A System for Automatic Testing of Embedded Software in Undergraduate Study Exercises

Voin Legourski, Christian Trödhandl, and Bettina Weiss
Vienna University of Technology,
Embedded Computing Systems Group E182-2,
Treitlstr. 3/2, 1040 Vienna, Austria
Email: legourski@yahoo.com, {troedhandl,bw}@ecs.tuwien.ac.at

Abstract— **As student numbers in embedded systems lab courses increase, it becomes more and more time-consuming to verify the correctness of their homework and exam programs. Automatic verification can vastly improve the speed and quality of such tests. This paper describes a system that can carry out black-box tests to verify whether the embedded software running on a target system meets predefined requirements. To this aim, we employ a special test board using an ATmega128 microcontroller which is connected to both the target system and to a host computer. Tests can be selected and started remotely, the results are presented to the user on the host. Monitoring and control via Internet is also easily possible. A special meta-language is used to describe the correct behavior of the tested program, and this description is compiled and downloaded to the test system via a standard RS-232 interface, where the test is executed. The same interface is used to control the tests and for transfer of data and end results.**

Index Terms—**embedded software education, automatic testing, black-box testing**

I. Introduction

Laboratory courses in embedded systems programming are characterized by tight resource constraints and a significant expenditure of time and effort per student. One of the major factors contributing to the time expenditure is the verification of student programs, be they homework or exam solutions. To facilitate this work, we proceeded to develop a test system which allows to verify the functionality of simple embedded software. In a first trial run, we plan to employ the test system in our introductory microcontroller course to automatically verify the correctness of student exam programs. The Microcontroller lab course is held once every year and is attended by 120 to 150 undergraduate students. It teaches the basics of microcontroller programming, such as the usage of timers, a/d converters, UARTs, and so on in assembler and C language on simple 8 bit, 16 bit, or 32 bit microcontrollers, and is thus particularly well suited for a first trial. In the long run, the tool will be used to automatically verify student programs in several different embedded systems courses.

This work is part of the FIT-IT project SCDL "Seamless Campus: Distance Labs"[1], which aims to develop a remote teaching concept for embedded systems laboratory courses. This concept incorporates the required web infrastructure and, depending on the requirements of the course, customized learning packages and remote workplaces. The distant goal of the project is to provide a remote learning environment that can be used for most of the embedded systems courses at the Vienna University of Technology. This will enable us to increase our training capacity and to better support working or handicapped students.

There are other projects similar to the SCDL project, with the goal to extend the education in embedded systems in a way to meet the requirements of the constantly growing embedded systems industry [1]. Such projects are described in [2], [3], [4]. In the past years, many systems have been developed which can test embedded software and hardware in different ways. For instance, testing by fault-injection is performed by the tools described in [5] and [6]. The object-oriented scenario-based test framework shown in [7] places its emphasis upon test re-usability in case of product families or new product versions. In contrast to our solution, which performs black-box functional tests, their approach requires knowledge of the underlying software structure of the tested system.

Our test system employs a special language to describe tests and expected results. There are several other projects that use special languages to test embedded systems. For example, SystemC is a language that defines abstract models of software and hardware components, and [8] explores the possibilities of using SystemC for testing embedded systems.

The system described in [9] is a generic test equipment for embedded software. It is based on the C++ interpreter CINT and adopts the C++ language and its features.

Another design of a generic test and maintenance node for embedded system testing is the subject of [10]. It is built on the IEEE 1149.1 standard test bus and is designed to provide a cheap solution for the design-for-test and built-in-test overhead cost problem.

Our own test system [11] is executed on an ATmega128 microcontroller and is programmed and controlled via a Linux host computer. The basic idea of our system is to supply the tested target system with input signals on its I/O pins and to evaluate its responses in order to compare them with predefined patterns. Hence, we do black-box tests only and are mainly interested in the timing of the response signals.

[1]The "Seamless Campus: Distance Labs" project received support from the Austrian "FIT-IT Embedded systems" initiative, funded by the Austrian Ministry for Traffic, Innovation and Technology (BMVIT) and managed by the Austrian Research Promotion Agency (FFG) under grant 808210. See http://www.ecs.tuwien.ac.at/Projects/SCDL/ for further information.

The tests and expected results are defined in a special meta-language, which in practice allows any type of program behavior to be tested. The test system performs and evaluates the tests automatically and provides reports to a human operator on a local or remote computer. New tests can easily be loaded to the system and executed.

Compared to the other works cited above, our approach corresponds well to the goals of an embedded software test system for the education. In contrast, [7], [9] are well suited for testing high-level functionalities and expect high-level communication of some kind in order to perform tests, and [8] is based on abstract modeling.

