# MODI – A proposal of a visual tool to simulate and synthesize software applied to embedded systems.

da Silva, Marcos Antônio Vieira, MSc. - LPCI/ITI.
de Souza, Antônio Heronaldo, Dr.- DEE/UDESC.
Ferreira, Elnatan Chagas, Dr. - DEMIC/FEEC/UNICAMP.

**Abstract**

This work presents a visual environment applied to embedded systems based on microcontrollers. The system proposed uses the concepts of visual programming, that is, it works with visual elements such as fluxogram blocks and icons, to represent the algorithms that will translated to the final code of the application. The objective of this kind of approach is to increase the productivity of the systems developers, because it is not necessary to write any line of code even in C or Assembly language. The system is able to simulate and generate the Assembly code of the application.

**Introduction**

Embedded systems had always their programming associated with Assembly language, because of their low amount of memory and because they are always used in applications strongly tied to hardware.

However, two other languages are often used: BASIC and C language. In spite of their different advantages and disadvantages, these languages have in common the fact that the source-program is expressed by a text. Thus, they are close to human language. Nevertheless, the level of details in the program instructions, synthesized in text format, causes its writing and maintenance, in many cases, too complicated.

The current concepts of development tools explore the means of working in visual operational environments (Windows©, Openwindows, etc).

We can verify by looking for programs such as LabView[1] and LabWindows (National Instruments), MPLab (Microchip), among others, that the use of symbols and images to express ideas and algorithms is being more accepted by the software programers.

**Text languages *versus* visual languages**

The life cycle of the system can be synthesized by the following steps[2]: a) problem definition and feasibility study; b) analysis and system design; and c) implementation and maintenance.

Frequently, in short applications or in applications that require quick solution, it is common to start directly from the implementation step, what leads to scarce or inexistent documentation. Even in applications that take into account the whole developing cycle, the specifications generated in the system analysis can be misunderstood in the implementation step, because they are usually executed by different persons.

One solution found, considering the troubles due to text languages, was to make visual compilers that allow an automatic program generation for text language or finally for machine code. With this solution, problems in the communication between systems analysts and programmers are eliminated, and also the hard job of the codification step. Besides, the use of visual languages turns the documentation always updated, because visual compilers use their own tools to generate documentation as input source.

It is known that the human being has more ability to interpret an algorithm by using visual language than text language.

Researches in the neurocognition area (Springer and Deutsch[3]) showed that the left brain hemisphere processes information sequentially, orally and logically. The right hemisphere processes information simultaneously, visually and spatially.

The text mechanisms have very little spatial information, thus, the right hemisphere do not contribute significantly with these mechanisms. Consequently, only half of the brain can be influenced in the comprehension process, when these techniques are used.

Contrary to text technics, the visual mechanisms of algorithms comprehension, like structured fluxograms, tend to stimulate the whole brain. The

visual technics have also sequential, logical and, in less proportion, verbal information. Thus the left hemisphere of the brain is stimulated. Besides, the visual technics also stimulate the right hemisphere, because it contains lots of visual-spatial information.

Research made by B. A. Calloni and D. J. Bagert [4] showed that visual languages are more intuitive, and facilitate learning and algorithm comprehension.

They developed a procedural programming language, based on icons to teach programming, called BACII and performed a comparison with the text language PASCAL, widely used in universities to the teaching of basic programming.

The research was made following the results obtained in the experimental work of David A. Scanlan in the algorithms comprehension area. After the tests performed in 1988 [5] and 1989 [6], it was possible to concluded that visual methods enable the necessary abstraction of the syntax details and show themselves better than the text methods in the intuitive mental process to the teaching (and comprehension) of algorithms.

Recently, the visual compilers or visual environments are consolidating an important role in the computer programming. This tendency has been increasing thanks to the great technological development that enabled the price reduction of high-resolution video monitors.

## Embedded systems development tools

Analyzing the development tools of systems based on microcontrollers, we can notice that they present an IDE (Integrated Development Environment) based on the textual description (generally in C language) of the code to be generated, which contradicts the current tendency of visual environments.

The use of visual language implies in the substitution of the typing of text that contains the mnemonics, in the case of the programming in Assembly, or the commands or functions, in the case of C language, by visual elements that assist in the final understanding of the system. This approach also eliminates errors in the code attributed to mistyping.

Considering the approaches of the visual programming languages established by M. Burnett and M. Baker[7], MODI has characteristics of data flow (VPL-II.A.3), diagramming language (VPL-II.B.1), with flow control (VPL-III.B), translators (VPL-IV.D), language of general application (VPL-V.A) and theory of icons (VPL-VI.B).

