eForth and Zen

Dr. C. H. Ting
Second Edition

Ofeete Enterprises

2013

eForth and Zen

Contents

1. Forth and Zen

Forth is often mentioned not only as a computer language but also a religion, because of its feverish followers. Among the religions, Zen is considered to be the closest to Forth. As popularly known, Zen is understood as a synonym of simplicity, brevity, light, understanding, wisdom and enlightenment. These are also attributes to Forth as a language and a philosophy. Indeed, Zen and Forth also bear striking similarity in their historical development and evolution. This aspect of similarity between Forth and Zen has not been well documented, and this is one purpose of this paper.

Zen and Forth both started as revolutions towards well entrenched establishments--the priesthood.

Zen and Forth both started as oral traditions because of the lack in supporting literature and the high concentration of expertise.

Zen and Forth both stressed that their matters of subject were simple and straightforward. However, it was in the interests of the entrenched establishment to make things complicated and inaccessible to common folks.

Zen and Forth both stressed that the true enlightenment and understanding were within the grasp of individuals. The rituals and the accepted practices exercised in the establishments had nothing to do with these goals.

Here I would like to compare Zen and Forth in greater details so that we all have a better appreciation of these two seemingly unrelated topics. Forth programmers may be encouraged to press on using this language with a higher hope that their work will be recognized by their peers in the future.

Comparing Forth and Zen

The most important contributions of Huineng, the Sixth Patriarch of Zen in China, was that he had his lectures recorded by his disciples. He also had the lectures printed and distributed as identifications of discipleship to his teach and his philosophy. In striking contrast to the Buddhist Sutras translated from Sanskrit, the lectures were plain, easy to read and to comprehend. This collection of lectures was the only book written by a Chinese to be granted the status of a Sutra in the Buddhist literature, and is commonly call the Platform Sutra. (Sutras were the teaching of Buddha himself.)

People compared Forth and Zen in a very superficial manner, mostly without the understanding of either. I feel that there is a need to treat both subjects fully in a single treatise. There is no better way than to present the very original text of Zen and the source code of Forth side by side. In the course of preparing the complete documentation for eForth, I thought it would be very useful to lay down the eForth text in parallel with the original text of the Platform Sutra. Of course, it is impossible to correlate the text of eForth directly with the text of the Platform Sutra. However, one should be able to see the common threads in them, as the themes of simplicity, personal understanding, enlightenment, and the struggles against prevailing established doctrines recurring time and again in both texts.

As I intended this book for both English and Chinese readers, I would like to include texts and discussions in both English and Chinese. In effect, we would have four threads running in parallel: the original Platform Sutra in Chinese, its English translation, the text of eForth, and its Chinese translation. This is how this book is laid out.

Historically, Zen was a grand synthesis, combining the essence of Buddhism, Confucianism, and Taoism after about one thousand years of inter fertilization. It was the results of the many Chinese minds, struggling for emancipation after a thousand years of conflicts between the traditional humanistic Confucianism, the nihilistic Taoism and the imported anti-materialistic Buddhism. It laid dormant for a hundred years, surviving through five generations of Zen masters, passing the doctrine orally from heart to heart. Eventually, during the reign of the Sixth Patriarch, Huineng (638-714 AD), it blossomed in full and became the dominating religious philosophy in China ever since. Its influence spread into Japan, Korea, and Southeast Asia. Lately, it also became fashionable in Western Europe and America.

The history of Forth is too short for meaningful comparison with that of Zen. It was virtually unknown to the world in its first decade of existence until the late 70's. It was invented by a lone programmer, Charles H. Moore, outside of the main stream of computer industry and computer sciences. In the early 70's, it was only used in astronomy, as he helped programming minicomputers to automate telescopes and observatories. Then it blossomed with microcomputer revolution, promoted by Forth Interest Group in the early 1980's. Since 1985, the Forth Interest Group has been in steady decline, as C became the dominant programming language. The advantages of Forth, such as elegance in its architecture, simplicity in its syntactic construction, and economy in memory utilization, seem irrelevant in the age of mega-resources, where MIPS, megabytes of RAM memory, gigabytes of disk storage are commodities easily affordable.

As the operating systems and applications grow to fill the available RAM and disk storage, at some point people will ask the question whether these huge programs are worthy of the resources they consume. People will have to ask whether the direction we are heading will lead us to better lives and better environment. When we stop equating bigger to better, and more to happier, then we can re-evaluate the computer technology in a new light. Then, maybe Forth will shine again.

Zen as an Oral Tradition

Buddhism was founded by Gautama Siddhartha, a religious philosopher and teacher who lived in India (~560-480 B.C.) He was called Buddha which means the enlightened one. He attracted a large following. His teaching in essence was that one could reach

Nirvana, a divine state of release from earthly pain, sorrow, and desire, by the right living, the right thinking and self denial. He left no writing behind, while Buddhism flourished in India for a long time.

In the two hundred years after Buddha's death, many schools formed after different personalities and there were great arguments and debates concerning what were his true teaching. Great conventions were held to debate the issues and codified his teachings as Sutras in Sanskrit.

The earlier dominant school was Hinayana, which spread to the south and is still flourishing in Sri Lanka, Burma, and Thailand. The later dominant school was Mahayana, which spread into China, Tibet, Korea, and Japan.

The Hinayana School emphasize the mystic power of Buddha and the personal salvation through one's own efforts. The Mahayana School emphasized eclecticism and in common search for salvation.

After the introduction of Mahayana into China in 200 AD, it arose great interests in the intellectuals as well as the peasants. Many emperors and their courts were converted and spent great efforts in building temples and spreading the Buddhism. A continuing effort over three hundred years was devoted to translate Sanskrit Sutra and related literature into Chinese. In Tang Dynasty (618-907 AD), more than 5000 volumes of Buddhist literature were translated and assembled.

Most of the Sutra translations were done poorly and required a priesthood for the interpretation and dissimulation. The vast amount of literature also caused sectarian divisions and arguments among the priesthood, continuing the Hindu tradition.

Zen was introduced into China by a legendary Indian monk Buddhidharma in 527 AD. He stayed at the Shaolin Temple for 9 years, spending all his time meditating in front of a stone wall. He was known as the 'Indian Monk Looking at a Wall'. He didn't use any Sutra, and he didn't write anything. He taught a few students and encouraged them to find enlightenment in themselves. His teaching was summarized as:

My teaching is outside Buddhist tradition,
As truth cannot be conveyed by writings;
Cleanse your mind to reveal your true nature;
One can reach Nirvana directly.

Buddhidharma passed his garment and bowl to his student Huiko as evidence of the discipleship and commanded him to do the same for five generations. He then returned to India. Huiko passed the teach with the garment and the bowl to Sengtsan. Sengtsan passed them to Taohsin. Taohsin passed them to Hungjen. Finally, Hungjen passed them to Huineng (638-714 AD).

For a hundred years, Zen was passed from mouth to mouth, and from heart to heart.

Very few people knew of its existence. Even fewer knew its philosophy and teachings. In China, Buddhism flourished when supported by the emperors and by high officers. A number of times the Buddhism was almost completely destroyed when the country was in turmoil and when the Confucian officers could convince the emperor that Buddhists were threats to the state. All the while, the Zen masters orally passed their teachings from one generation to the next.

Forth as an Oral Tradition

Forth was invented by Charles (Chuck) H. Moore who was trained as a physicist in MIT but wandered into programming.

In early days, he built an interpreter so that it would execute words on punched cards. Later he found that these words could be more conveniently compiled into lists, which could be executed by the computer more conveniently. The interpreter with very small modification, could be made to compile anything and everything, and the whole scheme evolved into a programming language. It was named Forth, as abbreviated from Fourth, meaning the fourth generation of programming language when the third generation of computers bases on integrated circuits were becoming prevailing in the computing industry.

Very early in the development of Forth, a state of closure was reached. Chuck was able to generate new Forth systems from an existing Forth system through meta-compilation. He did not need other programming tools to build new Forth systems, and Forth started to evolve independent of the existing operating systems and programming languages. This state of closure was very interesting, like the ying-yang cycle. One could not find an entry point once the cycle was closed. In that Chuck had the monopoly on Forth, because very few people possessed the understanding to cut in the cycle in order to build new Forth systems. He felt quite secure in giving users the complete source listings, fairly sure of that nobody could reversed engineered the Forth technology, even though the source listings were complete and truthful.

Indeed, the source code was very difficult to read, because a Forth system was generated by the meta-compiler, and the meta-compiler was written in Forth. To understand Forth, one had to understand the meta-compiler. To understand the meta-compiler, one had to understand Forth completely. Where do you start?

Forth thus became a legend. The astronomers loved it so much that they made it the standard language for observatory automation. It was fairly easy to use but very difficult to understand. The source code traveled to the far corners of the world with the telescopes, but the knowledge and understanding of Forth was only passed from mouth to mouth and heart to heart. Hence Forth became an oral tradition these days. Forth code tended to be concise and often packed tightly in blocks. In-line documentation and comments were deemed too expensive, and most code was poorly commented. Forth thus acquired the reputation of a write-only language.

Several manuals were circulated among the observatories, documenting a few of the most popular Forth implementations. These manuals mostly contained a short section introducing Forth and discussing how to use that particular Forth system, and a long dictionary documenting what each word did. These manuals told the users what Forth was, but provided very little help as to how Forth worked.

Acceptance of Zen

Huineng, the Sixth Patriarch, was a genius. He couldn't read because he was borne poor and gathered wood for a living, but he could explain the Sutras when people read them to him. He went to learn from the Fifth Patriarch Hungjen, and Hungjen sent him to labor in the kitchen. As Hungjen got old and wanted to pass on the garment and the bowl, he asked his students to write poems to show him their understanding of the enlightenment. His best student Shenhsiu wrote the following poem:

Our body is the bodhi tree,
And our mind a mirror bright.
Carefully we wipe them hour by hour,
And let no dust alight.

Hearing this poem, Huineng asked a scholar to write down his own poem, because he couldn't write himself:

There is no bodhi tree,
Nor stand of a mirror bright.
Since all is void,
Where can the dust alight?

When Hungjen saw this poem, he passed the garment and the bowl to Huineng and told Huineng: "You are the one Buddhidharma prophesied. Zen will flourish in China through you. Take the garment and the bowl to be the Sixth Patriarch, but do not pass them on any more.'" Hungjen was in such a hurry to pass things to Huineng that he didn't even shave Huineng's hair (to admit him to Buddhist order), as Huineng was still a layman.

About 20 years later, when Huineng was well established as the Master of Zen, he was asked by a Provincial Officer to give lectures on Zen. The Officer had Huineng's eldest student Fahai recorded his lectures and had the lectures printed as the 'Platform Sutra, Lectures by the Six Patriarch'. When Huineng was about to die, Fahai asked him: "What are you going to do with the garment and the bowl? Who's going the inherit them?" Huineng said: "As commanded by Buddhidharma, the garment and the bowl will not be passed on. But now you have the Platform Sutra. Go forth and teach others according to this Sutra. Everything I learned I put down in it. When you read it, it is as if you am talking to me."

20 years after Huineng died, the Northern School was favored by the royal court and

dominated the Buddhist landscape. The Southern School was scattered and mute in Southern China. One of Huineng's student, Shenhui (686-760 AD), went to the capital and challenged the doctrine and the Zen inheritance of the Northern School in a series of lectures and public debates. He convinced the court and the public of the historical significance of Huineng and established the Platform Sutra as the orthodox doctrine of Zen Buddhism.

Acceptance of Forth

The major breakthrough in the Forth arena was due to the Forth Interest Group, which was founded by Bill Ragsdale in 1978 in the Silicon Valley. The most important contribution FIG made was to reverse-engineer a Forth system from ground zero, thus breaking the infinite ying-yang cycle. FIG organized a Forth Implementation Team which built and released 6 Forth implementations for the 6 then most popular microprocessors based on the figForth model. These implementations were written in assembly code of the native microprocessors. People who were familiar with the assembly code could then easily implement figForth on their own microcomputers. figForth thus trained a new generation of Forth programmers outside the Forth oral tradition.

There was a host of Forth literature appearing in the early 1980's, which further helped the popularization of Forth among the personal computer users. Among them was Leo Brodie's 'Starting Forth' , 'Thinking Forth', and the 1979 Special Forth Language Issue in the Byte magazine. 'Forth Dimensions' from FIG and 'Journal of Forth Applications and Research' from the Forth Institute were the two major publications on Forth. These literature showed that Forth penetrated into many different scientific communities and technical industries.

My most important contribution to Forth was the publication of 'Systems Guide to figForth', first released in 1979. Instead of telling people what Forth did, it systematically explored how Forth did things and why things were done the ways they were. It put to rest the myth that Forth was a write-only language by showing that Forth could be understood by averaged user with somewhat casual studying. It showed how the inner interpreter and the outer interpreter worked, and why words and dictionary in Forth were constructed the ways they were by necessity. It proved that the understanding of Forth could be transmitted through paper medium without personal interaction. The impact of this work, I would like to believe myself, was similar to what Huineng caused with his Platform Sutra on Zen.

Simplicity in Zen

Buddhism is very complicated because it is not a monolithic system of thoughts and philosophy. It accumulated many centuries of cultural and philosophical development. The Sutras were all attributed to Buddha but were most likely written by people remotely associated with Buddha. Lots of the mystic Hinduism found their ways into Buddhism, which was inevitable because Buddhism was developed in the Hindu environment, like the 33 layers of heaven, 18 layer in hell, the reincarnation of all animals, etc.

There were many different theories about how life, death, and Nirvana. There were many sectors and schools about how one could attain Nirvana to avoid the reincarnation into a lower animal form. Thing got complicated and confusion reigned supreme. In essence, everybody just picked what he believed and convinced others that his was the best and most logical way to deal with life and all its ramifications.

The general consensus was that vegetarianism was good, giving to the temple was good, kindness to people and animal was good, reciting Sutra was good, meditation was good, worshipping Buddha and other Buddhist deities was good, dedicating to priesthood was good, etc. Could one attain Nirvana by doing all these? Maybe. Maybe not.

Zen was a great simplification of all these. Huineng maintained that Buddha hood and enlightenment could not be achieved through generally accepted Buddhist practices, like reciting Sutras, making offerings, meditation in special sitting positions. As everybody already had the Buddha nature in him, all he had to do is looking inward to find the true Buddha. Our senses and our thoughts tended to veil us from the Buddha nature and they should not be trusted. The process of Zen (Dhyana, Ch'an, meditation), was to reject the influences of senses and thoughts, and to arrive at a state of ideallessness, nonobjectivity, and nonattachment. In this state, nothing external of ourselves and within our own minds can influence us and dragged us back to the earthy existence.

Simplicity in Forth

The poem by Shenhsiu and the poem by Huineng provide the best contrast for us to compare the conventional wisdom in the current computer industry against the Forth philosophy. Let me paraphrase the poems to show my point. From the mainstream of the computer science, one will advice our youngster:

Hardware is complicated,
Software even more so.
Study hard day and night,
Maybe you can be productive.

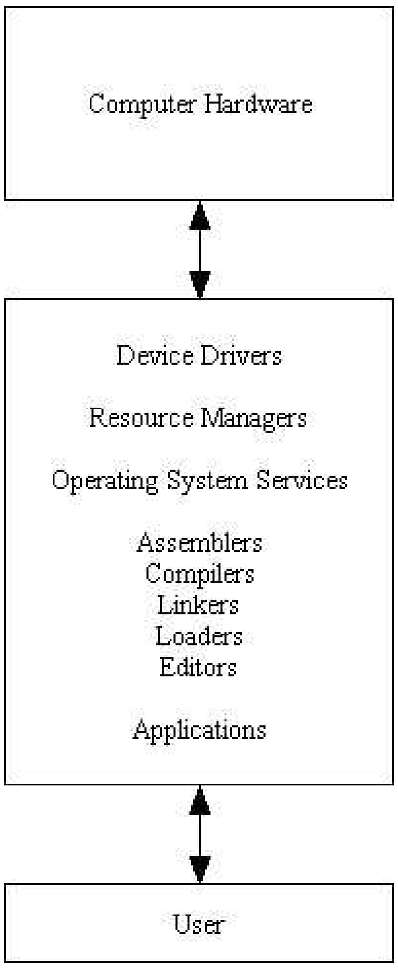
From the point of view of Forth, we might say:

Hardware is the reality,
Software but an illusion.
Learn your Forth well,
And you can conquer them both.

Computer hardware is difficult. Computer software is even more difficult. We have volumes and volumes of literature to prove them. Hardware and software are difficult, only because people are not given the right tools to deal with the complexity in these systems. If we insist on asking whether the complexity is necessary, we can convince ourselves that they should not be complicated. The computer hardware evolved trying to solve the perceived software problems. The software evolved trying to solve the perceived hardware problems and the problems in the human interface. If these preconceived problems do not exist at all, the hardware can be simple and powerful. The software can be simple and powerful as well.

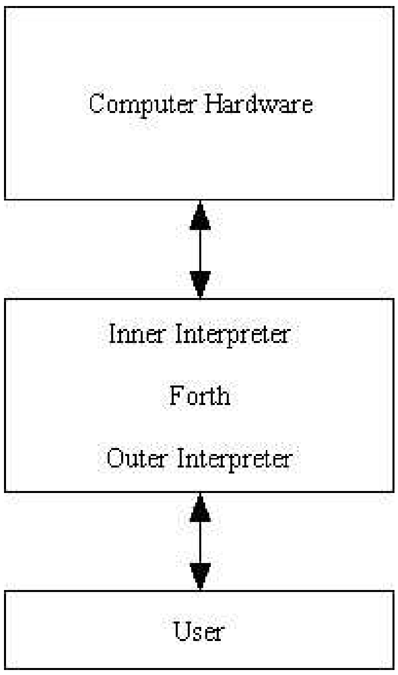
In a typical computer system, there are layers upon layers of software between the user and the computer hardware. The operating system and its utilities, the compilers, assemblers, editors, linkers and loaders are all very complicated and mostly proprietary programs. They helped the user to start his journey into this computer juggle. After a while, they tend to hinder the users progress, because they insulate the user from the hardware, and deliberate efforts were expended to prevent the user to fully make use of the capabilities built in the hardware.

This issue of simplicity can be illustrated in the following diagram. The operating systems and the applications separate the user and the computer hardware. The software protects the hardware, because the user is not to be trusted. Leaving to the user, he will abuse the hardware and causes the system to crash. In the days of mainframe computers, the greatest sin was to crash the computer, because the livelihood of the computer priesthood depended upon the continuing operation of the computer. The hardware and the operating system had to be protected at all costs.



In this age of personal computers, the priority is turned by 180 degrees. The user owns the computer. He is responsible for its operation. Is it necessary to protect the computer against its owner?

Forth provides a much simplified interface between the user and his computer as show in the following diagram. Forth is a simple and integrated interface between the user and his computer.



Through Forth, the use can directly control the computer hardware, because every part of the computer system is freely accessible to the user. Therefore, the user can explore the best way to use the computer system to suit his applications.

With the freedom to access computer hardware comes the responsibility to use the hardware properly. The user may crash the system frequently. It will cause no harm as long as the computer can recover quickly from the crashes with minimal damages to the data stored in the computer system. After all, the user owns the computer.

There are other examples which show that complexity does not necessarily mean performance. Simple systems are generally faster and more resilient.

Registers in the CPU are designed to hold temporary data so that the CPU does not have to go to the slow memory to fetch and store data. However, large number of registers become a burden when you call subroutines, and the registers must be saved before the call and restored after the return. In order to speed up a computer, you tend to have more and more registers and make less and less subroutine calls. If we recognize that in high level languages and in structure programming the subroutine is the most important mechanism, we should instead optimize the memory access in the subroutine call-return and put all the intermediate data on the data stack.

