SOFTWARE ENGINEERING FOR ELECTRONIC ENGINEERS

A. F. T. WINFIELD
Department of Electronic Engineering, University of Hull, England

The introduction of software engineering into an undergraduate electronic engineering curriculum poses a number of special questions for an engineering lecturer faced with its implementation. In particular,
(i) what exactly is software engineering, when in an electronic engineering context, (since the computer science approach may well prove inappropriate here),
(ii) should the course have a predominantly hardware, or software bias,
(iii) which (computer) hardware, and language(s), should be chosen as the principal vehicle for illustrations and problems, and, related to this,
(iv) how much emphasis should be placed on high-level languages, since these are now feasible and indeed desirable for developing software engineering applications, and finally,
(v) can a unified software design methodology be presented which would prove useful for any target hardware, or language?

This paper aims to address each of these problems, with particular reference to a third year undergraduate software engineering course, run, by the author, during the past two years. Firstly the scope and aims of a software engineering course, for electronic engineers, must be established (or, at least, sketched out).

SOFTWARE ENGINEERING DEFINED
The advent of low-cost processing in the shape of the microprocessor and its associated memory and input/output devices, has meant that in many engineering applications it is now more cost effective to use a microcomputer than to design dedicated discrete hardware to perform the same function. Indeed 'micro-controllers', (small single board microcomputers, incorporating RAM, PROM sockets, and I/O), are now standard off-the-shelf items, so that often the principal design effort is reduced to, first, interfacing between the embedded micro-controller and the real-world (i.e. signal conditioning, A/D, D/A conversion etc.), and, second, developing the software to run the micro-controller. The important feature of this new approach is that the functionality of the machine is realised through software design rather than hardware.

This paper was first presented at the Conference on Electronic Engineering – The Way Ahead, held at the University of Hull, England, in March 1984.
This approach has a number of distinct advantages. In particular, costs are reduced, both by reduced hardware design effort and the use of low-cost mass-produced micro-controllers. More significant, however, is the increased flexibility of designing with software; modifications or enhancements are simply a matter of installing new software by replacing PROM devices, and the functional complexity that can be achieved in software is often far greater than would be feasible using dedicated hardware design.

Of course these benefits cannot be realised unless the engineer is a competent programmer. However, writing the software for embedded micro-controllers involves, in the view of the author, far more than just programming skills; and herein lies the distinction between programming and software engineering.

Since the software (which shall hereafter be referred to as 'the application'), running in the micro-controller will inevitably be performing some monitoring and/or control function, it will be intimately concerned with the hardware input/output interfaces. Many applications will be required to run a number of simultaneous processes in real-time, with the resulting need for multi-tasking and interrupt driven as well as polled input/output. An effective software engineer must understand these techniques, as well as having detailed knowledge of the hardware.

To summarise this section: in a modern electronic engineering context, software engineering might be defined as 'the discipline of developing, installing and maintaining the software for embedded micro-controllers'. A software engineering course should not teach either programming or microcomputer hardware, but should draw together these two disciplines within the framework already described. It will be predominantly software biased, but with frequent reference to the hardware, and should not be based on any particular hardware configuration.

**LANGUAGES FOR SOFTWARE ENGINEERING**

Often the choice of which language to write the application in is made for the software engineer. If the micro-controller has limited memory, or the application is time critical and needs to run as quickly as possible, then the choice of language must be assembler. Few high-level languages have the facility for designing multiple-tasks or concurrent processes, or for arranging interrupt servicing. Indeed few compilers will generate a ROMable stand-alone 'package' of code, so, again, assembler is the only option.

Assembler must, therefore, be the main vehicle for illustrating the techniques of software engineering, and the author has adopted Intel 8080 and DEC PDP-11 MAC-11 assemblers as the course standards. These were chosen both because of the availability of development facilities, and because they represent one of the simplest and most sophisticated assembly languages respectively, in common use. Students taking this course are already familiar with programming in assembler, but not necessarily with these two machines. At this stage (third year) they can be expected to 'pick up' a new language without too much difficulty and, indeed, this seems to be the case. In any event, handouts describ-
ing the two instruction sets in detail, together with details of how to access the various development facilities, are given at the start of the course. The course then begins with a number of 'primer' lectures to cover a few key features of 8080 and MAC-11 assembler including addressing modes and stack mechanisms.

Despite the foregoing discussion, the software engineer will increasingly have the option of writing the application in a high-level language. Largely because of the advent of personal computing, there are now a number of low-cost high-level language compilers for microprocessors, and a few languages are emerging which are purpose-built as assembler replacement applications languages. For example 'C', PASCAL and 'FORTH' are becoming increasingly important, and are worthy of mention in a software engineering course. Also, surprisingly, 'BASIC' is available as an on-board language (as is FORTH) on a number of micro-controllers, allowing the exciting possibility of the target micro-controller doubling up as the software development system in low speed applications.