In the following paper, we first give an overview of our test system in Section II, then proceed to describe the system implementation in more detail in Section III. Section IV shows the system in action with a small example, Section V discusses the usefulness of the system, and Section VI lists some possible extensions. Section VII concludes the paper.

II. Test System Overview

Resources for undergraduate studies are rather limited – in obligatory courses, one or two faculty members have to handle up to 150 students, supported only by about ten tutors. Yet, embedded systems programming skills are best conveyed by letting students program a lot, so many such student programs have to be evaluated. In the Microcontroller laboratory course that will be used for a trial run, for example, students have to solve about 10-15 programming exercises during the term, may voluntarily submit up to another 15 programs to improve their grade, and additionally have to attend exams. The course offers three practical exams, and students need to do at least one of these exams to pass, but may also come to all three exams to further improve their grade. On the average, about 120 students attend these exams. Each exam consists of two simple programming tasks which the students have to solve in the given time. Students work on the hardware during these exams, so they can try out their solutions and debug their programs. Whenever a student believes he or she is finished, (s)he can submit the solution.

In our experience, much of the time spent per student in undergraduate embedded programming courses is invested into correcting homeworks and exams, so it makes sense to look into means of automating this process. Since our undergraduate courses generally tend to use simple exercises to teach the basic concepts, automatically verifying the correctness of programs appeared to be an achievable goal.

We wanted our test system to be versatile and to be useful to a wide variety of courses, ranging from FPGA programming over different microcontrollers to industrial automation with SPS hardware. So simulators or code instrumentation were no viable options. Instead, we focus on the interface of the control unit to the hardware, in a microcontroller's case its I/O pins, and use black-box tests to verify program functionality and timing. Of course, these tests have their limitations, but they are the most flexible solution as far as the supported target system hardware is concerned.

Our test hardware consists of the *test system* itself, the tested board, which we will call the *target system* in the remainder of the paper, the *host computer*, and the connections between them. The whole structure is shown in Figure 1.

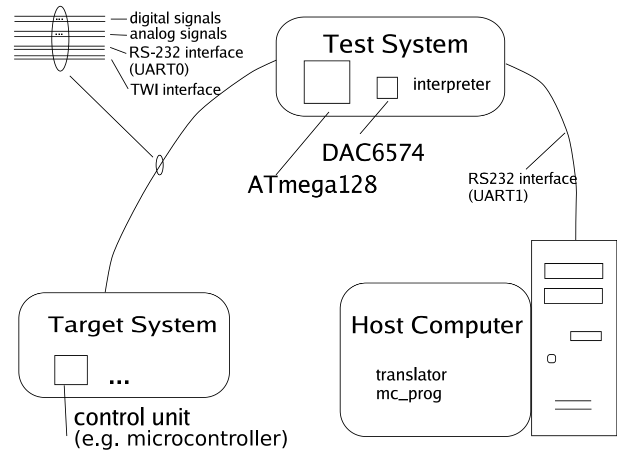

Fig. 1. Test system hardware and connections

Currently, the test system is a universal ATmega128 board, which was developed in the Department of Computer Engineering at Vienna University of Technology. In addition to the microcontroller, a digital to analog converter is used in order to extend the functionality of the tests and to support analog test signals. There are four types of signals that can be connected to the target system board: Digital signals, analog signals, RS-232, and the TWI (I^2C) interface.

The ATmega128 microcontroller executes a C program, the interpreter, whose main function is the interpretation and execution of the actual test program instructions, which are written in a special meta-language. The interpreter is also responsible for the communication with the host computer and the loading of new tests. Note that due to the interpreter, the test program itself is system independent. It can be executed on any hardware that provides the interpreter. So although we are currently using the ATmega128, we could easily change the system by porting the interpreter.

Two software tools are located on the host computer: The meta-language translator, which translates test descriptions into "executable" code, and the `mc_prog` programmer tool, which can download the tests to the test system and manipulate them. The results from the test evaluation as well as all messages are received at the host computer via the standard serial port in text format.

The language developed to describe the test behavior is referred to as "meta-language". It is an assembler-type language whose purpose it is to facilitate the writing of test programs and to make them more understandable. It offers a set of instructions including arithmetical and logical operations, memory manipulations, compare and jump instructions, time measurement, analog and digital signals I/O, TWI and UART communication interfaces, and test control instructions. Currently, the meta-language is the top-level interface to the

test system.

III. Implementation

A. Hardware

As we already mentioned, the test system is implemented on a 5V Atmel ATmega128 microcontroller, which is driven by a 16 MHz clock. An external digital to analog converter, the DAC6574 of Texas Instruments, is connected via a TWI interface. Figure 2 shows a block schematic of the test system hardware and its available connections.