Another example is the prefix arithmetical notation prevailing in conventional programming languages. The prefix notation is unnatural and was forced upon all young minds in their algebra lessons. It is much more natural for both computers and humans to think in linear lists, sequentially executed. Thinking algebraically, you need two-pass compilers to break the equations down into pieces and reassemble them for the computer to execute. Thinking in the postfix terms, a simple one-pass interpreter can be drafted to perform all the functions required of a complicated compiler.

Enlightenment in Zen

Zen was the cumulative synthesis of the Buddhist philosophy and the traditional Chinese Confucianism and Taoism. Zen was also a revolution against the Buddhist traditions and establishments. It discredited the Buddhist practices, which emphasized ceremonies and outer appearances, while claimed that the enlightenment exists only in the minds of individuals.

A huge amount of Buddhist literature had been translated from Sanskrit to Chinese. Because the translations were difficult to understand, a priesthood was established for its dissimulation and interpretation. Towards this literal tradition, the Zen masters proclaimed that enlightenment could not be transmitted by written words, but had to be handed done orally from heart to heart.

In traditional Buddhist theories, it was very difficult to attain Nirvana or Buddhahood. It required a long time of studying, and the practice of self-denial. In the end, there was still no assurance that one could attain it. Even if one attained it in this life, there would be the possibility of losing it in the next life. There were external forces which we could not know and we could not avoid.

Zen placed the possibility and the capability to attain enlightenment and Buddhahood squarely in the individual, by declaring that the Buddha nature is part of the human nature and it exists in everybody. The Buddha nature is vile and corrupted by the worldly desires and thoughts. These desires and thoughts can be purged, the individual can thus be enlightened, and his own Buddha nature can reveal itself. The enlightenment is the realization of this self-sufficient Buddha nature.

As to how one became enlightened, there were two major schools of thoughts. Huineng insisted that enlightenment came suddenly and Senghsui maintained that it ought to be the results of diligent study, mediation and searching. These were the Southern Sudden School and the Northern Gradual School. However, as Huineng maintained, how enlightenment is achieved is not important. The important thing is its realization. People are all different, and they are enlightened in different ways. A master can teach, but he can not enlighten. The enlightenment comes from within. The best a master can do is to inspire, to help, to lead the way, and maybe to strike a sharp blow on the head at the right time.

Enlightenment in Forth

What is the enlightenment in Forth? I think it is the complete understanding of a computer in terms of its operations and its interface to the user. This understanding is not as complicated as we have all being lead to believe. It involves two components of Forth: the inner interpreter which Forth imposes on the computer hardware to execute Forth lists, and the outer interpreter which executes words typed in by the user. If one understands both the inner interpreter and the outer interpreter, he has the complete understanding of computer, in the sense that he can go out and build a Forth system on any computer and make that computer do what he wants it to do. He then will realize that operating systems and languages enslave people to do what the system permits them to do, while Forth gives them freedom to tell the computer to do what they want to do. This is enlightenment.

Operating systems and programming languages are designed to enslave the users, by their sheer sizes and their complexities. They are too complicated to be understood by individual users. Forth shows that an operating system and a high level language do not have to be complicated. In fact, they can be very simple and can be mastered by individuals with some limited efforts. Once the principles are mastered, they can be applied to all computers. Then, the user can become the master, and the computers become obedient but powerful slaves. To be the master of powerful slaves is very enlightening and satisfying. Once you taste the freedom and the satisfaction of being the master, you will not want to be enslaved again by a computer through its operating systems and programming languages.

2. eForth Model

eForth is a series of Forth implementations based on a very small and simple model developed in the Silicon Valley Chapter of the Forth Interest Group, in the San Francisco area. Many people inside this chapter and all over the world had made significant contributions to it. The first model was released in 1991. As it stands now, there are 14 different implementations of this model on various microprocessors, including 8086, 80386, Z80, 8096/98, 8051, 68000, 68332, 68HC11, PIC17C42, Transputer, and the experimental P21.

The basic model used the direct-threaded code to implement the inner interpreter, but there are several versions of subroutine threaded code around. The basic model used the Microsoft MASM as the software platform, but there are many versions using the native assemblers of the targeted microprocessor. The basic model came with a minimum set of utilities, but there are several versions highly optimized and with lots of utilities.

With eForth we hoped to achieve two goals: one is to encourage people who do not have any prior Forth experience to learn Forth and evaluate it for possible applications. The other was to enable experienced Forth programmer to port this model to newer and more powerful microprocessors. It seems that we have made good progress in both directions.

It is amazing that eForth had gained that much ground without a detailed and complete documentation. I hope that by going through the source code of the 8086 implementation line by line in this book, users can understand this system better and can make better use of this model in their applications.

Origin of eForth

eForth is the name of a Forth model designed to be simple so that it can be easily understood by programmers with some elementary knowledge about assembly languages and easily portable to a large number of newer and more powerful processors which are available now and are becoming available in the near future. Originally it was called pigForth. However, because of very strong objections to the term 'pig' among many experienced Forth programmers and users, a less provocative name 'eForth' was adopted after much heated debates at a beer party in the 1990 Rochester Forth Conference, held in June 1990, Rochester, New York.

PIG originally stood for Post-Forth Interest Group, a term used by Mr. Andreas Gopold, a Forth programmer from Germany, to encourage people to think the future of Forth, in light of the recent advancements in computer hardware and software technologies. His feeling was that Forth in its present shape and form will not be useful for the programmers in the future, because it does not provide enough support for program development, especially for large and very demanding projects. He thus felt that Forth has to develop in the direction as Tom Zimmer's FPC, Mitch Bradley's ForthMacs, and Leibnitz he developed. These systems are huge because they incorporate lots of tools and utilities useful in the programming processes.

I am stealing this acronym 'pig' from him for a different purpose. As we are marching into the nineties, we are confronting a host of new and very powerful microprocessors of different architecture and designs. The excitement is not the least less intense than that in the seventies when the first crop of 8 bit microprocessors made their appearance. Forth Interest Group captured the imagination of the first generation of personal computer users by releasing the figForth model riding on six different microprocessors. It is thus very interesting to see if we can build a new Forth which can be easily ported to many, if not all, of the new and future microprocessors. This is the original design goal of eForth.

eForth Model

eForth model defines a minimal Forth system which turns a computer into a system that the user can build applications and to test the applications interactively. At the topmost layer, the outer (text) interpreter accepts words from the user and executes them. In this layer, a rich set of utility words are available to allow the user to compile new words into the dictionary, decompile words in the dictionary, inspect and change memory, inspect and change the data stack, and download files through the terminal. These are functions a normal operating system provides to the user. In eForth, these functions are provided in one simple, elegant, and integrated program.

At the lowest layer, the eForth is built upon a set 31 primitive words which are coded in the native machine instructions. This set of primitive words was selected for portability, because we intended that the eForth model to be implemented on many different computers. Ideally, one could port eForth to a new computer by recoding only this set of primitive words, and the rest of the system would be ported over unmodified. This portability has been proven in many cases, although there were cases where we had to make minor changes to port it to a computer with strange architecture.

This set of primitive words imposes the virtual Forth engine on a real life computer and make it behave like an idealized Forth computer, with two stacks and a dictionary. By limiting the number of machine specific primitive words, we also impose a performance penalty on eForth, which does not take advantages of the resources designed into a real computer. However, it is expected that the user will improve its performance by replacing the most often used high level words with native machine instructions.

Changing Environment

In the seventies, personal computers were novelties, totally stand-alone systems that you put on your desk and had them do work for you. (Or, more precisely, you willingly became enslaved to them.) A personal computer, however, had to contain the programming environment to be useful. Thus one needed a CPU with memory, a keyboard, a display monitor, one or more disk drives, and an operating system including languages and utilities. A personal computer must be self-sufficient to be able to stand-alone.

figForth was originally written by William Ragsdale for an Apple II computer. Bill organized the Forth Interest Group in 1978. Among the first things he did was gathering a group of programmers to port it to other personal computers popular at the time.

figForth, as all other Forth systems at that time, had to be self-sufficient also. It had to be able to support the keyboard input, screen and printer output, and the disk drive for mass storage. An editor was always the first application a Forth user did, because that was the only way he could proceed to do serious programming.

figForth was adopted in many commercial Forth systems, which generally added an editor, an assembler, and some demonstration programs. It was enthusiastically accepted because it was the only high level language beside BASIC which could conformably reside in 16 Kbytes of RAM memory.

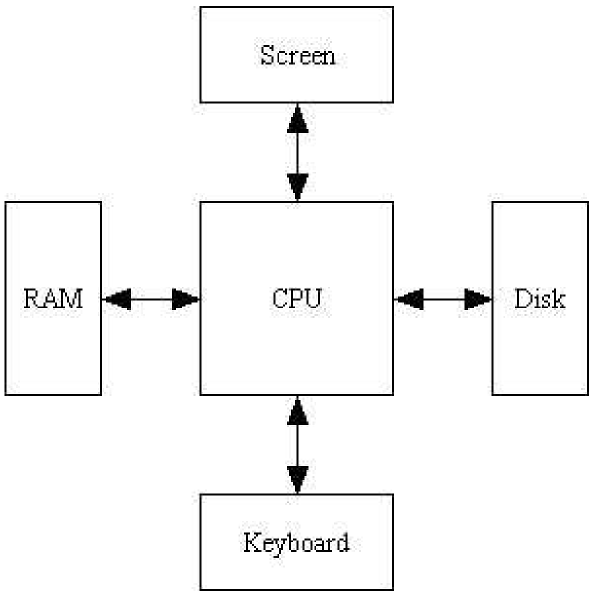
F83 was developed by Henry Laxen and Michael Perry in the early 1980's. It was a giant step forward from figForth, and much human engineering went into it to facilitate programming. It included a much better editor, more extensive debugging tools, and smoother interface to the operating system. Nevertheless, F83 was still an environment to itself, tailored to a single user using a single machine.

The largest change in the eighties was the separation of the processor from the programming environment, due to the proliferation of standard personal computers; i. e., IBM-PC and the clones. PC's become the default programming tool for other processors in separated boxes. I do not deny that there are still lots of programming done on the PC's, for the PC's. However, a very large portion of the programming activities for newer and more advanced processors are carried out on PC's separated from the target computers.

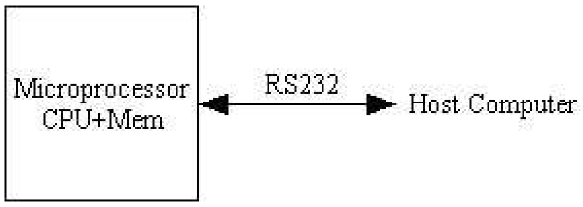
The separation of the programming environment from the target processor has a great impact on the design of eForth, which I presume will go into the target processor, not the host computer. It is also of great advantage to the design of eForth, because eForth does not have to include utilities supported by the host computer, and the eForth design requirements can be greatly simplified. In a sense, the eForth is addressing the needs of the embedded processors, the latest buzz word in the microprocessor world.

Universal Microcomputer

The microcomputer for which figForth was designed can be shown schematically here. It contained a CPU, some memory, and it had to talk to a keyboard, a monitor, and a disk.



The microcomputer eForth has to deal with is much simpler, as shown in the following. It is a CPU with some memory, perhaps with some peripherals which are yet to be specified. It is connected to the outside world through a RS232 cable. This RS232 line may not be used in real applications, but it is connected to a host computer for programming, testing and debugging. In certain applications, the RS232 umbilical cord may never be severed because then the host can download specific code to the target to perform specific tasks, depending on different circumstances. Many of the new generation of instruments are designed using this capability.



Universal Forth

As we have defined an Universal Microcomputer, it is logical that we can define and build a Universal Forth for it. There are many trade-offs that we have to consider in designing this Universal Forth, which we assume will be ported to many different CPU's. There are also many models of Forth which we can used to base our design on, taking advantages of the wisdom accumulated over the last twenty years in the Forth community.

There are three most important Forth models from which eForth derives its strength: figForth by Bill Ragsdale, cmForth by Chuck Moore, and the bForth by Bill Muench.

Here is a list of important features we liked to include in this Universal Forth:

1. A small set of kernel words, which is machine dependent. Only this set of words are rewritten for a specific CPU. A minimum kernel word set encourages the porting of eForth to new CPU's.
2. The high level definitions must be portable to all target CPU's, including 8, 16, and 32 bit machines.
3. The only I/O words are KEY, EMIT, and ?KEY, because the only I/O device in the target is the RS232 port.
5. Editor, file server, and other utilities are provided in the host system. Forth does not have to provide these services.
6. Assume that the host is an PC/MS-DOS system which is the lowest cost, and the most available platform for programming.
7. Source code is provided in the MASM assembly file to avoid problems in metacompilation..

The guiding principles for this Forth are easy to understand, easy to modify, and easy to port to other microprocessors. The letter 'e' in eForth thus stands for easy, educational, embedded, elegant, and maybe, evangelical.

DOS Implementation

The ideal of eForth was discussed extensively in the Silicon Valley FIG Chapter in the period of 1990-91. Various models were evaluated to decide which one was the best for its implementation. The available models considered including figForth, F83, cmForth, LaForth, ZenForth, Fcode, ANS Standard Forth, and bForth. The consensus favored bForth, whose principal author was Bill Muench. It had a very small core of machine specific word set, and had the most of the features desired in eForth. Bill distributed a preliminary specification based on bForth for the group to work on. I took the model and implemented it on an IBM-PC, in the MASM format. This implementation formed the basis of the eForth Model.

The eForth Model is targeted to the Intel 8086 processors under the DOS environment. It relies on the DOS to provide the serial I/O services. It is a fully functional Forth system which allows a user to exercise it and evaluate the functions of its word set.

Following are some of the special features in this eForth Model are:

1. 31 machine dependent words and 193 high level words.
2. Direct Threaded Code.
3. Separated name and code dictionaries.
4. All system variables are defined as user variables for ROMmability.
5. Vectored ?KEY, KEY, and EMIT.

6. File handling through the serial I/O interface.
7. CATCH-THROW error handler.
8. Only the single indexed FOR-NEXT loop is provided.
9. Track the proposed ANS Forth Standard.
10. Compile-only words are trapped in interpretive mode.
11. Tools include DUMP, WORDS, SEE and .S .
12. Flexibility in memory mapping.

Porting eForth

The eForth Model was designed for portability and every effort was made to facilitate the porting process. The eForth system is relative small wherein the code dictionary is 5 Kbytes long and the name dictionary is 2 Kbytes long. The following procedure is suggested for porting it to a new CPU.

1. Determine the memory map in the target system. Set the memory pointers in the EQU section properly to reflect the physical memory assignments in the target system; i.e., ROM, RAM, stacks, user area, code dictionary and name dictionary.

2. Study the words in the machine dependent kernel. Rewrite them in the assembler of the target CPU. If you have access to an assembler of the target CPU, use it to assemble this set of code words. If you do not have an assembler, hand assemble the code words. It seems difficult to do hand assembly, but you have only 31 words to worry about.

3. If you have tools to exercise the assembly code words, try to debug them.

4. The binary object from the assembler has to be entered into the eForth source code using DB and DW statements, which can be handled by MASM.

5. Assemble eForth source code by MASM to produce a binary object code.

6. Move the binary object code into your target system via EPROM's or other means.

7. Debug the target system.

The eForth Model assumes that the CPU can address bytes in memory. All the memory accessing words use byte addresses. If your CPU cannot address bytes, you have to synthesize a byte addressing space from the cell addressing space and provide a mechanism to translate byte addresses into cell addresses and vice versa. The eForth Model does enforce words alignment to the cell boundaries to facilitate the byte to cell address translation.

Hand assembling machine code is not the most pleasant task in programming. However, it is impossible to provide the eForth Model in all the assembler formats used by all the CPU manufacturers. Adopting MASM as the common source code processing environment establishes the largest common denominator for porting eForth to the largest number of CPU's.

After the generic eForth Model is ported to a target CPU, you might want to consider optimizing it to improve its performance. Following is a list of words which are primary candidates to be recoded in machine instructions:

1. Recode +, -, UM* and UM/MOD if you want to do number crunching.

2. Recode 'parse' and NUMBER? to speed up interpretation and compilation.

3. Recode doUSER and doVAR. Use CALL doUSE and CALL doVAR to replace the high level mechanisms in the eForth Model.

4. Recode 'find' to improve the dictionary searching.

5. Add ONLY-ALSO mechanism to use multiple vocabularies.

3. eForth Overview

Before diving directly into eForth, I would like to discuss the general principles of a virtual Forth engine and many other system design issues so that you have a better overall view of the eForth system when investigating the detailed structures and code in eForth.

I think the following list of topics are the most important in understanding Forth:

```
     Address interpreter
     Text interpreter
     Dual stack architecture
     User variables
     List of execution addresses
     Linked vocabulary
     Memory map
```

Using a real eForth implementation helps to make these topics clear. It is also interesting to note that these topics appear quite naturally in the beginning of the eForth MASM source listing as we set up the assembler before starting assembling any code.

In effect, it allows us to read the source code from the beginning to the end in its natural sequence. I had some reservation in that we might not be able to present the eForth system clearly without going back and forth over the source listing, because the logic of the system might not follow the loading order. In the end, it worked out perfectly.

Inner and Outer Interpreters

Like Prajna and Samadhi in Zen, the most important concepts in Forth can be summarized in two components, the inner interpreter and the outer interpreter. The inner interpreter controls the actions of the computer hardware to execute compiled Forth commands in the form of address lists. The outer interpreter is the user interface in Forth. It accepts commands from the user in the text form and executes these commands. Through the outer interpreter, the user can interactively control the entire computer system to do what he wants to accomplish.

A typical line of Forth commands is as follows:

```
'  ?TX  HERE OVER - DUMP
```

This line of commands dumps the entire eForth system in binary form on the screen. The meaning of the commands can be explained more fully as the following:

' ?TX ' is a Forth command, which searches the Forth command ?TX in memory and returns its address. ?TX happens to be the first Forth command in memory. HERE returns the address of the end of the last Forth command in memory. OVER - subtract the address of ?TX from the address returned by HERE. The result is the length of the memory which contains the entire Forth system. DUMP displays the contents of memory from ?TX to the last command in memory

The Forth outer interpreter reads the line of commands, parses out each command and execute them in the sequence by which the commands are entered by the user. The commands are separated by spaces so that the outer interpreter can easily pick up the commands one after the other and executes them in sequence.

Forth uses very simple syntax rules, which makes the outer interpreter extremely simple yet very powerful

A typical Forth program creates a new command in the memory which performs a sequence of existing commands. An example is:

```
: dumpCode ['] ?TX HERE OVER - DUMP ;
```

This new command dumpCode does exactly what the command line in the previous example did. It dumps the entire memory containing all the Forth commands. This new command is compiled and added to the memory. After dumpCode is so defined, the user can type: 'dumpCode' and all the actions in the command line will be executed.

The Forth command : changes the behavior of the outer interpreter so that the outer interpreter will compile commands into the memory for later execution, rather than execute them interactively. DumpCode in the memory is represented by a list of execution addresses:

```
dumpCode:
     addr of [']
     addr of ?TX
     addr of HERE
     addr of OVER
     addr of -
     addr of DUMP
     addr of ;
```

When dumpCode is executed, it is the inner interpreter which runs through this address list and causes each command to be executed in sequence. The interesting thing about the inner interpreter in Forth is that it is not a separated program which takes the address list, extracts the execution addresses out of the list and executes the commands. The inner interpreter is actually part of the code in every command. When a command is executed, at the end of its execution, the inner interpreter is executed which causes the next command to be pulled in and executed.