MODI can be described as a fluxogram generator where several functional blocks can be placed .

These blocks contain a set of instructions represented by icons that implement some tasks of the application in development.

MODI also uses the resources of ONAGRO[8][9] to compile the algorithm to the Assembly language of a microcontroller specified (the microcontroller used, in this first version, was Intel 8051).

The simulator makes the logical simulation of the application to detect 'bugs' in the generated code. The visualization of the simulation parameters can be done by using some virtual instruments such as: voltimeter, osciloscope, thermometer, display, leds and others. We can interact with the system, in simulation time, through virtual devices such as: switches, push-buttons, etc.

Finally, by using the serial port of the microcomputer, the transfer of the generated code can be made to a system that burns EPROM memories, closing the cycle of software development.

This software can be executed in any PC microcomputer with the Windows© system.

Reference [10] brings a comparative among programs similar to MODI that can highlight the qualities of this software related to characteristics such as: usability, functionality and flexibility.

## The software description

A complete vision of the MODI IDE, with its main windows activated, can be seen below in Figure 1.

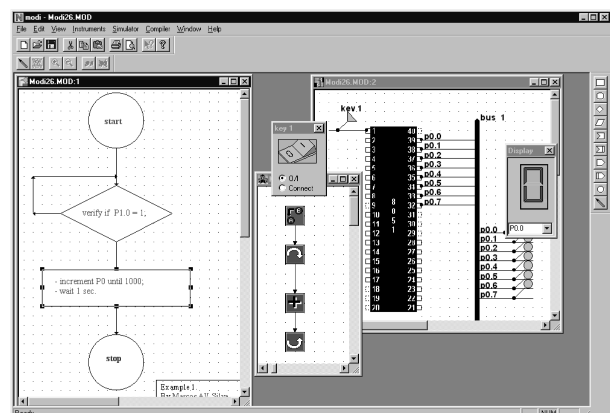


Figure 1: MODI framework.

To start a project in MODI, the user has, in the initialization screen, a window for fluxogram edition where the division of the project in operational blocks is made. The adopted symbology followed the pattern recommended by CCITT[11]. In this screen we can also put some textual information that helps us to remember the aspects the block is implementing.

After the description of the fluxogram of the system in development, we pass to the next step that is the functional description of each block by using icons. These icons represent the operations accomplished by the microcontroller.

## The operation icons

The *operation icons* represent the basic operation that can be executed in the language. They are similar to operation codes of a machine instruction, *i.e.*, they essentially indicate the nature of processing that is required to be executed. It is necessary, in addition, that we indicate the parameters that will be used in the operation. These parameters are usually variable identifiers, IN/OUT ports and constant identifiers.

The language offers icons already defined to execute the most common operations. However, icons more specific are offered to dedicated applications. Besides, the user can create its own *operation icons* and assign which task should be executed, because the language has a type of icon defined by the user.

They are organized in operational classes, where each class is represented by a different figure and indicates a group of similar operation. In most of the cases the icon that represents the class is the same that indicates the most used operation in that class (see figure 2). However, there are classes that have one specific icon to represent it, different from the other class *operation icons*. This is shown in figure 2.

The operation icons were divided in the following functions:

- Attribution;
- Arithmetic operations;
- Logical operations;
- Displacement of bits;
- Comparisons;
- Looping;
- Subroutines;
- Delay;
- Edge generation ;
- Seek in vector;
- Operation of serial communication;
- Vectorial transformation;
- Junction of bits;
- Operation defined by the user;
- Connections;
- Junction;
- Beginning and end of the program.

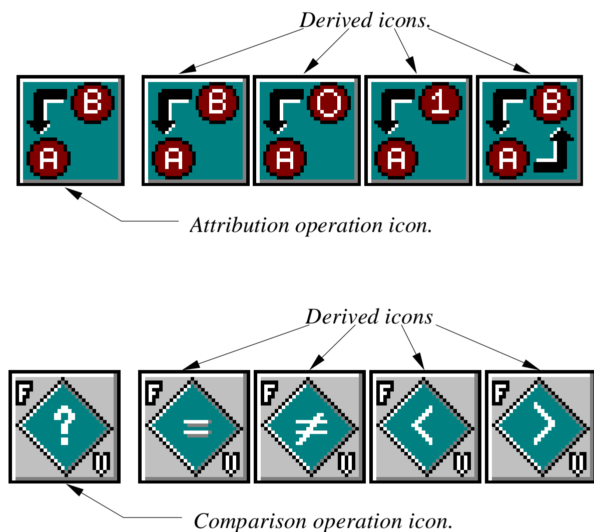



Figure 2: Some operation icons.

