A Target Environment Programming Language to Improve Developer's Productivity – A Case Study

José Metrôlho, Mónica Costa, Fernando Reinaldo Ribeiro, Eurico Lopes

***Abstract*—** This paper presents a target-specific programming language (TSL) that was designed to improve the design cycle of code generation for an industrial embedded system. The native assembly code, the new language structure and their constructs, are presented in the paper. The proposed TSL is expressed using words and terms that are related to the target's domain and consequently it is now easier to program, understand and to validate the desired code. It is also demonstrated the language efficiency by comparing some code described using the new language against the previous used code. The design cycle is improved with the usage of the TSL because description time and debug time are significantly reduced with this new software tool. This is also a case of university-industry partnership.

***Keywords*—**Compilers and Interpreters, Embedded Systems, Programming Languages.

I. Introduction

THE development time in industrial informatics systems, in industry environments, is a very important issue for competitiveness. Companies that develop solutions for industry usually deal with several levels of abstractions, from high level languages to assembly. As we move toward the high to low level languages the effort is greater and the developers generally want to work with more abstract levels. However, it is very common for these companies to handle with specific embedded devices, that require specific programming languages, mainly low level programming languages. Although low-level languages have the advantage that they can be written to take advantage of any peculiarities in the architecture of the microprocessor/microcontroller and can be extremely efficient, writing a low-level program takes a substantial amount of time, as well as a clear understanding of the inner workings of the processor itself.

Domain-specific languages (DSL) can play an important role in facilitating the software developers' task increasing its productivity. DSL are programming languages for solving problems in a particular domain. They are much more expressive in their domain and allow faster development of programs allowing solutions to be expressed in the idiom and at the level of abstraction of the problem's domain. DSL and TSL provide several advantages over general purpose programming languages, namely [1] concrete expression of domain knowledge, direct involvement of the domain expert, expressiveness, modest implementation cost, reliability, training costs and design experience. These types of programming languages are usually small, more declarative than imperative, less expressive and more attractive than general-purpose languages because of easier programming, systematic reuse, better productivity, reliability, maintainability, and flexibility.

In this paper we describe a TSL to improve developer's productivity in industrial embedded systems in the scope of University-Industry collaboration. Preliminary tests show that the TSL decreases the development time and increases developers' productivity.

The remainder of this paper is structured as follows: in Section 2, we introduce the target environment and in Section 3 we describe the native language of the hardware. In Section 4 we present the formalism of the TSL and in Section 5 we present preliminary tests. Finally, Section 6 concludes this paper with a discussion of the pre and pos systems implementation and pointed out some directions of future work..

II. The Target Environment

Due to confidential constraints, we will not present details about the module used by the company. This company develops industrial informatics solutions for other companies, mainly to the automotive industry. But in general terms, and to introduce the theme, we can inform that the target module (see figure 1) is used to actuate over relays and has several internal units like timers and I/O ports (see table 1) that can be configured using a dedicated assembly language. Some module features are: 6 Digital I/O pins; 3 Transistor Outputs; 1 Relay outputs; 2 Analog inputs; 1 counter and 8 32 bit timer with a time resolution of 1 ms.

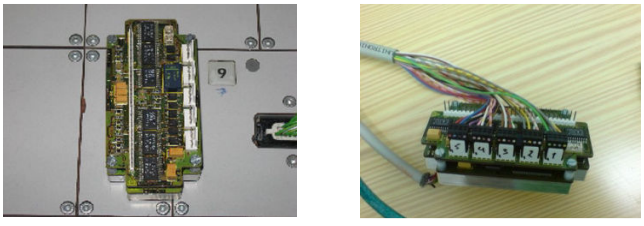

Fig. 1 Hardware module

Those modules have a set of registers whose bits have particular meanings. These registers can be of different types: read, write or read/write. A feature of the assembly language is that any time the designer wants to read or write something, he must knew the register number and each the bits meaningful. This demands a lot of manual readings and becomes repetitive for some applications.

Another feature is that the necessary instructions to build applications are scarce and all well defined. As example a read or write relay operation is almost the same, but requires knowing the name of the register and to know the bit number that must be set or reset to act according the desired action. Additionally the code is only readable and understandable by developers that have knowledge about that particular assembly. A language that could be more intuitive and make code more documented and understandable was desired.

This leads to the idea that a high-level programming language, more adapted to the field, can be designed with proper and intuitive constructs, like in this case relay(on), or relay(off) avoiding details and constants that are well known and thus improving developers' productivity.

