The Principle of Vaccination in Teaching

Tadej Tuma, Iztok Fajfar, Janez Puhan
University of Ljubljana, Faculty of Electrical Engineering
1000 Ljubljana, Traška 25, Slovenia
E-mail: tadej.tuma@fe.uni-lj.si

Abstract – *This paper describes a very natural teaching mechanism that we have incorporated in our higher education curriculum. Any parent knows that some children are very difficult to persuade not to touch hot objects. The more you warn them the more your offspring is attracted by the burning candle. The thing to do is, let the child touch the flame and it will avoid all hot objects henceforth. The trick is to control the experience in order to keep the damage as small as possible. We have observed the same stubborn attitude of students toward certain subjects. In our experience good programming style is one of those things that need to be learned the hard way. We therefore let the students first experience bad programming and then make them start over, this time employing a proper approach. Our concept has not only been successful regarding the improved programming manners, but has also considerably increased the popularity of the laboratory.*

Introduction

There are things in life that you have to learn by experience. No matter how smart you are, there is just no way around making certain painful mistakes before you can understand. For instance, one has to suffer at least one serious data loss due to unsaved work in order to learn the simple routine of regularly pressing a save button. Preaching theory, discussing, simulating, or warning is all of little use. The best way is to just let it happen and keep the damage as small as possible. After a “patient” has actually lost two hours of typing he/she is cured for the rest of his/her live. This is exactly the way vaccination works: a dose of a virus is deliberately injected: small enough not to cause serious damage but large enough to provide a lifelong resistance.

Well, not every disease can be prevented by vaccination, but bad programming technique is certainly one of them. Throughout many courses we constantly preach about the importance of a systematic (procedural, modular, or object) and orderly approach [1]. The students are asked to document every line of their source code. They have to write detailed reports as their work proceeds. And, of course, they do their homework, they learn and understand all the arguments in favor of good programming techniques. Many decades of educational experience, however, teach us that a majority of graduate students will abandon any planning and documenting as soon as direct supervision of their programming techniques disappears. The academic line of arguing obviously isn't convincing enough [2]. They tend to memorize the theory and obey the instructions in their practical work, without grasping the need for a systematical approach to programming.

In 1993 we have redesigned the laboratory for control software development, in order to let the students experience bad programming. By not insisting on proper techniques we deliberately let the students work disorganized until they start getting desperate. Then we help them with some very effective tips. As simple as the idea may seem, there are quite a few practical problems:

1. The students must not be aware of being first misled and then corrected. On the other hand, it is not fair to push them in the wrong direction.
2. The laboratory assignments must be especially selected to emphasize the difference between good and bad programming techniques.
3. The assignments must also be highly motivating, otherwise the students will not wade through the crisis.
4. Inevitably, there is a considerable loss of time since the average student needs from two to three weeks to produce a sufficiently messy code.

In the following we explain in detail how we have dealt with those problems and present the feedback we have received from the students over the past six years.

The Hardware

In order to concentrate on software development we have designed our target hardware as simple as possible. We use two types of training boards, both based on a M6803 microcontroller with 8Kbytes of external RAM and 8Kbytes of EPROM. The boards differ only in their I/O devices as can be seen in Figures 1 and 2.

Both system types include an onboard download utility as well as a simple onboard debugger residing in the system EPROM and the M6803's internal RAM. This software replaces the usual EPROM and processor emulators, thus simplifying the students’ initial preparations and cutting down laboratory costs.

The development system is thereby reduced to a standard PC running an M6803 cross assembler. Actually the shaded units in Figures 1 and 2 are also part of the development system in spite of the fact that they reside on the target system.

