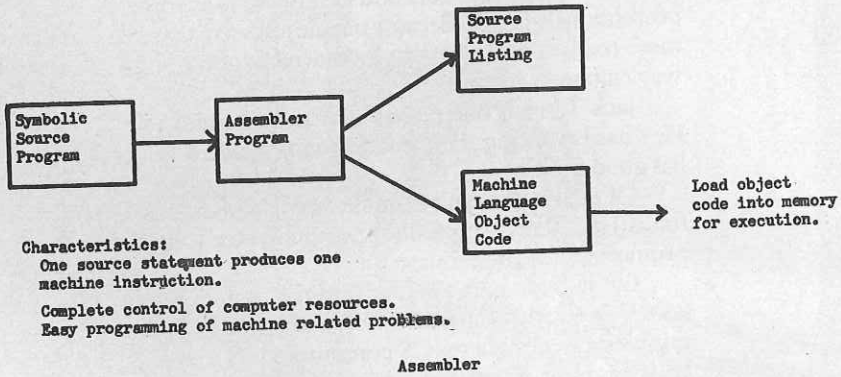
KOMPUTER KORNER

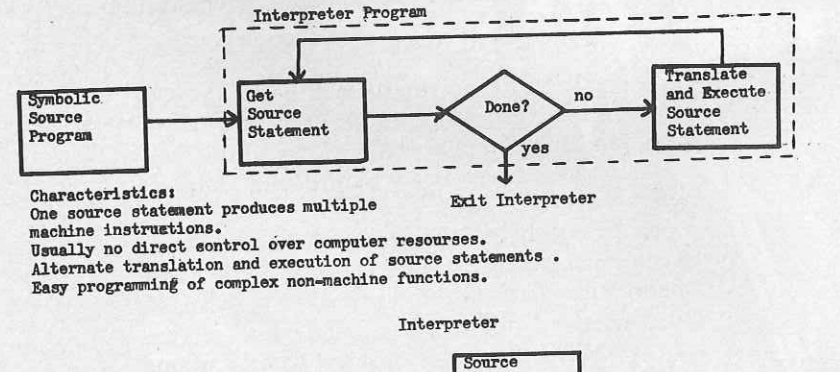
We are introduced to software—a classification covering appplications and systems programs.

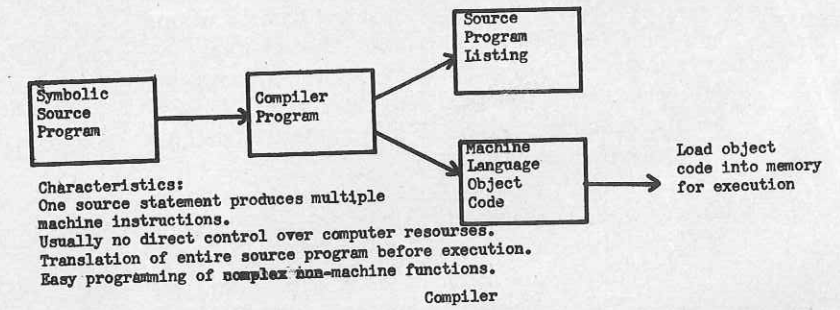
TIM BARRY

SOFTWARE IS A TERM THAT HAS ALMOST AS many definitions as there are programmers. In the broadest sense, software can be considered to be any program written for use with a computer. There are a considerable number of present misconceptions about software, and hopefully these columns will help put them to rest. The feeling that programming is an "art" or in some way mysterious has been around for some time. This is a self-defeating attitude that is probably particularly irritating to hardware designers learning software. While extremely sophisticated or elegant programming can approach an art form, so can hardware design, tennis serving or any other activity when performed by a highly trained person. Most of us will have to be content with getting the job done. This is no problem once you realize that 99% of all programming is a straightforward application of a handful of easily learned concepts and techniques.