The drawing of these icons tried to be as intuitive as possible, and each icon has a dialog to input some parameters such as variable names, numerical values, comments and others.

The icons are linked in such way that they form a sequence of tasks that represents the operation block.

## The simulator description

After finishing the detailing of the blocks, we can simulate the application to detect any logical errors. To do that, we have a third window, called schematics window, where we can enter some hardware elements that represent the micocrontroller itself, latches, wires, buses, etc., and in these elements we can attach some observers (meters) such as voltimeters, leds, displays, osciloscope, etc., that help us to analyze if the

project is working as desired. Some of these meters

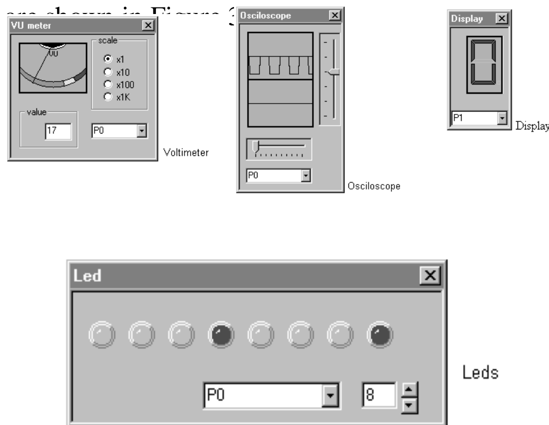



Figure 3: Observers or meters.

Besides the meters, actuators can be added to schematics window to interact with the application in simulation time. The aim of this interaction is to try to simulate application that collect external data through the microcontroller ports. These actuators represent basic functions such as:

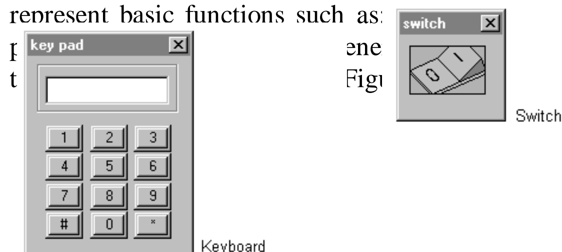



Figure 4: Actuators

The simulator has three operation modes:

- Automatic: Where the application is simulated block to block following the fluxogram.
- Step-by-step (fluxogram): The simulator waits for the user to press a key to follow to the next block of the fluxogram to be simulated.
- Step-by-step (Icons): In the icons edition window, the user simulate each icon individually.

In any operation mode, the simulator assigns the block or icon that is being simulated, and this helps us to follow the execution flux.

After simulating the application, the Assembly code can be generated through the compiler of icons, and the final result is shown in illustration 5.

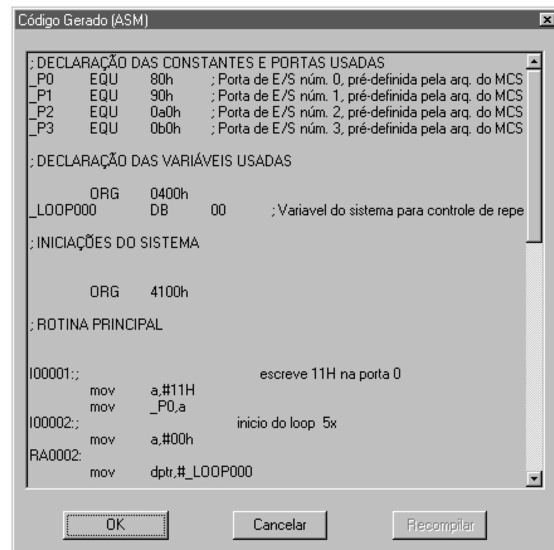



Figure 5: An example code.

**The code structure and the translation mechanisms**

MODI has icons oriented code, i.e., the operations are described by standard symbols (the *operation icons*) arrangement. These arrangements have a syntax guided by a graphic editor, in such a way that only the right operational statements are allowed. It not only reduces the syntax tests in excess that occur in traditional languages, but also generates a code structure more predictable to the compiler.

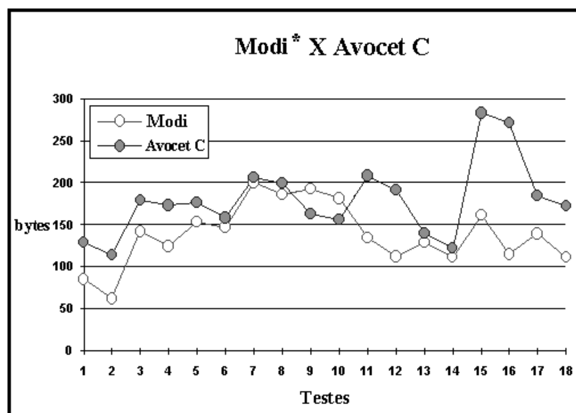
In the code generation, it is tried to choose a mechanism that was the most generic as possible, accordig to the microcontroller type. Thus, the major part of generated code uses the essential instructions and the microcontrollers architecture basic registers. Obviously, in the operations related to

microcontrollers characteristics, this is impossible, because each microcontroller has its particularities.