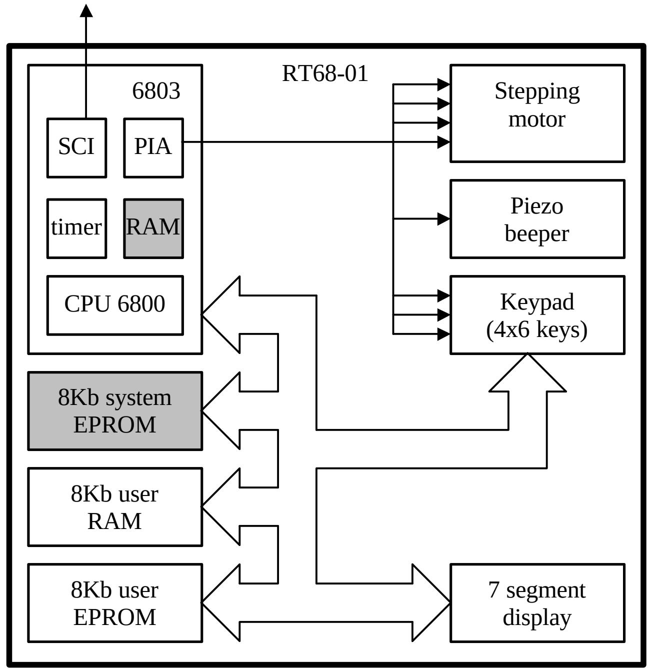


Figure 1. The basic target system

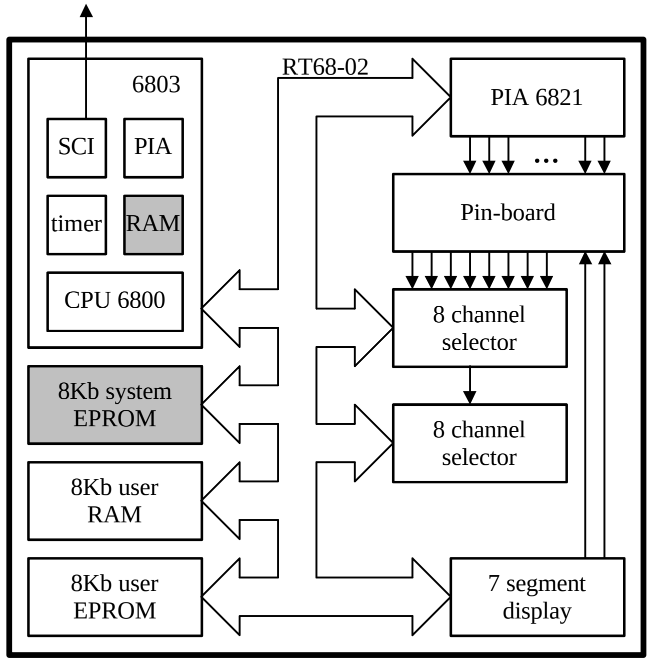


Figure 2. The advanced target system

A typical laboratory session starts with the assembly language coding on a PC [3]. The cross assembler is then used to produce a standard Motorola S-code file, which is simply copied to the PCs serial communications port. On the other end the download utility converts the S-code to machine code and places the latter into the user EPROM, which is in fact a write locked RAM. At this moment a standard terminal emulator is run on the PC to communicate with the onboard debugger. The program is then traced in a single-step mode with only a few basic commands. The onboard single stepping enables the students to monitor everything that is happening inside the registers and memory locations. At the same time all I/O devices can be observed, working in a “slow motion” mode.

The Projects

At the beginning of the laboratory, every student is working on his own PC with an attached target system. The first task is to get familiar with the development system. For this purpose everybody is asked to write a simple keyboard driver for the target system. At this stage the students are still guided by the teaching assistants. Many potential real-time problems like the bouncing of mechanical contacts are being brought to their attention. The students are led rather strictly to a uniform and optimal solution for the keyboard driver.

In the next phase the conception of a small real-time operating system is handed out and discussed. Some core routines like a simple task scheduler are already included in assembler source code while others are just described from the caller's point of view. The keyboard driver, which has just been developed collectively, is of course part of this mini operating system.

At that point the laboratory curriculum changes dramatically. The students are grouped into teams of three to five and each team is assigned a practical project. Although the operating system conception is being recommended it is made clear that the project functionality is all that matters. After an initial briefing the teaching assistants start behaving as consultants - the teams have to make their own decisions.

The relatively short leash in the beginning makes the students impatient. The following sudden freedom of choosing their teammates as well as the project, also suggests a more “relaxed” way of programming, which is exactly what we intend, only misleading, not pushing the wrong way.

Let us now take a closer look at the assigned projects. All projects are complete applications well known to every lay person and not just parts of some sophisticated application [4]. We all know what a remote control, a credit card reader, a railway crossing, a code lock or an elevator do. It took us quite some time to design small toy-like models for each application. These models are connected directly to the digital and analog interfaces of our target systems. By successfully completing their projects, the students can actually read the code from their fathers credit card, they can analyze the pulses of an infra-red car key, see the movable arm of the mini railway crossing go up while lights are flashing and so forth. Although this may seem a little childish, it is most important for the students' motivation! Beside that, the innocent looking toy-like models make the students initially underestimate the control problems.