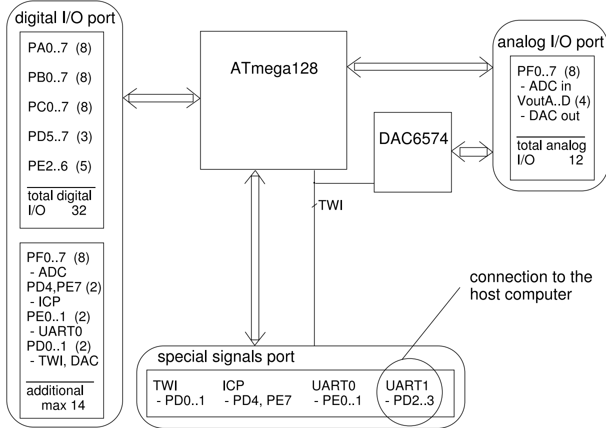

Fig. 2. Test system block schematic

32 digital inputs and outputs are available for the digital signals. Additionally, up to 14 other pins could be used for digital signals if their primary functions are not required. These are the analog inputs, the two UART interfaces, and the TWI interface. However, in order to prevent or at least minimize changes to the connections between the test system and the target system, it is recommended that the special function pins not be used for digital I/O.

In addition to the digital I/O, the ATmega128 has 8 analog input pins with 10 bit resolution. A resolution of 8 bit is also supported by the test system. The external d/a converter is controlled via the TWI (I^2C) serial bus of the ATmega128 and provides 4 analog outputs with 10 bit resolution. Again, an 8 bit resolution is supported as well. More converters could be attached to the bus as necessary, and the test system currently supports two DACs and thus 8 analog outputs. Note that the TWI interface can also be used to test TWI code on the target system, independently of the function of the DAC module.

One of the two UART interfaces of the test system is used for communication with the host computer. Over this interface, the user can control the tests, and data can be sent back to the host. The second UART interface can be used to test UART code running on the target system.

B. Connections

Our test system can dynamically assign the functionality of a pin. So if, for example, a certain pin PB0 of the target system is used to output a PWM signal in one test and utilized as an input from a switch in another, the test system can handle this change in functionality without requiring a change in connections. Hence, it is of no particular importance how the digital I/O pins of the target system are connected to the test system.

There could be one problem when connecting the test system to the target system, and this pertains to the analog pins of the target system, which may either be used as digital I/O or as analog I/O. In the case of digital I/O, they should be connected to digital I/O pins of the test system. If the target requires analog inputs, however, the pins must be connected to the outputs of the external d/a converter. Different tests may entail different functionality of these pins, thus necessitating a change in the connections.

In this version of the test system we do not have any built-in solution for this problem, so currently the user has to reconnect the pins if the change in functionality is required. In a future version, we will solve this problem with an external multiplexer, which can of course in turn be controlled by the test system, see Figure 3. This technique allows to dynamically switch between testing analog input functionality and digital I/O (or analog output). Of course, it requires one more pin to control the multiplexer, so it might be sensible to control several analog inputs/outputs of the test system simultaneously if the extra pins cannot be afforded.

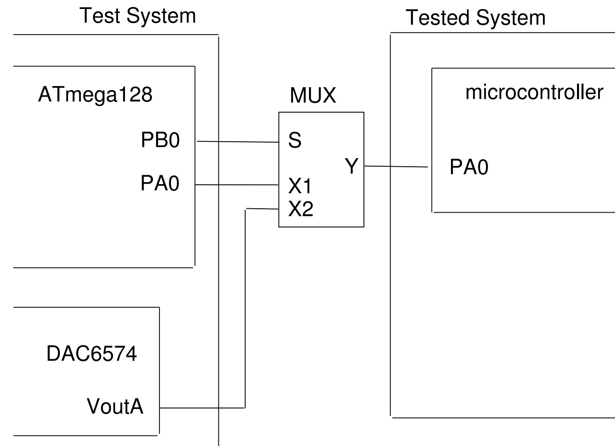

Fig. 3. Connecting more than one test signal

Note that in most cases the problem will not occur anyway, since normally the connection between hardware and the control unit are fixed, so the pin functionalities will remain the same over all exercises and exam tasks that use this hardware. For courses that do not use such statically assigned connections, it may still be feasible to formulate program specifications that keep these problematic pins fixed. So this problem does not pose much of a limitation on the test system in its current form, and a couple of such multiplexers will be sufficient to support the few pins that for whatever reasons must remain flexible.

Another aspect we do not currently address is the problem of different power supplies for target and test system. If the target system happens to work with different voltage levels,

level converters must be employed to bridge the gap between the target and the 5V levels of the test system.

C. Software

The software of the test systems consists of the translator tool and the programmer tool on the host, and the interpreter on the test system itself, see Figure 4.

