Teaching assembly and C language concurrently

Janet Puhar, Árpád Bármbyn, Tadej Tuma and Iztok Fajfar
Faculty of Electrical Engineering, University of Ljubljana, Ljubljana, Slovenia
E-mail: iztok.fajfar@fe.uni-lj.si

Abstract The paper discusses whether (and how) to teach assembly coding as opposed to (or in conjunction with) higher programming languages as part of a modern electrical engineering curriculum. We describe the example of a very simple cooperative embedded real-time operating system, first programmed in C and then in assembler. A few lines of C language code are compared with the slightly longer assembly code equivalent, and the advantages and drawbacks are discussed. The example affords students a much deeper understanding of computer architecture and operating systems. The course is linked to other courses in the curriculum, which all use the same hardware and software platform; this lowers prices, reduces overheads and encourages students to reuse parts of a written code in subsequent courses. A student learns that badly written and poorly documented code is very difficult to reuse.

Keywords programming; embedded systems; real-time systems; microcontroller; target system; code reuse

In the three decades since the advent of the first microprocessor, there has been immense development in this exciting field of engineering. Nowadays, microcontrollers offer tremendous computing power at extremely low prices and negligible power consumption. It is no wonder more and more products are billed as embedded systems based on such microcontrollers. The architecture and consequently the machine code and assembly language used for advanced microcontrollers are rather complicated and therefore are rarely directly manipulated. Almost all the software is written in higher programming languages such as C or Bascom. Only microcontroller designers and compiler developers still need to master the machine code and assembly language, while embedded system software designers write their code in higher languages and therefore do not need to have any assembly language programming skills. Or so it seems.

A closer look at the industrial reality reveals that a major part of coding is indeed done in higher languages, but in spite of this one cannot avoid assembly programming altogether. There are number of cases where understanding what exactly is going on at the assembly level is crucial. Examples include detecting and avoiding undetermined behaviour due to compiler code optimisations, writing a special extremely fast subroutine for inclusion within a higher-level language as an assembly piece of code, debugging complicated errors such as stack overflow, and real-time embedded trace monitoring in a target environment, to name just a few. After all, a disassembly tool is always included in any professional development environment to handle such difficulties. Thus, assembly language programming skills are not obsolete; rather, they are very useful in the everyday work of a software engineer dealing with embedded systems.

Teaching assembly against higher programming language dilemma

For these reasons the decision was made at the Faculty of Electrical Engineering in the University of Ljubljana that the teaching of an assembly language should not be completely abandoned and totally replaced by higher languages in the electrical engineering curriculum. The questions then arose regarding the extent of such teaching, and how best to interweave the teaching of assembly code and higher-level language. There were several goals which we considered important. A student should master programming skills in both languages. Further, both the higher-level and the assembly languages should be modern and ubiquitous, not some exotic version. That is, widely used industrial microcontroller/compilers should be the subject of consideration, although this then involves a complicated assembly language. At the beginning of the course, complex assembly details, such as the confusing multitude of different kinds of addressing modes, had to be skillfully avoided to keep the students' attention and not to scare them. Later on students could develop a feeling for when to use a higher language, and when and why they should resort to assembly programming. As an answer to all the posed objectives, we developed the approach of teaching an assembly and higher languages in parallel.
As the students were, thus, already familiar with C syntax and basic programming logic when they took the course on embedded system software design, it would have been highly demotivating to stick exclusively to an assembly language at this point. Students could perceive this as a step backward and therefore uninteresting, especially in light of the complexity of the ARM-based 32-bit RISC processor assembly language, which is by definition not an exciting matter. In spite of that, some of the assembly language should somehow remain present. Our first attempts to complement C code with an explanation of the corresponding assembly code were fairly unsuccessful. A sole link between the two was a compiler, which generated the machine code. It can be disassembled, but the code obtained is still far too complicated for teaching assembly language syntax. A simple passing of arguments to a C function generates half a screen full of hard-to-read assembly code, which is quite a painful experience for a beginner. We finally gave up with one-to-one C-to-assembly transformations when standard C functions like printf() and floating point arithmetic without a math coprocessor came into the game. Compiler-generated assembly code was simply too long and too complex.

