Modular programming conventions in assembly languages

by SHY-MING JU
University of North Carolina at Charlotte
Charlotte, North Carolina

ABSTRACT

The methodology of modular programming has received increased attention in which the most basic yet important part is the intermodule communication. Circumstances may dictate the use of module language in software development. However, linkage conventions at this level are generally lacking or unworkable. This paper proposes a calling sequence convention and a calling sequence handler for intermodule communication. This scheme is simple to use and enhances good programming practices, such as simplicity, flexibility, comprehensibility and integrity.

INTRODUCTION

The methodology of modular programming has received increased attention in recent years. It permits parallel development of modules and affords product flexibility and comprehensibility. Criteria for modularization have been recommended to design each module around a suitable abstraction or to have each module hide only one design decision. If the system configuration and software tools are adequate, modular programming is usually undertaken in high-level languages. However, when memory storage is a limiting resource, or no suitable high-level language is available, or run-time support software is insufficient, or execution efficiency is of paramount importance, then assembly language programming will be the last resort. Programming in assembly language especially prevails among minicomputer users because most of the existing minicomputers suffer from one or more of the above mentioned deficiencies.

Indeed most programming languages directly or indirectly make use of assemblers, linkage editors, and possibly macro processors, therefore modularly designed software is nothing more than a set of independently assembled subroutines and data blocks when it is viewed as a runnable representation. Consequently the standards and conventions on intermodule communication at this level are the most basic yet important part of modular programming. The author has observed various manufacturers' software and found that such standards and conventions are generally lacking, poor or unworkable. This lack of standards and conventions in the software has resulted in incompatibility within the same installation, as well as between different installations.

In the following sections we will propose conventions and supporting mechanisms for intermodule communication which are simple to use and enhance good programming practice.

LINKAGE CONVENTIONS

A well-documented and well-known linkage convention is the one established for IBM System/360 which requires the called module to save the values contained in the general registers as soon as it gets control, and then restore those values prior to returning control. It also requires the calling module to provide a "save area" for that purpose, and to pass the address of the save area to the called module through Register 13. It further requires the calling module to use registers 15, 14, and 1 in the calling sequence, in which Register 15 will contain the entry point, Register 14 the return address, and Register 1 the parameter or the address of a list of parameters.

To an assembly language programmer, this kind of convention is certainly not enjoyable, especially if a macro facility is not available. As far as solving the problem is concerned, this linkage mechanism is not part of the algorithm; therefore it tends to be overlooked by the programmer and thus causes errors when intermodule communication takes place.

The save area reserved in each module automatically eliminates the potential for recursion or reentrancy, thus contradicting a goal of modular programming.

In our proposal, the responsibility for saving and restoring the contents of registers, and that of providing a save area, are delegated to a "calling sequence handler" which is invoked by a calling sequence through a software "trap" such as a supervisor call. To illustrate the complete process of intermodule communication, we will choose the Interdata/M70 minicomputer as our base of representation, partly because it is similar to yet much simpler than an IBM System/360, partly because our idea has been thoroughly tested on it.

* This work was supported in part by a Faculty Summer Research Grant sponsored by the Foundation of the University of North Carolina at Charlotte.
The Interdata/M70 minicomputer is a halfword-oriented byte-addressable machine. It has sixteen general registers but does not support base-register addressing scheme. It accommodates up to sixteen supervisor call services; the SVC instruction has a format similar to that of a RX-type instruction in IBM System/360.

The format of the proposed calling sequence in a calling module is:

```
SVC 0,(entry point)
DC (arg1), (arg2), . . . , (argN)
```

```
ENTRY CALLER,STACK
* CALLING SEQUENCE HANDLER
* REGISTER 15 IS RESERVED FOR STACK POINTER
* REGISTER 0 AND 1 WILL NOT BE RESTORED
*
CALLER XHR 1,1
CALL CHI 15,STACK+160
BNL STKOVF
STM 0,0(15)
LI 0,'X'94'
LI 2,'X'60'
LI 1,'X'90'
SIS 1,2
STH 1,2(15)
CALL1 AIS 1,2
LH 3,0(1)
CLHI 3,32
EPL CALL2
NHI 3,'X'F'
SLLS 3,1
STH 3,LC+2
LI 3,'H400F'
OHR 3,2
STH 3,LC
```

```
DH 4
SII 2,'X'10'
CALL2 AHI 15,32
BR 0
EXIT SII 15,32
CHI 15,STACK
BL STKUNF
LI 1,2(15)
LHI 1,'X'98'
SLLS 3,1
STH 3,STH+2
LH 3,'H400F'
OHR 3,2
STH 3,STH
TH DS 4
CALL3 AIS 2,4(15)
LD 4,0(1)
```

```
BR 1
STKOVF DC 'X'B000'
STKUNF DC 'X'B000'
H400F LH 0,15
H400F STH 0,15
STACK DS 320
END
```