Generally, software is divided into two broad classifications: applications programs and systems programs. An application program is written by a user to solve problems that are not related to the control of the computer system. Systems programs are used to translate and control the execution of applications or systems programs. For example, a home burglar alarm program would be an applications program, while a Basic interpreter would be a systems program. We will be concerned with applications programs in this series (games, home control, hobby use, etc.), but we must discuss briefly a type of systems program known as language translators.







As discussed earlier, for the microcomputer to execute a program, the machine language representation of that program must be present in memory. We then start the processor executing where the program begins and it executes the instructions. Unfortunately, the language that the com-

(continued on page 22)

KOMPUTER KORNER

(continued from page 20)

puter understands (1's and 0's) is not necessarily the most convenient for the human operator to use. A microcomputer or minicomputer will have several hundred machine instructions (the 8080 has 243), and a large computer will have thousands. Each machine instruction is represented by a unique code. The misplacement of a single bit can result in radically different program operations. Even worse, the insertion or deletion of a single instruction may require us to move or change many other instructions. While it is possible to write very simple programs in machine language, as the programs become more complex, the task of making changes and corrections becomes almost impossible.

To improve this situation we use *symbolic languages*. This allows us to use easily remembered symbols to represent machine instructions, addresses, and data constraints. The symbols used to represent the machine instructions are called *mnemonics*. When using a symbolic language, we still have complete control of the machine and it frees us from having to keep track of the large number of machine instruction codes and the value and location of each program element. The translation from this symbolic language (or *source* form) into the actual machine language (or *object* code) it represents is the process called *assembly*. Assembly can be accomplished by hand (we will learn how in the first programming lesson) or by a program called an *assembler*.

Assemblers

The assembler frees us from many of the easily bungled mechanical details of machine language coding and allows us to concentrate on the solution to the problem. For example, the machine code represented by the mnemonic ADD A might be 63. When writing a program it would not be necessary to remember the code, only the mnemonic. During the assembly process, any time the symbol ADD A is encountered, the assembler would automatically substitute the code 63. Similarly, the assembler would translate the entire program, producing one machine code for each mnemonic encountered. Also, the assembler handles all of the assignments of program addresses and symbol values. This makes a tremendous difference when it is time to make corrections to a program by inserting or deleting steps.

The only real disadvantage of assemblers for small systems is that they require a larger memory and more I/O devices than may be available on many hobbyist systems. Normally, a system with at least 4096 bytes of read/write memory, a teletypewriter, and a tape read/write device (paper or magnetic) are required to run an assembler program. However, the simple hand-assembly technique will offer many of the advantages of symbolic languages with no additional hardware. This will probably be the main way that programs will be written for small systems.

The next step up from the assembler in the software hierarchy are the so called "higher level" languages (see diagram). The higher-level languages are structured in a much more English-like fashion. Where assembly languages use mnemonics to represent single machine instructions, higher-level languages use words and operators to specify procedures that represent groups of machine instructions. For example, a program in assembly language to multiply two numbers and print their product on a terminal may take 30 or more instructions. Using a higher level language, the same function can be accomplished in three or four statements.

Another advantage of the higher-level languages is that they make it considerably easier to learn to program. The language processor handles all of the machine details, leaving you free to concentrate on solving the problem. Also, since the language syntax is independent of the machine structure, programs written in a higher-level language can be run on different machines with few modifications. All that is required is that the different computers have the translator for the language used in the program. Ease of programming complex functions combined with shorter program development times and program portability make higher-level languages the usual choice for applications programs on large computer systems.

Higher-level languages are divided into *interpreters* and *compilers*, based on how they produce and execute the final machine-code. An interpreter translates to machine language and executes each source statement each time it is encountered. This means that if a statement is used several times (in a program loop, for example), the statement will be translated every time it is encountered. A compiler, on the other hand, translates the entire program into machine language before it is executed. This means that each source statement is only translated once-per-compilation.

Basic and APL are interpreters while PL/M and FORTRAN are compilers. Compilers are generally more efficient but they are less responsive to frequent program changes. The entire program must be re-compiled if a single source statement is changed. Interpreters translate source statements as they are encountered, making for rapid program changes at the expense of less efficient program execution. Compilers also require more memory and are more complex to implement than interpreters, particularly on small systems. The smaller memory requirements and simpler structure of the interpreter will probably make it the most common type of higher-level language for the hobbyist.

