What This Country Needs Is a Good 8-Bit High Level Language

Editorial
by
Carl Helmers

Have you ever tried to talk to a person who speaks a language other than your own? You know that the person must think as a rational, sentient being or you wouldn't make the attempt to communicate. As a *human being* your conversational partner's basic thought patterns are of necessity similar. Yet you can't understand him and he can't understand you. There are ways around this problem, given a sincere interest in communications by both parties.

An analogous problem exists in the field of personal computing as it is practiced by the readers of BYTE. If you translate the words referencing human beings into words referencing computers and reread the preceding paragraph, the result will be a corresponding statement about the computer communication problem: In the home brew computer world, there are a number of different computer architectures, all speaking different machine languages. There are even dialects of machine languages since each home brew designer or manufacturer makes specific choices about address space allocations and input/output port assignments used with the standard computer chip designs. Yet all of these machines are potentially programmable to do similar functions.

Like two human beings facing each other but speaking different languages, computers can talk to each other on compatible channels. But making sense — that is to say, executing the same programs — is another matter. It is possible to connect an Altair up to a Scelbi 8H product using an RS-232 interface. The link can be made and every bit of every byte sent (for example) from the Altair will be received and digested by the Scelbi. But unless the machines have common referents and means of translation, the information sent will be no better than "noise" — just as a foreign language perceived by a person conveys no more meaning than an arbitrary sound until the language is learned. Having a working communications channel does not guarantee that the communications sent will mean anything.

Levels of Intelligence

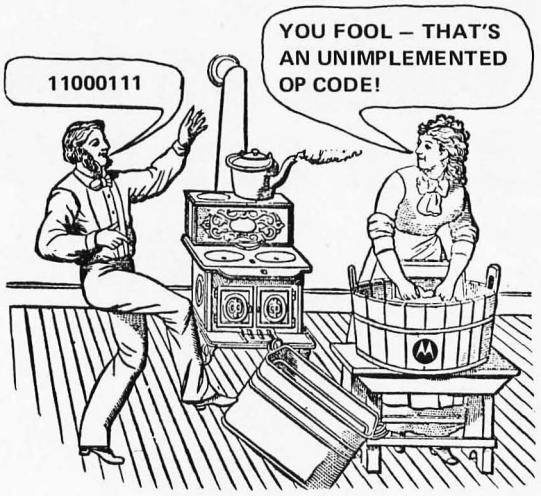
Given two or more different computers and compatible electrical interfaces between them, the simplest and easiest form of common understanding is at the level of raw data to be processed by the different computers. A binary number is a binary number independent of the machine for which it was generated. The fact that the bit string 11000111 means "load accumulator from memory" in the machine language of an 8008 and means "You fool — that's an unimplemented op code" for the Motorola 6800 is an accident of hardware design at two different companies. In either case, as data, the binary number 11000111 means an integer value of 199. Similarly, an ASCII character is an ASCII character independent of the place and time of its creation. Given the decision to use ASCII for communications — or binary numbers, for that matter — both parties to the communication can talk characters or numbers. These simple data formats provide an easily implemented link between computers which

does not require a large amount of software intelligence on the part of receiving and sending computers. It does not matter whether such a link is in the form of tape cassettes sent in the mail or a direct RS-232 connection when all the neighborhood hackers get together for a multiprocessor powwow and computerized crap game. ASCII and binary numbers can be shuffled back and forth with the assurance that, at this data level of coding, the information will be understood by both parties.

The communication represented by binary numbers or ASCII encoded data is not at all what might be called *true understanding* between the two computers involved. Sending data is a first step, but it is by no means the ultimate. The significance of the communication at this level must be determined by the *human beings* who manipulate the data being sent. There is no direct way of affecting the understanding – the programming – of the computer which is dutifully receiving the ASCII or binary data in this simple fashion.

Simple transmission of data across a parallel-serial-parallel or parallel-parallel interface enables the *users of the computers* to talk to one another, but the computers which carry out the exchange "couldn't care less." The computer in this type of an exchange is simply serving as a "dumb" transmission channel.

BASIC is an adequate language – but it has its drawbacks.