Once on their own, many teams will start off by thinking: “Who needs this systematic approach stuff to control a few lights?” Sooner or later, however, they discover that controlling an innocent looking toy-robot requires the exactly same approach as does controlling a professional one. On their way to this discovery the teams will not only drown themselves in messy programs but will also encounter communication problems. Sometimes they will even start quarreling about who messed up what. It is very important to reach this frantic state as fast as possible without forcing it upon the students, because this procedure is actually wasting precious time.

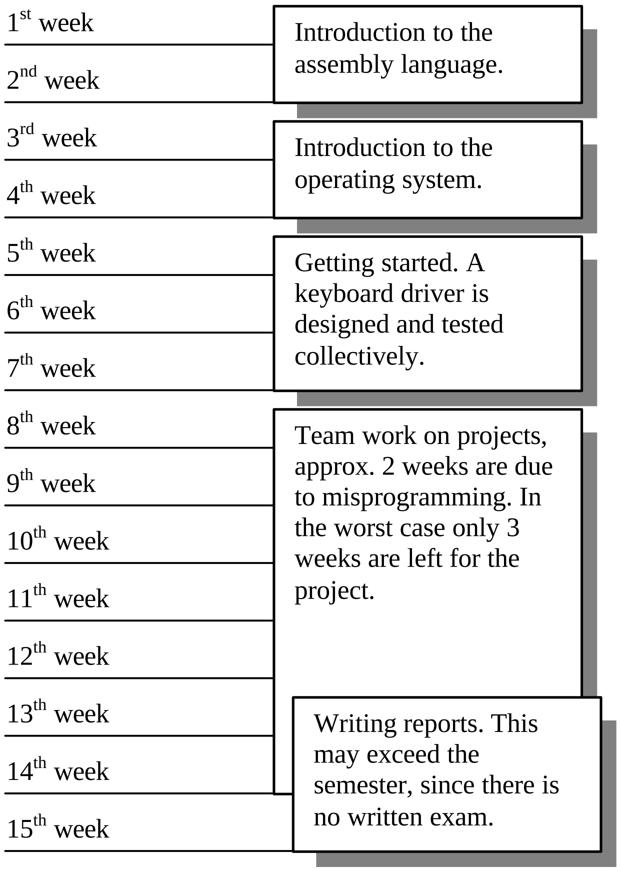
This is the point where an intervention becomes necessary since all our projects are almost impossible to complete without a sound software engineering technique. The teams are encouraged to start over, this time following the code of good programming. The unpleasant experience they have just had now makes them treasure the operating system conception which has been put forward to them in the beginning.

Of course some students are clever enough to use a systematic approach from the beginning, others are just obedient enough. Still others have been warned by senior students, which is just as well. The ones who need the hard lesson most will receive it.

As soon as a project is completed each member of the team is asked to write a detailed report on his work. The students are then assessed individually according to three criteria. The most important criterion is the student's programming proficiency, but also his behavior in the team as well as the quality of his written report are considered. The most outstanding reports are continuously published on our Web site at http://fides.fe.uni-lj.si/tuma/mpeoms.html. In spite of being written in Slovenian language, these reports are very illustrative, showing great enthusiasm. By publishing superior projects we award the best teams. At the same time, the accumulating reports are a very good study material for future student generations.

The time scale in Figure 3 summarizes the laboratory schedule. After the introductory two weeks, it takes the students two weeks to become familiar with the concept of time slicing and another three weeks to build a keyboard drive. The teamwork on individual projects is scheduled for the next seven weeks. The average team loses approximately two weeks by trying to hack itself through the project, though this time is not entirely wasted.

The time needed to complete the project depends on the team. Some teams take only four weeks; others have no yet completed their project by the end of the semester. The average group however needs six weeks and has two weeks to spare. The writing of reports is not bound to the semester and is considered individual homework. Since there is no written examination the report writing may continue after the semester terminates.



Introduction to the assembly language.

Introduction to the operating system.

Getting started. A keyboard driver is designed and tested collectively.

Team work on projects, approx. 2 weeks are due to misprogramming. In the worst case only 3 weeks are left for the project.

Writing reports. This may exceed the semester, since there is no written exam.

Figure 3. The laboratory schedule. One semester=15 weeks

A Typical Lab Session

Let us observe a team of three students (Jack, Susan and Paul) who have to design the controlling software for a pedestrian crossing. The model consists of two traffic light posts, one for the pedestrians with two LEDs and one for the cars with three LEDs. There is also a beeper for blind persons and a button for pedestrians to request crossing. The model is connected to the parallel interface occupying six outputs and one input. The target system RS232 line is used to simulate the communication between the pedestrian crossing and a central computer.