The inner interpreter in Forth is generally a very short code fragment inside the executable portion of every command in the memory.

There are several classes of Forth commands in a typical Forth system. Each class of Forth commands employs an unique inner interpreter to provide special run time behavior to this class of commands.

Virtual Forth Engine

Forth is a computer model which can be implemented on any real CPU with reasonable resources. This model is often called a virtual Forth engine. The minimal components of a virtual Forth engine are:

1. A dictionary to hold all the execution procedures called words.
2. A return stack to hold return addresses of procedures yet to be executed.
3. A data stack to hold parameters passing between procedures.
4. A user area in RAM memory hold all the system variables.
5. A CPU to move date among stacks and memory, and do ALU operations to items stored on the data stack.

The eForth Model is a detailed specification of a universal virtual Forth engine which can be implemented on many different CPU's and forces them to behave identically in executing an identical instruction set. This first implementation of eForth used Intel 80x86 CPU's as a guiding model to implementations on other CPU's. Here we will try to describe precisely the behavior of the generic eForth Model. When the logic description is not clear enough, we will use the 80x86 implementation to clarify the specification.

The following registers are required for a virtual Forth Engine:

Forth Register Function

```
IP   (SI)           Interpreter Pointer
SP   (SP)           Data Stack Pointer
RP   (RP)           Return Stack Pointer
WP   (AX)           Word or Work Pointer
UP   in memory      User Area Pointer
```

In the dictionary, each procedure (more commonly called word in Forth terminology) occupies an area called code field, which contains executable machine code and data required by the code. There are two types of words used in eForth: code word whose code field contains only machine instructions, and colon word whose code field contains an colon word interpreter and a list of tokens. A token is the execution address of the word in the dictionary. 4 bytes are allocated for the colon interpreter. Tokens are 2 bytes in length, and are pointers to code fields of words in the dictionary. The length of a code field varies depending upon the complexity of the word.

In the code field of a code word there is a list of machine instructions of the native CPU. The machine instructions are terminated by a group of instructions, generally specified as a macro instruction named $NEXT. The function of $NEXT is to fetch the token pointed to by the Interpreter Pointer IP, increment IP to point to the next token in a token list, and execute the token just fetched. Since a token points to a code field containing executable machine instructions, executing a token means jumping directly to the code field pointed to by the token. $NEXT thus allows the virtual Forth engine to execute a list of tokens with very little CPU overhead. In the 80x86 implementation, $NEXT is a macro assembling the following two machine instructions as shown below.

In other CPU's, especially the less capable 8 bit processors, $NEXT could assemble a JMP <NEXT> instruction. <NEXT> is then a centralized routine which causes the next token to be fetched and executed while IP is incremented to point to the next token.

```
;       Assemble inline direct threaded code ending.
$NEXT   MACRO
        LODSW                   ;;load next token into WP (AX)
        JMP     AX              ;;jump directly to the token thru WP
        ENDM                    ;;IP (SI) now points to the next token
```

This scheme of jumping to the execution address pointed to by a token is commonly referred to as 'Direct Threaded Code'. The other scheme 'Indirect Thread Code' used in many other Forth systems puts a pointer at the head of a code field. This pointer points to the executable code of an inner interpreter which defines the execution behavior of the token. A third scheme uses the pointer to point to a token table where addresses of executable code can be stored. This is called 'Token Threaded Code'. Direct Threaded Code is chosen in eForth because it is conceptually simpler and faster in execution.

In a colon word, the first four byte in the code field must be a machine instruction to process the token list following this first instruction. This token list processing routine is too complicated to fit into a four byte space; therefore, it is generally implemented as a CALL DOLST instruction. DOLST pushes the contents in IP onto the return stack, copies the address of the first token in its code field into IP and then calls $NEXT. $NEXT will then execute the list of tokens in the code field.

The last token in the token list of a colon word must be EXIT. EXIT is a code word which undoes what DOLST accomplished. EXIT pops the top item on the return stack into the IP register. Consequently, IP points to the token following the colon word just executed. EXIT then invokes $NEXT which continues the processing of the token list, briefly interrupted by the last colon word in this token list.

```
;    doLIST    ( a -- )
;    Process colon list.
     $CODE     COMPO+6,'doLIST',DOLST
     XCHG BP,SP          ;exchange pointers
     PUSH SI        ;push return stack
     XCHG BP,SP          ;restore the pointers
     POP  SI        ;new list address
     $NEXT

;    EXIT ( -- )
;    Terminate a colon definition.
     $CODE     4,'EXIT',EXIT
     XCHG BP,SP          ;exchange pointers
     POP  SI             ;pop return stack
     XCHG BP,SP          ;restore the pointers
     $NEXT
```

$NEXT, DOLST and EXIT are often call the 'inner interpreters' and 'address interpreters' of Forth. They are the corner stones of a virtual Forth Engine.

Based on the above mechanism to execute code words and colon words, a Forth engine can be constructed using a small set of machine dependent code words and a much larger set of colon words, giving it the capability of accepting commands from a user through a terminal device and compiling more words to extend the functionality of the basic system. The word at the highest level which interact with the user through a terminal is call the text interpreter and outer interpreter. The Forth text interpreter is equivalent to a conventional operating system with an integral compiler.

To learn Forth, it is important to keep in mind constantly the functions of the inner/address interpreter and of the outer/text interpreter. They are the crucial elements which make a Forth engine tick.

The text interpreter has the capability of compiling new words into the dictionary. A new word contains a list of existing words, like a string of subroutines. The user can thus expand the scope and capability of an elementary Forth system by adding more words to the dictionary. By adding more words based upon previously defined words in the dictionary, a Forth system can easily grow and encompasses solutions to a wide range of applications.

eForth Words

There are 190 high level words in eForth, built on the 31 low level primitive words. The high level word set is required to build the outer interpreter and the associated utility words. As the outer interpreter itself represents a fairly substantial application, the word set necessary to build the outer interpreter forms a very solid foundation to build most other applications. However, for any real world application one would not expect that this eForth word set is sufficient. The beauty of Forth is that in programming an application, the user designs and implements a new word set best tailored to his application. Forth is an open system, assuming that no operating system can be complete and all-encompassing. The user has the best understanding of his own needs, and he knows the best way to accomplish his goal.

Another way of looking at the eForth word set is to divide it into the commonly used words and the system words which are needed to build the outer interpreter, but are rarely used in applications. Among the 221 words in eForth, 134 words can be classified as commonly useful words and 87 words are the system words.

The set of 134 common Forth words contains the Forth words which are universally supported in most Forth systems. To use Forth fluently, you have to fully understand these words and use them efficiently to compose words which will solve your applications. They include data stack and return stack words, memory accessing words, math and logical words, control structure words, defining words, and some utility words.

The set of 87 system words are defined to support the construction of the outer interpreter in eForth. They are necessary in building a Forth system and are not needed for day-today programming. If you are building applications on an eForth system, only in rare occasions you will have to use words in this set. If you have to build an eForth system or port it to a special CPU, you must understand this word set very well because there are tools you must have to build or change the eForth system.

eForth Word Set

Device Dependent I/O Words
```
BYE  ?RX  TX!  !IO
```
Kernal Words
```
doLIT  doLIST  EXIT  EXECUTE  next  ?branch  branch  !  @  C!  C@ RP@  RP!  R>  R@  >R
SP@  SP!  DROP  DUP  SWAP  OVER  0<  AND  OR  XOR  UM+
```
System Variables
```
doVAR  UP  doUSER  SP0  RP0 '?KEY  'EMIT  'EXPECT  'TAP  'ECHO  'PROMPT  BASE  tmp  SPAN
>IN  #TIB  CSP  'EVAL  'NUMBER  HLD  HANDLER  CONTEXT  CURRENT  CP  NP  LAST
```
Common Functions
```
doVOC  FORTH  ?DUP  ROT  2DROP  2DUP  +  NOT  NEGATE  DNEGATE  -  ABS  =  U<  <  MAX  MIN
WITHIN
```
Divide and Multiply
```
UM/MOD  M/MOD  /MOD  MOD  /  UM*  *  M*  */MOD  */
```
Miscellaneous
```
CELL+  CELL-  CELLS  ALIGNED  BL  >CHAR  DEPTH  PICH
```
Memory Access
```
+!  2!  2@  COUNT  HERE  PAD  TIB  @EXECUTE  CMOVE  FILL  -TRAILING  PACK$
```
Numeric Output
```
DIGIT  EXTRACT  <#  HOLD  #  #S  SIGN  #>  str  HEX  DECIMAL
```
Numeric Input
```
DIGIT?  NUMBER?
```
Basic I/O
```
?KEY  KEY  EMIT  NUF?  PACE  SPACE  TYPE  CR  do$  $"|  ."|  .R  U.R  U.  .  ?
```
Parsing
```
parse  PARSE  .(  (  \  CHAR  TOKEN  WORD
```
Dictionary Search
```
NAME>  SAME?  find  NAME?
```
Terminal Response
```
^H  TAP  kTAP  accept  EXPECT  QUERY
```
Error Handling
```
CATCH  THROW  NULL$  ABORT  abort"
```
Text Interpreter
```
$INTERPRET  [  .OK  ?STACK  EVAL
```
Shell
```
PRESET  xio  FILE  HAND  I/O  CONSOLE  QUIT
```
Compiler
```
'  ALLOT  ,  [COMPILE]  COMPILE  LITERAL  $."  RECURSE
```
Structures
```
FOR  BEGIN  NEXT  UNTIL  AGAIN  IF  AHEAD  REPEAT  THEN  AFT  ELSE  WHILE  ABORT"  $"  ."
```
Name Compiler
```
?UNIQUE  $,n
```
Forth Compiler
```
$COMPILE  OVERT  ;  ]  call,  :  IMMEDIATE
```
Defining Words
```
USER  CREATE  VARIABLE
```
Tools
```
_TYPE  dm+  DUMP  .S  !CSP  ?CSP  >ANME  .ID  SEE  WORDS
```
Hardware Reset
```
VER  hi  'BOOT  COLD
```

Memory Map

The most important contribution by von Neumann to the computer design was the recognition that a single, uniform memory device can be used to store program and data, contrasting to the then prevailing architecture in which program and data were stored separately and most often using very different storage media. It greatly simplified the design of computers and had become the dominant computer architecture for all the important computer families ever since.

Memory space is a concept of paramount importance in computer hardware and assembly programming, but often hidden and ignored in most conventional high level languages. High level languages and operating systems hide the addressable memory space from the user in order to protect the operating system, because there are very sensitive areas in the memory space and unintentional alterations to the information stored in these areas would cause the system to malfunction or even to crash. The point of view from the operating system and from the computer priesthood, these sensitive areas must be protected at all cost, and they are the reserved territory of the systems programmers. Ordinary applications programmers are allocated only enough space to run their programs safely, for their own good.

Forth opens the entire memory space to the user. The user can freely store data and code into memory and retrieve them from the memory. Coming with the freedom is the responsibility of handling the memory correctly.

```
;; Memory allocation     0//code>--//--<name//up>--<sp//tib>--rp//em
EM   EQU  04000H    ;top of memory
COLDD     EQU  00100H    ;cold start vector
US   EQU  64*CELLL  ;user area size in cells
RTS  EQU  64*CELLL  ;return stack/TIB size
RPP  EQU  EM-8*CELLL     ;start of return stack (RP0)
TIBB EQU  RPP-RTS   ;terminal input buffer (TIB)
SPP  EQU  TIBB-8*CELLL   ;start of data stack (SP0)
UPP  EQU  EM-256*CELLL   ;start of user area (UP0)
NAMEE     EQU  UPP-8*CELLL    ;name dictionary
CODEE     EQU  COLDD+US  ;code dictionary
```

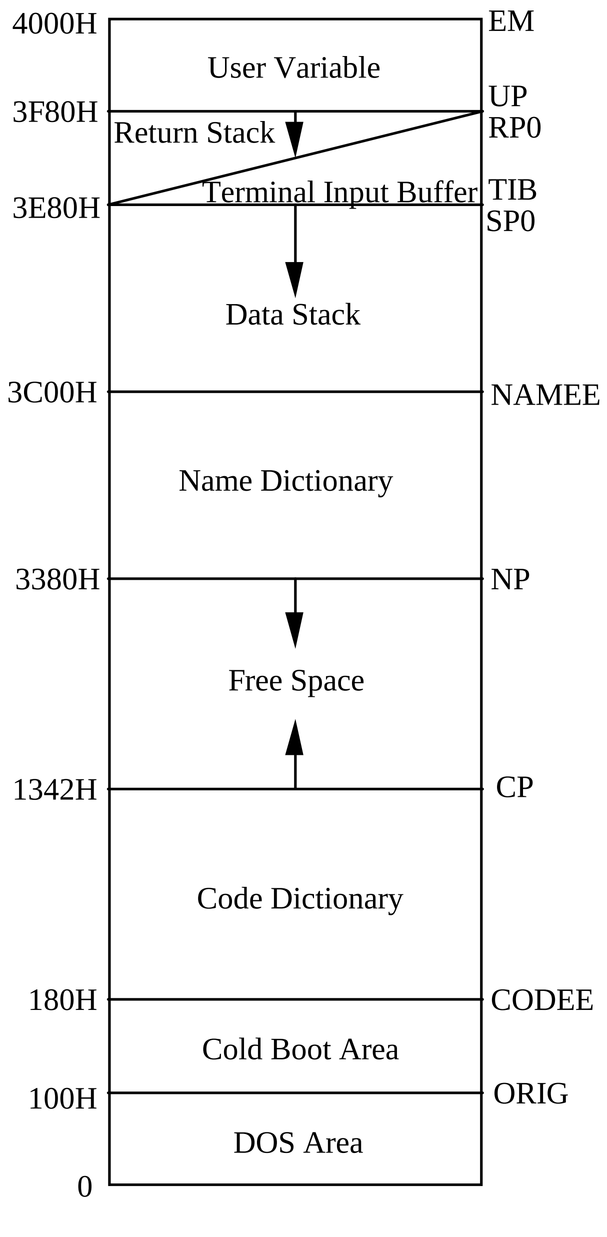
Memory used in eForth is separated into the following areas:

```
Cold boot           100H-17FH           Cold start and variable initial values
Code dictionary     180H-1344H          Code dictionary growing upward
Free space          1346H-33E4H         Shared by code and name dictionaries
Name/token          33E6H-3BFFH         Name dictionary growing downward
Data stack          3C00H-3E7FH         Growing downward
TIB                 3E80H-              Growing upward
Return stack        -3F7FH              Growing downward
User variables      3F80H-3FFFH
```

These areas are allocated by assembly constants and can be changed conveniently to suit the target environment.

The memory map follows conventional Forth system like figForth, except that the headers of words are put in a separated dictionary. A header consists of the following fields:

Field	Length	Function
Token	2 bytes	code address (ca)
Link	2 bytes	name address (na) of previous word
Length	1 byte	length and lexicon bits
Name	n bytes	name of word
Filler	0/1 byte	fill to cell boundary



The name dictionary is a linked list of all the word headers. This list grows downward towards the code dictionary. Thus the name dictionary shares the free space with the code dictionary. This arrangement has the advantage that the name dictionary can be eliminated from a target system which does not use an interpreter.

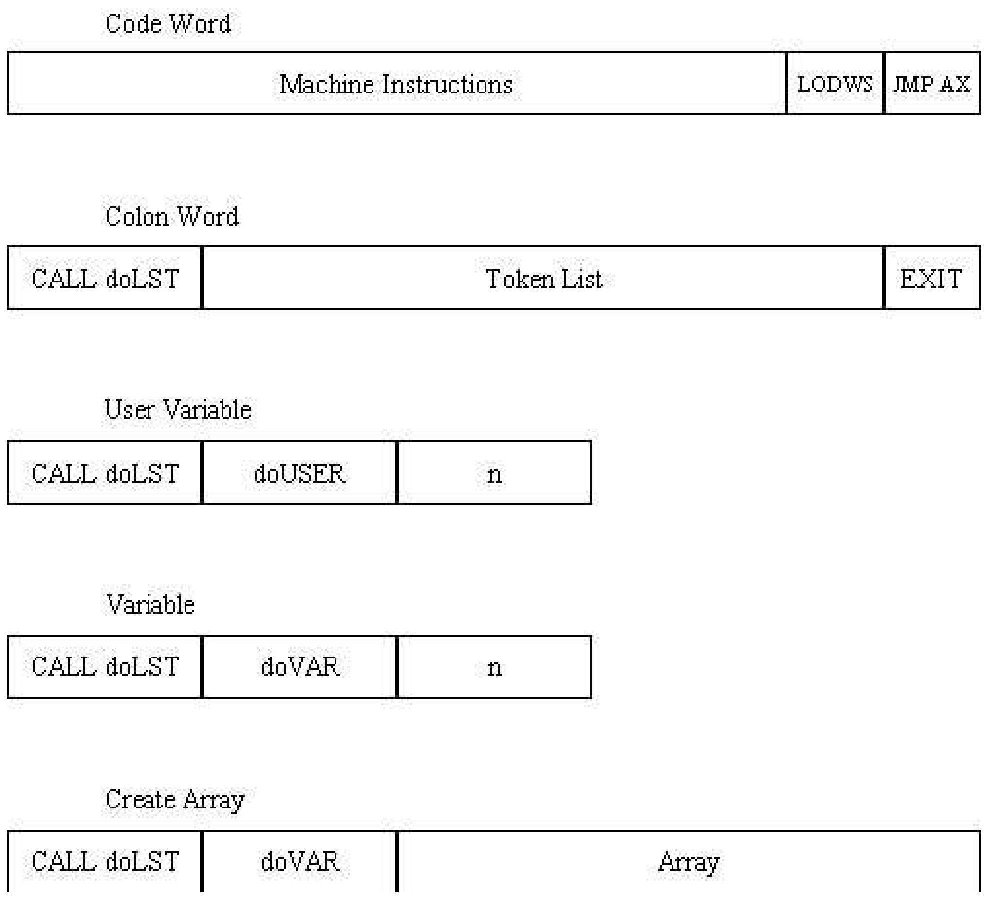
eForth is built using the 'Direct Threaded Code' technique. Each word is allocated a Code Field in the code dictionary. The starting address of the Code Field is stored in the Pointer Field in the header. This code address is considered the Token of this word. The code field contains executable code in Direct Threaded Code, contrasting to a pointer to executable code in Indirect Threaded Code scheme. In a low level machine code definition (a code word), the code field contains executable code terminated by the Forth inner interpreter $NEXT. $NEXT is defined as an in-line expanded macro (LODWS JMP AX) which fetches the next token from a token list and executes that token.

In a high level colon definition (a colon word), the code field begins with a CALL DOLIST instruction, which process the rest of the code field as a list of tokens. This is the only other code field construct used in eForth besides the code word construct. Other familiar constructs in conventional Forth systems like constants, variables, arrays, and user variables are derived from the colon word construct. The eForth implementations of these constructs (defining words) are shown in the following page.

EXIT, doLIT, doUSER and doVAR are themselves code word.

In the name dictionary, headers are linked through the link fields. The contents of a link field is the address of the name field in the previous header. This threading scheme optimizes the dictionary search. With the link field address on the data stack, @ @ will yield the length byte and the first character in the name of the previous word. This 16 bit value is compared to the length and first byte of the name to be searched for, and a quick decision can be reached to look for the next header or to compare the rest of the name. Since the name fields are NUL filled to the cell boundary, name comparisons proceed one cell at a time and can be made very efficient.