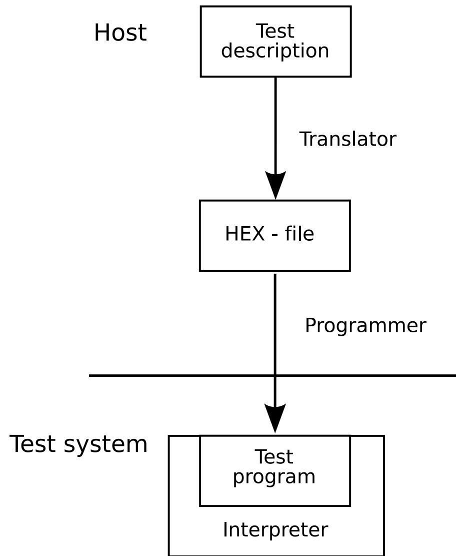

Fig. 4. Software structure of the test system

The test description is written in the meta-language and is translated to a test program that can be executed by the interpreter on the test system. The programmer tool is responsible for uploading the test program to the test system, and for controlling the tests.

The test program itself consists of one or more phases, which can be individually controlled by the programmer. Each phase tests a particular aspect of the target program. Among the things that can be tested are the timing of signals, timing relations between two signals, signal states, or analog outputs. To perform the tests, the test system can generate diverse stimuli ranging from simple digital or analog signals over changing analog values to more complex actions like UART communication.

1) Meta-language and Translator: The meta-language is a full image of the instructions implemented on the test system, so it resembles an assembler language with an emphasis on the functionality required for the tests. Additionally, some higher-level constructs are implemented. Contrary to [9], our meta-language is built entirely on the test functionality and not on an existing programming language. It is also not object-oriented as are [7] and [9].

Each test description can contain up to three sections. The most commonly used section is the `#PROGRAM` part, which contains the sequence of instructions which form a test program. `#VARIABLES` contains pairs of addresses and sequences of values which are to be stored starting at the given address. The `#PHASES` section contains one or more groups of addresses and phase numbers.

Our meta-language also supports labels. Each label points to the address of the next instruction and does not need to be previously declared to be used in the code. Labels can for example be used in instructions which control the program flow, like jump instructions. It does not matter whether the instruction expects relative or absolute addresses. Another application where labels are useful is in the definition and subsequent call of subroutines.

Definitions can be used to associate a variable name with a number in the range of 0 to 255. The value of the definition cannot be changed once assigned, so it only allows the usage of definitions as constants.

As we have already indicated, our meta-language supports the definition and use of subroutines. They can be placed on any address, usually at the end of the test program. Each subroutine must start with a label and end with the return instruction. Parameters can be passed only implicitly.

The translator tool is used to transform programs in the meta-language into a hex-code that is ready to be transferred to the test system. The translator should not be referred to as a compiler, because no special high-level language constructions are translated. One of the basic features that the translator provides is the transformation of complex instruction parameters into a tightly packed form, which is thus not a responsibility of the test developer. The parameters are packed in spaces of less than 1 byte, so that one parameter byte can contain one, two or more values relevant to the instruction.

2) Programmer Tool: The `mc_prog` programmer tool realizes the communication between the host PC and the test system. It can download test programs or variable sets to the system, and can start or stop a test. It accepts a hex-file as input. The hex-file consists of hex-numbers as they are to be uploaded into the memory of the test system, and of the directives (`#PROGRAM`, `#VARIABLES`, `#PHASES`) which tell the programmer what to do with the data following it. The hex-file must begin with a directive.

The upload to the test system is done through the serial port of the host computer.

3) Interpreter: The test system executes an instruction interpreter which interprets the instructions that describe the test behavior. All data required by the interpreter, such as the size of the last uploaded test program and a pointer to the currently executed instruction, are stored in global variables. The test description itself is stored in the microcontroller's 4kB SRAM.

Every test description consists of several instructions. Most instructions are 2 bytes wide (1 byte for the instruction code, 1 byte for the parameters), with the exception of some extended instructions, which are 4 bytes in length and represent functionalities which either cannot be coded in two bytes or which are an extension of the corresponding 2 byte instruction for the purpose of optimization. Extensions perform a functionality

in one instruction which would otherwise take two or more two-byte instructions. This leads to faster program execution, because the instruction decoding time is roughly the same as the instruction execution time.

A set of local variables can be used for the purposes of the test process. They can be initialized during upload of the test program. The memory size for the variables is configurable and is currently set to 512 bytes. Paging is used in order to allow the addressing of all variables. All instructions which use 8 bit addresses refer to the active page, paging itself is controlled by special instructions.

