Int. J. Engng Ed. Vol. 14, No. 4, p. 289–293, 1998
Printed in Great Britain.

0949-149X/91 $3.00+0.00

A Microcontroller Laboratory for Electrical Engineering*

T. TUMA, F. BRATKOVIČ, I. FAJFAR and J. PUHAN
University of Ljubljana, Faculty of Electrical Engineering, 1000 Ljubljana, Tržaška 25, Slovenia
E-mail: tuma@fe.uni-lj.si

In spite of the fact that students of the electrical engineering curriculum receive an adequate education in software development we are observing a rapid degradation in their programming style as soon as supervision of their programming ceases. We have therefore redesigned our laboratory for microcontroller software development, employing some unusual alternatives. We 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 methodology but has also considerably increased the popularity of laboratory.

INTRODUCTION

TODAY most electrical engineering curricula include several courses on the subject of software development [1]. First-year students typically learn basic programming language, later there is usually a course dealing with object-oriented programming, where a systematic approach and good programming 'manners' are taught. Finally, there may even be a course on real-time programming or operating system development.

Throughout the courses the lecturers constantly preach about the importance of a systematic and structural approach. 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, as they do their homework, they learn and understand all the arguments in favour of good programming techniques.

Many decades of educational experience, however, teach us that a majority of graduated 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. For this reason we decided from 1992 to 'teach the students a lesson they will never forget'.

Our basic idea was: 'Wer nicht hoeren will, muss fuehlen'. For that purpose we have redesigned our laboratory for microcontroller software development in the forth year in order to let the students *experience* bad programming. By not insisting on proper techniques [2] we deliberately let the students work unorganized until they start getting desperate. Then we help them with some very effective tips. As simple as the idea may seem, there are some practical problems with this scheme.

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. The laboratory assignments must be specially selected to emphasize the difference between good and bad programming techniques. The assignments must also be highly motivational, otherwise the students will not wade through the crisis. Inevitably, there is a considerable loss of time, since the average student needs from two to three weeks to produce a sufficiently messy program.

In the following we will explain in detail how we have overcome these difficulties.

THE DEVELOPMENT AND TARGET SYSTEM

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 Fig. 1.

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 emulator, 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 dashed units in Fig. 1 are also part of the development system in spite of the fact that they reside on the 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 PC's serial communications port. On the other

* Accepted 15 February 1998.