**Simplified cooperative real-time operating system**

As an alternative approach, we decided to teach both languages through pairs of programming cases written from scratch in the two languages simultaneously. For motivation purposes, the examples chosen were not to be trivial ones. As a solution, the idea of programming a simplified, yet operational, cooperative real-time operating system (OS) was born.

As a real-time OS is a special case of a multitasking system, we decided to use the concept of time slicing, as this is common to all multitasking systems. This basic principle of quasi-parallel running of two or more program threads is fairly simple and can be easily explained. At this point we introduce the first simplification to our OS: all time slices are of the same length, i.e., the time is chopped into equidistant intervals, \( \Delta t_{slice} \). For this we used a timer interrupt. At the beginning of a time slice the OS core is called and reduced to a simple scheduler, which simply calls the next task waiting to be executed. This is the second simplification we introduced to our OS: namely, there are no priorities. Tasks are called one after another in a simple round-robin manner. If there are \( n_{task} \) tasks, then each is executed once, over a total period of \( \Delta t = n_{task} \times \Delta t_{slice} \). The calling mechanism of our OS is depicted in Fig. 1.

![Time partition among tasks and main program.](image)

The OS core algorithm (see Fig. 2) or scheduler, \( sch\_int \), is in fact a timer interrupt service routine. It requires cooperation over tasks and is simple enough to serve as a code example. It is evident from Fig. 1 that every task has to terminate before the end of the time slice, which is the third simplification imposed upon our OS. The task cannot be interrupted and resumed later. If a particular task is too long, the system hangs in an infinite loop. Such system is called a non-pre-emptive OS.

In order to maintain absolute control over the timings, no interrupts other than the timer interrupt used to invoke the scheduler were allowed. This was the fourth and final simplification – interrupts disallowed – which meant that we had to use polling. This simplification represented the most serious drawback of our simplified cooperative real-time OS.

**Programming equivalent code in two ways**

As our students were already familiar with the syntax of the C programming language, they were first required to code the operating system core (i.e. the scheduler – see Fig. 2) in C language. Coding such a simple algorithm might be expected to
be trivial, but it turns out that it is not quite so. Several questions arise during the
coding, including how to initiate/subscribe to the timer interrupt, and how to enable/disable
nested interrupts. It becomes obvious that when programming an embedded system,
some knowledge of the underlying hardware architecture is essential. Apart from
that, students notice that the usual standard C syntax becomes more complicated
when they are trying to define for example the address of a register or pointer to the
function containing the code of the next task. Students thereby get to know so-called
embedded C language.
The C code of the scheduler sch_int for the Philips ARM-based microcontroller
LPC2138\textsuperscript{33} is given below. At this point, the selection of a concrete microcontroller is
necessary. We discuss the hardware and software platform in more detail in the
next section.

```c
#define mr0_interrupt 0x00000001
// System status
#define task_completed 0
#define task_running 1
// Registers
#define T0IR (*((volatile unsigned long *)
0xe0004000))
#define VICVectAddr (*((volatile unsigned long *)
0xffffffff030))

typedef void (* voidfuncptr)();
// System variables
int sch_tst, sch_idx;
// Scheduler
extern void task1();
extern void task2();
extern void task3();
voidfuncptr sch_tab[] = {task1, task2, task3};
// Enable IRQ interrupts
void enable_irq() {
    asm("stmfd sp!, {r0}");
    asm("mrs r0, cpsr");
    asm("bic r0, r0, #0x80");
    asm("msr cpsr_c, r0");
    asm("ldmfd sp!, {r0}");
}
// Disable IRQ interrupts
void disable_irq() {
    asm("stmfd sp!, {r0}");
    asm("mrs r0, cpsr");
    asm("orr r1, r1, #0x80");
    asm("msr cpsr_c, r0");
}

asm("ldmfd sp!, {r0}");
}
// Real time operating system core
void sch_int() {
    if(sch_tst == task_running) while(1);
    sch_tst = task_running;
    T0IR = mr0_interrupt;
    VICVectAddr = 0;
    enable_irq();
    (*((sch_tab[sch_idx]))());
    sch_idx = sch_idx + 1;
    if(sch_idx == sizeof(sch_tab) / sizeof(voidfuncptr))
    sch_idx = 0;
    disable_irq();
    sch_tst = task_completed;
}
```