The first 16 variables can also be addressed with a special 4-bit addressing mode. This type of addressing is provided so that the remaining 4 bits in the instruction word are free for other parameters. For example, the instruction `test_bit 3, 13` (in hex: 0x06 0x3D) tests the 3rd bit of the variable at address 13 (the 14th variable). The result is implicitly stored in variable 0. The 16 variables are not affected by paging, so our example would have the same effect no matter which page is currently active. This addressing mode thus provides an easily reachable set of 16 variables and can be compared to the concept of registers in a microprocessor. Variable 0 is used as an accumulator, which means that it is used as an implicit parameter of some instructions or stores the results of an operation.

A special array contains values of user-defined timeouts. These timeouts define the test system reaction in case of instructions which depend on external signals, like for instance time measurement instructions or instructions that wait for a bit change. This prevents hang-ups in case of unexpected or missing signal patterns. We can provide timeouts in the range of 1 μs up to over a minute.

An important issue is the addressing of the digital and analog pins of the test system. To this aim, all of the microcontroller's registers can be accessed by using the instructions `load_acc_reg` and `store_acc_reg`. The names of all I/O registers of the microcontroller are defined and can be used in the test descriptions. Instructions for manipulating separate bits of a register are also available. Thus, all digital I/O signals of the ATmega128 can be manipulated through their ports' registers. The analog I/O pins as well as the special signals like TWI and UART can be manipulated through special instructions, depending on the function. Please refer to [11] for a detailed description.

The test system offers two types of time measurement: Common blocking mode realized through polling, and input capture mode, which works in parallel to the test execution. Time measurement through polling can be performed by every pin of the ATmega128. It blocks the test program execution until the measurement is finished or a timeout occurs, so it is for example not possible to check the timing of two such signals at the same time. Time measurement through input capture can only be performed by some pins of the ATmega128, but has the distinct advantage that the measurement is interrupt driven. Hence, it is possible to check the timing of two signals at the same time. In both cases, the time is measured in milliseconds, microseconds or in processor clock cycles.

The test system can also perform wait loops. Wait loops are for example useful when a time delay has to be implemented in the test program, or when the test needs an external event (signal change) to continue. Note that the value of the waiting period is stored in a variable and thus does not have to be set in advance, but can be dynamically computed during the test execution.

A variant of the wait loop is implemented by the wait-until-condition instructions. Depending on the parameters, these instructions wait for a positive or negative edge on a particular pin. The test execution is blocked until either the bit change takes place, or until a previously defined timeout occurs.

IV. EXAMPLE

This section shows a practical example in the form of a simple program executed on an ATmega16 target system. The program should wait for the press of a button and then start to generate a PWM signal with a given period and ratio. A timing diagram of the desired behavior of the tested target system is shown in Figure 5.

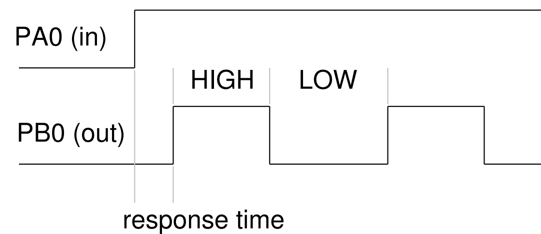

Fig. 5. Expected time behavior of the tested target

The following connections are required between the target and the test system:

```
ATmega128   -   ATmega16
PA0         -   PA0
PB0         -   PB0
```

The positive edge of PA0 is the start condition for the ATmega16. The output is on PB0. The width of the PWM signal is 240μs for LOW and 220μs for HIGH. The response signal on PB0 should be tested with an accuracy of 5μs.

To test the behavior of the target, we must write the test description using the meta-language. The program consists of several parts, which are described below.

The test program begins with the `#PROGRAM` directive, after which the addresses of some variables are declared. For instance, `var2=2` shows that the name `var2` refers to the variable at address 2.

```
#PROGRAM
// definitions
```

```
acc=0  // accumulator
var10=10
var2=2
var6_32=6
var6_32_LW=8

var0_32=6
var0_32_LW=8

pin0=0
low_duration=12
high_duration=4
accuracy=14
```

The first instructions initialize pins PA0 and PB0, which are used later on.

```
begin:
  // label; defines start of the program
// INIT
cbi_reg  ADDR_DDRB  pin0  // PB0 as input
sbi_reg  ADDR_PORTB  pin0  // PB0 -> pull-up
cbi_reg  ADDR_PORTA  pin0  // PA0 -> 0
cbi_reg  ADDR_DDRA  pin0  // PA0 -> output
// initialize timeout
set_timeout  0 ms var10
```