In eForth, all variables used by the system are defined as user variables and their initial values are stored in the cold boot area. During the cold booting process, all user variables are copied from the cold boot area to the user variable area in the high memory. This design is especially advantageous in a ROM based target system in which the cold boot area and the code dictionary are burnt into ROM.


Code Word
Machine Instructions
LODWS
JMP AX
Colon Word
CALL doLST
Token List
EXIT
User Variable
CALL doLST
doUSER
n
Variable
CALL doLST
doVAR
n
Create Array
CALL doLST
doVAR
Array

Assembly Macros

Several macros are defined in the MASM assembly source code to simplify the assembling of the object code. The macro $CODE is complicated because it is designed to assemble both a name dictionary and a code dictionary. Another problem is that the name dictionary must be laid down backward, from high memory to low memory as new entries are added to the name dictionary. The name field also required special attention because it must be NULL filled to the cell boundary.

$CODE begins by adjusting the assembly pointer $ to the next cell boundary in the code dictionary. A label is created marking the code address, which are later referred to as tokens. The assembly pointer is then saved in the assembly variable _CODE before assembling an entry in the name dictionary. The length of the next name entry is computed from the length of the new name, with due respect to the cell boundary at the end of the new name field. Now the assembly pointer is switched to the name dictionary, previously saved in _NAME, an offset to the beginning of the new name entry. From there, the code pointer field, the link field and the name field of the new name entry are assembled.

```
;; Initialize assembly variables
_LINK     = 0            ;force a null link
_NAME     = NAMEE        ;initialize name pointer
_CODE     = CODEE        ;initialize code pointer
_USER     = 4*CELLL      ;first user variable offset
;; Define assembly macros
;    Adjust an address to the next cell boundary.
$ALIGN  MACRO
     EVEN                ;;for 16bit systems
     ENDM
;    Compile a code definition header.
$CODE     MACRO     LEX,NAME,LABEL
     $ALIGN              ;;force to cell boundary
LABEL:                   ;;assembly label
     _CODE     = $       ;;save code pointer
     _LEN = (LEX AND 01FH)/CELLL   ;;string cell count, round down
     _NAME     = _NAME-((_LEN+3)*CELLL);;new header on cell boundary
ORG  _NAME               ;;set name pointer
     DW    _CODE,_LINK   ;;token pointer and link
     _LINK     = $       ;;link points to a name string
     DB   LEX,NAME       ;;name string
ORG  _CODE               ;;restore code pointer
     ENDM
;    Compile a colon definition header.
```

After the new header is completed, the address of the new pointer field is saved back in _NAME. The assembly pointer $ is then restored from _CODE, ready to assemble new code into the code field of this new word. For a code definition, machine code are assembled in this code field. For a colon word, a NOP and a CALL doLIST are assembled. The NOP is used to align the CALL to a cell boundary.

$CODE, $COLON and $USER are macros to build headers and the initial entries of code, colon and user variable definitions. $COLON uses $CODE to assemble a header and then assembles NOP CALL doLIST to begin a token list. $USER adds doUSE and an user variable offset to the CALL doLIST instruction. The user variable offset is managed by another assembly variable _USER so that the user variable list can be generated automatically.

D$ is a special macro to construct string literal in a colon definition. It first assembles a string literal token FUNCT, followed by the length byte of the string and then the string itself. The string is NULL filled to the cell boundary so that MASM is ready to resume assembling new tokens into the colon definition.

```
$COLON  MACRO  LEX,NAME,LABEL
     $CODE     LEX,NAME,LABEL
     NOP                ;;align to cell boundary
     CALL DOLST         ;;include CALL doLIST
     ENDM
;    Compile a user variable header.
$USER     MACRO     LEX,NAME,LABEL
     $CODE     LEX,NAME,LABEL
     NOP                ;;align to cell boundary
     CALL DOLST         ;;include CALL doLIST
     DW   DOUSE,_USER   ;;followed by doUSER and offset
     _USER     = _USER+CELLL  ;;update user area offset
     ENDM
;    Compile an inline string.
D$   MACRO     FUNCT,STRNG
     DW   FUNCT         ;;function
     _LEN = $           ;;save address of count byte
     DB   0,STRNG       ;;count byte and string
     _CODE     = $      ;;save code pointer
ORG  _LEN               ;;point to count byte
     DB   _CODE-_LEN-1  ;;set count
ORG  _CODE              ;;restore code pointer
     $ALIGN
     ENDM
```

Address Interpreter

Finally, the macro $NEXT is defined to assemble NEXT, the inner interpreter, at the end of every code word. It adds three bytes to a code word, the same length as a JMP <NEXT> instruction. The in-line $NEXT macro makes the code run much faster by eliminating an unnecessary jump. Using Direct Threaded Code and in-line $NEXT, eForth has the potential to be very fast and efficient. However, the goal of eForth is not performance but portability. The speed is intentionally throttled down because many important words are defined in high level colon words. Implementers and users are encouraged to optimize their systems, taking advantages of the clean and efficient mechanism built-in in eForth.

In a colon definition, tokens (execution addresses) are assembled using DW assembly commands. To execute a list of tokens, we first initialize the IP register (SI in 8086) to point to the head of this list and then execute $NEXT. LODSW reads an execution address into the WP register (AX in 8086), and the IP register is automatically incremented to point to the next token in the token list. JMP AX simply jumps to the token's execution address and causes the token to be executed. At the end of the execution sequence of the token, there will be another $NEXT, which causes the next token to be read and executed, and so forth.

This $NEXT is a very powerful mechanism allowing token lists to be scanned and executed in sequence. It is thus commonly referred to as the address interpreter in Forth.

It is also called the inner interpreter of Forth, in contrast to the text interpreter which is also called the outer interpreter of Forth. With an address interpreter embedded in every word, each code word in Forth is a self contained piece of code. This property makes Forth words reentrant so that one copy of the word can be executed by many different tasks independently.

```
;    Assemble inline direct threaded code ending.
$NEXT     MACRO
     LODSW          ;;next code address into AX
     JMP  AX        ;;jump directly to code address
     ENDM
```

Cold Boot

DOS starts executing the object code at 100H. The eForth Model is configured for a DOS machine. It can be modified to jump to any memory location from where the CPU boots up. What we have to do here is to set up the 8086 CPU so that it will emulate the virtual Forth engine as we discussed before. All the pertinent registers have to be initialized properly. Since eForth is very small and fits comfortably in a single 64 Kbyte code segment, we will use only one segment for code, data, as well as the two stacks. Therefore, both the DS and SS segment registers are initialized to be the same as the CS register. Then, the data stack pointer SP and the return stack pointer RP (BP in 8086) are initialized. To prevent the eForth from being forced back into DOS accidentally, the Control-C interrupt is made benign by vectoring it to a simple IRET instruction.

Now we are ready to start the Forth engine. Simply jumping to COLD will do it. COLD is coded as a colon word, containing a list of tokens. This token list does more initialization in high level, including initializing the user area, and setting up the terminal input buffer. At the end, COLD executes QUIT, the text interpreter, which contains an infinite loop to receive commands from a user and executes them repeatedly.

```
;; Main entry points and COLD start data
MAIN SEGMENT
ASSUME  CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN
ORG  COLDD               ;beginning of cold boot
ORIG:     MOV  AX,CS
     MOV  DS,AX          ;DS is same as CS
     CLI                 ;disable interrupts, old 808x CPU bug
     MOV  SS,AX          ;SS is same as CS
     MOV  SP,SPP         ;initialize SP
     STI                 ;enable interrupts
     MOV  BP,RPP         ;initialize RP
     MOV  AL,023H        ;interrupt 23H
     MOV  DX,OFFSET CTRLC
     MOV  AH,025H        ;MS-DOS set interrupt vector
     INT  021H
     CLD                 ;direction flag, increment
     JMP  COLD           ;to high level cold start
CTRLC:IRET               ;control C interrupt routine
```

Initializing User Variables

The user area contains vital information for Forth to perform its functions. It contains important pointers specifying memory areas for various activities, such as the data stack, the return stack, the terminal input buffer, where the code dictionary and the name dictionary end, and the execution addresses of many vectored words like KEY, EMIT, ECHO, EXPECT, NUMBER, etc.

The user area must be located in the RAM memory, because the information contained in it are continuously updated when eForth is running. The default values are stored in the code segment starting at UZERO and covering an area of 74 bytes. This area is copied to the user area in RAM before starting the eForth engine. The sequence of data in UZERO must match exactly the sequence of user variables.

```
; COLD start moves the following to USER variables.
; MUST BE IN SAME ORDER AS USER VARIABLES.
$ALIGN              ;align to cell boundary
UZERO:    DW   4 DUP (0) ;reserved
     DW   SPP       ;SP0
     DW   RPP       ;RP0
     DW   QRX       ;'?KEY
     DW   TXSTO     ;'EMIT
     DW   ACCEP     ;'EXPECT
     DW   KTAP      ;'TAP
     DW   TXSTO     ;'ECHO
     DW   DOTOK     ;'PROMPT
     DW   BASEE     ;BASE
     DW   0         ;tmp
     DW   0         ;SPAN
     DW   0         ;>IN
     DW   0         ;#TIB
     DW   TIBB      ;TIB
     DW   0         ;CSP
     DW   INTER     ;'EVAL
     DW   NUMBQ     ;'NUMBER
     DW   0         ;HLD
     DW   0         ;HANDLER
     DW   0         ;CONTEXT pointer
     DW   VOCSS DUP (0)  ;vocabulary stack
     DW   0         ;CURRENT pointer
     DW   0         ;vocabulary link pointer
     DW   CTOP      ;CP
     DW   NTOP      ;NP
     DW   LASTN     ;LAST
ULAST:
```

4. Machine Dependent Kernel

For the very beginning, we wanted to develop a complete Forth system with a minimum set of primitives. In the Forth community, this has been an active argument since day one. What had been selected as the eForth kernel word set were all the operations we can not synthesize conveniently and effectively. Actually, eForth provided a good platform to test the effectiveness of this set of primitive Forth words. We had observed that in certain small 8-bit controller like 8051, this primitive word set really slows down the processor. However, the same primitive words set proved to be quite adequate in the 16-bit processors like 8086 and 68000. In the 32-bit implementation for 80386 in the protected mode, eForth is blazingly fast.

Since most microprocessors have fairly good machine instruction sets, it is quite easy to rewrite the eForth kernel for any target microprocessor. The assembly code in the eForth Model serves to clarify any discrepancy in the functional specifications of the primitive words.

eForth Kernel

One of the most important feature of eForth is the small machine dependent kernel, which allows its to be ported to other CPU's very conveniently. The selection of words in this kernel is based on the criteria that they are very difficult if not impossible to synthesize from other words as high level colon definitions. From this set of kernel words, all other Forth words have to be built. The kernel words can be classified as following:

```
System interface:        BYE, ?rx, tx!, !io
Inner interpreters:      doLIT, doLIST, next, ?branch, branch, EXECUTE, EXIT
Memory access:           ! , @,  C!,  C@
Return stack:            RP@,  RP!,  R>,  R@,  R>
Data stack:              SP@,  SP!,  DROP,  DUP,  SWAP,  OVER
Logic:                   0<,  AND,  OR,  XOR
Arithmetic:              UM+
```

BYE returns control from eForth back to the operating system. !io initializes the serial I/O device in the system so that it can interact with the user through a terminal. These two words are not needed once the eForth system is up and running, but they are essential to bring the system up in DOS. ?rx is used to implement ?KEY and KEY, and tx! is used to implement EMIT. eForth communicates with the user through these words which supports terminal interactions and file download/upload.

Here these words are defined using the DOS service calls.   For embedded controllers, these three words must be defined for the specific I/O devices.

```
ORG    CODEE        ;start code dictionary
;; Device dependent I/O
;    BYE  ( -- )
;    Exit eForth.
     $CODE     3,'BYE',BYE
     INT  020H      ;MS-DOS terminate process
```

?RX is a unique design invented by Bill Muench to support serial input .   ?RX provides the functions required of both KEY and KEY? which accept input from a terminal.   ?RX inspects the terminal device and returns a character and a true flag if the character has been received and is waiting to be retrieved.   If no character was received, ?RX simply returns a false flag.   With ?RX, both KEY and KEY? can be defined as high level colon definitions.

TX! sends a character on the data stack to the terminal device.   Both ?RX and TX! are coded here as DOS calls.   In embedded applications, they will have to be coded in machine specific code to handle the specific serial I/O device.

!IO initializes the serial I/O device, which is not necessary here because it is taking care of by the DOS.   In embedded systems, the I/O device must be initialized by !IO.

```
;    ?RX  ( -- c T | F )
;    Return input character and true, or a false if no input.
     $CODE     3,'?RX',QRX
     XOR  BX,BX          ;BX=0 setup for false flag
     MOV  DL,0FFH        ;input command
     MOV  AH,6           ;MS-DOS Direct Console I/O
     INT  021H
     JZ   QRX3           ;?key ready
     OR   AL,AL          ;AL=0 if extended char
     JNZ  QRX1           ;?extended character code
     INT  021H
     MOV  BH,AL          ;extended code in msb
     JMP  QRX2
QRX1:     MOV  BL,AL
QRX2:     PUSH BX        ;save character
     MOV  BX,-1          ;true flag
QRX3:     PUSH BX
     $NEXT

;    TX!  ( c -- )
;    Send character c to the output device.
     $CODE     3,'TX!',TXSTO
     POP  DX             ;char in DL
     CMP  DL,0FFH        ;0FFH is interpreted as input
     JNZ  TX1            ;do NOT allow input
     MOV  DL,32          ;change to blank
TX1: MOV  AH,6           ;MS-DOS Direct Console I/O
     INT  021H           ;display character
     $NEXT

;    !IO  ( -- )
;    Initialize the serial I/O devices.
     $CODE     3,'!IO',STOIO
     $NEXT
```

Colon Word Interpreter

The words doLIST and EXIT encapsulate a token list in a colon definition, which begins with a CALL doLIST, followed by a list of tokens, and terminated by EXIT. doLIST pushes the current Instruction Pointer (IP), which is in the SI register, on the return stack and then pops the address of the token into IP from the data stack. When NEXT is executed, the tokens in the list are executed consecutively.

EXIT is at the end of all token lists. EXIT pops the execution address saved on the return stack back into the IP register and thus restores the condition before the colon word was entered. Execution of the calling token list will continue.

doLIST and NEXT are therefore the interpreter of high level colon definitions. They are the crucial mechanism which executes a high level token in a token list and then returns control to the calling token list.

EXECUTE takes the execution address from the data stack and executes that token. This powerful word allows the user to execute any token which is not a part of a token list.

```
;    doLIST    ( a -- )
;    Process colon list.
     $CODE     COMPO+6,'doLIST',DOLST
     XCHG BP,SP          ;exchange pointers
     PUSH SI             ;push return stack
     XCHG BP,SP          ;restore the pointers
     POP  SI             ;new list address
     $NEXT

;    EXIT ( -- )
;    Terminate a colon definition.
     $CODE     4,'EXIT',EXIT
     XCHG BP,SP          ;exchange pointers
     POP  SI             ;pop return stack
     XCHG BP,SP          ;restore the pointers
     $NEXT

;    EXECUTE   ( ca -- )
;    Execute the word at ca.
     $CODE     7,'EXECUTE',EXECU
     POP  BX
     JMP  BX             ;jump to the code address
```

Integer Literals

In the token list of a colon definition, it is generally assumed that tokens are execution addresses, which can be executed sequentially by the address interpreter $NEXT. However, occasionally we do need to compile other types of data in-line with the tokens. Special mechanisms must be used to tell the address interpreter to treat these data differently. All data entries must be preceded by special tokens which can handle the data properly. A special token and its associated data form a data structure. Data structures are extensions of tokens and can be thought of as building blocks to form lists in colon definitions with regular tokens.

In eForth, three types of data structures are allowed: integer littorals preceded by doLIT, address literals preceded by 'next', '?branch' and 'branch', and string literals (which will be discussed in the section of compiler). doLIT pushes the next token onto the data stack as an integer literal. It allows numbers to be compiled as in-line literals, supplying data to the data stack at run time. eForth uses this mechanism to replace CONSTANTs as we know in conventional Forth systems. Integer literals (doLIT n) are compiled by LITERAL which will be discussed in the compiler section.

Integer literals are by far the most numerous data structures in colon definitions other than regular tokens. Address literals are used to build control structures. String literals are used to embed text strings in colon definitions. We will discuss address literals and string literals later.

In the 8086 eForth implementation, doLIT is very simple and very fast. It is the preferred way to introduce integers to the data stack than constants and variables.

```
;    doLIT     ( -- w )
;    Push an inline literal.
     $CODE     COMPO+5,'doLIT',DOLIT
     LODSW               ;get the literal compiled in-line
     PUSH AX             ;push literal on the stack
     $NEXT               ;execute next token after literal
```

Address Literals

eForth uses three different types of address literals. 'next', '?branch' and 'branch' are followed not by tokens but by addresses of tokens in a list to be executed next. These address literals are the building blocks upon which looping and branching structures are constructed. An address token is followed by a branch address which causes execution to be transferred to that address. The branch address most often points to a different location in the same token list.

Address literals are used to construct control structures in colon definitions. 'next' is compiled by NEXT. '?branch' is compiled by IF, WHILE and UNTIL. 'branch' is compiled by AFT, ELSE, REPEAT and AGAIN.

It is interesting to note that eForth supplies only the down-counting FOR-NEXT definite loop structure. The more conventional DO-LOOP structure and its variations are not supported. The dual indexed DO-LOOP structure is much more complicated than the single index FOR-NEXT structure. Omitting DO-LOOPs simplifies eForth greatly.

```
;    next ( -- )
;    Run time code for the single index loop.
     $CODE     COMPO+4,'next',DONXT
     SUB  WORD PTR [BP],1     ;decrement the index
     JC   NEXT1          ;?decrement below 0
     MOV  SI,0[SI]       ;no, continue loop
     $NEXT
NEXT1:ADD BP,CELLL       ;yes, pop the index
     ADD  SI,CELLL       ;exit loop
     $NEXT

;    ?branch   ( f -- )
;    Branch if flag is zero.
     $CODE     COMPO+7,'?branch',QBRAN
     POP  BX             ;pop flag
     OR   BX,BX          ;?flag=0
     JZ   BRAN1          ;yes, so branch
     ADD  SI,CELLL       ;point IP to next cell
     $NEXT
BRAN1:MOV SI,0[SI]  ;IP:=(IP), jump to new address
     $NEXT

;    branch    ( -- )
;    Branch to an inline address.
     $CODE     COMPO+6,'branch',BRAN
     MOV  SI,0[SI]       ;jump to new address unconditionally
     $NEXT
```

Memory Access

Four memory accessing words are included in the eForth kernel: !, @, C! and C@. ! and @ access memory in cells, whose size depends on the CPU underneath. eForth assumes that the CPU can access memory in bytes and that all addresses are in the units of bytes. Porting eForth into a cell addressing CPU, C! and C@ must used synthesized byte addresses and one must be able to switch between cell addresses and byte addresses conveniently.