With this simple example, we can also show that some things cannot be done with C. For instance, the principle of enabling and disabling interrupt nesting is easy: one
just has to manipulate the interrupt flag in the program status register. However,
while a few lines of assembly code do the trick, this cannot be done at the C level.
With this easy-to-understand real case, students instinctively start to appreciate
assembly language.

The next step for students is to try to rewrite the scheduler sch_int wholly in
assembly language. For that purpose we explain to students only a tiny subset of the
available instructions, accompanied with necessary assembly syntax. We also reveal
some of the theoretical aspects of the central processing unit and the registers. The
student-written assembly code should look something like that shown below. Each
student is required to write their own version. After the task is completed we encour-
ge a discussion of the different versions of code (all of course handling the same
problem). The debate eventually leads into a comparison of the original C code with
the assembly versions. The parts of the code that correspond to particular C lines
can be identified and isolated from the assembly program. This helps students to understand what a compiler actually does.

```c
/* Constants */
.equ i, 0x80
.equ t0mr0_int, 0x01
.equ word_len, 0x04
.equ rtos_active, 0x01
.equ rtos_inactive, 0x00
/* Registers */
.equ t0ir, 0xe0004000
.equ vicvectaddr, 0xffffffff030
/* Global symbols */
.global task1
```
Teaching assembly and C

mov r1, #rtos_inactive
str r1, [r0]
ldmfd sp!, (r0-r5, lr)
mov pc, lr

Throughout the course, students are required to write accompanying subroutines that perform well defined and isolated tasks, such as initiating the operating system. Each assignment exposes a theoretical problem and proposes a simple solution. For instance, students deal with problems like simultaneous access to shared resources, task length measurement, delays, communication between tasks, accessing memory buffers, and so on. Thus, the learning of assembly language is wrapped into the design of a simple cooperative multitasking real-time OS, which is quite an exciting area. Second-year undergraduate students customarily see an OS as a huge and complicated piece of software needed to run computer programs. The idea of writing their own OS seems unthinkable. So a student’s interest grows with the realisation that the very elementary principles of an operating system are not such a big deal. Some basic knowledge of OSs is acquired in addition.

In the teaching of assembly code some comments can readily be made on the embedded system hardware design and the microcontroller’s architecture. Essential points like memory mapping, stack positioning and operating modes can be effectively explained. Because the students are dealing with the actual code, the theory becomes more readily absorbed. We are able painlessly to highlight notions like linker script and microcontroller initialisation, which are hidden underneath the C programming level, usually buried in some assembly run-time file.

Since the notion of an OS is well known to the average student, the requirements which arise in writing the code are intuitive. The subject can always be presented in such a way that a problem and a demand for a solution are created first. We found this particular pedagogical approach to be very effective, since solving a well defined task with a clear and practical goal is more fun. The generally unpleasant assembly syntax, with its multitudes of instructions and addressing modes, becomes a bit more bearable.

Hardware and software platform

When we designed the course, we wanted to use only a general, hardware-independent system of embedded software. However, as it is necessary to present students with specific cases, some particular microcontroller platform has to serve as a role model. As stated above, we selected a Philips LPC2138 microcontroller, on the basis of few requirements which had to be met.

At our faculty, several courses included in the bachelor degree in electrical engineering deal with embedded systems. The course we discuss in this paper is only students’ first contact with the subject. It is best to use a single microcontroller system at all levels, even at the introductory C programming level (which precedes all the other courses on embedded systems), as then students can focus on the essentials of a particular course rather than losing time and getting confused learning the
There are intricate details of various microcontrollers. Further, this approach, apart from reducing overheads, can demonstrate to students the extremely important concepts of code reuse of writing code in an orderly manner, with the inclusion of many comments. Assembly code in particular is practically unreadable without being properly documented. An average student has to admit that even within a few weeks it is easier to reuse someone else's neatly aligned and documented code than their own bungled code with no comments whatsoever. These points are otherwise practically impossible to demonstrate practically or even to explain theoretically to an average student within a typical half-semester course.