A METHODOLOGY FOR SOFTWARE ENGINEERING IN ASSEMBLER
An effective software engineer is not only a good programmer in a number of assembler and high-level languages, and conversant with a whole range of hardware-orientated techniques, but also capable of writing applications which are, ideally, easy to debug and install, well documented, maintainable, and even portable. (A good test of maintainable software is; 'could an engineer other than the original author debug, or enhance the software?') Although much has been said for and against 'structured languages', most agree that a structured, i.e. disciplined and modular, approach to programming and documentation does reduce development effort and enhance maintainability.

A structured approach to software engineering is relatively easy when the application is being written in a high-level language, particularly if the language is 'structured' in the formal sense. Assembler, however, is not self-documenting, and does not readily lend itself to any kind of structured approach. An application written in assembler is very rarely portable or easy to maintain, especially if the original author used obscure machine instructions in the coding.

One solution to this problem, developed by this author over a number of years of writing software engineering applications, is to write in a structured, 'gotoless' pseudo assembler. The concept of a structured assembly language is not new, although it is not widely recognised, and a number of practical languages have been developed; PL/M 8080 is particularly interesting, as are the macro-assembler based schemes of Walker, Harris and Hughes and Worrell. However, these are all actual languages, which need compilers (or special assemblers), and have syntax which must be adhered to. The scheme presented here is not a programming language, but a way of developing
algorithms, for assembler, which is quick, more readable than flowcharts and more easily translated into true assembler.

Fig. 1 illustrates this technique with a structured assembler description of a routine, for 8080, to print the contents of memory in the familiar hex plus ASCII format. This is a mixture of actual assembler instructions: 'call' and 'return', some self-documenting arithmetic: A = peek(HL+C), and some high-level control structures, (except that the 'variables' are 8080 registers). The only (self-imposed) rules of this language are that it should be readable, and that it must use the four control structures: if . . . then . . . else . . . endif, while . . . endwhile, repeat . . . until and for . . . next (or whichever words are personally preferred). The use of the structures is considered crucial, for the following reasons:

(i) They immediately clarify the flow of execution; and impart a readability to the algorithm normally reserved only for high-level languages.

; Display the contents of memory in hex and ASCII
; from start to end. In the format...
;  addr  hex bytes  ASCII
;  0000  41 42 43 00 FF 31 32 0D  ASC..12.
;
; Registers used: HL, C, A.
;
; Subroutines called: print_crlf, print_hl_in_hex
; print_space, print_a_in_hex, print_dot,
; print_char_in_a.
Display:
for HL=start to end step 8

  call print_crlf
  call print_hl_in_hex
  call print_space

; loop thru bytes, printing in hex
for C=0 to 7
  call print_space
  A=peek(HL+C)
  call print_a_in_hex
next C

  call print_space

; and again in ASCII
for C=0 to 7
  A=peek(HL+C)

; Only print if 20H <= A <= 7FH, i.e.
; an ASCII printable character. Print
; a dot if outside this range.
if A<20H then
  call print_dot
else
  if A>7FH then
    call print_dot
    else
    call print_char_in_a
    endif
next C

next HL
return

FIG. 1 Display routine in structured assembler.
(ii) They avoid the need, while designing the algorithms, to become involved in the distracting and error-prone construction of loops, and conditional jumps, in assembler. (Incorrect jump instructions are reckoned to account for about 80% of bugs in assembler programs). Furthermore,

(iii) since there are only four well-defined control structures, efficient and guaranteed bug-free code may be constructed once-only to implement these structures, so that their translation becomes very straightforward.

To illustrate the third point Fig. 2 lists the code, for 8080, to implement the 'if' structure, where the value to be tested is in one of the 8-bit registers B,C,D,E,H or L. Clearly, six different conditional tests must be worked out, although only two are illustrated here. Using code like this, the routine of Fig. 1 can be hand-translated into actual 8080 assembler very rapidly. Only the 'A = peek(HL + C)' requires any thought, representing only 5 instructions out of the total of 44 in the translated code. The majority of instructions are generated by control structures, with a reasonable likelihood of being error-free. After the translation, the original 'structured assembler' is not thrown away, but included in the comment field of the final assembler listing, thus providing useful self-documentation. Notice also that the algorithm, is now somewhat portable, since the only machine-dependent aspect is in the reference to processor registers.