## Results

Eighteen tests were made by using the icon compiler of MODI in 5 applications: Transductor Linearization, Keyboard Scanning, Step-Motor Control, Temperature Control and Eprom Writer. AVOCET C V1.217 was chosen to be a reference for the comparison of the size of the code generated. Figure 6 brings the results of these tests. The size of de code was influenced by factors such as: Memory model (Large or Small) and I/O Ports Mapping (Memory mapped or Internal).

The code generated has some restrictions: the type of data is limited to Byte and Word and there is no support to floating point data and polydimensional vectors. But, in spite of these limitations, the size of the generated code to the 5 applications mentioned above, was smaller, in most of the cases, than the code generated by AVOCET, which is a complete C compiler.




*The icon compiler used in Modi is called ONAGRO.

Figure 6: Size of the code.

## Conclusion

This paper showed a proposal of development tool that uses visual programming applied to embedded systems. The aim of this approach was to make the microcontroller programmer more independent of the typing of lines of code in Assembly or C language. The automatically generated code is free of errors and its size, comparing with others compilers, is not so extensive.

At the moment we are working in collaboration with ITI (Information Tecnological Institute) to apply this approach to develop systems that use ASIC based on microprocessor core embedded with a gate array area.


## Acknowledges

We would like to thank CAPES for the financial support, DEMIC/UNICAMP and LPCI/ITI for the material support for this project.


## References


[1] JamaL, Rahman; Wenzel, Lothar: " The Applicability of the Visual Programming Language LabVIEW to Large Real-World Applications" - Proceedings 11th International IEEE Symposium on Visual Languages - Darmstadt, Germany – September/ 1995.

[2] Davis, Willian S. *Systems Analysis and Design*. Addison-Wesley, New York (1983).

[3] Springer, S. P., Deutsch, G. *Left Brain, Right Brain*. W. H. Freeman and Company, New York (1985).

[4] Calloni, Ben A., Bargert, Donald J. "ICONIC Programming in BACII vs Textual Programming: which is a better Learning Environment ?" *SIGCSE Bulletin,* 26 (1), (March, 1994), 188-192.

[5] Scalan, David A. "Should Short Relatively Complex Algorithms be Taught Using Both Graphical and Verbal Methods ?" *SIGCSE Bulletin,* 20 (1), (February, 1988), 185-189.

[6] Scalan, David A., "Structured Flowcharts Outperform Pseudocodes: An Experimental Comparison." *IEEE Software,* 6 (5), (September, 1989), 28-36.

[7] Brunett, Margaret M., Baker, Maria J.; "A Classification System for Visual Programming

Languages" – Department of Computer Science-Oregon State University; Technical Report – 1994.

[8] de Sousa, Antonio Heronaldo; "Onagro Um Ambiente Gráfico para Desenvolvimento de Software para Microcontroladores" – Master thesis – UNICAMP/1995.

[9] de Sousa, Antonio Heronaldo; da Silva, Marcos Antonio V.; Ferreira, Elnatan Chagas; *ONAGRO - A Graphical Environment to the Development of Microcontrollers Software*; II World Automation Congress - WAC II; 7/05/1996 a 30/05/1996 Montpellier - FRANCE.

[10] Valaer, Laura A.; Babb II, Robert G.; "Choosing a User Interface Development Tool" - IEEE SOFTWARE Vol. 14, No. 4: JULY/AUGUST 1997, pp. 29-39.

[11] "Functional Specification and Description Language (SDL)"; CCITT ( The International Telegraph and Telephone Consultative Committee) – Volume VI – fascicle VI.7. Recomendations Z.100-Z.104 (http://www.itu.ch/itudoc/itu-t/rec.html).

[12] Freeman, Elisabeth; Gelernter, David; Jagannathan, Suresh : "In Seach of a Simple Visual Vocabulary" – Proceedings 11th International IEEE Symposium on Visual Languages - Darmstadt, Germany – September/ 1995.

[13] Nickerson, Jeffrey V. "Visual Programming" - Ph.D. Dissertation. New York University- 1994.

[14] Sears, Andrew; Lund, Arnold M.; "Creating Effective User Interfaces" - IEEE SOFTWARE Vol. 14, No. 4: JULY/AUGUST 1997, pp. 21-24.