Thus, a requirement of the microcontroller selected was that it be able to serve a student over many semesters of work at different levels. One on the hand it had to be plain enough to accommodate relatively simple assembly and C programming language sessions, while on the other hand it had to be general and powerful enough to fulfil the different (semi-professional) requirements of higher-level courses. In many cases the advanced courses require additional specific devices. Therefore, a sufficient number of general-purpose I/O pins was also a necessity.

The microcontroller should also be a modern one, widely used in industrial applications. Beside the microcontroller board, a professional development environment was also needed. And above all, every student had to be able to purchase their own system at an affordable price, which should not be higher than the cost of an average textbook. The development kit included both hardware (the board) and software (the development environment). With their own embedded system, the students could work at home right from the beginning. They could try out the cases discussed in the classroom or laboratory, or even use some imagination and work on their own projects.

Looking for such an an all-round workhorse, we decided to take our chances with the ARM7 core by Philips implemented in the LPC2138 microcontroller. We speculated that this technology would be around for at least a decade. In order to keep cost as low as possible and still meet professional standards, we had to find some sponsorship. After much negotiation, a consortium of eight companies was ready to develop and finance our new ARM7 microcontroller board.

According to the requirements discussed above, we designed the basic development module as depicted in Fig. 3. The highlight of the system is the integrated on-board debugging hardware that links the ARM7 CPU to the well known professional development environment winIDEA™ by iSystem,26 which runs on any standard personal computer. The PC is connected to a target board via USB cable and is provides the necessary power supply as well.

The students have a fully functioning development system. The proprietary software on the PC is locked to the on-board debugging hardware, which serves as a dongle, in order to prevent unauthorised professional use of the system. This is an original concept that both protects the copyright of winIDEA™ and gives students full development power at the same time.

The microcontroller board has powerful debugging capabilities but very few input/output devices. That is because we needed to keep the initial costs as low as possible. There are four keys, four small LEDs, a potentiometer connected to an A/D input, a general-purpose operational amplifier at a D/A output, a pair of RS232 serial ports and the facilities to mount a standard LCD piggyback. All these devices can be disconnected by simply taking off the corresponding jumpers. Such a configuration turned out to be more than enough for a beginners course on embedded systems.

To accommodate the needs of more advanced courses, we have provided connections to all CPU general-purpose I/O pins. Any number of sophisticated specific add-on boards can be attached to these pins. Individual teachers are designing add-on boards for their specific needs in smaller quantities. Senior students are encouraged to experiment with add-ons in their project work. Many master theses are based on developing and testing add-ons. Optionally, an external embedded trace monitor can be connected to a special port, enabling students to trace their programs in real time. This, however, requires relatively expensive additional hardware.

**Observations and conclusions**

We have developed a basic course on the design of embedded system software, based on a simplified cooperative real-time OS. Students are required to write a piece of code in assembly and C programming languages concurrently, which gives them a hands-on insight into microcontroller system architecture and function, as well as solid programming skills. This basic course is a springboard to all higher courses.
on embedded systems, in some of which assembly is at the forefront, while in others
C is the main programming language. The course helps students to understand both
approaches better. They can then choose between approaches and are therefore able
to find the most appropriate solution to a specific problem.

We have been running the new course for four years now. The early feedback has
been very encouraging. Our teaching approach was well accepted by students, and
the microcontroller board and accompanying development environment has become
popular. This is evident both in the steep increase in the number of students volun-
tarily undertaking individual project work in the number of systems sold. We have
also received some favourable responses from industry, as many students worked as
summer interns. Their theoretical knowledge is better supported by practical pro-
gramming skills on a modern 32-bit microcontroller. The only drawback we have
noticed so far is the fact that our course – and consequently, as explained above, the
entire part of the curriculum on embedded systems – is strategically dependent on
a single microprocessor architecture and a single development system.