@ and C@ allow the user to inspect any memory location in the computer, and they can be executed harmlessly. On the other hand, ! and C! are dangerous. You can mistakenly store wrong data into the dictionary, the user variable area, and the stack area. When this happens, most likely the Forth system will crash, or behave erratically. However, it is very easy to reboot the system ans start over again. If one develop a system incrementally and save the source code often, crashes do not seriously impede the progress.

```
;    !    ( w a -- )
;    Pop the data stack to memory.
     $CODE     1,'!',STORE
     POP  BX        ;get address from tos
     POP  0[BX]          ;store data to that adddress
     $NEXT

;    @    ( a -- w )
;    Push memory location to the data stack.
     $CODE     1,'@',AT
     POP  BX        ;get address
     PUSH 0[BX]          ;fetch data
     $NEXT

;    C!   ( c b -- )
;    Pop the data stack to byte memory.
     $CODE     2,'C!',CSTOR
     POP  BX        ;get address
     POP  AX        ;get data in a cell
     MOV  0[BX],AL  ;store one byte
     $NEXT

;    C@   ( b -- c )
;    Push byte memory location to the data stack.
     $CODE     2,'C@',CAT
     POP  BX        ; get address
     XOR  AX,AX     ;AX=0 zero the hi byte
     MOV  AL,0[BX]  ;get low byte
     PUSH AX        ;push on stack
     $NEXT
```

Return Stack Words

Return stack is used by the virtual Forth engine to save return addresses to be processes later. It is also a convenient place to store data temporarily. The return stack can thus be considered as a extension of the data stack. However, one must be very careful in using the return stack for temporary storage. The data pushed on the return stack must be popped off before EXIT is executed. Otherwise, EXIT will get the wrong address to return to, and the system generally will crash.

RP@ and RP! are only used to initialize the system and are seldom used in applications.

>R pops a number off the data stack and pushes it on the return stack.. R> does the opposite. R@ copies the top item on the return stack and pushes it on the data stack.

```
;    RP@  ( -- a )
;    Push the current RP to the data stack.
     $CODE     3,'RP@',RPAT
     PUSH BP             ;copy address to return stack
     $NEXT               ;pointer register BP

;    RP!  ( a -- )
;    Set the return stack pointer.
     $CODE     COMPO+3,'RP!',RPSTO
     POP  BP             ;copy (BP) to tos
     $NEXT

;    R>   ( -- w )
;    Pop the return stack to the data stack.
     $CODE     2,'R>',RFROM
     PUSH 0[BP]          ;copy w to data stack
     ADD  BP,CELLL       ;adjust RP for popping
     $NEXT

;    R@   ( -- w )
;    Copy top of return stack to the data stack.
     $CODE     2,'R@',RAT
     PUSH 0[BP]          ;copy w to data stack
     $NEXT

;    >R   ( w -- )
;    Push the data stack to the return stack.
     $CODE     COMPO+2,'>R',TOR
     SUB  BP,CELLL       ;adjust RP for pushing
     POP  0[BP]          ;push w to return stack
     $NEXT
```

Data Stack Initialization

Data stack is initialized by SP!. The depth of data stack can be examined by SP@. These words, as RP@ and RP! are only used by the system and very rarely used in applications. These words are necessary in the Forth kernel because you cannot operate a stack-based computer without these instructions. However, in a true Forth engine like NC4000 and RTX2000 which have build-in circular stacks in independent stack spaces, which are completely separated from the main memory, these stack words are unnecessary.

SP! and RP! are not needed because the stacks will be initialized by hardware. SP@ and RP@ are not needed because the stacks are circular and they do not overflow. The system does not have to constantly watch over them to prevent them from overflowing or underflowing.

In true Forth engines, we can eliminate these four instructions. However, to impose virtual Forth engines on conventional CPU's, we have to retain them to manage the stacks. It is very awkward to justify their existence in the eForth kernel since they do occupy 4 slots among the 31 primitive eForth words, but are rarely used in applications.

```
;    SP@  ( -- a )
;    Push the current data stack pointer.
     $CODE     3,'SP@',SPAT
     MOV  BX,SP          ;use BX to index the stack
     PUSH BX             ;push SP back
     $NEXT

;    SP!  ( a -- )
;    Set the data stack pointer.
     $CODE     3,'SP!',SPSTO
     POP  SP             ;safe
     $NEXT
```

Classic Data Stack Words

The data stack is the centralized location where all numerical data are processed, and where parameters are passed from one word to another. The stack items has to be arranged properly so that they can be retrieved properly in the Last-In-First-Out (LIFO) manner. When stack items are out of order, they can be rearranged by the stack words DUP, SWAP, OVER and DROP. There are other stack words useful in manipulating stack items, but these four are considered to be the minimum set.

In this eForth Model, we use the system stack as the data stack. We can use the machine instructions PUSH and POP to access the data stack. Because 8086 is a register-based CPU, all arithmetic and logic operations are performed in the CPU registers, it is necessary to POP items from the data stack into the registers for these operations, and then PUSH the results back on the data stack. This is the overhead we have to pay in using a register machine to emulate a stack machine.

```
;    DROP ( w -- )
;    Discard top stack item.
     $CODE     4,'DROP',DROP
     ADD  SP,CELLL       ;adjust SP to pop
     $NEXT

;    DUP  ( w -- w w )
;    Duplicate the top stack item.
     $CODE     3,'DUP',DUPP
     MOV  BX,SP          ;use BX to index the stack
     PUSH 0[BX]
     $NEXT

;    SWAP ( w1 w2 -- w2 w1 )
;    Exchange top two stack items.
     $CODE     4,'SWAP',SWAP
     POP  BX             ;get w2
     POP  AX             ;get w1
     PUSH BX             ;push w2
     PUSH AX             ;push w1
     $NEXT

;    OVER ( w1 w2 -- w1 w2 w1 )
;    Copy second stack item to top.
     $CODE     4,'OVER',OVER
     MOV  BX,SP          ;use BX to index the stack
     PUSH CELLL[BX]      ;get w1 and push on stack
     $NEXT
```

Logical Words

The only primitive word which cares about logic is '?branch'. It tests the top item on the stack. If it is zero, ?branch will branch to the following address. If it is not zero, ?branch will ignore the address and execute the token after the branch address. Thus we distinguish two classes of numbers, zero for 'false' and non-zero for 'true'. Numbers used this way are called logic flags which can be either true or false. The only primitive word which generates flags is '0<', which examines the top item on the data stack for its negativeness. If it is negative, '0<' will return a -1 for true. If it is 0 or positive, '0<' will return a 0 for false.

The three logic words AND, OR and XOR are bitwise logic operators over the width of a cell. They can be used to operate on real flags (0 and -1) for logic purposes. The user must be aware of the distinct behaviors between the real flags and the generalized flags.

```
;    0<   ( n -- f )     ;Return true if n is negative.
     $CODE     2,'0<',ZLESS
     POP  AX
     CWD                 ;sign extend AX into DX
     PUSH DX             ;push 0 or -1
     $NEXT

;    AND  ( w w -- w )   ;Bitwise AND.
     $CODE     3,'AND',ANDD
     POP  BX
     POP  AX
     AND  BX,AX
     PUSH BX
     $NEXT

;    OR   ( w w -- w )   ;Bitwise inclusive OR.
     $CODE     2,'OR',ORR
     POP  BX
     POP  AX
     OR   BX,AX
     PUSH BX
     $NEXT

;    XOR  ( w w -- w )   ;Bitwise exclusive OR.
     $CODE     3,'XOR',XORR
     POP  BX
     POP  AX
     XOR  BX,AX
     PUSH BX
     $NEXT
```

Primitive Arithmetic Word

The only primitive arithmetic word in the eForth kernel is UM+. All other arithmetic words, like +, -, * and / are defined from UM+ and some logic word as colon definitions. Bill Muench pioneered this design in his bForth. This design emphasize portability over performance, because it greatly reduces the efforts in moving eForth into CPU's which do not have native multiply and divide instructions. Once eForth is implemented on a new CPU, the multiply and divide words are the first ones to be optimized to enhance the performance.

UM+ adds two unsigned number on the top of the data stack and returns to the data stack the sum of these two numbers and the carry as one number on top of the sum. To handle the carry this way is very inefficient, because most CPU's have carry as a bit in the status register, and the carry can be accessed by many machine instructions. It is thus more convenient to use carry in machine code programming. eForth provides the user a handle on the carry in high level, making it easier for the user to deal with it directly.

Since it is slower to handle the carry in high level code, we expect that the user will enhance the eForth system by recoding many of the high level ALU words in assembly so that the eForth system can run faster in real applications.

```
;    UM+  ( w w -- w cy )
;    Add two numbers, return the sum and carry flag.
     $CODE     3,'UM+',UPLUS
     XOR  CX,CX          ;CX=0 initial carry flag
     POP  BX
     POP  AX
     ADD  AX,BX
     RCL  CX,1           ;get carry
     PUSH AX             ;push sum
     PUSH CX             ;push carry
     $NEXT
```

5. High Level Forth Words

Following are the eForth words defined as high level colon definitions. They are built from the primitive eForth words and other high level eForth words, including data structures and control structures. Since eForth source is coded in Microsoft MASM assembler, the token lists in the colon definitions are constructed as data in MASM, using the DW directive. This form of representation, though very effective, is very difficult to read. The original model of eForth as provided by Bill Muench was in the form of a Forth source listing. This listing is much simpler and easy to read, assuming that the reader has some knowledge of the Forth syntax. This listing is also a very good source to learn a good coding style of Forth. I therefore think it is better to present the high level Forth colon definitions in this form. As the 8086 eForth implementation deviates slightly from the original Forth model, I tried to translate the 8086 implementation faithfully back to the Forth style for our discussion here.

The sequence of words is exactly the same as that in the MASM assembly source listing. The reader is encouraged to read the MASM source listing along with the text in this book. Reading two descriptions of the same subject often enable better comprehension and understanding.

User Variables

The term user variable was codified in earlier Forth systems on the mini-computers in which multitasking was an integral part of the Forth operating system. In a multitasking system, many user share CPU and other resources in the computing system. Each user has a private memory area to store essential information about its own task so that the system can leave a task temporarily to serve other users and return to this task continuing the unfinished work. In a single user environment, the user variables have the same functionality as system variables.

In eForth, all variables used by the system are merged together and are implemented uniformly as user variables. A special memory area in the high memory is allocated for all these variables, and they are all initialized by copying a table of initial values stored in the cold boot area. A significant benefit of this scheme is that it allows the eForth system to operate in ROM memory naturally. It is very convenient for embedded system applications which preclude mass storage and file downloading.

In an application, the user can choose to implement variables in the forms of user variables or regular variables when running in RAM memory. To run things in ROM, variables must be defined as user variables. Although eForth in the original model allows only a small number of user variable to be defined in an application, the user area can be enlarged at will by changing a few assembly constants and equates.

```
:    doVAR ( -- a )
     ( Run time routine for VARIABLE and CREATE.)
     R> ;                ( where data is stored)

VARIABLE  UP   ( -- a )
     ( Pointer to the user area.)

:    doUSER    ( -- a )
     ( Run time routine for user variables.)
     R> @                ( retrieve user area offset)
     UP @ + ;            ( add to user area base addr)
```

eForth provides many functions in the vectored form to allow the behavior the these functions to be changed dynamically at run time.   A vectored function stores a code address in a user variable. @EXECUTE is used to execute the function, given the address of the user variable.

```
USER SP0  ( -- a )
     Pointer to bottom of the data stack.
USER RP0  ( -- a )
     Pointer to bottom of the return stack.
USER '?KEY     ( -- a )
     Execution vector of ?KEY.  Default to ?rx.
USER 'EMIT     ( -- a )
     Execution vector of EMIT.   Default to tx!
USER 'EXPECT   ( -- a )
     Execution vector of EXPECT.  Default to 'accept'.
USER 'TAP ( -- a )
     Execution vector of TAP.  Defulat the kTAP.
USER 'ECHO     ( -- a )
     Execution vector of ECHO.  Default to tx!.
USER 'PROMPT   ( -- a )
     Execution vector of PROMPT.  Default to '.ok'.
USER BASE ( -- a )
     Storage of the radix base for numeric I/O.  Default to 10.
USER tmp  ( -- a )
     A temporary storage location used in parse and find.
USER SPAN ( -- a )
     Hold character count received by EXPECT.
USER >IN  ( -- a )
     Hold the character pointer while parsing input stream.
USER #TIB ( -- a )
     Hold the current count and address of the terminal input buffer.
     Terminal Input Buffer used one cell after #TIB.
USER CSP  ( -- a )
     Hold the stack pointer for error checking.
USER 'EVAL     ( -- a )
     Execution vector of EVAL.  Default to EVAL.
USER 'NUMBER   ( -- a )
     Execution vector of number conversion.  Default to NUMBER?.
USER HLD  ( -- a )
     Hold a pointer in building a numeric output string.
USER HANDLER   ( -- a )
     Hold the return stack pointer for error handling.
USER CONTEXT   ( -- a )
     A area to specify vocabulary search order.  Default to FORTH.
     Vocabulary stack, 8 cells follwing CONTEXT.
USER CURRENT   ( -- a )
     Point to the vocabulary to be extended.  Default to FORTH.
     Vocabulary link uses one cell after CURRENT.
USER CP   ( -- a )
     Point to the top of the code dictionary.
USER NP   ( -- a )
     Point to the bottom of the name dictionary.
USER LAST ( -- a )
     Point to the last name in the name dictionary.
```

Vocabulary and Search Order

In eForth only one vocabulary is used. The name of this vocabulary is FORTH. When FORTH is executed, the address of the pointer to the top of the dictionary is written into the first cell in the CONTEXT array. When the text interpreter searches the dictionary for a words, it picks up the pointer in CONTEXT and follow the thread through the name dictionary. If the name dictionary is exhausted, the text interpreter will pick up the next cell in the CONTEXT array and do the search. The first cell in CONTEXT array containing a 0 stops the searching. There are 8 cells in the CONTEXT array. Since the last cell must be zero, eForth allows up to 8 context vocabularies to be searched.

There are two empty cells in the code field of FORTH. The first cell stores the pointer to the last name field in the name dictionary. The second field must be a 0, which serves to terminate a vocabulary link when many vocabularies are created. Vocabularies are useful in reducing the number of words the text interpreter must search to locate a word, and allowing related words to be grouped together as logic modules. Although the eForth itself only uses one vocabulary, the mechanism is provided to define multiple vocabularies in large applications.

The CONTEXT arrays is designed as a vocabulary stack to implement the ONLY- ALSO concept of vocabulary search order first proposed by Bill Ragsdale in the Forth 83 Standard.

CURRENT points to a vocabulary thread to which new definitions are to be added.

```
: doVOC   ( -- )
     ( Run time action of VOCABULARY's.)
     R>                  ( get vocab pointer)
     CONTEXT ! ;         ( make it context vocab)

: FORTH   ( -- )
     ( Make FORTH the context vocabulary.)
     doVOC               ( retrieve FORTH pointer)
                         ( and make it context vocab)
     0 ,                 ( vocabulary head pointer)
     0 ,                 ( vocabulary link pointer)
```

Multitasking Considerations

As discussed before, Forth systems early on had incorporated the multitasking features through the use of user variables.   Multitasking is a very advanced feature we chose not to implement in eForth, because the design goal of eForth is to present a simple and portable Forth system to first time Forth users.   Nevertheless, the design of eForth has provisions to become a multitasking system if the user so desires.

The first four cells in the user area are reserved for implementing multitasker in eForth. Four cells are sufficient to build a Round Robin Multitask Switching mechanism on most CPU's.   Implementing multitasking is an advance topic beyond the scope of eForth. Interested reader should consult polyForth, F83 or FPC to get more detailed information on how a multitasking system works and how to build a multitasking system.

More Stack Words

This group of Forth words are commonly used in writing Forth applications. They are coded in high level to enhance the portability of eForth. In most Forth implementations, they are coded in machine language to increase the execute speed. After an eForth system is ported to a new CPU, this word set should be recoded in assembly to improve the run time performance of the system.

This group of words are stack operators supplementing the four classic stack operators DUP, SWAP, OVER and DROP.

ROT is unique in that it accesses the third item on the data stack. All other stack operators can only access one or two stack items. In Forth programming, it is generally accepted that one should not try to access stack items deeper than the third item. When you have to access deeper into the data stack, it is a good time to re-evaluate your algorithm. Most often, you can avoid this situation by factoring your code into smaller parts which do not reach so deep.

When you have to dig deep into the stack, the code becomes very difficult to read because of all the stack operations. This is one of the reasons why Forth acquired the reputation of a write-only language. Proper factoring is the best cure against this tendency.

```
: ?DUP    ( w -- w w | 0 )
     ( Dup top of stack if its is not zero.)
     DUP
     IF DUP THEN ;       ( add another copy if not 0)

: ROT     ( w1 w2 w3 -- w2 w3 w1 )
     ( Rot 3rd item to top.)
     >R                  ( save top item)
     SWAP                ( get 3rd to top)
     R>                  ( retrieve top)
     SWAP ;              ( get 3rd to top)

: 2DROP   ( w w -- )
     ( Discard two items on stack.)
     DROP DROP ;

: 2DUP    ( w1 w2 -- w1 w2 w1 w2 )
     ( Duplicate top two items.)
     OVER OVER ;
```

More Arithmetic Operators

This group of arithmetic operators are simple extensions from the primitive word UM+. It is interesting to see how the more commonly used arithmetic operators are derived. + is UM+ with the carry discarded. NOT returns the ones compliment of a number, and NEGATE returns the two's compliment. Because UM+ preserves the carry, it can be used to form multiple precision operators like D+. Later we will see how UM+ is used to do multiplication and division.

```
: +  ( w w -- sum )
     ( Add top two items.)
     UM+                 ( return sum and carry)
     DROP ;              ( discard carry)

: D+ ( d d -- d )
     ( Double addition, as an example using UM+.)
     >R SWAP >R          ( save high parts)
     UM+                 ( add low parts with carry)
     R> R> +             ( add high parts)
     + ;                 ( add carry)

: NOT     ( w -- w )
     ( One's complement of tos.)
     -1 XOR ;

: NEGATE  ( n -- -n )
     ( Two's complement of tos.)
     NOT 1 + ;

: DNEGATE ( d -- -d )
     ( Two's complement of top double.)
     NOT >R              ( complement and save high)
     NOT 1 UM+           ( complement low part)
     R> + ;              ( add carry to high)

: -  ( n1 n2 -- n1-n2 )
     ( Subtraction.)
     NEGATE + :

: ABS     ( n -- n )
     ( Return the absolute value of n.)
     DUP 0<              ( negate if negative)
     IF NEGATE THEN ;
```

More Comparison

The primitive comparison word in eForth is ?branch and 0<. However, ?branch is at such a low level that it can not be readily used in high level Forth code. ?branch is secretly compiled into the high level Forth words by IF as an address literal. For all intentions and purposes, we can consider IF the equivalent of ?branch. When IF is encountered, the top item on the data stack is considered a logic flag. If it is true (non-zero), the execution continues until ELSE, then jump to THEN, or to THEN directly if there is no ELSE clause.

The following logic words are constructed using the IF...ELSE...THEN structure with 0< and XOR. XOR is used as 'not equal' operator, because if the top two items on the data stack are not equal, the XOR operator will return a non-zero number, which is considered to be 'true'.

U< is used to compared two unsigned numbers. This operator is very important, especially in comparing addresses, as we assume that the addresses are unsigned numbers pointing to unique memory locations. The arithmetic comparison operator < cannot be used to determine whether one address is higher or lower than the other. Using < for address comparison had been the single cause of many failures in the annals of Forth.

```
: =  ( w w -- t )
     ( Return true if top two are equal.)
     XOR            ( compare all bits)
     IF 0 EXIT THEN ( return 0 if mismatch)
     -1 ;           ( match completely, return -1)

: U< ( u1 u2 -- t )
     ( Unsigned compare of top two items.)
     2DUP XOR 0<         ( compare sign bits)
     IF SWAP DROP   ( sign bit different)
          0< EXIT   ( t follows u2 )
     THEN
     - 0< ;         ( same sign, subtract )

: <  ( n1 n2 -- t )
     ( Signed compare of top two items.)
     2DUP XOR 0<         ( compare sign bits)
     IF DROP        ( sign bit different)
          0< EXIT   ( t follows n1 )
     THEN
     - 0< ;         ( same sign, subtract )
```

MAX retains the larger of the top two items on the data stack. Both numbers are assumed to be signed integers.