Then the signal that is expected by the tested target program is produced, and the test system checks whether the response has come within the predefined response time. If not, the test program executes the jump instruction and outputs an error message.

```
// START
// apply positive edge on PA0
sbi_reg  ADDR_PORTA  0 // sbi PA0
wait_us_imm  50        // wait for about 50us
 // (response time)

// test for response
load_acc_reg  ADDR_PINB
test_bit_var  0 acc  // test bit 0 of acc
if0_var  acc  // no response
jump_rel_imm  error_no_response
```

In the next section, the duration of the low pulse is measured. The first negative edge starts an internal counter and the measured value is obtained with the next positive edge. The `measure_pulse_width` instruction provides a resolution of the ATmega128 clock cycle. Two possible errors are checked here: Whether a timeout has occurred during the measurement, or whether the measured value does not match the requirements within the bounds set by the desired measurement accuracy. In case of an error a corresponding jump is performed.

```
// MEASURE duration of the LOW pulse
use_timeout  0 // define use of timeout
// pulse width in us, 0=LOW pulse, 1=one pulse, on
// pin0; store into var6_32 (32 bits), of IO port PINB
measure_pulse_width  us 0 1 pin0 var6_32  ADDR_PINB
// CHECK whether timeout has occurred
if_flags  flag_timeout
jump_rel_imm  error_timeout
memory_transfer  2 0 0 var2 var6_32_LW
  // save measured time in var2
operation_var16_var  sub var6_32_LW  low_duration
  // var6_32_LW  -= low_duration
comp_var16  var6_32_LW  accuracy
  // compare difference to the stored accuracy
// CHECK whether the difference is greater than the
// accuracy
if_flags  >
jump_rel_imm  error_low
```

Similar instructions are used to measure the duration of the high pulse. If the test has been successful, no jump has been performed and the next section is executed. It outputs a message to the computer and terminates the test.

```
// TESTING DONE! EVALUATE AND PRINT RESULTS

// PROGRAM TERMINATED WITH SUCCESS
sbi_reg  ADDR_PORTA  0 // turn off PA0
pc_printstr  0x80  // accuracy=+-
pc_printdec16  accuracy
pc_printstr  0x90  // us
pc_printstr  0x60  // new line
stop_test  0 1   // if here - test success
```

Now all that is left is the reaction for all error cases. Each one begins with a label and ends with a termination of the test. User messages are passed as well.

```
// PROGRAM TERMINATED WITH ERROR
error_low:
sbi_reg  ADDR_PORTA  0 // turn off PA0
pc_printstr  0x10  // message  "LOW pulse duration="
pc_printdec16  var2  // print duration value
pc_printstr  0x60  // new line
pc_printstr  0x50  // message must value=
pc_printdec16  low_duration
pc_printstr  0x60  // new line
pc_printstr  0x80  // message accuracy=+-
pc_printdec16  accuracy
pc_printstr  0x90  // message us
pc_printstr  0x60  // new line
stop_test  0 0   // measured time not ok or timeout

error_high:   // similar to error_low

error_timeout:
sbi_reg  ADDR_PORTA  0 // turn off PA0
pc_printstr  0x70  // message timeout
pc_printstr  0x60  // new line
stop_test  0 0   // measured time not ok or timeout

error_no_response:
sbi_reg  ADDR_PORTA  0 // turn off PA0
pc_printstr  0x93  // message no response
pc_printstr  0x60  // new line
stop_test  0 0   // measured time not ok or timeout
// PROGRAM TERMINATED WITH ERROR
```

This concludes the program part of the test description. The next section contains no instructions, but defines initial values for the program variables, as well as the user messages. After each `#VARIABLES` directive, we place first an address and then constants that should be stored at this location. This is done more than once, since each message begins at a different location.

```
// UPLOAD PREDEFINED CONSTANTS
#VARIABLES
0x0000  // start @ address 0

0 0 0 0
0 220 // high_duration  must value
0 0 0 0
0 10 // var10 contains timeout
0 240 // low_duration  must value
0 5  // accuracy

#VARIABLES
0x0010  // start @ address 0x10
'L' 'O' 'W' 32 'p' 'u' 'l' 's' 'e' 32
    'd' 'u' 'r' 'a' 't' 'i' 'o' 'n' '=' 0

#VARIABLES
0x0050  // start @ address 0x50
'm' 'u' 's' 't' 32 'v' 'a' 'l' 'u' 'e' '=' 0

#VARIABLES
0x0060  // start @ address 0x60
13 10 0 // new line

#VARIABLES
```

```
0x0070    //  start   @  address   0x70
't'  'i'  'm'  'e'  'o'  'u'  't'  '!'  0

#VARIABLES
0x0080    //  start   @  address   0x80
'a'  'c'  'c'  'u'  'r'  'a'  'c'  'y'  '='  '+'  '-'  0

#VARIABLES
0x0090    //  start   @  address   0x80
'u'  's'  0
'n'  'o'  32  'r'  'e'  's'  'p'  'o'  'n'  's'  'e'  '!'  0

//  END  OF  TEST  PROGRAM
```

In our test description, both automatic messages and texts defined by the test developer are returned via the serial interface. The following list shows the resulting output on the host PC in some important cases.

