Computer Bits

By Hal Chamberlin

ASSEMBLERS

ACCORDING to a recent magazine survey, one of the most popular applications of personal computers is software development, or simply writing programs. As anyone who has been bitten by the programming bug undoubtedly knows, each new program is always bigger and fancier than the last. Beyond a certain point in program complexity, however, the use of an assembler program is almost mandatory to eliminate most of the drudgery associated with hand coding in octal or hex. This is particularly true when one wishes to make a "small improvement" to a hand-assembled program which otherwise requires it to be rewritten.

Functions of an Assembler. Using an assembler in machine language program development has three important advantages over hand coding. First, an assembler allows the programmer to use operation mnemonics such as "LDA" for the "load register A" operation rather than the octal code 072 (8080 microprocessor). When looking at a program you wrote several weeks ago or one written by somebody else, the LDA is much more meaningful than the 072, which in turn makes the program easier to understand.

The second and most important advantage is that the addresses of sections of code and data items can be given *symbolic names* and referred to by name. Again, a name like TAXTAB used to refer to a table of tax rate data is more meaningful than its address which might be 005:120. The most important benefit of symbolic names comes when a program is changed for some reason. With a hand-coded program, some of the addresses used in the program would probably have to change as sections of the program and data are shuffled around to make room for additions. Then, every reference to addresses that were changed would also have to be changed. The result is that, in a large program, a considerable number of changes may be necessary for what would otherwise be a minor addition. With symbolic names, the assembler can do all of the address shuffling when the program is reassembled and the programmer need be concerned only with the additions. The concept is analogous to solving an equation in general using symbols and algebra and then substituting actual values into the *solution* rather than solving the equation for each set of values needed.

A third advantage is that the use of an assembler tends to develop good program documentation habits which adds to the value of a program. All assemblers allow the latter part of each statement to be used for comments. A well-written program has an English explanation of what the machine instructions are accomplishing as comments on nearly every statement. A neat assembly listing of a program is also much easier to reproduce and read than hand scrawls on coding sheets. Conversely, buying a machine language program without documentation in the form of commented assembly listings is like buying electronic equipment without a schematic.

Using the assembler program itself is generally quite simple. First the assembly language program which is called a *source* program is converted into machine readable form. Such a form may be ASCII characters on paper tape, audio or digital cassette records, floppy disk sector records, or even ASCII data in memory depending on the system and assembler used. Usually some kind of program editor is used to aid in entering and editing the source program. Next the assembler is loaded and executed. During execution, the assembler will scan the source program and produce a *listing* file containing a copy of the source program along with the octal machine codes and an *object* file containing only the machine codes.

The assembler may also flag some statements as having errors. Common errors that an assembler can catch include using non-existent instruction mnemonics and undefined symbols. The latter is the case when a reference is made to a symbolic address but an actual address is never assigned to the symbol. These and other errors detected by the assembler are usually caused by typing mistakes. After editing the source program to eliminate errors and reassembly, the object program is ready to be loaded into memory and executed.

Types of Assemblers. Although all assemblers perform basically the same function, there is considerable variety in the implementation and use details. Perhaps the most distinguishing characteristic is the number of scans or *passes* over the source code done by the assembler.

A classical assembler makes two passes over the source program. During the first pass, all symbol definitions are searched out and placed in a *symbol table* maintained by the assembler. During the second pass, the mnemonics are translated into their octal equivalents and the listing file and object file are generated. The two passes are needed because a reference to a symbolic address may occur in the program ahead of the definition of the symbol. This is called forward referencing. If the assembler is to know what octal address to substitute for the symbol, it will have to see the definition first.

Several attempts have been made at one-pass assemblers and a couple of these are available on hobbyist systems. The advantage of a one-pass assembler is increased assembly speed since the source file, which may be many thousands of characters in length, needs to be read only once. Often however the one-pass assembler imposes

```
        .MACRO
*       MACRO DEFINITION FOR A DOUBLE PRECISION ADD FROM MEMORY
*       MACRO-INSTRUCTION
*       ADDS THE CONTENTS OF $ADDR AND $ADDR+1 TO REGISTERS B AND
*       C WITH THE RESULT IN B AND C, CONDITION FLAGS UNAFFECTED

$LBL    DPAD $ADDR              DOUBLE PRECISION ADD PROTOTYPE

        PUSH H                  SAVE H AND L
        LHLD $ADDR              GET TWO BYTES TO ADD IN H AND L
        DAD  B                  ADD THEM TO B AND C
        MOV  B,H                COPY RESULT INTO B AND C
        MOV  C,L
        POP  H                  RESTORE H AND L
        .MEND
```

Fig. 1. Example of macro definition.

restrictions on program organization and the free placement of symbols. This is due to the "look ahead" problem mentioned earlier. Sometimes a one-pass assembler is "faked" by a two-pass one. In this case the source file is read for the first pass and then saved in memory for the second pass which is invisible to the user. The difficulty with this approach is that a large amount of memory is needed to assemble a reasonably large program.

