarly, the functions in text

languages are connected with variables. The first variables are declared when configuring the input and output ports. Each value of an input pin is transferred to a variable and each value of an output pin is loaded from a variable. Once in each cycle, all inputs and outputs are updated by their corresponding variables.

Name	Output /user/ constant	Type (boolean, int, long)	Initial value
CountButtonsPressed	Output	long	0
SpeedSetA	user	int	40

	Update formula,			
Name	Variable or number	op	Variable or number	Condition to update (Boolean logical expression)
CountButtonsPressed	CountButtonsPressed	+	ControlInputRise	
SpeedSetA				

Fig. 5. Defining the variables (one table shown in two pict.)

Logical variables are created in a special worksheet "variablesLogical" in the form of a table, as shown in Fig. 5. In the "Name" column, the name of the variable is specified, then some information about its use, type and optionally an initial value. The next three columns "Update formula" are optional and a formula is entered there to update this variable. In the next column, which is also optional, a condition is entered that determines when the variable is updated. If the column remains empty, the variable is always updated. The feature to update the logical variables automatically and in every cycle was developed from the experience of several projects. The code for declaring the variables and the corresponding update formula are created automatically. There are also some other specific variable types that are created in special tabs, e.g. variables used in timers that are created in the "variablesTime" tab and statistics variables that are created in the "variableStatistics" tab and others.

VI. Worksheets for Writing Free Text

If the programmer needs to write actual code, there is a special tab for writing free text, i.e. as it is written in the Arduino IDE. The new code is written in the form of functions on the "Functions" worksheet. The newly created functions are entered in a list and the programmer can select them for use in other tabs such as the state-related tabs.

There is also a special category of functions that are entered as free text on the "Configuration" tab shown in Fig. 6, they are used to set up some variables before starting a new state. In the garage door project, for example, the direction of rotation of the motor is set differently when closing and opening the door. As the functions are mainly about setting variables, the columns "Variable", "Assignment","Value","Variable" are used for assigning variables and values to make it easier for the programmer. In the "Variable" column, the programmer selects from the existing variables in the project.

AA	Type	Name of function		Function		Variable	Assignment	Value	Variable
1	void	ConfigDoNothing() {							
				ConfigStop()					

Fig. 6. Worksheet "Configuration", for defining environment variables before starting a state

The WIDE programming environment provides a dedicated space for writing and organizing free text code, allowing programmers to develop and integrate custom algorithms seamlessly. Additionally, the ability to configure environment variables before starting a new state streamlines the development process, enhancing the overall efficiency and organization of the project.

VII. Downloading the Program to the Board and Debugging

The program is written automatically when the worksheets are filled with data. There are two important worksheets that are filled in automatically. One is the "Setup" worksheet, which collects all the setup instructions from the other worksheets and transfers their contents to the setup function of the Arduino program. In this worksheet there are also some rows where the programmer can add commands that need to be executed during setup.

	A	B	C	D
1	void loop()	{	Formula of col. A	void loop() {
2	loupdateDigital()	;	=$pinsDigital.AF2	loupdateDigital() ;
3	loUpdateAnalog()	;	=$IOUpdateAnalog.B18	loUpdateAnalog();
4	variableIOupdateAnalogDif()	;	=$pinsAnalog.AK1	variableIOupdateAnalogDif();
5	variableIOupdateDigitalDif()	;	=$pinsDigital.BJ2	variableIOupdateDigitalDif();
6	variableUpdateLogical()	;	=$variablesLogical.T1	variableUpdateLogical();
7	TimingUpdate()	;	=$variablesTime.Y1	TimingUpdate();
8	//	;	#N/A	//;
9	variableUpdateKeypad()	;	=$variablesKeypad.X2	variableUpdateKeypad();
10	variableUpdateCounters()	;	=$varCounters.R1	variableUpdateCounters();
11	variableUpdateSerial()	;	=$serialRead.G5	variableUpdateSerial();
12	//	;	=$I2C.P16	//;
13	//	;	#N/A	//;
14	//	;	#N/A	//;
15	//	;	#N/A	//;
20	EmergencyStateFlow()	;	=$emergency.N2	EmergencyStateFlow() ;
21	LoopOfOperation()	;	=$stateDescr.J5	LoopOfOperation();
22	LoopOfStateFlow()	;	=$stateFlow.N2	LoopOfStateFlow() ;
23	//	;	#N/A	//;
24	variableUpdateAnalog0()	;	=$pinsAnalog.AN1	variableUpdateAnalog0();
25	variableUpdateDigital0()	;	=$pinsDigital.BM2	variableUpdateDigital0();
26	}			}

Fig. 7. Arduino function "Loop", continuously executed functions in the loop function

The other worksheet is the "Loop" shown in Fig. 7, where all the functions to be executed in the Arduino function loop are collected. In this loop, the commands are entered in the order in which they are to be executed. First, the digital inputs are transferred to the corresponding variables and the output pins are updated with the values of the corresponding variables that were calculated in the previous run shown in rows 2 and 3. Then the differential variables are calculated from the current values and the previous values of the input and output variables, as shown in rows 4 and 5. The logical variables are updated in row 6, taking into account the variables updated variables above in rows 2 to 5. This is followed by further updates for variables and communication channels shown in rows 7 to 12, followed by some empty lines for the programmer or for future additions to WIDE. Then there is a check for emergency signals that could put the program into an emergency state (row 20). This is followed by the actual operation "LoopOfOperation()" in the state that was selected in the previous cycle and whose code can be found in the "stateDescr" worksheet. Then the "LoopOfStateFlow()" function determines whether there is a transition to a new state. Finally, the variables containing the values of the variables from the previous cycle are updated shortly before the end of the loop in rows 24 and 25. Column "Formula of col. A" contains the formula contained in column "A" as well as the tabs from which the code is collected. Column "D" contains the code that is inserted in the tab containing the entire program.