- No response within the predefined response time is recognized:
  ```
  STARTING   COMPLETE   TEST
  no  response!
  TEST  FAILED!
  ```
- The test is successful, the time properties of the measured signal meet the predefined expectations, the response time meets the requirements as well:
  ```
  STARTING   COMPLETE   TEST
  accuracy=+-5us
  TEST  OK!
  ```
- The low pulse duration is not correct, considering the predefined accuracy:
  ```
  STARTING   COMPLETE   TEST
  LOW  pulse  duration=249
  must  value=240
  accuracy=+-5us
  TEST  FAILED!
  ```
- A timeout is reached while measuring the pulse width. This can be the case when the pulse is too long or if the signal does not change at all:
  ```
  STARTING   COMPLETE   TEST
  timeout!
  TEST  FAILED!
  ```

This test description occupies 146 bytes of the memory of the test system. Additionally, 16 bytes are used for variables and constants, and 93 bytes contain the user-defined messages.

Of course, the tool is also capable of processing more sophisticated tests. By using conditional instructions, logical operations, and timeouts, various types of state machines could be implemented and therefore the complexity of the test scenario is only limited by the memory size.

V. Discussion

The tool should facilitate program submission for both students and instructors: From the students' point of view, the tool allows them to submit (and thus check) their programs at any time and to get instant feedback. In the case of homework, this makes remote submissions with feedback possible. During exams, students can check the correctness of their programs at any time, without the need to call and wait for a supervisor. If the instructor desires, the tool can also give feedback on the nature of problems in case of failed submissions, although such features should probably be used with care to prevent students from employing a mindless trial-and-error strategy.

From the instructor's point of view, the tool saves a lot of time otherwise spent on manually checking student programs, which generally is a tedious task. It also eliminates human error from the verification process, allows to conduct exams in a distance learning setting, i.e., at a remote location without qualified supervisors on site, and scales to a large number of students.

These features are bought with the additional effort necessary to generate automated tests. Here, exam programs reap the highest benefit from the tool, since they tend to be fairly simple and thus the correctness tests are easily constructed. As a rule of thumb, we estimate that the time it takes to write the test for an exam task is approximately equal to the time it takes to set up the task description and program its solution. So doing automated tests roughly doubles the time required to set up the exam. However, verifying a student solution by hand certainly takes at least ten times as long as doing it automatically, and this factor increases with increasing task complexity. Hence, even a moderate amount of students already makes automated testing economic. Add to that the time-independent advantages of no room for errors, getting rid of a tedious labor, and the advantages for students, and we feel that automated tests are justified even for smaller student numbers like 15-20 students.

Homework exercises pose more problems since they tend to be more complex. Furthermore, homework is generally not only about writing a correct program, but also about writing a good (well-designed) program. Since the test system does only a black-box test, it cannot be used to verify whether the student has employed special features of the target system, like using the timer to automatically generate a PWM signal, and it can of course not be used to evaluate higher-level issues like programming style or efficiency. Still, it does free the instructors from the rather tedious, uninteresting, and error-prone task of checking the correctness of a large number of programs and leaves more time to concentrate on high-level issues like programming style. Furthermore, the test system allows students to remotely submit their homework with instant feedback on the correctness, which will certainly be appreciated. So here it will certainly be worthwhile to investigate methods to generate test programs more easily, as we will elaborate in the next section.

As we have seen in the previous section, the result of a test is either that the test has completed successfully within the given constraints, or that an error has occurred. It is the responsibility of the test programmer to specify what output message the test system should print in case of errors, so detailed information about the nature of the failure is possible. Therefore, the test system can be used both for binary tests (program works/failed) and for more fine-grained evaluation and can thus also be used to assign partial credit (e.g., award $k\%$ of the full points if the program has successfully passed $k\%$ of the test).

VI. Possible Extensions

As one can already see from the example, the test system is well capable of handling timing relations. However, setting up the test description is currently done by implementing a test program in the meta-language. Although not particularly

difficult, it is still time-consuming, so an obvious extension of the system would be to provide a high-level translator that can directly take timing diagrams as input and translate them into an appropriate test description. Such a tool would greatly facilitate the design of test descriptions. Of course, in case of a failed test, the system should still be able to tell the user which part of the test failed. But since we only check the timing of signals, it should be relatively easy to automatically generate test cases for all errors that can occur during a test.