MIN retains the smaller of the top two items on the data stack. Both numbers are assumed to be signed integers.

WITHIN checks whether the third item on the data stack is within the range as specified by the top two numbers on the data stack. The range is inclusive as to the lower limit and exclusive to the upper limit. If the third item is within range, a true flag is returned on the data stack. Otherwise, a false flag is returned. All numbers are assumed to be unsigned integers.

```
: MAX     ( n1 n2 -- n )
     ( Return the greater of two top stack items.)
     2DUP <              ( if n1<n2 )
     IF SWAP THEN        ( drop n1)
     DROP ;              ( else drop n2)

: MIN     ( n1 n2 -- n )
     ( Return the smaller of top two stack items.)
     2DUP SWAP <         ( if n1>n2 )
     IF SWAP THEN        ( drop n1)
     DROP ;              ( else drop n2)

: WITHIN  ( u ul uh -- t )
     ( Return true if u is within the range of ul and uh; ul<=u<uh.)
     OVER - >R           ( distance between ul and uh)
     -                   ( distance between u and ul)
     R> U< ;        ( compare the distances)
```

More Math Words

This group of words provide a variety of multiplication and division functions. The most interesting feature of this word set is that they are all based on the primitive UM+ operator in the kernel. Building this word set in high level has the penalty that all math operations will be slow. However, since eForth needs these functions only in numeric I/O conversions, the performance of eForth itself is not substantially affected by them. Nevertheless, if an application requires lots of numeric computations, a few critical words in this word set should be recoded in assembly. The primary candidates for optimization are UM/MOD and UM*, because all other multiply and divide operators are derived from these two words.

UM/MOD and UM* are the most complicated and comprehensive division and multiplication operators. Once they are coded, all other division and multiplication operators can be derived easily. It has been a tradition in Forth coding that one solves the most difficult problem first, and all other problems are solved by themselves.

UM/MOD divides an unsigned double integer by an unsigned signal integer. It returns the unsigned remainder and unsigned quotient on the data stack.

```
: UM/MOD  ( udl udh u -- ur uq )
     ( Unsigned divide of a double by a single. Return mod and quotient.)
     2DUP U<
     IF NEGATE           ( negate u for subtraction)
          15 FOR         ( repeat 16 times for 16 bits)
               >R        ( save -u)
               DUP UM+   ( left shift udh)
               >R >R DUP UM+  ( left shift udl)
               R> + DUP  ( add carry to udh)
               R> R@ SWAP     ( retrieve -u)
               >R UM+    ( subtract u from udh)
               R> OR     ( a borrow?)
               IF >R DROP     ( yes, add a bit to quotient)
                     1 + R>
               ELSE DROP      ( no borrow)
               THEN R>   ( retrieve -u)
          NEXT           ( repeat for 16 bits)
          DROP SWAP EXIT ( return remainder and quotient)
     THEN DROP 2DROP     ( overflow, return -1's)
     -1 DUP ;
```

M/MOD divides a signed double integer by a signed signal integer. It returns the signed remainder and signed quotient on the data stack. The signed division is floored towards negative infinity.

/MOD divides a signed single integer by a signed integer. It returns the signed remainder and quotient.

MOD is similar to /MOD, except that only the signed remainder is returned.

/ is also similar to /MOD, except that only the signed quotient is returned.

In most advanced microprocessors like 8086, all these division operations can be performed by the CPU as native machine instructions. The user can take advantage of these machine instructions by recoding these Forth words in machine code.

```
: M/MOD   ( d n -- r q )
     ( Signed floored divide of double by single. Return mod and quotient.)
     DUP 0<             ( n negative?)
     DUP >R             ( save a copy of flag)
     IF   NEGATE >R     ( take abs of n )
          DNEGATE R>    ( negative d also)
     THEN >R
     DUP 0<             ( if d is negative)
     IF R@ + THEN       ( floor it)
     R> UM/MOD          ( now divide)
     R>                 ( if n is negative)
     IF SWAP NEGATE SWAP THEN ;    ( negative remainder also)

: /MOD    ( n1 n2 -- r q )
     ( Signed divide. Return mod and quotient.)
     OVER 0<            ( sign extend n1)
     SWAP M/MOD ;       ( floored divide)

: MOD     ( n n -- r )
     ( Signed divide. Return mod only.)
     /MOD DROP ;        ( discard quotient)

: /  ( n n -- q )
     ( Signed divide. Return quotient only.)
     /MOD SWAP DROP ;   ( discard remainder)
```

UM* is the most complicated multiplication operation. Once it is coded, all other multiplication words can be derived from it.

UM* multiplies two unsigned single integers and returns the unsigned double integer product on the data stack.

M* multiplies two signed single integers and returns the signed double integer product on the data stack.

* multiplies two signed single integers and returns the signed single integer product on the data stack.

Again, advanced CPU's generally have these multiplication operations as native machine instructions. The user should take advantage of these resources to enhance the eForth system.

```
: UM*     ( u1 u2 -- ud )
     ( Unsigned multiply. Return double product.)
     0 SWAP             ( u1 sum u2 )
     15 FOR             ( repeat for 16 bits)
          DUP UM+ >R >R  ( left shift u2)
          DUP UM+        ( left shift sum)
          R> +           ( add carry to u2)
          R>             ( carry shifted out of u2?)
          IF >R OVER UM+      ( add u1 to sum)
               R> +      ( carry into u2)
          THEN
     NEXT               ( repeat 16 time to form ud)
     ROT DROP ;         ( discard u1)

: *  ( n n -- n )
     ( Signed multiply. Return single product.)
     UM* DROP ;         ( retain only low part)

: M* ( n1 n2 -- d )
     ( Signed multiply. Return double product.)
     2DUP XOR 0< >R      ( n1 n2 have same sign?)
     ABS SWAP ABS UM*    ( multiply absolutes)
     R> IF DNEGATE THEN ;     ( negate if signs are different)
```

Scaling Words

Forth is very close to the machine language that it generally only handles integer numbers. There are floating point extensions on many more sophisticated Forth systems, but they are more exceptions than rules. The reason that Forth has traditionally been an integer language is that integers are handled faster and more efficiently in the computers, and most technical problems can be solved satisfactorily using integers only. A 16-bit integer has the dynamic range of 110 dB which is far more than enough for most engineering problems. The precision of a 16-bit integer representation is limited to one part in 65535, which could be inadequate for small numbers. However, the precision can be greatly improved by scaling; i.e., taking the ratio of two integers. It was demonstrated that pi, or any other irrational numbers, can be represented accurately to 1 part in 100,000,000 by a ratio of two 16-bit integers.

The scaling operators */MOD and */ are useful in scaling number n1 by the ratio of n2/n3. When n2 and n3 are properly chosen, the scaling operation can preserve precision similar to the floating point operations at a much higher speed. Notice also that in these scaling operations, the intermediate product of n1 and n2 is a double precision integer so that the precision of scaling is maintained.

*/MOD multiplies the signed integers n1 and n2, and then divides the double integer product by n3. It in fact is ratioing n1 by n2/n3. It returns both the remainder and the quotient.

*/ is similar to */MOD except that it only returns the quotient.

```
: */MOD   ( n1 n2 n3 -- r q )
     ( Multiply n1 and n2, then divide by n3. Return mod and quotient.)
     >R M*              ( n1*n2)
     R> M/MOD ;         ( n1*n2/n3 with remainder)

: */ ( n1 n2 n3 -- q )
     ( Multiply n1 by n2, then divide by n3. Return quotient only.)
     */MOD              ( n1*n2/n3)
     SWAP DROP ;        ( discard remainder)
```

Memory Alignment Words

The most serious problem in porting system from one computer to another is that different computers have different sizes for their addresses and data. We generally classify computers as 8, 16, 32, ... , bit machines, because they operate on data of these various sizes. It is thus difficult to port a single programming model as eForth to all these computers. In eForth, a set of memory alignment words helps to make it easier to port the eForth model to different machines.

We assume that the target computer can address it memory in 8 bit chunks (bytes). The natural width of data best handled by the computer is thus a multiple of bytes. A unit of such data is a cell. An 16 bit machine handles data in 2 byte cells, and a 32 bit machine handles data in 4 byte cells.

CELL+ increments the memory address by the cell size in bytes, and CELL- decrements the memory address by the cell size. CELLS multiplies the cell number on the stack by the cell size in bytes. These words are very useful in converting a cell offset into a byte offset, in order to access integers in a data array.

ALIGNED converts an address on the stack to the next cell boundary, to help accessing memory by cells.

```
: CELL+   ( a -- a )
     ( Add cell size in byte to address.)
     2 + ;

: CELL-   ( a -- a )
     ( Subtract cell size in byte from address.)
     -2 + ;

: CELLS   ( n -- n )
     ( Multiply tos by cell size in bytes.)
     2 * ;

: ALIGNED ( b -- a )
     ( Align address to the cell boundary.)
     DUP 0 2 UM/MOD      ( divide b by 2)
     DROP DUP            ( save remainder)
     IF 2 SWAP - THEN    ( add 1 if remainder is 1)
     + ;
```

Special Characters

The blank character (ASCII 32) is special in eForth because it is the most often used character to delimit words in the input stream and the most often used character to format the output strings.   It is used so often that it is advantageous to define an unique word for it.   BL simply returns the number 32 on the data stack.

>CHAR is very important in converting a non-printable character to a harmless 'underscare' character(ASCII 95).   As eForth is designed to communicate with a host computer through the serial I/O device, it is important that eForth will not emit control characters to the host and causes unexpected behavior on the host computer.   >CHAR thus filters the characters before they are sent out by EMIT.

```
: BL ( -- 32 )
     ( Return 32, the blank character.)
     32 ;

: >CHAR   ( c -- c )
     ( Filter non-printing characters.)
     127 AND DUP         ( mask off the MSB bit)
     127 BL WITHIN       ( if it is a control character)
     IF DROP 95 THEN ;   ( replace it by an underscore)
```

Managing Data Stack

The data stack is one of the most important resources in a Forth system. The user has to have the complete knowledge of whets on the stack in order to be sure that his program is operating properly. The Forth words discussed so far allows the user to access only the top 3 items on the stack. In coding small modules, the user is not expected to access stack items more than 3 levels deep. However, in running significant applications, there are occasions when the user needs to reach deeper into the stack.

In eForth there is an utility word .S which dumps non-destructively the contents of the data stack for the user to examine. The stack words DEPTH and PICK are included in the eForth system mainly to support this stack dump utility.

DEPTH returns the number of items currently on the data stack to the top of the stack. PICK takes a number n off the data stack and replaces it with the n'th item on the data stack. The number n is 0-based; i.e., the top item is number 0, the next item is number 1, etc. Therefore, 0 PICK is equivalent to DUP, and 1 PICK is equivalent to OVER.

As discussed in an earlier section, DUP, SWAP, OVER and ROT should be able to handle most situations in Forth programming. If you find yourself in a situation that you have to use PICK, something is wrong and you should look more carefully in your code to see if there are ways to simplify your code.

```
: DEPTH   ( -- n )
     ( Return the depth of the data stack.)
     SP@                ( current stack pointer)
     SP0 @ SWAP -       ( distance from stack origin)
     2 / ;              ( divide by bytes/cell)

: PICK    ( ... +n -- ... w )
     ( Copy the nth stack item to tos.)
     1 + CELLS          ( bytes below tos)
     SP@ + @ ;          ( fetch directly from stack)
```

Memory access

Here are three useful memory operators. +! increments the contents of a memory location by an integer on the stack. 2! and 2@ store and fetch double integers to and from memory.

There are three buffer areas used often in the eForth system. HERE returns the address of the first free location above the code dictionary, where new words are compiled. PAD returns the address of the text buffer where numbers are constructed and text strings are stored temporarily. TIB is the terminal input buffer where input text string is held.

@EXECUTE is a special word supporting the vectored execution words in eForth. It takes the token address stored in a memory location and executes the token. It is used extensively to execute the vectored words in the user area.

```
: +! ( n a -- )
     ( Add n to the contents at address a.)
     SWAP OVER @         ( get contents in a)
     +                   ( add n to it)
     SWAP ! ;            ( store the sum back)

: 2! ( d a -- )
     ( Store the double integer to address a.)
     SWAP OVER !         ( store high part)
     CELL+ ! ;           ( store low part)

: 2@ ( a -- d )
     ( Fetch double integer from address a.)
     DUP CELL+ @         ( fetch low part first)
     SWAP @ ;            ( fetch high part)

: HERE    ( -- a )
     ( Return the top of the code dictionary.)
     CP @ ;              ( top of code dictionary)

: PAD     ( -- a )
     ( Return the address of a temporary buffer.)
     HERE                ( top of code dictionary)
     80 + ;              ( leave 80 byte gap)

: TIB     ( -- a )
     ( Return the address of the terminal input buffer.)
     #TIB CELL+ @ ;      ( 1 cell after #TIB)

: @EXECUTE     ( a -- )
     ( Execute vector stored in address a.)
     @                   ( fetch the execution address)
     ?DUP                ( do nothing if addr is 0)
     IF EXECUTE THEN ;   ( execute it only if non-zero)
```

Memory Array and String Words

A memory array is generally specified by a starting address and its length in bytes. In a string, the first byte is a count byte, specifying the number of bytes in the following string. This is called a counted string. String literals in the colon definitions and the name strings in the name dictionary are all represented by counted strings. Following are special words which handles memory arrays and strings.

COUNT converts a string array address to the address-length representation of a counted string.

CMOVE copies a memory array from one location to another.

FILL fills a memory array with the same byte.

Arrays and strings are generally specified by the address of the first byte in the array or string, and the byte length. This specification of course is the consequence that the memory is byte addressable. In a CPU which address memory in cells, these words must be defined in terms of an artificial byte space.

```
: COUNT   ( b -- b+1 +n )
     ( Return count byte of a string and add 1 to byte address.)
     DUP 1 +            (  increment b)
     SWAP C@ ;          ( get byte from b)

: CMOVE   ( b1 b2 u -- )
     ( Copy u bytes from b1 to b2.)
     FOR                ( repeat u+1 times)
          AFT           ( skip to THEN the first time)
               >R DUP C@     ( fetch from source)
               R@ C!    ( store to destination)
               1 +      ( incerement source address)
               R> 1 +   ( increment destination address)
          THEN          ( repeat after AFT)
     NEXT 2DROP ;       ( done. discrad addresses)

: FILL    ( b u c -- )
     ( Fill u bytes of character c to area beginning at b.)
     SWAP               ( get u to the top)
     FOR                ( loop u+1 times)
          SWAP          ( get b to top)
          AFT           ( skip to THEN)
               2DUP C!  ( store c to b)
               1 +      ( increment b)
          THEN          ( repeat after AFT)
     NEXT 2DROP ;       ( done. discard b and c)
```

-TRAILING removes the trailing white space characters from the end of a string. White space characters include all the non-printable characters below ASCII 32. This word allows eForth to process text lines in files downloaded from a host computer.   It conveniently eliminates carriage-returns, life-feeds, tabs and spaces at the end of the text lines.

PACK$ is an important string handling word used by the text interpreter.   It copies a text string from on location to another.   In the target area,   the string is converted to a counted string by adding a count byte before the text of the string.   This word is used to build the name field of a new word at the bottom of the name dictionary.   PACK$ is designed so that it can pack bytes into cells in a cell addressable machine.

A cheap way to implement eForth on a cell addressable machine is to equate cell addresses to byte addresses, and to store one byte in a cell.   This scheme is workable, but very inefficient in the memory utilization.   PACK$ is a tool which helps the implementor to bridge the gap.

```
: -TRAILING    ( b u -- b u )
     ( Adjust the count to eliminate trailing white space.)
     FOR                ( scan u+1 characters)
          AFT           ( skip the first loop)
               BL       ( blank for comparison)
               OVER R@ + C@ < ( compare char at the end)
               IF R> 1 +     ( non-white space, exit loop)
                    EXIT     ( with adjusted count)
               THEN
          THEN          ( else continue scanning)
     NEXT 0 ;           ( reach the beginning of b)

: PACK$   ( b u a -- a )
     ( Build a counted string with u characters from b. Null fill.)
     DUP >R             ( save address of word buffer )
     2DUP C!            ( store the character count first )
     1 + 2DUP +         ( go to the end of the string )
     0  SWAP !          ( fill the end with 0's )
     SWAP CMOVE         ( copy the string over )
     R> ;               ( leave only word buffer address )
```

6. Text Interpreter

The text interpreter is also called the outer interpreter in Forth. It is functionally equivalent to an operating system in a conventional computer. It accepts command similar to English entered by a user, and carries out the tasks specified by the commands. As an operating system, the text interpreter must be complicated, because of all the things it has to do. However, because Forth employs very simple syntax rules, and has very simple internal structures, the Forth text interpreter is much simpler that conventional operating systems. It is simple enough that we can discuss it completely in a single chapter, admitted that this is a long chapter.

Let us summarize what a text interpreter must do:

```
     Accept text input from a terminal
     Parse out commands from input text
     Search dictionary
     Execute commands
     Translate numbers into binary
     Display numbers in text form
     Handle errors gracefully
```

Forth allows us to build and integrate these required functions gradually in modules. All the modules finally fall into their places in the word QUIT, which is the text interpreter itself.

You might want to look up the code of QUIT first and see how the modules fit together. A good feeling about the big picture will help you in the study of the smaller modules. Nevertheless, we will doggedly follow the loading order of the source code, and hope that you will not get lost too far in the progress.

Numeric Output

Forth is interesting in its special capabilities in handling numbers across the man-machine interface. It recognizes that the machine and the human prefer very different representations of numbers. The machine prefers the binary representation, but the human prefers decimal Arabic digital representations. However, depending on circumstances, the human may want numbers to be represented in other radices, like hexadecimal, octal, and sometimes binary.

Forth solves this problem of internal (machine) versus external (human) number representations by insisting that all numbers are represented in the binary form in the CPU and in memory. Only when numbers are imported or exported for human consumption are they converted to the external ASCII representation. The radix of external representation is controlled by the radix value stored in the user variable BASE.

Since BASE is a user variable, the user can select any reasonable radix for entering numbers into the computer and format ting numbers to be shown to the user. Most programming languages can handle a small set of radices, like decimal, octal, hexadecimal and binary.

DIGIT converts an integer to a digit.

EXTRACT extracts the least significan digit from a number n. n is divided by the radix in BASE and returned on the stack.

```
: DIGIT ( u -- c )
     ( Convert digit u to a character.)
     9 OVER <           ( if u is greater than 9)
     7 AND +            ( add 7 to make it A-F)
     48 + ;             ( add ASCII 0 for offset)

: EXTRACT ( n base -- n/base c )
     ( Extract the least significant digit from n.)
     0 SWAP UM/MOD      ( divide n by base)
     SWAP DIGIT ;       ( convert remainder to a digit)
```

Number Formatting Tools

The output number string is built below the PAD buffer. The least significant digit is extracted from the integer on the top of the data stack by dividing it by the current radix in BASE. The digit thus extracted are added to the output string backwards from PAD to the low memory. The conversion is terminated when the integer is divided to zero. The address and length of the number string are made available by #> for outputting.