The principal virtue of this approach to developing assembly language software engineering applications is its speed. Indeed, Lewis\(^7\), in an excellent book, has proposed a similar but more rigid scheme, which he calls 'speedcode'. Since its introduction (experimentally, at first), into the software engineering course given by the author, two interesting reactions have been noted. Firstly, students already familiar with programming in assembler report that work

```
  'if'  control structure                        assembler
    if <reg> <condition> <value> then          * JCF else
      dothis                                   ; do this
    else                                      JMP endif
      dothat                                   else: ; do that
    endif                                     endif:
```

* There are six different varieties of the Jump if Condition False (JCF) code, corresponding to the six conditions: =, >, <, >= and <=. For example,

```
  MOV A,<reg> ;if <reg> = <value> then
  CPI <value>
  JNZ else
```

or,

```
  MOV A,<reg> ;if <reg> >= <value> then
  CPI <value>
  JC else
```

**FIG. 2** Translation of 'if' structure into assembler.
planned in a structured assembler is 'cleaner' and easier to debug; their programming 'style' is improved. More significant however, is that those students who have difficulty with assembler, being overwhelmed by its need for attention to detail, feel much more confident using this new approach.

CONCLUSION
Many aspects of software engineering have not been covered in this paper, particularly the computer science-orientated skills, such as requirements specification, testing and validation and the managerial problems of large software projects. While these skills are not always needed within an electronic engineering environment, the engineer should be aware of them, and directed reading is probably sufficient here, for example Sommerville\textsuperscript{8}.

However, this paper has tried to show that software engineering for electronic engineers does need a new approach, because of both its hardware orientation, and the small target-machine aspect. The paper has outlined a few of the broad range of techniques that need to be covered in such a course, and, in particular, the 'structured assembler' methodology which has been found particularly useful. The full course syllabus is given in the Appendix, although this is in a continuous state of evolution, as any electronics course must be, to reflect the rapidly changing nature of the subject.

REFERENCES

APPENDIX

Third year Software Engineering 20 hours

1. \textit{Overview} (1 hour)
   What is software engineering? Guidelines on selection of processor, development system and language. The concepts of maintenance and portability.

2. \textit{The use of symbolic assembler} (2 hours)
   The importance of documentation, comment and layout in symbolic assembler. A typical set of assembler directives.

3. \textit{Examples from 8080 and MACRO-11 assembler} (5 hours)
   Including the use of assembler directives, conditional assembly and macros. Some features of 8080 and MAC-11; the programmers model, register usage, addressing modes and the stack. The distinction between macros and subroutines, parameter passing and recursion.
4. *Structured assembler* (5 hours)
The four control structures and their usage; IF, FOR, WHILE and UNTIL. Their implementation in 8080 and MAC-11 assembler. A major design exercise example illustrating the structured approach to design and documentation; top-down design using structured assembler, bottom-up translation and testing in actual assembler.

5. *High-level languages for software engineering* (2 hours)
Which languages are available and when to use them. Examples from C and FORTH. Interfacing between assembler and high-level languages; examples from MAC-11 and FORTRAN.

6. *The software/hardware interface* (4 hours)
A review of special I/O devices; PIO, UART, etc. polled and interrupt driven input/output techniques. Synchronisation of processes, and the handling of multiple tasks with different levels of priority. Interfacing constant rate data input or output with asynchronous software; cycling buffering.

7. *Advanced debugging techniques* (1 hour)
Including simulation and in circuit emulation (ICE).

**ABSTRACTS—ENGLISH, FRENCH, GERMAN, SPANISH**

**Software engineering for electronic engineers**
The introduction of software engineering into an electronic engineering curriculum can pose special problems for the lecturer faced with its implementation; foremost of which is ‘what exactly is software engineering when applied to current microprocessor engineering practice?’ This paper aims to answer this and related questions, and hence propose a course syllabus suitable for undergraduate electronic engineers.

**Ingénierie en logiciel pour ingénieurs électroniciens**
L'introduction de l'ingénierie en logiciel dans un programme de cours pour ingénieur électronicien peut poser des problèmes spéciaux à l'enseignant confronté à son implantation; le premier de ceux-ci étant la question: ‘qu'est exactement l'ingénierie en logiciel appliquée à la pratique courante d'utilisation des microprocesseurs?’. Cet article a pour but de répondre à cette question ainsi qu'à d'autres sur le même sujet, et de là propose un programme de cours adapté à des étudiants ingénieurs électroniciens.

**Software-Technik für Elektroingenieure**

**Ingeniería de software para ingenieros electrónicos**
La introducción de la ingeniería de software en un curriculum de ingeniería electrónica presenta problemas especiales en su implantación; lo más importante es contestar esta pregunta: ¿Qué significa exactamente ingeniería de software cuando se aplica a la práctica corriente de la ingeniería de microprocesadores?’. Este artículo intenta responder a esta y otras preguntas relacionadas, y propone el temario de un curso apropiado para alumnos de ingeniería electrónica.