As we have mentioned, the test system is intended for checking both exam programs and homework programs. Exam programs tend to be relatively simple, so the current memory space is more than sufficient for our purposes. If more complex homework is tested, however, it might be necessary to extend the test system's SRAM. To allow completely automatic testing of diverse programs, we should also add some multiplexers for the analog pins to our target hardware.

It might sometimes be interesting not only to test the external signals, but also to verify internal information like register initializations of the target microcontroller. This can for example be useful to check whether a student has programmed the PWM generator of the target microcontroller as demanded in the exercise task, or is simply using a manually tuned busy-wait loop to achieve the same effect. If such additional checks are desired, the course instructor has to provide some additional (target-dependent) code which is linked to the student code and which verifies register initializations. This code could then use some free pins of the target system to communicate its results to our test system, which could in turn incorporate this additional information into its tests. No change in the test system is required to use such additional data provided by the target.

Of course, the meta-language can still be enhanced by adding complementary functionality to some instructions. For example, the duty cycle of a digital signal could be measured in one instruction and thus one would not have to measure the positive and negative pulse durations separately. Or, we could add some special types of comparison instructions which already compare with a certain accuracy. This would be useful when determining whether a measured time or an analog value is in the required interval.

VII. Conclusion and Future Work

The test system described in this paper is intended to automatically check undergraduate student exam programs and homework exercises in the area of embedded systems programming for correctness, that is, in compliance with a given specification. To this aim, we do a black-box test on the I/O pins of the target system, for example a microcontroller. We provide the microcontroller with stimuli, thus simulating the behavior of hardware input elements like switches, and check the output responses of the microcontroller in both time and value domain against the program specification. The output of the system can be used both for simple binary grading (passed/failed), but also for assigning partial credit for partial functionality.

The tool will allow us to increase the cost efficiency of our teaching activities, since faculty staff is relieved from the monotonous task of correcting more than hundred program examples per exam. Additionally, the system enables us to increase the number of students and to devise courses where exams can be held at other locations than our own university. Students benefit from the tool as well, since they can submit their programs anytime and from any location, can expect fast and correct verification of their programs, and can also get feedback in case of errors. Therefore we are planning to try out this system in our next year's microcontroller laboratory course, and in the long run intend to employ the tool in all undergraduate courses on embedded systems.

The system is currently available as a prototype. We are now working on enhancements of the basic functionality, most notably on a good and easy to use interface for developing test descriptions.

References

[1] W. Wolf and J. Madsen, "Embedded systems education for the future," *Proceedings of the IEEE*, vol. 88, no. 1, pp. 23–30, Jan. 2000.

[2] M. W. Mutka and A. Bakic, "Teaching undergraduate computer science and engineering students techniques for the design and analysis of real-time applications," in *28th Annual Frontiers in Education Conference*, vol. 3, Nov. 4–7, 1998, pp. 1079–1084.

[3] B. Haberman and M. Trakhtenbrot, "An undergraduate program in embedded systems engineering," in *18th Conference on Software Engineering Education and Training)*, Apr.18–20, 2005, pp. 103–110.

[4] J. Sztipanovits, G. Biswas, K. Frampton, A. Gokhale, L. Howard, G. Karsai, T. J. Koo, X. Koutsoukos, and D. Schmidt, "Introducing embedded software and systems education and advanced learning technology in engineering curriculum," *ACM Transactions of Embedded Computing Systems, Special Issue on Education*, 2005.

[5] A. Benso, P. L. Civera, M. Rebaudengo, and M. S. Reorda, "A low-cost programmable board for speeding-up fault injection in microprocessor-based systems," in *Annual Reliability and Maintainability Symposium (RAMS'99)*, 1999, pp. 171–177.

[6] A. Sung and B. Choi, "An interaction testing technique between hardware and software in embedded systems," in *9th Asia Pacific Software Engineering Conference (APSEC'02)*, 2002, pp. 457–464.

[7] W.-T. Tsai, Y. Na, R. J. Paul, F. Lu, and A. Saimi, "Adaptive scenario-based object-oriented test frameworks for testing embedded systems," in *26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, 2002, pp. 321–326.

[8] A. Fin, F. Fummi, M. Martignano, and M. Signoretto, "SystemC: A homogenous environment to test embedded systems," in *International Conference on Hardware Software Codesign*, 2001, pp. 17–22.

[9] H. J. Zainzinger, "Testing embedded systems by using a C++ script interpreter," in *11th Asian Test Symposium (ATS'02)*, 2002, pp. 380–385.

[10] J. D. Lofgren, "A generic test and maintenance node for embedded system test," in *IEEE International Test Conference on TEST: The Next 25 Years*, 1994, pp. 143–153.

[11] V. Legourski, "Test system for embedded software," Bachelor Thesis, Vienna University of Technology, Institute of Computer Engineering, 2005.