When a macro is activated, the worksheet is saved and the entire program is copied from the first tab. The program is then pasted into the Arduino IDE using the key combination “Ctrl-A,Ctrl-V,Ctrl-S”. The entire file is first selected, then the clipboard is pasted and then saved. The program is downloaded from the Arduino IDE to the board.

During the compilation phase, the Arduino IDE creates an error list in which the line number and the type of error are specified. Using the line number, the programmer can find the code in the same line number in the "AllProgram"

worksheet, where all the code is located. Next to the code is a formula that shows the worksheet and the cell from which the code was copied. Each tab fulfills a specific task, so that the error is contained in its environment and can logically be corrected. Experience with WIDE has shown that it is easy to correct errors.

In WIDE it is easy to add print statements and interactive menus so that the programmer can interact with the program. WIDE has no breakpoints, but it is easy to set interactive menus in different states.

The WIDE programming environment has a simple process of downloading the program to the board and debugging, allowing programmers to develop and test their microcontroller projects efficiently. The use of a spreadsheet-based preprocessor provides a structured approach to programming, making it easier for inexperienced programmers to develop microcontroller projects.

VIII. Conclusion

This article introduces a Worksheet Integrated Development Environment (WIDE) that acts as a preprocessor for the Arduino IDE and supports a finite state machine model with data paths and a hierarchical state model. The proposed integrated worksheet development environment can be used to easily teach undergraduate and graduate students how to program Arduino-based microcontrollers.

By automatically generating the code, syntactical errors are avoided, which leads to early success and encourages the programmer to add more features and complete a project with many functions. The programmer is expected to be familiar with variables, logical expressions, logical comparisons and simple math skills.

The programmer begins with WIDE in the idea phase by filling the tables with variables and states to create a very simple project that they can test and debug, then add more variables and states. Incremental development makes debugging a project much easier and success is an important factor for engagement in this area.

In the future, more projects need to be developed in WIDE for a variety of Arduino-based boards to test the method for ease of programming. The use of WIDE provides a structured approach to programming, making it easier for inexperienced programmers to develop microcontroller projects.

Overall, the WIDE programming environment provides a powerful tool for developing microcontroller projects, simplifying the process of programming and testing, and encouraging engagement and success among students and hobbyists alike.

References

[1] P. Marwedel,”Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things”, Fourth ed. Springer, 2021, pp 7.

[2] P. Marwedel,”Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things”, Fourth ed. Springer, 2021, pp VII.

[3] P. Caspi, J. Jackson, “2005 Workshop on Embedded Systems Education”, (a satellite event of EMSOFT 2005), New Jersey, 2005

[4] H. Barragán, "The Untold History of Arduino", https://arduinohistory.github.io/, (retrieved Jan. 29, 2024)

[5] M. Banzi, "History of Arduino", https://forum.arduino.cc/t/history-of-arduino/118744 (retrieved on Jan. 29, 2024.

[6] P. Marwedel, “Towards a common basis for education in embedded systems development”, presented at the “2005 Workshop on Embedded Systems Education”, NJ, USA, .

[7] Hernando Barragán, “Wiring: Prototyping Physical Interaction Design”, Dissertation, | Interaction Design Institute Ivrea | June 2004,

[8] "Engineering Biology in Cambridge", https://www.engbio.cam.ac.uk/news/no-code-programming-biology-workshop-recordings- now-available Cambridge College (retrieved Jan. 29, 2024)

[9] "Eduino project at the University of Edinburgh", https://eduino.ed.ac.uk/wp/, (retrieved Jan. 29, 2024)

[10] Furman University, "PHY-433 Introduction to Embedded Systems", https://catalog.furman.edu/preview_course_nopop.php?catoid=17&coid=33815, (retrieved Jan. 29, 2024)

[11] Coursera, Univeristy of Irvine, California,"An introduction to programming in the Internet of Things (IOT)specialization", https://www.coursera.org/specializations/iot, (retrieved Jan. 29, 2024)

[12] Arduino, "Announcing the Arduino IDE 2.0 (beta)", https://blog.arduino.cc/2021/03/01/announcing-the-arduino-ide-2-0-beta/, (retrieved Jan. 29, 2024)

[13] XOD. "A visual programming language for microcontrollers", https://xod.io/ (retrieved Jan. 29, 2024)

[14] "MicroBlocks is a blocks programming language for physical computing inspired by Scratch", https://microblocks.fun/, (retrieved Jan. 29, 2024)

[15] "Ardublockly Visual Programming for Arduino", https://ardublockly.embeddedlog.com/index.html, (retrieved Jan. 29, 2024)

[16] Ottodiy, "Standalone software for coding robots and IoT projects", https://www.ottodiy.com/software (retrieved Jan. 29, 2024)

[17] F. Vahid and T. Givargis, "Embedded System Design: A Unified Hardware/Software Approach", http://dsp-book.narod.ru/ESDUA.pdf, 1999, pp 8-5