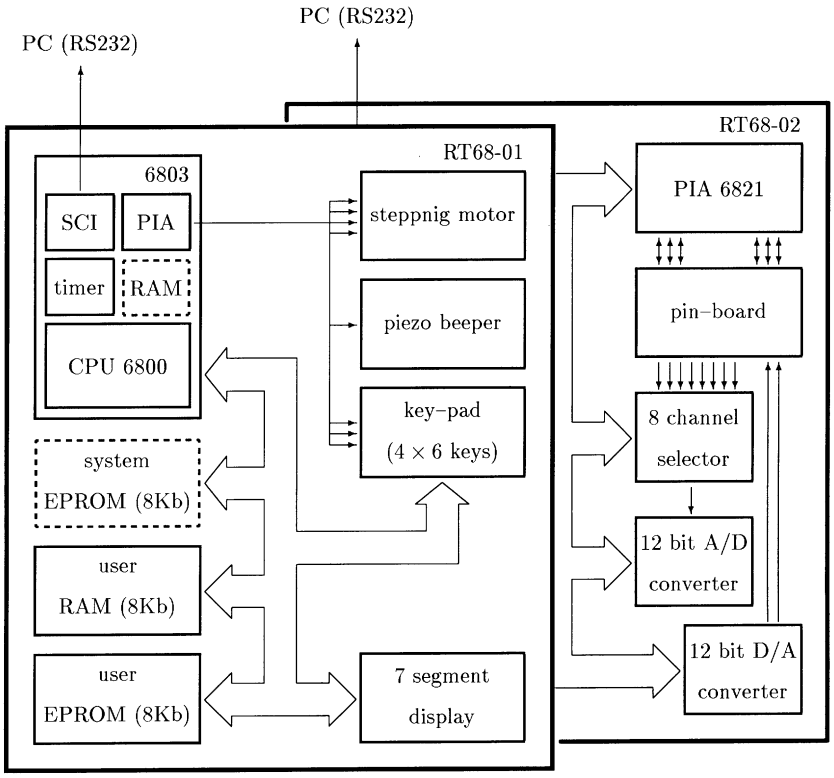

Fig. 1. Two types of target systems.

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 single-step mode with only four basic commands: `T[num]`—trace num instructions, `P[num]`—proceed 'num' instructions; `R[num]`—run 'num' instructions; `D[addr]`—display 256 memory locations starting at 'addr'. 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 LABORATORY SCHEDULE

At the beginning of the laboratory, every student is working on his own PC with an attached target system. The fist 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.

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 father's credit card, they can analyse 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, which is exactly what we want.

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 a toy robot requires exactly the 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 quarrelling about who messed up what.

This is the point where 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 graded 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 time-scale in Fig. 2 summarizes the laboratory schedule. After the introductory two weeks, it takes the students three weeks to become familiar with the development system and another two weeks to grasp the concept of time slicing. The team-work 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 not completed their project by the end of the semester. The average group however needs six weeks and has two week to spare. The writing of reports is not bound to the semester and is considered individual homework.

AN EXAMPLE

Let us observe a team of three students 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 RS-232 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 Susanne 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 there is really no need to scan the pedestrian button neither is there any communication via RS-232 necessary. So far everything is sequential. The only 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 Suzanne'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 Suzanne'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 students choose initially will eventually lead them to the only sound solution—the time-slicing technique of a scheduler. This 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 foreground task allowed, namely the scheduler.

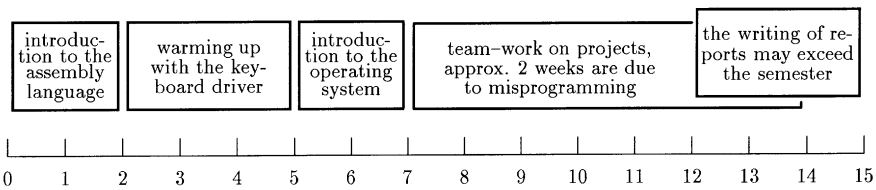

Fig. 2. The laboratory schedule in weeks. One semester = fifteen weeks.

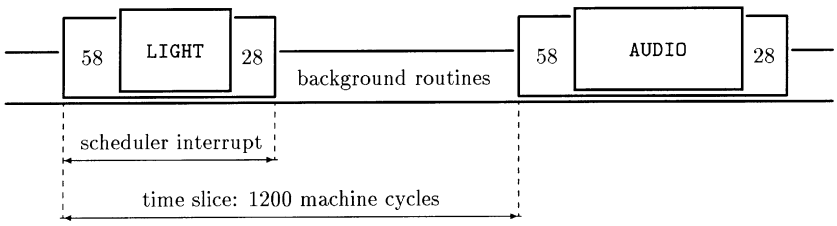

Fig. 3. The principle of time slicing. All numbers are in machine cycles.

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, which will fit together with its task schedule into a few lines shown in Fig. 4.

The students only need to include these 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 Fig. 4.

After discussing the proposed scheduler the students have configured the task schedule in Fig. 4 for their particular pedestrian crossing. We can see Paul's light controlling task `LIGHT` running concurrently with Suzanne's beeper driver `AUDIO`, both with a 1/64s duty cycle. Suzanne 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 1/256 s duty cycle.

By following this scheme our three 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.