Figure 1—Listing of calling sequence handler
Modular Programming Conventions in Assembly Languages

where (entry point) is usually an external symbol declared in the called module, and \( \langle \text{arg1} \rangle, \langle \text{arg2} \rangle, \ldots, \langle \text{argN} \rangle \) are register numbers representing any of the sixteen registers.

The format of return sequence in a called module is simply:

\[
\text{SVC 0,0}
\]

For each calling sequence the calling module should set to each register in the “argument list” either an operand (i.e., call by value) or an address of a list of operands (i.e., call by address), as dictated by the called module. The registers in the argument list will be called “argument registers.” The calling module can further “flag” some of the argument registers with a ‘+16’ to indicate that the contents of the flagged ones may be updated after the call is completed. Note that when a flagged argument register contains an operand, it is essentially a “call by value and result.”

Before getting into more detail, let us consider a module which searches for the first occurrence of a specified character through a character string in a 80-bytes buffer area pointed to by a pointer. If such a character is found, its relative position is reported, otherwise the relative position is set to 80.

The assembly language code of this module is listed below:

```
ENTRY FIND
* THE CALLING SEQUENCE IS:
* SVC 0,FIND
* DC PTR,TARGET,INDEX+16
*
PTR EQU 15
TARGET EQU 14
INDEX EQU 13
WRK EQU 2
*
FIND XHR INDEX,INDEX SET INDEX T0 0
G0 LB WRK,(PTR) LOAD A CHARACTER FROM STRING
CHR WRK,TARGET COMPARE IT WITH THE TARGET CHAR
BE RETURN IF MATCHED, DONE.
AHI PTR,1 INCREMENT PTR BY 1
AHI INDEX,1 INCREMENT INDEX BY 1
CHI INDEX,80 TEST IF BUFFER EXHAUSTED
BNE G0 IF NOT, GET NEXT CHARACTER
RETURN SVC 0,0
END
```

A calling module may be outlined as:

```
EXTRN FIND
.
.
LENGTH EQU 5
CHAR EQU 8
PTR EQU 10
.
.
LHI CHAR,C'$'
LHI PTR, BUFFER
.
.
SVC 0,FIND
DC PTR,CHAR,LENGTH+16
.
.
BUFFER DS 80
END
```

It should be noted that even though the module “FIND” has updated both PTR and INDEX, only the updated value of INDEX will be carried back to the calling module because it was flagged in the calling sequence. Also noted is that the called module accesses actual parameters through registers starting from Register 15 and in descending order, without knowing which registers were used by the calling module as argument registers.

To discuss the implications of the proposed calling sequence, we need to describe the functions of the calling sequence handler first.

THE CALLING SEQUENCE HANDLER

The calling sequence handler maintains a “save stack” for each “job.” By job we mean a sequence of logically related processes, each process represents a running module. The stack area is inaccessible to programmers.

When the handler is invoked by a calling sequence, it first stores the contents of all registers and the return address into its stack, and adjusts the stack pointer. It then copies the contents of the argument registers into Register 15, 14, . . . , in descending order so that the correspond-
ence between the actual argument registers and the formal argument registers is established. After this is done, it transfers control to the called module.

When the handler is invoked by a return sequence, it first copies the current contents of the corresponding registers into the flagged argument registers, and restores the remaining registers. It then retrieves the return address from the stack, adjusts the stack pointer, and transfers control to the instruction following the specific calling sequence.