The development of applications, before the new tool described in this paper, was done by writing assembly code that is uploaded to the modules by a proprietary application. This fosters a deep knowledge about the assembly and about the registers and the meaning of its bits. To develops applications with a low time to market a more abstract tool is needed, this s the goal of our approach. This paper describes a tiny language designed and implemented to allow quicker developing time and also generated assembly code documented and indented properly to foster faster detection of software bugs.

III. THE NATIVE LANGUAGE

Here we present some of the assembly language features. The following piece of code (see figure 2) shows a sample of the type of details and structure which must be introduced by the programmer.

```
$init
  ...
MOVI(T0VAL,0)
MOVI(T0MAX,1000)
MOVI(T1VAL,0)
MOVI(T1MAX,500)
...
WREG(A2,5,255)
MOVI(A13,2)

$code
RREG(A4,6)
ANDI(A10,A4,8)
SRI(A10,A10,3)
ANDI(A11,A4,16)
SRI(A11,A11,4)
ANDI(A12,A4,32)
SRI(A12,A12,5)
IFEQ(T0VAL,T0MAX)
ORI(A10,A10,2)
MOVI(T0VAL,0)
ENDIF
....
$end
```

Fig. 2 Sample of native assembly code.

As it can be observed in Fig. 2, the user must be aware of the native assembly and a constant set of variables that can be used and must deal with information about the registers and also regarding timers, he/she must convert the time unit to milliseconds. These details are prone to generate errors.

So this case-study has fostered the design of a tiny language to describe applications for an embedded device that is used in industrial environments. The main goals of the new language are, transform the design of new programs as high level as possible, use intuitive constructs, allow some verifications to avoid errors, make the code documented and automatically idented. In other terms, make the design time shorter with less design effort for the designers of applications involving that embedded microcontroller.

IV. THE NEW LANGUAGE

Here we will describe the developed tool. First we will present the structure and then the constructs of the new language.

A. The new language structure

The new structure has 2 sections, one for declarations and other for code. This is similar to the target assembly, however the section delimiters are now '{' as in common languages.

Within each section the user will now avoid details and will focus on actions or constructs that are common to programmers and for designers of that kind of applications. The constructs were defined to make clear the programs, and to avoid details. The tool will then generate the proper code..

B. The new language constructs

Number After studying the possible instructions and the final result in the module, we define a set of keywords to allow an easy and intuitive definition of those instructions. As example to control a digital output the bit 0 of the module register 7 must be set/reset. In assembly this is dome using the instruction WREG(A0,7,1). As we can observe the user must put the number of the target register, a variable that transport the value that must be put over the bit (ex: since A0=0 then the bit 1 will be reset), and the number of the bit that will suffer the change (in this case is the 1st bit). However based on the "clients" feedback we notice that this output is always used for relay control. So, we defined a language construct "relay" with a single switch that makes this description easy and intuitive. Next we present in the left the new language construct usage and on the right the generated/corresponding assembly.

```
relay(on);      →  WREG(A0,7,1)
relay(off);     →  WREG(A1,7,1)
```

Other examples of usage of the new language constructs and the corresponding assembly:

```
var A31=2;      →  MOVI(A31, 2)
attr A31=A5;    →  MOV(A31, A5)
IN (0,A3);      →  RREG(A3, 8)
                   ANDI(A3, A3, 1)
startT(0);      →  MOVI (T0VAL,0)
defT(1,1500);   →  MOVI (T1MAX,1500)
stopT(1);       →  MOVI(T1VAL,1501)
```

Fig. 3 New language constructs.

We've defined a set of keywords for the language, in small number due to the simplicity of the assembly. The total of keywords is 28 and all them are presented in the following table.

TABLE I
LANGUAGE CONSTRUCTS

init	JMP	INOUT_R	INPUTS_W	attr	defT
code	JMPI	INOUT_W	if	rele	tstTLimit
end	JMPIX	OUTPUTS_R	elif	delay	stopT
OUT	IOCTL_R	OUTPUTS_W	else	startT	
IN	IOCTL_W	INPUTS_R	var	setT	

This is also interesting because a small set of keywords represents a small time to learn the language.

C. The generation chain

To implement this code converter, from the new language to the target assembly, the software chain can be represented as in Figure 4.