As useful as they are, higher-level languages do have some drawbacks. Since they produce multiple machine instructions for each program instruction, higher-level languages do not always produce the best machine language. A problem solved using a higher-level language is apt to be longer and less efficient than the same task programmed directly in the machine's assembly language. (Efficiency, is of course relative. If it takes you an extra two weeks to make that assembly language program run in an application that is not speed or memory sensitive, it is arguable whether or not the more efficient code is worth the extra time.) Also, by learning a higher-level language, you are insulated from the actual working of the computer. You learn to program a "Basic" computer and not an

(continued on page 108)

KOMPUTER KORNER

(continued from page 22)

actual machine. Your programs will be largely independent, but you will not develop the understanding of computer operation and machine architecture that you will gain from programming in a machine's assembly or machine language.

Another factor to consider is size and availability. Higher-level language processors are at least as large as assemblers and require the same I/O devices. They are also considerably harder to develop.

All examples for this column will be presented in the assembly language of the 8080 microprocessor. This will give us maximum coverage of both hardware and software interaction while providing a good medium for programming.

Microcomputer evolution

The microprocessor has received a lot of press as being a revolutionary device that sprang fully formed from the logic designer's forehead. In actuality, the microprocessor is a logical evolution of increasingly sophisticated integrated circuit technology combined with program-controlled logic.

Early SSI (*S*mall *S*cale *I*ntegration) integrated circuits were simple gate functions. Logic designers combined these together to form more complex functions and systems. As the technology improved, it became possible to provide more complex functions as building blocks for logic design. At this point the integrated circuit designers were faced with the problem of what basic functions to build. Since there were many more devices on the chip than there were pins available for I/0, some decisions had to be made as to which inputs, outputs, and functions should be provided to make generally useful devices.

The families of MSI registers, counters, adders, memories, and so on are the partitioned logic devices that resulted. As the trend to more complex devices continued, the question of what types of general devices to build became a crucial factor. Eventually the decision came down to hardwired functions which would be built for large volume special applications (clock chips, calculator chips, custom controllers, memories, etc.) and general purpose devices whose function could be altered by means of programming (microprocessors). Actually, the calculator chips are really special purpose microprocessors, combining memory, control, ALU, and some I/O all on a single chip. While very cost effective for their specific functions, the calculator chip's architecture was compromised to the point of making it difficult to use for anything else. The more general purpose microprocessors, on the other hand, did not have their main memory on the same chip as the ALU and control. This extra room allowed the designers to implement more versatile architectures for use in general applications.

The first generation of microprocessors to be introduced were 4-bit machines bearing a strong resemblance to their dedicated calculator relatives. They were designed for use in arithmetic applications such as advanced calculators, process controllers and computer peripherals, and as a result their architectures were optimized for arithmetic operations on 4-bit BCD numbers. These were followed by the first generation of 8-bit machines. These were intended for use in more sophisticated control and data processing systems, eight-bits conveniently storing one ASCII character. The early four- and eight-bit machines all suffered from extreme compromises forced by the process and packaging technology. Heavily multiplexed data paths, limited instruction sets, and complex interface and timing requirements made these devices difficult to use.

The introduction of the 8080 marked the beginning of the second generation of microprocessors. Featuring a sound general purpose architecture with a broad instruction set, unmultiplexed data paths, and TTL-compatible N-channel technology, this device offered major increases in both speed and useful computing power. It is this microprocessor and the others of its generation that have brought us to the day of a computer in every home.

Whats next? To try and predict the trends in the microprocessor evolution at this point is apt to be futile, since the field changes at a bewildering pace. What we will try to do is establish some sound programming and analysis procedures that will allow us to keep up on the technology as it evolves. To give a basis for future comparisons we will use these principles on a real processor. In this manner we will develop skills that will be real and readily transferred to any other devices as the technology advances. **R-E**