Apart from purely empirical conclusions, we have also obtained some numerical
results showing the advantages of our approach. At the end of each year we ask our
students to answer a short questionnaire. All responses are on a continuous scale
from 1 to 5. The questions related to the relevance of the particular subject, how
interesting the subject was, what the overall workload for the students was, how
much they learnt, and their general feeling about the subject. The questions are
strongly correlated, as this allows the relevance of the answers to be tested. It is well
known that if students find the subject relevant they are automatically more moti-
vated and interested, they feel better, learn more and feel less workload. For the
most part, that is exactly the pattern we found in the responses to the questionnaire.
On average, the answers from the last four years – i.e. since we have used the new
approach – differ more than half a point from those obtained before the new
approach was introduced. We are not surprised that, due to a higher level of motiva-
tion, students feel less workload, although, in reality, the actual workload as esti-
1ated by the teachers has not changed: the number of contact hours has not changed
and there was not much change in the teaching goal or the examination questions.
What actually came as a bit of surprise to us was the fact that the students indicated
they had learned more, whereas the average grade they received did not change.
This observation will need further attention from the teaching staff. Either there are
additional things the students learned that were not sufficiently tested in the exam
or the teachers quickly adapted to the new level and unconsciously adopted the same
(Gaussian) distribution of grades as before. It is also possible that the students only
felt they had learned more, although we feel this is a less likely possibility.

Acknowledgement
The authors would like to thank the Ministry of Education, Science and Sport
(Ministrstvo za šolstvo, znanost in šport) of the Republic of Slovenia for co-funding
our research work through programme P2-0246, Algorithms and Optimisation
Methods in Telecommunications.

References
2 MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set. Doc. Num. MD00086,
Rev. 2.50 (MIPS Technologies Inc., 2005).
3 Coldfire Family Programmer’s Reference Manual. Doc. Num. CFPRM, Rev. 3 (Freescale Semi-
conductor Inc., 2005).
International Conference on Computational Science – ICCS 2004, Lecture Notes in Computer
5 I. Aedo, P. Dias, C. Fernandez, G. M. Martin and A. Berlanga, ‘Assessing the utility of an interac-

tive electronic book for learning the PASCAL programming language’, IEEE Transactions on
Education, 43 (2000), 403-413.
6 D. Davenport, ‘Experience using a project-based approach in an introductory programming course’,
7 J. S. Song, S. H. Hahn, K. Y. Tak and J. H. Kim, ‘An intelligent tutoring system for introductory C
8 C. Departine and G. Gay, ‘Active participation of integrated development environments in
9 S. Xingogalo and M. Satratzemi, ‘An integrated programming environment for teaching the object-
oriented programming paradigm’, in First European Conference on Information and Communication
10 T. Tuma, F. Bratkovic, I. Fajfar and J. Puhan, ‘A microcontroller laboratory for electrical engineer-
11 D. L. Maskell and P. J. Grabau, ‘A multidisciplinary cooperative problem-based learning approach
12 M. Hedley, ‘An undergraduate microcontroller systems laboratory’, IEEE Transactions on Education,
14 B. Martin-del-Brio and C. B. Al-Ruiz, ‘A software tool for teaching microcontroller system
15 T. Tuma, I. Fajfar, J. Puhan and Á. Bürlmen, ‘A non-virtual distance education course in software
engineering’, International Conference on Information Technology Based Higher Education
16 A. Nissokoff, T. Tuma, I. Fajfar, J. Puhan and Á. Bürlmen, ‘A PSD based HC11 microcontroller
development board: building on past experience’, Proceedings of World Multiconference on
17 A. H. G. Al-Dhahir, ‘Integrating hardware and software for the development of microcontroller-
18 J. W. Jeon, ‘A microprocessor course designing and implementing personal microcomputers’, IEEE
20 C. Mu, S. Liu and J. Chen, ‘Hardware/software integrated training on embedded systems’, Interna-
21 S. Nooshabadi and J. Garside, ‘Modernization of teaching in embedded systems design – an inter-