Occasionally a "three-pass" assembler is seen. These are really two-pass assemblers with the second pass split in two to accommodate a Teletype with built-in paper tape. These machines cannot punch the object file at the same time as printing the listing file so a separate pass is required for each function.

A *conversational* assembler is another variation. Basically a combination of a simple text editor and a conventional assembler, the conversational assembler is very convenient for experimentation and testing of short programs and subroutines. Operation of a conversational assembler is much like most BASIC language systems. The program is typed in line-by-line and edited using line numbers and simple editing commands. When a RUN command is given, the program is quickly assembled directly into memory and executed. Program size is limited since the source program ASCII text, symbol table, and object program as well as the conversational assembler program itself must all fit into memory at once.

Advanced Assembler Features. As assembly language programming experience increases, some of the more sophisticated assembler features available will be appreciated. Although these features have been rare in hobbyist oriented systems, the assemblers being supplied with recently announced floppy disk systems generally have most of them.

One such feature is *macro-instruction* capability. A macro-instruction (often abbreviated as "macro") is one that may generate many machine language instructions when assembled. When writing a program, macro-instructions may be used just as if the microprocessor actually had them as real instructions in its repertoire.

```
                *       EXAMPLE PROGRAM SEGMENT ILLUSTRATING USE OF DPAD MACRO

001:100 116             MOV  C,M        LOAD ORIGINAL RAW VALUE (16 BITS)
001:101 043             INX  H
001:102 106             MOV  B,M
001:103                 DPAD CORR       ADD IN CORRECTION FACTOR
001:103 345         -   PUSH H          SAVE H AND L
001:104 052 200 001 -   LHLD CORR      GET TWO BYTES TO ADD IN H AND L
001:107 011         -   DAD  B          ADD THEM TO B AND C
001:110 104         -   MOV  B,H        COPY RESULT INTO B AND C
001:111 115         -   MOV  C,L
001:112 341         -   POP  H          RESTORE H AND L
001:113 160             MOV  M,B        UPDATE WITH CORRECTED VALUE
001:114 053             DCX  H
001:115 161             MOV  M,C
```

Fig. 2. Example of use of a macro-instruction.

Macros can be defined by the programmer at the beginning of his program according to his needs. Although exact details of macro definitions and usage differ among various assemblers, a typical macro definition is shown in Fig. 1. The .MACRO on the first line alerts the assembler that a macro definition follows rather than ordinary program instructions. The next line gives the macro *prototype* which defines how the macro-instruction would look in a source program. The symbols preceded by dollar marks are sometimes called "dummy arguments" because, when the macro-instruction is actually expanded by the assembler, they are effectively replaced by the actual symbols used in the macro-instruction. Following the prototype are the actual machine instructions that would be generated when the macro-instruction is used. Note the use of the same dummy argument in the LHLD instruction as in the prototype. The .MEND signals the assembler that the macro definition is complete. The definition is then saved by the assembler in a special table in memory reserved for that purpose.

Figure 2 shows the use of this macro-instruction in a program (octal). In this example all of the instructions generated when the macro was expanded are shown on the listing with a preceding minus sign. Generally the assembler will have a command that would suppress printing of these expansion instructions if desired. With a good library of macro definitions, assembly language programming may become almost as easy as programming in a higher level language.

Another advanced feature is called "relocatable object code" capability. An assembler having this feature supplies additional information in the object file so that it may be later loaded into memory *anywhere* desired completely automatically without difficulty. A special *relocating loader* must be used to interpret this extra information and load the object file into memory. Not only are the addresses of all jump, call, and direct addressing instructions changed, but address constants and other location dependent symbolic references are changed. An additional feature of the relocating loader allows several object files that were generated at different times to be linked together into a single coherent program with all calls and jumps between the separate "modules" properly adjusted. This feature greatly facilitates the use of subroutine libraries without having to copy all of the source code into the program being developed every time a subroutine from the library is needed.

With this little bit of background, the reader should be able to evaluate more fully the assembly language program development facilities of a particular system. ◇