Borrowing again from the analogy to human beings, this *data level* communication between computers might be compared to the non-verbal emotional forms of interpersonal exchange. Common languages and verbal understanding are not required for humans to exchange emotional states, using music, facial gestures and other body motions which are inherent in the nature of the beast. But to exchange knowledge and practical data humans must speak a common language; it is not an accident that verbal activity and the tools of language are so much a part of the dominant species on this planet. If merely sending data represents a low level of communications intelligence between computers, what does it mean to transmit a higher level of intelligence?

Program Level Intelligence

To transmit data between computers is the first step toward a higher level of communications among diverse types of computers. Every computer on the market can handle 7-bit ASCII and other forms of information encoded into 8-bit bytes. (Of course, it may be easier to program ASCII data manipulations on some machines than on others.) Having a computer which can easily be *programmed* (possibly with no human intervention) as a result of an ASCII exchange with a computer of a different internal architecture is the essence of this next level of intelligence in intercomputer communications. In such *program level* exchanges the computers are speaking to each other in the form of *abstract program representations* which can be automatically translated into specific machine language representations for execution.

In the previous analogy, this is like human beings exchanging information and thought in a commonly understood language – for example, English. Each person who understands the language has incorporated within his mind a "translator" which creates an internal understanding based upon what was heard. The result of this translation – which must be consciously performed – is an understanding of the message which can be used as a basis for further thoughts. Emotional data is a much more directly perceived input (although it may of course be colored by thoughts). But verbal inputs require cogitation and analysis before they can be used and integrated.

The Goal: Exchange of Programs

The goal of *program level* interchange between computers is thus the ability to communicate understandable and potentially executable *programs* between computers of different design. When this goal is achieved it will be possible for a reader in one corner of the technological world – for instance an 8080 user – to develop a neat little utility program which can be sent to a friend in another corner of the technological world who has just completed a different processor such as a PACE machine. Since the program is recorded and communicated using the *program level* techniques, the recipient need only read the ASCII representation from the communications channel and process this data with a suitable "translator" in order to obtain a new executable program for a different machine design.

This program level of exchange is a well known technique which has been developed very thoroughly over the past 15 years after computer science left its formative years of the 1950's. It is the technique of *high level languages* and *compilers*. The *language* is the machine independent *notation* for the programs which are to be exchanged. The *compiler* is the computer program which carries out the translation. (Variations on this technique of course exist; for instance some languages like BASIC and FOCAL rarely have compilers, but typically use "interpreters" instead to compile, then execute statements one by one.) In the computer world at large, of course, there is no unanimity about choices of languages – and there no

doubt will be considerable variation in program representation philosophies in this personal microcomputer field. Be that as it may, exchanges at the program level are needed and computer languages/compilers are the technique for accomplishing such exchanges with *minimum* machine dependence.

So that's the story behind the need for a good 8-bit high order language. The home brew computing field is much more extensive than the confines of just one computer architecture, and the technological problem of passing 8-bit bytes all over the place is not at all impossible. The need is there, but can it be satisfied?

What Exists Now?

What currently exists in the way of high level languages for the field of home brew computers is limited. Currently available is only one language — BASIC — which to a certain extent satisfies the need for a *good* language. BASIC now exists for the MITS Altair, and will soon be offered by several other manufacturers. As such it is the only high level language which both exists and will (hopefully) run the same programs on any one of these small computer systems. As a high level language, BASIC is adequate but it has a few drawbacks:

— Descriptive names of variables are impossible with single character identifiers.

— Only a primitive GOSUB/RETURN facility exists for subroutine linkage, and parameter passing is not built into the language.

— The language BASIC is interpretatively executed, which means that each statement is "compiled on the fly" and executed whenever it is encountered. (Pre-scanning of programs and reduction of the source text is sometimes done, however.) An interpreter is necessarily slower than an equivalent compiler's object code.

— BASIC is missing the more advanced software tools such as the IF-THEN-ELSE construct, and statement grouping constructs, like the PL/1 DO . . . END block.

— Only line numbers may be used to label places in the program.

Now don't make the mistake of concluding from this criticism that BASIC is useless. Far from it. Any high level language which works as well as BASIC is better than none at all in the majority of programming circumstances. This is because for most problems the minute details of execution are unimportant, provided that certain functional building blocks of software (provided by the higher order language) are available to use.