CONCLUSIONS

We have introduced an unusual alternative to laboratory work in the forth year of the electrical engineering curriculum. Of course we realize that most important for any laboratory concept is its

```
;------------------------ TASK SCHEDULE ----------------------------------------
SCHTAB  FDB   SCI          ;Jack's serial communications task
        FDB   TIM          ;Real time clock.
        FDB   SCHRTS       ;Void task.
        FDB   SCHRTS       ;Void task.
        FDB   SCI          ;Jack's serial communications task
        FDB   TIM          ;Real time clock.
        FDB   LIGHT        ;Paul's light sequence task
        FDB   AUDIO        ;Suzanne's audio sequence task
        FDB   SCI          ;Jack's serial communications task
        FDB   TIM          ;Real time clock.
        FDB   SCHRTS       ;Void task.
        FDB   BUTTON       ;Suzanne's button scanning task
        FDB   SCI          ;Jack's serial communications task
        FDB   TIM          ;Real time clock.
        FDB   SCHRTS       ;Void task.
        FDB   SCHRTS       ;Void task.
SCHRTS  rts
;------------------------ THE SCHEDULER INTERRUPT ROUTINE (~=86) ---------------
                           ;12 cycles between interrupt and _OCF (worst case)!
_OCF    ldaa  SCHTST       ; 4 Get test byte.
        beq   SCHOK        ; 3 Branch if previous interrupt completed,
SCHERR  bra   SCHERR       ;   fatal error otherwise -- can't continue!
SCHOK   inc   SCHTST       ; 6 Set test byte to indicate running interrupt.
        ldaa  _TCSR        ; 3 Clear TOF by reading TCSR.
        ldd   _OCR         ; 4 Load output compare register,
        addd  #1200        ; 4    increment it by time slice
        std   _OCR         ; 4    and restore it to OCR.
        ldx   SCHPTR       ; 5 Get pointer to current 1/64s period task.
        ldx   0,X          ; 5 Get the task's entry address.
        cli                ; 2 Allow interrupts.
        jsr   0,X          ; 6 EXECUTE THE TASK.
        ldaa  SCHPTR+1     ; 4 Get high byte of SCHPTR,
        adda  #2           ; 2    increment it,
        anda  #%00011110   ; 2    overlay 0's
        staa  SCHPTR+1     ; 4    and restore it to SCHPTR.
        clr   SCHTST       ; 6 Reset test byte to indicate end of interrupt.
        rti                ;10 Return from OCF interrupt
```

Fig. 4. The task schedule data structure and the scheduler code.

pedagogical efficiency, which can be seen from the students' feedback.

So far five student generations have passed our new laboratory course. 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.*, **8,** 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 Trans. Education*, **31,** 3 (1988).
4. P. I. Lin, Microcomputer hardware/software education in electrical engineering technology: a practical approach, *Proc. ASEE-91*, pp. 791–794, New Orleans, LA (1991).

Dr. Tadej Tuma is an Assistant Professor at the University of Ljubljana, Faculty of Electrical Engineering. He received his Dipl. Ing. degree in 1988 and the Doctor's degree in 1995. In the past six years Tadej Tuma has developed several laboratory courses.

Dr. Franc Bratkovic is a Professor at the University of Ljubljana, Faculty of Electrical Engineering. He graduated in electrical engineering at the University of Ljubljana in 1960. In the year 1972 he received the Doctor's degree. Dr. Bratkovic has been teaching as an Assistant Professor from the beginning of his academic career in 1961 and was elected Professor in 1974. In this period he has published five text books. Currently he is Dean of the Faculty of Electrical Engineering in Ljubljana.

Dr Iztok Fajfar is an Assistant Professor at the University of Ljubljana, Faculty of Electrical Engineering. He received his Dipl. Ing. degree in 1991 and the Doctor's degree in 1997. His teaching interests are mainly in the field of motivation. He has been studying the role of practical teaching as a supportive element in understanding theory.

Janez Puhan is a Teaching Assistant at the Faculty of Electrical Engineering. He is very experienced in practical teaching of laboratory courses.