An output number conversion is initiated by <# and terminated by #>. Between them, # converts one digit at a time, #S converts all the digits, while HOLD and SIGN inserts special characters into the string under construction. This set of tokens is very versatile and can handle many different output formats.

```
: <# ( -- )
     ( Initiate the numeric output process.)
     PAD HLD ! ;         ( use PAD as the number buffer)

: HOLD    ( c -- )
     ( Insert a character into the numeric output string.)
     HLD @               ( get the digit pointer in HLD)
     1 - DUP HLD !       ( decrement HLD)
     C! ;                ( store c where HLD pointed to)

: #  ( u -- u )
     ( Extract one digit from u and append the digit to output string.)
     BASE @              ( get current base)
     EXTRACT             ( extract one digit from u)
     HOLD ;              ( save digit to number buffer)

: #S ( u -- 0 )
     ( Convert u until all digits are added to the output string.)
     BEGIN               ( begin converting all digits)
          # DUP          ( convert one digit)
     WHILE               ( repeat until u is divided to 0)
     REPEAT ;

: SIGN    ( n -- )
     ( Add a minus sign to the numeric output string.)
     0<                  ( if n is negative)
     IF 45 HOLD THEN ;   ( add a - sign to number string)

: #> ( w -- b u )
     ( Prepare the output string to be TYPE'd.)
     DROP                ( discard w)
     HLD @               ( address of last digit)
     PAD OVER - ;        ( return address of 1st digit and length)
```

Number Output Words

With the number formatting word set as shown above, one can format numbers for output in any form desired. The free output format is a number string preceded by a single space. The fix column format displays a number right-justified in a column of pre-determined width. The tokens ., U., and ? use the free format. The tokens .R and U.R use the fix format.

```
: str     ( n -- b u )
     ( Convert a signed integer to a numeric string.)
     DUP >R              ( save a copy for sign)
     ABS                 ( use absolute of n)
     <# #S               ( convert all digits)
     R> SIGN             ( add sign from n)
     #> ;                ( return number string addr and length)

: HEX     ( -- )
     ( Use radix 16 as base for numeric conversions.)
     16 BASE ! ;

: DECIMAL ( -- )
     ( Use radix 10 as base for numeric conversions.)
     10 BASE ! ;

: .R ( n +n -- )
     ( Display an integer in a field of n columns, right justified.)
     >R str              ( convert n to a number string)
     R> OVER - SPACES    ( print leading spaces)
     TYPE ;              ( print number in +n column format)

: U.R     ( u +n -- )
     ( Display an unsigned integer in n column, right justified.)
     >R                  ( save column number)
     <# #S #> R>         ( convert unsigned number)
     OVER - SPACES       ( print leading spaces)
     TYPE ;              ( print number in +n columns)

: U. ( u -- )
     ( Display an unsigned integer in free format.)
     <# #S #>            ( convert unsigned number)
     SPACE               ( print one leading space)
     TYPE ;              ( print number)

: .  ( w -- )
     ( Display an integer in free format, preceeded by a space.)
     BASE @ 10 XOR       ( if not in decimal mode)
     IF U. EXIT THEN     ( print unsigned number)
     str SPACE TYPE ;    ( print signed number if decimal)

: ?  ( a -- )
     ( Display the contents in a memory cell.)
     @ . ;               ( very simple but useful command)
```

Numeric Input

The Forth text interpreter also handles the number input to the system. It parses words out of the input stream and try to execute the words in sequence. When the text interpreter encounters a word which is not the name of a token in the dictionary, it then assumes that the word must be a number and attempts to convert the ASCII string to a number according to the current radix. When the text interpreter succeeds in converting the string to a number, the number is pushed on the data stack for future use if the text interpreter is in the interpreting mode. If it is in the compiling mode, the text interpreter will compile the number to the code dictionary as an integer literal so that when the token under construction is later executed, this literal integer will be pushed on the data stack.

If the text interpreter fails to convert the word to a number, there is an error condition which will cause the text interpreter to abort, posting an error message to the user, and then wait for the user's next line of commands.

Only two words are needed in eForth to handle input of single precision integer numbers.

DIGIT? converts a digit to its numeric value according to the current base, and NUMBER? converts a number string to a single integer. NUMBER? is vectored through 'NUMBER to convert numbers.

```
: DIGIT? ( c base -- u t )
     ( Convert a character to its numeric value. A flag indicates success.)
     >R                 ( save radix )
     [ CHAR 0 ] LITERAL -     ( character offset from digit 0 )
     9 OVER <           ( is offset greater than 9? )
     IF   7 -           ( yes.  offset it from digit A )
          DUP           ( n n )
          10 <          ( if n<10, the flag will be -1, and )
          OR            ( OR with n, result will be -1 )
     THEN               ( if n>10, the flag will be 0 and )
     DUP                ( OR result will still be n )
     R> U< ;            ( if n=/>radix, the digit is not valid )
```

NUMBER? converts a string of digits to a single integer. If the first character is a $ sign, the number is assumed to be in hexadecimal. Otherwise, the number will be converted using the radix value stored in BASE. For negative numbers, the first character should be a - sign. No other characters are allowed in the string. If a non-digit character is encountered, the address of the string and a false flag are returned. Successful conversion returns the integer value and a true flag. If the number is larger than 2**n, where n is the bit width of the single integer, only the modulus to 2**n will be kept.

```
: NUMBER? ( a -- n T, a F )
     ( Convert a number string to integer. Push a flag on tos.)
     BASE @ >R                ( save the current radix in BASE )
     0 OVER COUNT             ( a 0 a+1 n --, get length of the string )
     OVER C@                  ( get first digit )
     [ CHAR $ ] LITERAL =     ( is it a $ for hexadecimal base? )
     IF   HEX                 ( use hexadecimal base and adjust string )
          SWAP 1 +            ( a 0 n a+2 )
          SWAP 1 -            ( a 0 a+2 n-1 )
     THEN OVER C@             ( get the next digit )
     [ CHAR - ] LITERAL =     ( is it a - sign? )
     >R ( a 0 b' n')          ( save the - flag )
     SWAP R@ -                ( adjust address b )
     SWAP R@ + ( a 0 b" n")   ( adjust count n )
     ?DUP                     ( do we still have digits left? )
     IF                       ( yes.  do conversion )
          1 - ( a 0 b" n"-1)  ( adjust loop count for FOR-NEXT loop )
          FOR  DUP >R         ( save address b )
               C@             ( get one digit )
               BASE @ DIGIT?  ( convert it according to current radix )
          WHILE SWAP          ( it is a valid digit )
               BASE @ * +     ( multiply it by radix and add to sum )
               R> 1 +         ( increment b, pointing to next digit )
          NEXT                ( loop back to convert the next digit )
               R@ ( ?sign)    ( completely convert the string.get sign )
               NIP ( a sum )  ( discard string address b )
               IF NEGATE THEN ( negate the sum if - flag is true )
               SWAP           ( sum a )
          ELSE                ( a non-digit was encountered )
               R> R>          ( a sum b" b index )
               2DROP          ( a sum b" )
               2DROP 0        ( a 0 , conversion failed )
          THEN      DUP       ( /sum a a/ if success; else /a 0/ 0 )
     THEN
     R> ( n ?sign)            ( retrieve the sign flag )
     2DROP                    ( discard garbage )
     R> BASE ! ;              ( restore radix )
```

Serial I/O Words

The eForth system assumes that the system will communicate with its environment only through a serial I/O interface.   To support the serial I/O, only three words are needed:

?KEY returns a false flag if no character is pending on the receiver. If a character is received, the character and a true flag are returned.   This word is more powerful than that usually defined in most Forth systems because it consolidate the functionality of KEY into ?KEY.   It simplifies the coding of the machine dependent I/O interface.

KEY will execute ?KEY continually until a valid character is received and the character is returned.   EMIT sends a character out throughout the transmit line.

?KEY and EMIT are vectored through '?KEY and 'EMIT, so that their function can be changed dynamically at run time.   Normally ?KEY executes ?RX and EMIT executes TX!.   ?RX and TX! are machine dependent kernel words.   Vectoring the I/O words allows the eForth system to changes its I/O channels dynamically and still uses all the existing tools to handle input and output transactions.

```
: ?KEY    ( -- c T | F )
     ( Return input character and true, or a false if no input.)
     '?KEY @EXECUTE ;

: KEY     ( -- c )
     ( Wait for and return an input character.)
     BEGIN     ?KEY      ( wait until a key is pressed)
     UNTIL ;

: EMIT    ( c -- )
     ( Send a character to the output device.)
     'EMIT @EXECUTE ;

: NUF?    ( -- t )
     Return false if no input, else pause and if CR return true.
     ?KEY DUP            ( wait for a key-stroke)
     IF   2DROP KEY
          13 =           ( return true if key is CR)
     THEN ;
```

Derived I/O Words

All I/O words are derived from ?KEY, KEY and EMIT. The following set defined in eForth is particularly useful in normal programming:

SPACE outputs a blank space character.

SPACES output n blank space characters.

CR outputs a carriage-return and a line-feed.

PACE outputs an ASCII 11 character to acknowledge lines received during file downloading.

NUF? returns a false flag if no character is pending in the input buffer. After receiving a character, pause and wait for another character. If this character is CR, return a true flag; otherwise, return false. This word is very useful in user interruptable routines.

TYPE outputs n characters from a string in memory.

```
: PACE    ( -- )
     ( Send a pace character for the file downloading process.)
     11 EMIT ;           ( 11 is the pace character)

: SPACE   ( -- )
     ( Send the blank character to the output device.)
     BL EMIT ;           ( send out blank character)

: SPACES  ( n -- )
     ( Send n spaces to the output device.)
     SWAP 0 MAX          ( avoid negative n )
     FOR  AFT            ( skip first loop)
               SPACE     ( emit one space)
          THEN
     NEXT DROP ;         ( repeat till done)

: TYPE    ( b u -- )
     ( Output u characters from b.)
     FOR  AFT            ( repeat u times)
               DUP C@ EMIT    ( get one byte and emit it)
               1 +       ( increment b)
          THEN
     NEXT DROP ;         ( repeat until done)

: CR ( -- )
     ( Output a carriage return and a line feed.)
     15 EMIT             ( send carriage-return)
     10 EMIT ;           ( and a line-feed)
```

String Literal Words

String literals are data structures compiled in colon definitions, in-line with the tokens. A string literal must start with a string token which knows how to handle the following string at the run time. Let us show two examples of the string literals:

```
: xxx     ...   " A compiled string"  ...   ;
: yyy     ...   ." An output string"  ...   ;
```

In xxx, " is an immediate word which compiles the following string as a string literal preceded by a special token $"|. When $"| is executed at the run time, it returns the address of this string on the data stack. In yyy, ." compiles a string literal preceded by another token ."|, which prints the compiled string to the output device.

Both $"| and ."| use the word do$, which retrieve the address of a string stored as the second item on the return stack. do$ is a bit difficult to understand, because the starting address of the following string is the second item on the return stack. This address is pushed on the data stack so that the string can be accessed. This address must be changed so that the address interpreter will return to the token right after the compiled string. This address will allow the address interpreter to skip over the string literal and continue to execute the token list as intended.

```
: do$     ( -- a )
     ( Return the address of a compiled string.)
     R>                  ( this return addres must be preserved)
     R@                  ( get address of the compiled string)
     R>                  ( get another copy)
     COUNT + ALIGNED >R  ( replace it with addr after string)
     SWAP                ( get saved address to top)
     >R ;                ( restore the saved retrun address)

: $"|     ( -- a )
     ( Run time routine compiled by $". Return address of a compiled string.)
     do$ ;               ( return string address only)

: ."|     ( -- )
     ( Run time routine of ." . Output a compiled string.)
     do$                 ( get string address)
     COUNT TYPE ;        ( print the compiled string)
```

Word Parser

Parsing is always thought of as a very advanced topic in computer sciences. However, because Forth uses very simple syntax rules, parsing is easy. Forth source code consists of words, which are ASCII strings separated by spaces and other white space characters like tabs, carriage returns, and line feeds. The text interpreter scans the source code, isolates words and interprets them in sequence. After a word is parsed out of the input text stream, the text interpreter will 'interpret' it--execute it if it is a token, compile it if the text interpreter is in the compiling mode, and convert it to a number if the word is not a Forth token.

PARSE scans the source string in the terminal input buffer from where >IN points to till the end of the buffer, for a word delimited by character c. It returns the address and length of the word parsed out. PARSE calls 'parse' to do the detailed works.

PARSE is used to implement many specialized parsing words to perform different source code handling functions. These words, including (, \, CHAR, WORD, and TOKEN are discussed in the next section.

```
: PARSE ( c -- b u \ <string> )
     ( Scan input stream and return counted string delimited by c.)
     >R                  ( save the delimiting character )
     TIB >IN @ +         ( address in TIB to start parsing )
     #TIB @ >IN @ -      ( length of remaining string in TIB )
     R> parse            ( parse the desired string )
     >IN +! ;            ( move parser pointer to end of string )
```

'parse'   (b1 u1 c -- b2 u2 n)   From the source string starting at b1 and of u1 characters long, parse out the first word delimited by character c.   Return the address b2 and length u2 of the word just parsed out and the difference n between b1 and b2.   Leading delimiters are skipped over.   'parse' is used by PARSE.

```
: parse ( b u c -- b u delta \ <string> )
     ( Scan string delimited by c. Return found string and its offset.)
     temp !                   ( save delimiter in temp )
     OVER >R DUP              ( b u u -- )
     IF                       ( if string length u=0, nothing to parse )
          1 -                 ( u>0, decrement it for FOR-NEXT loop )
          temp @ BL =         ( is delimiter a space? )
          IF                  ( b u' --, skip over leading spaces )
               FOR  BLANK
                    OVER C@   ( get the next character )
                    - 0<      ( is it a space? )
                    INVERT
               WHILE
               NEXT           ( b -- , if space, loop back and scan further)
                    R> DROP   ( end of buffer, discard count )
                    0 DUP EXIT     ( exit with -- b 0 0 , end of line )
               THEN
               1 -            ( back up the parser pointer to non-space )
               R>             ( retrieve the length of remaining string )
          THEN
          OVER SWAP           ( b' b' u' -- , start parsing non-space chars )
          FOR
               temp @         ( get delimiter )
               OVER C@ -      ( get next character )
               temp @ BL =
               IF 0<
               ELSE 1 +
               THEN
          WHILE               ( if delimiter, exit the loop )
          NEXT                ( not delimiter, keep on scanning )
               DUP >R         ( save a copy of b at the end of the loop )
          ELSE                ( early exit, discard the loop count )
               R> DROP        ( discard count )
               DUP 1 + >R     ( save a copy of b'+1 )
          THEN
          OVER -              ( length of the parsed string )
          R>  R> -            ( and its offset in the buffer )
          EXIT
     THEN ( b u)
     OVER                     ( buffer length is 0 )
     R> - ;                   ( the offset is 0 also )
```

Parsing Words

.(types the following string till the next). It is used to output text to the terminal.
(ignores the following string till the next). It is used to place comments in source text.

\ ignores all characters till end of input buffer. It is used to insert comment lines in text.

CHAR parses the next word but only return the first character in this word. Get an ASCII character from the input stream.

WORD parses out the next word delimited by the ASCII character c. Copy the word to the top of the code dictionary and return the address of this counted string.

TOKEN parses the next word from the input buffer and copy the counted string to the top of the name dictionary. Return the address of this counted string.

```
: .(      ( -- )
     ( Output following string up to next ) .)
     [ CHAR ) ] LITERAL PARSE      ( parse the string until next ) )
     TYPE ; IMMEDIATE              ( type the string to terminal )

: (  ( -- )
     ( Ignore following string up to next ) . A comment.)
     [ CHAR ) ] LITERAL PARSE      ( parse the string until ) )
     2DROP ; IMMEDIATE             ( and ignore it as a comment )

: \ ( -- )
     ( Ignore following text till the end of line.)
     #TIB @ >IN !                  ( store the length of TIB in >IN )
     ; IMMEDIATE                   ( in effect ignore the rest of a line )

: CHAR ( -- c )
     ( Parse next word and return its first character.)
     BL PARSE                      ( get the next string )
     DROP C@ ;                     ( return the code of the 1st character )

: TOKEN ( -- a \ <string> )
     ( Parse a word from input stream and copy it to name dictionary.)
     BL PARSE                      ( parse out next space delimited string )
     31 MIN                        ( truncate it to 31 characters )
     NP @                          ( word buffer below name dictionary )
     OVER - 2 - PACK$ ;            ( copy parsed string to word buffer )

: WORD ( c -- a \ <string> )
     ( Parse a word from input stream and copy it to code dictionary.)
     PARSE                         ( parse out a string delimited by c )
     HERE PACK$ ;                  ( copy the string into the word buffer )
```

Dictionary Search

In eForth, headers of word definitions are linked into a name dictionary which is separated from the code dictionary. A header contains three fields: a token field holding the code address of the word, a link field holding the name field address of the previous header and a name field holding the name as a counted string. The name dictionary is a list linked through the link fields and the name fields. The basic searching function is performed by the word 'find'. 'find' follows the linked list of names to find a name which matches a text string, and returns the address of the executable token and the name field address, if a match is found.

eForth allows multiple vocabularies in the name dictionary. A dictionary can be divided into a number of independently linked sublists through some hashing mechanism. A sublist is called a vocabulary. Although eForth itself contains only one vocabulary, it has the provision to build many vocabularies and allows many vocabularies to be searched in a prioritized order. The CONTEXT array in the user area has 8 cells and allows up to 8 vocabularies to be searched in sequence. A null entry in the CONTEXT array terminates the vocabulary search.

```
: NAME> ( na -- ca )
     ( Return a code address given a name address.)
     2 CELLS -               ( move to code pointer field )
     @ ;                     ( get code field address )

: SAME? ( a1 a2 u -- a1 a2 f \ -0+ )
     ( Compare u cells in two strings. Return 0 if identical.)
     FOR                     ( scan u+1 cells )
          AFT                ( skip the loop the first time )
               OVER          ( copy a1 )
               R@ CELLS + @  ( fetch one cell from a1 )
               OVER          ( copy a2 )
               R@ CELLS + @  ( fetch one cell from a2 )
               -             ( compare two cells )
               ?DUP
               IF            ( if they are not equal )
                    R> DROP  ( drop loop count )
                    EXIT     ( and exit with the difference as a flag )
               THEN
          THEN
     NEXT                    ( loop u times if strings are the same )
     0 ;                     ( then push the 0 flag on the stack )
```

find (a va -- ca na, a F) A counted string at a is the name of a token to be looked up in the dictionary. The last name field address of the vocabulary is stored in location va. If the string is found, both the token (code address) and the name field address are returned. If the string is not the name a token, the string address and a false flag are returned.