The BASIC language has been used as a tool for initially teaching computer programming concepts, and has done so from its inception in the early 1960's at Dartmouth. There are also innumerable tutorial books about BASIC, due to its widespread use in the educational field. It is certainly the case that in most implementations BASIC is a quick and conversational way to write simple programs at a terminal. The criticisms have to do with features in contemporary computer language technology which are not present in BASIC, but which are extremely useful when writing programs.

An Alternative to BASIC

Criticism without giving an alternative is an empty activity. The purpose of criticism is to find a way to a better approach. So what language exists, can be dreamed up, or adapted to the small systems context — and provides a better alternative to BASIC? At present there is one language which was expressly designed for systems programming and applications programming for microprocessors. This language is called PL/M, which is a registered trademark of the Intel Corporation. The origins of PL/M can be traced back to a book published in 1970 by three compiler specialists, W. M. McKeeman, J. J. Horning and D. B. Wortman called *A Compiler Generator* (Prentice-Hall, Englewood Cliffs NJ).

XPL is a subset of the IBM language PL/1. The XPL subset is designed to eliminate many extraneous bells and whistles from PL/1, retaining only those features most needed for writing compiler programs: Simple character string and binary data, manipulations of such data, and a block structured procedure oriented language. Another design criterion of XPL is that it had to be a simple language so that its compilers could easily generate efficient object programs without burning up incredible amounts of computer time. The authors of the language and the book describing it succeeded well, producing a powerful language design system which has been used in a number of large projects.

In most programming circumstances, any high level language is better than no high level language at all.

Now, as it turns out, the features which are in XPL are in many respects the features

BOMB: BYTE's Ongoing Monitor Box

BYTE would like to know how readers evaluate the efforts of the authors whose blood, sweat, twisted typewriter keys, smoking ICs and esoteric software abstractions are reflected in these pages. BYTE will pay a $50 bonus to the author who receives the most points in this survey each month. The following rules apply:

1. *Articles you like most get 10 points, articles you like least get 0 (or negative) points — with intermediate values according to your personal scale of preferences.*
2. *Use the numbers 0 to 10 for your ratings, integers only.*
3. *Be honest. Can all the articles really be 0 or 10? Try to give a preference scale with different values for each author.*
4. *No ballot box stuffing: Only one entry per reader!*

Fill out your ratings, and return it as promptly as possible along with your reader service requests and survey answers. Do you like an author's approach to writing in BYTE? Let him know by giving him a crack at the bonus through your vote.

Page No.	Article	LIKED LEAST										BEST
12	Ryland: Software Vacuum	0	1	2	3	4	5	6	7	8	9	10
20	Burr: Logic Probes	0	1	2	3	4	5	6	7	8	9	10
26	Baker-Errico: Test Clip	0	1	2	3	4	5	6	7	8	9	10
30	Peshka: Characters	0	1	2	3	4	5	6	7	8	9	10
48	Helmers: LIFE Line 3	0	1	2	3	4	5	6	7	8	9	10
58	Browning: Flip Flops	0	1	2	3	4	5	6	7	8	9	10
64	Lancaster: ROM Technology	0	1	2	3	4	5	6	7	8	9	10
70	Nelson: HP-65	0	1	2	3	4	5	6	7	8	9	10
72	Kay: Build a 6800	0	1	2	3	4	5	6	7	8	9	10
78	Zarrella: Altair 8800	0	1	2	3	4	5	6	7	8	9	10
82	Hogenson: Tell Time	0	1	2	3	4	5	6	7	8	9	10
94	Helmers: Photo Notes	0	1	2	3	4	5	6	7	8	9	10

PL/M is becoming an industry standard language: The computer language equivalent of a "black box" integrated circuit. BYTE would like to see a PL/M compiler adapted to the home brew computer context.

which are desirable for a programming language used with microcomputers for both applications and systems programming. XPL is not too far removed from assembly language and becomes very handy as a way to generate large programs without getting bogged down in details. This fact makes XPL a language of far more utility than a mere compiler writing tool.