The project is divided among the three students as follows: Jack is responsible for the serial communications with the PC, Paul takes care of the light sequence, while Susan is attending to the pedestrian button and the beeper.

The project would be fairly simple were it not for the audio signal. Managing the light sequence is straightforward since it can be programmed with simple delay loops. While the light sequence is running the pedestrian button might be scanned during the delay loops although this approach is already extremely messy. The communication via RS232 can wait until the sequence has finished. It is impossible however to stretch the “common-sense-approach” beyond this point. The crucial problem is the beeper, which has to be pulse driven simultaneously. The students usually fail to recognize this complication and start naively by designing and testing independent subroutines. Paul might even anticipate problems with the coexistence of his light sequence and Susan's audio signal but he will probably dismiss his doubts until later. He is inclined to think, "If everything else works fine, we'll somehow add Susan's audio driver". Once a team actually tried to solve this particular problem by squeezing the beeper pulses in-between the light sequence.

Whichever approach our three students choose initially will eventually lead them to the only sound solution - the time slicing technique of a scheduler. If they have gone the wrong way, they will have to start over. The proposed approach is not only well structured and systematic, but also extremely simple to understand provided, of course, the three are open-minded enough to look at things in a different manner.

The proposed scheduler has been derived from professional real time operating systems introducing four simplifications.

1. There is only one interrupt allowed in the system, namely the scheduler's.
2. All time slices are of exactly the same size of 1200 machine cycles.
3. Each task must terminate before its time slice expires.
4. The cyclic task schedule is composed of exactly 16 entries.

The four radical simplifications render an extremely simple scheduler [5], which will fit together with its task schedule into the few lines shown in Figure 4.

We have included the assembler code in this paper only to visualize how very little it takes to cross over from the "world of hacking" to a sound programming approach. In order to magnify the contrast between good and bad programming techniques, we have artificially brought the two concepts absurdly close together. In real life it takes much more than just a few simple lines to make a hopelessly messy piece of software work. Besides, there are usually many way in-between the "good" and the "bad" approach. But this is exactly what a vaccination is all about! Inflict an *unrealistically* small infection with an easy recovery and create a lifelong memory of it!

So – our three students only need to include the above 18 lines of assembler code and set up the scheduler data structure according to their needs. Since they have heard all the basic theory of multitasking in earlier courses they certainly are capable of understanding this extremely simple scheduler just by studying the commented source code in Figure 4.

After discussing the proposed scheduler the three students have configured the task schedule in Figure 4 for their particular pedestrian crossing. We can see Paul's light controlling task `LIGHT` running concurrently with Susan's beeper driver `AUDIO`, both with a 1/64s-duty cycle. Susan has even decided to grant the pedestrian button its own concurrent scanning routine `BUTTON`. There are also two high-speed tasks in the system; namely Jack's serial communications task `SCI` and an independent real-time clock task `TIM`. The latter tasks are each occupying four positions in the scheduler data structure, thus running with a 1/256s-duty cycle.

```
;---------------TASK SCHEDULE----------------
SCHTAB FDB    SCI    ;Jack's serial comm.
       FDB    TIM    ;Real time clock
       FDB    SCHRTS ;Void task
       FDB    SCHRTS ;Void task
       FDB    SCI    ;Jack's serial comm.
       FDB    TIM    ;Real time clock
       FDB    LIGHT  ;Paul's light sequence
       FDB    AUDIO  ;Susan's audio sequence
       FDB    SCI    ;Jack's serial comm.
       FDB    TIM    ;Real time clock
       FDB    SCHRTS ;Void task
       FDB    BUTTON ;Susan's button scan
       FDB    SCI    ;Jack's serial comm.
       FDB    TIM    ;Real time clock
       FDB    SCHRTS ;Void task
       FDB    SCHRTS ;Void task
SCHRTS rts

;-----THE SCHEDULER INTERRUPT ROUTINE---------
       ;12 cycles between interrupt and _OCF
_OCF   ldaa   SCHTST ;4 Get test byte
       beq    SCHOK  ;3 Branch if prev. ok
SCHERR bra    SCHERR ; fatal error otherwise
SCHOK  inc    SCHTST ;6 Set test byte
       ldaa   _TCSR  ;3 Clear TOF
       ldd    _OCR   ;4 Load OCR
       addd   #1200  ;4 incr. by time slice
       std    _OCR   ;4 and restore to OCR
       ldx    SCHPTR ;5 Get ptr to current
       ldx    0,X    ;5 Get task entry addr.
       cli           ;2 Allow interrupts
       jsr    0,X    ;6 EXECUTE THE TASK
       ldaa   SCHPTR+1;4 Get high of SCHPTR,
       adda   #2     ;2 increment it,
       anda   #%00011110; 2 overlay 0's
       staa   SCHPTR+1;4 and restore SCHPTR
       clr    SCHTST ;6 Reset test byte
       rti           ;10 Return from OCF
```

Figure 4. The schedule data structure and code

By following this scheme the students were able to split their problems into five independent tasks, all running quasi concurrently. It was actually impossible for them to divide the problem between themselves until they have reached this level of planning, which is why we insist on teamwork.

