A DESIGN STRATEGY FOR MICROPROCESSOR SOFTWARE

HARVEY A. COHEN and RHYS S. FRANCIS
Latrobe University

The CA linkage strategy involves the division of software into control and algorithmic parts and requires a precise specification of the barrier in-between. In using this strategy a macro-cross-assembler has been used to provide high level constructs for implementing software for the microprocessor based controllers used in the OZNAKI educational project. The code produced is very readily updated following hardware configuration changes and easily transferrable between different processors.

1.′ INTRODUCTION

What has recently become recognised as the microprocessor software crisis has two aspects, which can be termed the production problem and the update problem. There are major logistic issues in developing microprocessor software because of the intermixture of hardware and programming knowledge implicit in the overall design. But having developed functional software for a particular hardware configuration, the second great problem of update emerges. Quite minor changes in hardware require an extensive software effort of amendment. Moreover systems in the field cannot be easily reconfigured in line with product improvements. Yet for the foreseeable future there will be innumerable possibilities for economies and improvements in microprocessor based products as new integrated circuits including processor chips are released that embody functional enhancements but also subtle differences from their precursors.

This paper outlines an approach to software design, termed the ,CA linkage strategy, which experience suggests can completely eliminate some major update problems and simplify others while at the same time providing a basis for software development systems. This approach requires the complete separation of software into control and algorithmic code. To use the strategy involves imposing special requirements on the linking of control and algorithmic code, hence the name. Once these linkage specifications have been made the separate areas of software can be independently developed and assembled into machine code. If the linkage mechanism is realised via what we call a link wall then hardware changes which necessarily alter control code will not affect algorithmic machine code. Likewise enhanced versions of algorithmic software may be utilised without altering the control code. We have also developed a system that allows this algorithmic software to be written at a level higher than assembler code. This system has been used to develop processor independent code for the microprocessor based controllers needed in the OZNAKI educational research project.

2. A REAL SOFTWARE DISTINCTION

There is a traditional division between software and hardware which is seen as being as distinct as punching code and soldering ICs onto PC boards. Applied to microprocessor system development, this division has meant that hardware designers have seen their role limited to producing functional electronics. In this traditional set-up, software writers are then presented with a precise hardware specification replete with bit-specific details. If this hardware is changed, various software alterations have to be made. What is notable about developing microprocessor software is that it seems difficult to apply the lessons on structured programming learnt so painfully in the software engineering of mainframe computers. Altogether what has emerged is a situation that has been described as the microprocessor software crisis. This crisis has received wide discussion, recently in such places as the Journal of Software Engineering, the recent SIG PLAN devoted to "Programming Systems in the Small Processor Environment" (April 1, 1976) and the February 1977 issue of the "Computer Magazine".

To make any headway on this problem it is necessary to examine the nature of the software changes induced by hardware configuration changes. Major portions of any well-structured software are unaltered by such changes. What needs to be done, to eliminate once and for all the sheer messiness of having to make innumerable tiny alterations in what had been functional code, is to erect within the software a barrier. Outside of this barrier lies software that is not affected by hardware configuration changes. This software is essentially logical, configuration free and we call it algorithmic software. Within the barrier lies the control software which includes such items as input, output and timing routines, special execution mode routines and interrupt handlers. Our development strategy requires that properly constructed programs should never directly penetrate the barrier between algorithms and control. Instead, all communication between the two must be via specially provided bridges or links.

The first problem in system development is to decide the sorts of control operations that are needed. One must then specify the partitioning of memory space between the control and algorithmic software and the details of the bridges between them. At this stage the so-called software problem divides neatly into two parts. The details of the control software are irrelevant to the algorithmic programmers who only need to know the bridge specifications. Likewise for the control programmers.

3. BRIDGE SPECIFICATIONS

For small systems, where all the code is to be contained in one ROM, the bridges could consist of well defined subroutine calls which accept and return information in a standard manner. However, in large systems, where the control and algorithm code exist in separate ROMS, PROMS or RAMS, the link wall must be given a physical realization in memory. That is, a block of memory is assigned to be the link wall. In this memory block is placed a collection of jumps to the current start addresses of the control routines. For some processors, the link wall would also serve as a storage space for the return addresses associated with calls to control routines. As far as the algorithm programs are concerned, it is the address within the link wall of the jump to a routine that identifies that routine. It is this address that is used in a subroutine call to a control program. Thus, provided that the address of the link wall and the order of the jumps to routines within it are kept unchanged, the algorithmic code is fully portable across hardware alterations. Such alterations may well incur extensive recording of the control software. However the only change to the link wall will be, perhaps, different addresses for the jumps, and the algorithm code will require no changes at all.

The advantage of this approach to any multi-ROM system, is that reprogramming of code within one ROM is not reflected in address changes and consequent remasking of the other ROMS. In micro-computer systems, which are intended to be end user programmable, the adoption of this technique for the system monitor, which is basically a collection of control-type programs, leads to further advantages. If the link wall is maintained in RAM the end user can replace an appropriate address in the wall with the address of his own routine, thus selectively deactivating or updating routines stored in ROE. Of course the need to be able to transfer user programs across monitor and hardware updates is most apparent in this particular area.