When the time came for Intel to commission a high order language for programming of their microcomputers, the XPL language and compiler had been proven in several years of practical use by several compiler writing organizations. Its practicality as a systems programming tool no doubt resulted in the use of XPL as a model for the new PL/M language. PL/M is effectively an adaptation of XPL to the context of a microcomputer with 8-bit data quanta and 16-bit addressing. The result is a language which looks very much like XPL – with a few keyword substitutions and additional features. This resemblance is sufficiently close that at least one version of PL/M has been implemented simply by modifying a working XPL compiler, although Intel's original was implemented in FORTRAN.

PL/M as a language possesses many desirable attributes which are not found in BASIC. These attributes include the PL/1-like statements and statement groups, long descriptive names for variables and labels, block structure, and subroutine linkages with parameters. As a systems programming language for microcomputers, the PL/M language adopts some of the features of an absolute assembler – there are location counters for program code and data which can be set during a compilation. To top it all off, the PL/M language is a relatively simple one which can potentially be self-compiled upon a small (but not minimal) home brew system.

At the time of this writing, PL/M is fast on its way to becoming an industry standard. It is definitely a language which has the potential for adaptation to the software requirements of the more advanced programmers in BYTE's readership – yet at the same time it is simple enough for the novice to understand. At the present time, however, only *cross compilers* – large programs running on big machines – are available for PL/M. There are PL/M versions currently in the works or producing code for the 8080, the 6800 and PACE microcomputers – but all are cross compilers. These cross compiler versions are widely used via time sharing networks by a variety of industrial and commercial users of microprocessors. This acceptance indicates that PL/M is a language which is likely to be around for some time.

A Call For Compilers

So PL/M is the tool which the industrial and commercial world uses for efficient code generation with a high level language for microprocessors. Will this technology be made available in the home brew computer markets? Yes. One reason for writing this editorial is to point out the existence of PL/M and direct a few BYTE readers to appropriate sources of information. In future issues, BYTE will be getting into more of the details of PL/M as a language. Until then, the accompanying list of information sources will have to suffice.

A second reason for this editorial is to serve as a *call for compilers.* What is needed is a compiler for PL/M *or a similar language* which will run on a typical 16K (RAM) 8-bit microcomputer using as many as three serial I/O devices for multiple passes through the data of a source program. Ultimately there should be one such self-compiler program for each of the major microcomputer chip architectures. The compilers should be written with system design flexibility in mind (in other words, modularity throughout and isolation of hardware-dependent portions to specific modules). Who will be the first person, club or firm to provide such a self-compiler? ■

Information Sources

PL/M:

8008 and 8080 PL/M Programming Manual, MCS-451-0275-10K

Intel Corporation

3065 Bowers Ave., Santa Clara CA 95051

This describes 8008/8080 PL/M as originated by Intel.

PL/M6800:

PL/M6800 Programmers Reference and *PL/M6800 Language Specification*

Intermetrics, Inc.

701 Concord Ave.

Cambridge MA 02138

These manuals describe the version of PL/M being marketed for the 6800 processor.

As this issue goes to press, National Semiconductor has announced a version of PL/M called "PL/M+" for the PACE system. Further details will be provided by BYTE as they become available.

DIAGNOSTICS: Documentation of bugs in previous BYTEs.

BYTE #2, p. 54, Fig. 3. An inverter (e.g., 1/6 7404, or 1/4 7400) should be inserted between the CE inputs of the 7489 circuits and the 7400 which drives them in the original drawing. Thanks to Martin E. Haring, Edison NJ and several other readers for pointing this out.

Dan Clarke (105 Fir Court, Fredericton NB, Canada E3A 2E9) notes that the originate modem transmit frequency definitions (Fig. 14 and text of "Serial Interface"), page 35, BYTE #1, are incorrect. Using the Motorola *M6800 Microprocessor Applications Manual* page 3-32 as a source of data, the following table should correct the matter:

Mode	Data	Transmit Freq.	Receive Freq.
Originate	Mark	1270 Hz	2225 Hz
Originate	Space	1070 Hz	2025 Hz
Answer	Mark	2225 Hz	1270 Hz
Answer	Space	2025 Hz	1070 Hz