As soon as the students have grasped the advantages of this simple but effective scheduler, the only tricky obstacle left is the communication between individual tasks. Our three students have to deal with some classic arbitration and synchronization problems.

The Feedback

When we fist started the new laboratory curriculum, we were uncertain about its success, so we set out to monitor its progress very closely. Among other evaluation methods we started to collect some feedback with a short but well prepared anonymous questionnaire. After the course we wanted to know how they felt about four things on a scale from 1 to 5.

1. How much of a burden was our new approach to the students? We were afraid they might feel overworked when things started to go wrong.
2. How much relevance could they see in this kind of work? This is actually testing for the motivation.
3. How much stress did they feel during the course? It was our ambition to create a pleasant working atmosphere.
4. How much of a novelty is this laboratory compared to the experiences they have had so far?

The average answers to the above questions from the five past generations can be seen in Figure 5. Our approach was successful right from the start and we have steadily improved the course, as can be seen from the tendencies. The results from the ’98 generation are a bit different, since the course has been moved from third to second grade.

Generation:	’94	’95	’96	’97	’98
Population:	56	42	14	28	18
“How exhausting?”	3.45	3.40	3.07	3.07	3.61
“How relevant?”	4.16	4.10	4.43	4.68	4.67
“How pleasant?”	3.70	3.79	4.36	4.36	4.28
“How inventive?”	3.68	3.95	4.43	4.57	4.06

Figure 5. The results of the questionnaire on a 1 to 5 scale

Since we continuously publish the questionnaire answers on the Web we have included another very interesting question: “Some younger colleague of yours wants your opinion on the course he is about to take. What would be your advice to him?” So every new generation can get a collection of uncensored “insider tips and tricks” right from the Web. It is also very amusing to read, how the students blame *themselves* for having to start over with their projects! Our misleading is so subtle that no student has ever complained for being deceived.

Conclusions

We have introduced an unusual alternative to laboratory work in 4th year of the electrical engineering curriculum. Of course we realize that most important for any laboratory concept is its pedagogical efficiency, which can be seen from the students feedback.

So far five student generations have passed our new laboratory. The students programming skills have definitely improved in this period. Though - in our opinion - this is not the most important achievement. Amazingly, the laboratory has become extremely popular. In an anonymous questionnaire every third student claims to have learned more about programming than in all previous courses together. This means of course they have gained deeper understanding of previously learned methods.

Another feedback is the number of students who choose microcontroller software as their graduating thesis subject. This number is currently three times larger than before we introduced our “experience-bad-programming” laboratory. Several graduating students are currently designing new and interesting model applications, which will be used as laboratory assignments of future generations.

Each year some students decide to build their own target systems to work with after having passed the examination. Our integrated debugger actually makes an expensive development system superfluous, which is very important for inquisitive students who want to do some amateur controlling at home.

As a matter of fact, the enthusiastic feedback from our students has inspired us to write this article in the first place.

References

[1] T. F. Leibfried, R. B. MacDonald, “Where is Software Engineering in the Technical Spectrum?”, *Int. J. Engng Ed.*, Vol. 8, No. 6. pp. 419-426, 1992.

[2] D. M. Auslander, C. H. Tham, *Real-time Software for Control: Program Examples in C*, Prentice Hall, Englewood Cliffs, NJ 1990.

[3] M. C. Loui, “The Case for Assembly Language Programming”, *IEEE Transactions on Education*, Vol. 31, No. 3, 1988.

[4] P. I. Lin, “Microcomputer Hardware/Software Education in Electrical Engineering Technology: A Practical Approach”, *Proceedings ASEE-91*, pp. 791-794, New Orleans, LA, 1991.

[5] T. Tuma, F. Bratkovi , I. Fajfar, J. Puhan, “A Microcontroller Laboratory for Electrical Engineering”, *Int. J. Engng Ed*, Vol. 14, No. 4, pp. 289-293, 1998.