It is constructive to contrast our approach with that of Motorola who have manufactured several masked "monitor" ROMS for M6800 systems. As a crude attempt at portability, the original monitor program, of 256 bytes, was incorporated en block with the 512 bytes of their next monitor program, even though only a few of the old routines were required. The subsequent enhanced monitor ROM, of 1024 bytes, is completely incompatible with its forerunners. Had a link wall in RAM been conceived, all of these various products could have been compatible with user programs.

4. PROCESSOR INDEPENDENT ALGORITMS

The algorithmic software, having had hardware dependencies removed, lends itself to implementation in a higher level language. However traditional implementations of high level languages require massive effort to produce in the first place and, for microprocessors, an entirely ridiculous effort for update on processor variations. We have used a cross-assembler with advanced macro facilities to implement, in a few man months, an ALGOL like development language, provisionally called HELL for Highly Extensible 'Luverly' Language. This system has been used to produce software for micro-computers based on both the M6800 and the IN8080, generating the code for each from the same piece of high level software.

The modules of the system implementation language HELL are examples of what BROWN (1972) calls "textual macros". These macro modules were specially constructed to enable combinations to form statements in a higher level language. As all code is written only using these statements the resultant software can be made entirely processor independent.

5. MACROS

Traditionally macros are introduced as a means of extending the instruction set of an assembler. To quote the now classic example (where in this instance the expansion is for an IN8080 processor) the definition

```
   DEFINE SUM (Pl,P2,P3)<
          LDA     A,Pl
          ADD     P2
          STA     A,P3>
```

implies that the statement

```
SUM(100, NUMRESULT)
```

occurring in the assembly code will generate the machine code corresponding to

```
          LDA     A,100
          ADD     NUMB
          STA     A, RESULT
```

which adds the contents of the bytes addressed by the first two parameters, in this case 100 and NUMB and stores the sum in the byte addressed by the third parameter which in this case is RESULT.

Clearly the macro definition has added an entirely new and higher level construct to the assembly language. What has not been pointed out in the past is that the call or invocation of SUM contains no processor dependent information. By substituting a different definition of SUM, the identical calling sequence can be made to produce valid code for any other processor.

Starting from this realization, we have developed a comprehensive library of interacting macros which, using an assembly time stack, implement such structuring constructs as

```
    IF (relation) THEN
    . . . . . .
    ELSE
    . . . . . .
    ENDIF
```

and

```
    LOOP
    . . . . . .
    EXITIF (relation)
    . . . . . .
    ENDLOOP
```

along with ASSIGNMENT statements, PROCEDURE and DATA-type declarations and special macros which generate calls to specific control software, such as timing routines.

6. OBJECT CODE EFFICIENCY

The requirement of processor independence for macros enhances modularity but leads to the small overhead of storing intermediate results between macro statements. However using assembly-time facilities to collect and pack subroutines the overall organization is that of hand assembled code. In the domain of large machines it has been demonstrated by TANENBAUM (1976) that macro based languages can generate more compact and faster executing code than compiler based languages. For the major micro-processors the alternatives to using a macro based language would be compiler based systems such as PL/4 (see G. A. Kildall) or a much less efficient interpretive language such as BASIC. Of course for a bit-slice processor these alternatives do not exist. Moreover, the portability between processors achieved by HELL has no counterpart in any other system.

7. CONCLUSION

The effort expended in developing the system implementation language HELL has been recouped in faster implementation of application programs for the robotic controllers and TV graphics systems of the OZNAKI project. Moreover the subsequent portability of OZNAKI software means that the project is not restricted to today's hardware, but is ready to shift to the next generation. In fact the programs required assemble to between 2 and 4K bytes of code on the processors used, and it seems doubtful whether reliable code could be generated in assembly language from flow diagrams and the like. This experience suggests that even in the absence of a higher level language the CA linkage strategy has significant advantages in permitting the separate development of control and algorithmic code.

8. REFERENCES

P. J. Brown, “Macro Processors and Portable Software", Wiley, London, 1974.

A.S. Tanenbaum, "A General-Purpose Macro Processor as a Poor Man's Compiler-Compiler", Trans. Software Eng., Vol. SE-2 June 1976, pp. 121-125.

G.A. Kildall, "High-Level Language Simplifies Microcomputer Programming", Electronics, June 27, 1974, pp. 103-109.

J. D. Gannon and J. J. Horning, "The Impact of Language Design on the Production of Reliable Software", Proc. Intl. Conf. on Reliable Software, April 1975, pp. 10-22.

H.A. Cohen, "The OZNAKI Robotics Language OZ", Proc. 7th Aust. Comp. Conf., Vol 1, 1976, pp. 128-143.