A listing of the assembly language calling sequence handler is shown in Figure 1. In that implementation Register 15 was reserved for the stack pointer in order to eliminate the operations of loading and storing the stack pointer. Also, Register 0 and 1 were reserved as scratch registers so that intermediate results may be held there. Therefore, we will pretend that there were only fifteen general registers available to the programmer and Register 0 and 1 should only be used as scratch registers.

IMPLICATIONS OF THE PROPOSED SCHEME

In the proposed scheme, there is only one save stack for all modules of a job. Since it is unlikely to have all modules activated at the same time, the total memory storage allocated to the stack can be less than the total of all the save areas that would have been allocated to each of the modules.

The maximum number of argument registers allowed in a calling sequence is the number of registers available to the programmer in a specific implementation; for example, thirteen in our case. This limitation will not, however, impose a serious inconvenience when a large number of argument registers are required, because the parameters can be placed in a block of memory storage with only the block address passed through an argument register.

As assembly language programmers all know, efficiency in program execution and conservation of storage space can be attained by avoiding unnecessary memory accesses and by using RR-type instructions wisely. The convenient facilities of call-by-value and call-by-value-and-result for the calling sequence were designed to encourage this kind of good programming practice. Call-by-address was provided mainly for parameters which are structured data, such as an array.

It is interesting to note that if a module is pure and if its calling sequence involves only call-by-value and/or call-by-value-and-result, it will naturally be recursive and reentrant, as can be seen in the following example which computes the N-factorial:

```
* THE CALLING SEQUENCE IS:

ENTRY
SVC 0,FACT0R
SVC RESULT+16,N

RESULT EQU 14
N EQU 13
M EQU 2

FACT0R LHR N,N
BZ LBL IF N=0 THEN RESULT IS 1
CHI N,1
BE LBL IF N=1 RESULT IS ALSO 1
LHI M,-I(N) SET M TO N-1
SVC 0,FACT0R
DC RESULT+16,M COMPUTE FACT0RIAL OF N-1
LHR 1,RESULT
MHR 0,N
LHR RESULT,1 SET RESULT TO RESULT+RESULT
SVC 0,0 RETURN
LBL LHI RESULT,1 SET RESULT TO 0
SVC 0,0 END
```

It is recommended that any communication between a module and its global environment should be through argument registers, and the conventional way of accessing common area through "external declarations" should be used with great discretion or better yet, should be prohibited. This practice would achieve two advantages; on the one hand, the fact that arguments are explicitly shown in the calling sequence and only flagged arguments can be updated by a called module greatly improves the readability of the module logic; on the other hand, the fact that a module does not know and does not need to know its global environment except through the argument registers enhances module integrity. This practice is especially important in large software development efforts undertaken by a team of programmers.

The calling sequence handler can help in system integra-
tion testing by inserting code that prints a message whenever control enters or exits from a module. Should a fatal error occur during execution, the module at fault can be determined and the execution history retained in the save stack can be dumped out for diagnosis. The calling sequence handler can even help in resource management; for example, allocation and deallocation of work area to a module. Since the handler monitors module entrance and exit, it can also participate in dynamic linking when appropriately modified.

CONCLUSIONS

The proposed calling sequence conventions and the calling sequence handler have been used successfully in a large software project which employed modular programming in a chief programmer team approach. All modules were developed and tested independently, and during the system integration testing no major difficulties were encountered. The hierarchy of modules in its Chinese output subsystem is shown in Figure 2. By standardizing the calling sequence, some of these modules were able to be used without any modification in developing other software such as a text editor and a special-purpose language processor.

The proposed scheme should work equally well on machines with fewer general registers such as a PDP-11. In that case some storage locations can be set aside to simulate registers. A close look at the assembly language listing of the calling sequence handler should reveal that most of the instructions perform the functions which would be required in any linkage conventions. The only extra overhead is due to rearranging argument registers into descending ordered registers prior to entry and updating the flagged registers upon exit. However, this overhead is outweighed by gains in programming simplicity and enhancement. Ideally, the calling sequence handler should be hardwired or implemented in the form of a microprogram to further reduce overhead.

Figure 2—Hierarchy of modules in the Chinese output subsystem
ACKNOWLEDGMENT

The author is indebted to the EDP Center of the Directorate General of Budget, Accounting and Statistics of the Executive Yuan, Republic of China, for giving him the opportunity to test the idea presented herein during the development of the software project MODEST.

REFERENCES