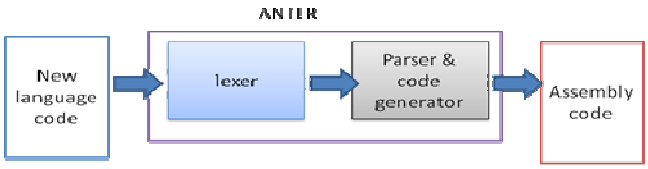

Fig. 4 Generation chain.

The code was developed using Java [2] and within the Eclipse IDE [3]. To implement the lexer and parser we used ANTLR (ANother Tool for Language Recognition) [4]. It provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages [4] including Java.

V. TESTS

In terms of tests the achievement of a smaller design time was the main goal. To test it we ask the development team of the partner company to give us their feedback. The feedback was positive since the new tool allows to reach sooner and on a more proper way the target assembly. The code becomes easily documented and the code is also readably.

In terms of the generated assembly the result is the same, as expected. However, now the user focus on the desired goals and the tool translates that for proper assembly.

In the following figures we present the code of a program in the new language and the resultant generated assembly.

```
program  Exemplo{
  init{
            IN(4, A10);
            startT(7);
              var  A10=0;
              var  A11=1;
              var  A12=100;
              rele(off);
              defT(0, 5s);
              startT(0);
              defT(1,5s);
              stopT(1);
              defT(2, 4h);
              startT(2);
              defT(6, 19h);
              stopT(6);
                    ...
              }
code{
            if(tstTLimit(0)){
              rele(on);
              startT(1);
              stopT(0);
            }
            if(A10==A12){
              stopT(0);
              stopT(1);
              startT(3);
              var A10=0;
            }
            ...
      }
}
```

Fig. 5 New language code.

As we can notice in Figure 6, the generated code is automatically commented and formatted. This gives the designer an easier method to understand code, even at the assembly level.

Also the generated code in the native language is more extensive than the new proposed language. This means that the present effort is less than previously when descriptions were done in assembly. Also the tools detects errors and signals the line were they occur, this fosters better software development in terms of developing time and code quality. Also the reduced number of language constructs fosters a quick learning of the language descriptions features.

```
The generated code:
## Init Section
$init
# Reserved variables
   MOVI(A0,0)
   MOVI(A1,1)
# Read Input 4
   RREG(A10, 8)
   ANDI(A10, A10, 5)
# Initialize Timer T7
   MOVI(T7VAL,0)
# Initialize variable
   MOVI(A10,0)
# Initialize variable
   MOVI(A11,1)
# Initialize variable
   MOVI(A12,100)
# Rlay OFF
   WREG(A0,7,1)
# Timer T0 Max value
   MOVI(T0MAX,5000)
# Initialize Timer T0
   MOVI(T0VAL,0)
# Timer T1 Max value
   MOVI(T1MAX,5000)
# Stop Timer T1
   MOVI(T1VAL,5001)
# Timer T2 Max value
   MOVI(T2MAX,14400000)
# Initialize Timer T2
   MOVI(T2VAL,0)
# Timer T6 Max value
   MOVI(T6MAX,68400000)
# Stop Timer T6
   MOVI(T6VAL,68400001)
## Code Section
$code
IFEQ(T0VAL,T0MAX)
   # Relay ON
      WREG(A1,7,1)
   # Initialize Timer T1
      MOVI(T1VAL,0)
   # Stop Timer T0
      MOVI(T0VAL,5001)
ENDIF
 IFEQ(A10,A12)
   # Stop Timer T0
      MOVI(T0VAL,5001)
   # Stop Timer T1
      MOVI(T1VAL,5001)
   # Initialize Timer T3
      MOVI(T3VAL,0)
   # Initialize variable
      MOVI(A10,0)
ENDIF
 ...
$end
```

Fig. 6 Generated code.

VI. Conclusion and Future Work

Preliminary experiments and tests show that using the new language a short effort and design time is needed to achieve better goals. The goals are the assembly code to be uploaded for embedded systems that is used for the automotive industry. The infrastructure can be easily adapted for other similar targets. The software is running on a platform independent basis, so portability would be not a problem to other environments.

As future work we want to implement and editor with code complete feature for our tool, to increase even more the development efficiency.

References

[1] D. Spinellis, "Notable design patterns for domain specific languages," Journal of Systems and Software, vol. 56, pp. 91-99, 2001.

[2] S. Microsystems, "Java," [Online] Available at: http://java.sun.com/, [Access date: 2009, October].

[3] E. Foundation, "Eclipse," [Online] Available at: http://www.eclipse.org/, [Access date: 2009, October].

[4] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, 2007.