To located a word, one could follow the linked list and compare the names of defined tokens to the string to be searched. If the string matches the name of a token in the name dictionary, the token and the address of the name field are returned. If the string is not a defined token, the search will lead to either a null link or a null name field. In either case, the search will be terminated and a false flag returned. The false flag thus indicates that the token searched is not in this vocabulary.

```
: find ( a va -- ca na, a F )
     ( Search a vocabulary for a string. Return ca and na if succeeded.)
     SWAP                     ( va a )
     DUP C@ 2 / temp !        ( va a -- , get cell count )
     DUP @ >R                 ( va a -- , save 1st cell of string )
     CELL+ SWAP               ( a' va -- , compare string with names )
     BEGIN                    ( fast test, compare only 1st cells )
          @ DUP               ( a' na na -- )
          IF                  ( na=0 at the end of a vocabulary )
               DUP @          ( not end of vocabulary, test 1st cell )
               [ =MASK ] LITERAL AND    ( mask off lexicon bits )
               R@ XOR         ( compare with 1st cell in string )
               IF             ( 1st cells do not match )
                    CELL+ -1  ( try the next name in the vocabulary )
               ELSE      CELL+     ( get address of the 2nd cell )
                    temp @    ( get the length of string )
                    SAME?     ( string=name? )
               THEN
          ELSE                ( end of vocabulary )
               R> DROP        ( discard the 1st cell )
               SWAP CELL- SWAP     ( restore the string address )
               EXIT           ( exit with ca na, na=0 is false flag )
          THEN
     WHILE                    ( if the name does not match the string )
          CELL- CELL-         ( a' la --, move to next word in vocab)
     REPEAT                   ( repeat until vocabulary is exhausted )
     R> DROP NIP              ( a match is found, discard 1st and va )
     CELL-                    ( restore name field address )
     DUP NAME>                ( find code field address )
     SWAP ;                   ( reorder and return.  -- ca na )
```

'find' runs through the name dictionary very quickly because it first compares the length and the first character in the name field as a cell. In most cases of mismatch, this comparison would fail and the next name can be reached through the link field. If the first two characters match, then SAME? is invoked to compare the rest of the name field, one cell at a time. Since both the target text string and the name field are null filled to the cell boundary, the comparison can be performed quickly across the entire name field without worrying about the end conditions.

NAME? (a -- ca na, a F) Search all the vocabularies in the CONTEXT array for a name at address a. Return the token and a name address if a matched token is found. Otherwise, return the string address and a false flag. The CONTEXT array can hold up to 8 vocabulary links. However, a 0 which is not a valid vocabulary link in this array will terminate the searching. Changing the vocabulary links in this array and the order of these links will alter the searching order and hence the searching priority among the vocabularies.

```
: NAME? ( a -- ca na, a F )
     ( Search all context vocabularies for a string.)
     CONTEXT                  ( address of context vocabulary stack )
     DUP 2@ XOR               ( are two top vocabularies the same? )
     IF                       ( if not same )
          1 CELLS -           ( backup the vocab address for looping )
     THEN
     >R                       ( save the prior vocabulary address )
     BEGIN
          R>  CELL+           ( get the next vocabulary address )
          DUP >R              ( save it for next vocabulary )
          @  ?DUP             ( is this a valid vocabulary? )
     WHILE                    ( yes )
          find                ( find the word in this vocabulary )
          ?DUP                ( word found here? )
     UNTIL                    ( if not, go searching next vocabulary )
          R> DROP EXIT        ( word is found, exit with ca and na )
     THEN
     R> DROP                  ( word is not found in all vocabularies )
     0 ;                      ( exit with a false flag )
```

Text Input from Terminal

The text interpreter interprets source text stored in the terminal input buffer. To process characters from the input device, we need three special words to deal with backspaces and carriage return from the input device:

kTAP (b1 b2 b3 c -- b1 b4 b5) Process a character c received from terminal. b1 is the starting address of the input buffer. b2 is the end of the input buffer. b3 is the currently available address in the input buffer. c is normally stored into b3, which is bumped by 1 and becomes b5. In this case, b4 is the same as b2. If c is a carriage-return, echo a space and make b4=b5=b3. If c is a back-space, erase the last character and make b4=b2, b5=b3-1.

TAP (b1 b2 b3 c -- b1 b4 b5) Echo c to output device, store c in b3, and bump b3.

^H   (b1 b2 b3 -- b1 b2 b4) Process the back-space character. Erase the last character and decrement b3. If b3=b1, do nothing because you cannot backup beyond the beginning of the input buffer.

```
: ^H      ( bot eot cur -- bot eot cur )
     ( Backup the cursor by one character.)
     >R OVER                 ( bot eot bot -- )
     R@ < DUP                ( bot<cur ? )
     IF   [ CTRL H ] LITERAL  ( yes, echo backspace )
          'ECHO @EXECUTE
     THEN                    ( bot eot cur 0|-1 -- )
     R> + ;                  ( decrement cur, but not passing bot )

: TAP ( bottom eot currrent key -- bottom eot current )
     ( Accept and echo the key stroke and bump the cursor.)
     DUP                     ( duplicate character )
     'ECHO @EXECUTE          ( echo it to display )
     OVER C!                 ( store at current location )
     1 + ;                   ( increment current pointer )

: kTAP ( bot eot cur key -- bot eot cur )
     ( Process a key stroke, CR or backspace.)
     DUP 13 XOR              ( is key a return? )
     IF   [ CTRL H ] LITERAL  ( is key a backspace? )
          XOR
          IF   BL TAP        ( non of above, replace by space )
          ELSE      ^H       ( backup current pointer )
          THEN
          EXIT               ( done this part )
     THEN                    ( key is a return )
     DROP NIP                ( discard bot and eot )
     DUP ;                   ( duplicate cur )
```

QUERY is the word which accepts text input, up to 80 characters, from the input device and copies the text characters to the terminal input buffer.   It also prepares the terminal input buffer for parsing by setting #TIB to the character count and clearing >IN.

EXPECT   (b u --)   Accept u characters to a memory buffer starting at b.   The input is terminated upon receiving a carriage-return.   The number of characters actually received is stored in SPAN.   EXPECT is called by QUERY to put characters into the terminal input buffer.   However, EXPECT is useful by itself because one can use it to place input text anywhere in the memory.   QUERY and EXPECT are the two words most useful in accepting text from the terminal.

accept   (b u1 -- b u2)   Accept u1 characters to b.   u2   returned is the actual count of characters received.

```
: accept ( b u -- b u )
     ( Accept characters to input buffer. Return with actual count.)
     BEGIN
          2DUP XOR            ( b+u = current pointer? )
     WHILE                    ( no, get next character )
          KEY                 ( get one more character )
          DUP BL - 95 U<      ( is it printable? )
          IF   TAP            ( yes, accept and echo it )
          ELSE      'TAP @EXECUTE  ( no, process control code )
          THEN
     REPEAT                   ( repeat until buffer full )
     DROP                     ( discard current pointer )
     OVER - ;                 ( leave character count )

: EXPECT ( b u -- )
     ( Accept input stream and store count in SPAN.)
     'EXPECT @EXECUTE         ( execute accept )
     SPAN !                   ( store character count in SPAN )
     DROP ;                   ( discard eot address )

: QUERY ( -- )
     ( Accept input stream to terminal input buffer.)
     TIB 80                   ( addr and size of terminal input buffer)
     'EXPECT @EXECUTE         ( execute 'accept' )
     #TIB !                   ( store number of characters received )
     DROP                     ( discard buffer address )
     0 >IN ! ;                ( initialized parsing pointer )
```

Error Handling

This error handling mechanism was first developed by Mitch Bradley in his ForthMacs and then adopted by the ANS Forth Standard. It is very simple yet very powerful in customizing system responses to many different error conditions.

CATCH setups a local error frame and execute the word referenced by the execution token ca. It returns a non-zero error code or a zero if no error occurred. As the assigned word at ca is executing, any error condition will execute THROW, which pushes an error code on the data stack, restore the return stack to the state before CATCH was executed, and execute the error handler stored in HANDLER. Since the error handler frame is saved on the return stack, many layers of safety nets can be laid down nested.

CATCH pushes SP and HANDLER on the return stack, saves RP in HANDLER, and then execute the token at ca. If no error occurred, HANDLER and SP are restored from the return stack and a 0 is pushed on the data stack.

THROW throws the system back to CATCH so that the error condition can be processed. CATCH is backtracked by restoring the return stack from the pointer stored in HANDLER and popping the old handler and SP off the error frame on the return stack.

```
: CATCH ( ca -- err#/0 )
     ( Execute word at ca and set up an error frame for it.)
     SP@ >R             ( save current stack pointer on return stack )
     HANDLER @ >R       ( save the handler pointer on return stack )
     RP@ HANDLER !      ( save the handler frame pointer in HANDLER )
     ( ca ) EXECUTE     ( execute the assigned word over this safety net )
     R> HANDLER !       ( normal return from the executed word )
                        ( restore HANDLER from the return stack )
     R> DROP            ( discard the saved data stack pointer )
     0 ;                ( push a no-error flag on data stack )

: THROW ( err# -- err# )
     ( Reset system to current local error frame an update error flag.)
     HANDLER @ RP!      ( expose latest error handler frame on return stack )
     R> HANDLER !       ( restore previously saved error handler frame )
     R> SWAP >R         ( retrieve the data stack pointer saved )
     SP!                ( restore the data stack )
     DROP
     R> ;               ( retrived err# )
```

NULL$ is the address of a string with a zero count. This address is used by ABORT and abort" to terminate the interpreting of the current command line. QUIT tests the address reported by CATCH. If this address is NULL$, the termination is normal and no error message will be issued. If CATCH reports a different address, QUIT will display the contents of the string at that address.

ABORT" is used only within a definition to compile an inline packed string terminated by the " double quote character. At run-time, if the flag is false, execute the sequence of words following the string. Otherwise, the string is displayed on the current output device, and execution is then passed to an error handling routine.

You have to study the code in QUIT carefully with this section to get a better understanding of the CATCH-THROW error handling mechanism.

```
CREATE NULL$  0 , 99 c, 111 c, 121 c, 111 c, 116 c, 101 c,
     ( Return address of a null string with zero count.)

: ABORT ( -- )
     ( Reset data stack and jump to QUIT.)
     NULL$              ( take address of NULL$ )
     THROW ;            ( and give it to current CATCH )

: abort" ( f -- )
     ( Run time routine of ABORT" . Abort with a message.)
     IF                 ( if flag is true, abort )
          do$           ( take address of next string )
          THROW         ( and give it to CATCH )
     THEN               ( if  flag is false, continue )
     do$ DROP           ( skip over the next string )
     ; COMPILE-ONLY
```

Let's look at how the CATCH-THROW pair is used. In QUIT, there is this indefinite loop:

```
BEGIN QUERY [ ' EVAL ] LITERAL          CATCH
?DUP UNTIL
```

QUERY get a line of text and CATCH causes EVAL to interpret the line. CATCH also sets up an error handling frame on the return stack and saves the return stack pointer in the user variable HANDLER. The error handling frame contains the current data stack pointer and the current contents in HANDLER. If no error occurred during EVAL, the error frame is popped off the return stack and a false flag is returned on the data stack. ?DUP UNTIL will loop back to QUERY and the interpretive process will continue.

While EVAL interprets the text, any word which decided that it detects an error condition and needs attention, it will execute THROW. THROW restores the return stack from the pointer stored in HANDLER, making the error handling frame available. THROW then restores HANDLER from the one saved in the error frame so that the error handling can be nested. The data stack pointer is also restored from the error frame. Now THROW passes the address of a error processing routine to the CATCH which built the error frame.

Here are some of the words which generate error conditions:

```
: ABORT   NULL$ THROW ;

: abort"   IF do$ THROW THEN do$ DROP ;

: ?STACK   DEPTH 0< IF $" underflow" THROW THEN ;

: $INTERPRET   ... 'NUMBER @EXECUTE  IF EXIT THEN THROW ;
```

$INTERPRET, ?STACK and abort" pass string addresses to THROW. The strings contains appropriate error messages to be displayed by the text interpreter. In QUIT, the words between UNTIL and AGAIN deal with the error conditions and then re-initialize the text interpreter.

Text Interpreter Loop

Text interpreter in Forth is like the operating system of a computer. It is the primary interface a user goes through to get the computer to do work. Since Forth uses very simple syntax rules--words are separated by spaces, the text interpreter is also very simple. It accepts a line of text from the terminal, parses out a word delimited by spaces, locates the token of this word in the dictionary and then executes it. The process is repeated until the source text is exhausted. Then the text interpreter waits for another line of text and interprets it again. This cycle repeats until the user is exhausted and turns off the computer.

In eForth, the text interpreter is encoded in the word QUIT. QUIT contains an infinite loop which repeats the QUERY EVAL phrase. QUERY accepts a line of text from the terminal and copies the text into the Terminal Input Buffer (TIB). EVAL interprets the text one word at a time till the end of the text line.

One of the unique features in eForth is its error handling mechanism. While EVAL is interpreting a line of text, there could exist many error conditions: a word is not found in the dictionary and it is not a number, a compile-only word is accidentally executed interpretively, and the interpretive process may bee interrupted by the words ABORT or abort". Wherever the error occurs, the text interpreter must be made aware of it so that it can recover gracefully from the error condition and continue on about the interpreting business.

$INTERPRET executes a word whose string address is on the stack. If the string is not a word, convert it to a number.

```
: $INTERPRET ( a -- )
     ( Interpret a word. If failed, try to convert it to an integer.)
     NAME?                    ( search dictionary for word just parsed )
     ?DUP                     ( is it a defined word? )
     IF   @                   ( yes.  examine the lexicon )
          [ =COMP ] LITERAL AND    ( is it a compile-only word? )
          ABORT" compile ONLY"     ( if so, abort with the proper message )
          EXECUTE EXIT        ( not compile-only, execute it and exit )
     THEN                     ( not defined in the dictionary )
     'NUMBER @EXECUTE         ( convert it to a number )
     IF EXIT THEN             ( exit if conversion is successful )
     THROW ;                  ( else generated the error condition )
```

Here are some of the nitty-gritty stuff on the text interpreter and how it actually works.

[activates the text interpreter by storing the execution address of $INTERPRET into the variable 'EVAL, which is executed in EVAL while the text interpreter is in the interpretive mode.

.OK prints the familiar 'ok' prompt after executing to the end of a line. 'ok' is printed only when the text interpreter is in the interpretive mode. While compiling, the prompt is suppressed.

?STACK checks for stack underflow. Abort if the stack depth is negative.

EVAL is the interpreter loop which parses tokens from the input stream and invokes whatever is in 'EVAL to handle that token, either execute it with $INTERPRET or compile it with $COMPILE.

```
: [ ( -- )
     ( Start the text interpreter.)
     [ ' $INTERPRET ] LITERAL      ( store $INTERPRET in 'EVAL )
     'EVAL !
     ; IMMEDIATE                   ( must be done even while compiling )

: .OK     ( -- )
     ( Display 'ok' only while interpreting.)
     [ ' $INTERPRET ] LITERAL
     'EVAL @ =
     IF ."  ok" THEN
     CR ;

: ?STACK  ( -- )
     ( Abort if the data stack underflows.)
     DEPTH 0<
     ABORT" underflow" ;

: EVAL ( -- )
     ( Interpret the input stream.)
     BEGIN
          TOKEN ( -- a )      ( parse a word and leave its address )
          DUP C@              ( is the character count 0? )
     WHILE                    ( no )
          'EVAL @EXECUTE      ( evaluate it )
          ?STACK              ( any stack error? overflow or underflow )
     REPEAT                   ( repeat until TOKEN gets a null string )
     DROP                     ( discard the string address )
     'PROMPT @EXECUTE ;       ( display the proper prompt, if any )
```

Operating System

Source code can be downloaded to eForth through the serial input device. The only precaution we have to take is that during file downloading, characters are not echoed back to the host computer. However, whenever an error occurred during downloading, it is more useful to resume echoing so that error messages can be displayed on the terminal. It is also convenient to send special pacing characters to the host to tell the host that a line of source code was received and processed correctly. The following words configure the eForth I/O vectors to have the proper behavior in normal terminal interaction and also during file downloading:

FILE turns off character echoing. After one line of text is processed correctly, a pacing character ASCII 11 is sent to the host. If an error occurred, send an ESC (ASCII 26) character. An error will also restore the I/O vectors into the terminal mode.

HAND resumes terminal interaction. Turn on character echoing, and send normal prompt message after a line is processed correctly.

```
: PRESET ( -- )
     ( Reset data stack pointer and the terminal input buffer. )
     SP0 @ SP!                ( initialize data stack )
     [ =TIB ] LITERAL
     #TIB CELL+ !             ( initialize terminal input buffer )
     ;

: XIO ( prompt echo tap -- )
     ( Reset the I/O vectors 'EXPECT, 'TAP, 'ECHO and 'PROMPT.)
     [ ' accept ] LITERAL 'EXPECT !     ( vector EXPECT )
     'TAP !                   ( init kTAP )
     'ECHO !                  ( init ECHO )
     'PROMPT ! ;                   ( init system prompt )

: FILE ( -- )
     ( Select I/O vectors for file download.)
     [ ' PACE ] LITERAL       ( send 11 for acknowledge )
     [ ' DROP ] LITERAL       ( do not echo characters )
     [ ' kTAP ] LITERAL       ( ignore control characters )
     XIO ;

: HAND ( -- )
     ( Select I/O vectors for terminal interface.)
     [ ' .OK  ] LITERAL       ( say 'ok' if all is well )
     'EMIT @                  ( echo characters )
     [ ' kTAP ] LITERAL       ( ignore control characters )
     XIO ;
```

CONSOLE initializes the serial I/O device for terminal interaction. ?KEY is vectored to ?RX and EMIT is vectored to TX!.

QUIT is the operating system, or a shell, of the eForth system. It is an infinite loop eForth will never get out. It uses QUERY to accept a line of commands from the terminal and then let EVAL parse out the tokens and execute them. After a line is processed, it displays 'ok' and wait for the next line of commands. When an error occurred during execution, it displays the command which caused the error with an error message. After the error is reported, it re-initializes the system using PRESET and comes back to receive the next line of commands.

Because the behavior of EVAL can be changed by storing either $INTERPRET or $COMPILE into 'EVAL, QUIT exhibits the dual nature of a text interpreter and a compiler.

```
CREATE    I/O  ' RX? , ' TX!
     ( Array to store default I/O vectors. )

: CONSOLE ( -- )
     ( Initiate terminal interface.)
     I/O 2@ 'KEY? 2!          ( get defaults from I/O )
     HAND ;                   ( keyboard input )

: QUIT ( -- )
     ( Reset return stack pointer and start text interpreter. )
     RP0 @ RP!                ( initialize the return stack )
     BEGIN
          [COMPILE] [         ( start text interpreter )
          BEGIN
               QUERY          ( get a line of commands from terminal )
               [ ' EVAL ] LITERAL
               CATCH          ( execute commands with error handler )
               ?DUP
          UNTIL ( a)          ( exit if an error occurred )
          'PROMPT @ SWAP      ( save the current prompt address )
          CONSOLE NULL$ OVER XOR   ( is error address=NULL$ ? )
          IF   SPACE          ( no.  display name of error word )
               COUNT TYPE
               ."  ?"         ( followed by a error code )
          THEN
          [ ' .OK ] LITERAL XOR    ( 'PROMPT has .OK ? )
          IF [ ERR ] LITERAL EMIT  ( no, send error code to file handler )
          THEN
          PRESET              ( reset the data stack )
     AGAIN ;                  ( go back get another command line )
```

7. eForth Compiler

After wading through the text interpreter, the Forth compiler will be an easy piece of cake, because the compiler uses almost all the modules used by the text interpreter. What the compile does, over and above the text interpreter, is to build various structures required by the new words we want to add to the existing system. Here is a list of these structures:

```
     Name headers
     Colon definitions
     Constants
     Variables
     User variables
     Integer literals
     String literals
     Address literals
     Control structures
```

A special concept of immediate words is difficult to grasp at first. It is required in the compiler because of the needs in building different data and control structures in a colon definition. To understand the Forth compiler fully, you have to be able to differential and relate the actions during compile time and actions taken during executing time. Once these concepts are clear, the whole Forth system will become transparent.

This set stage for enlightenment to strike.

Interpreter and Compiler

The Forth compiler is the twin brother of the Forth text interpreter. They share many common properties and use lots of common code. In eForth, the implementation of the compiler clearly reflects this special duality. Two interesting words [and] causes the text interpreter to switch back and forth between the compiler mode and interpreter mode of operation.

Since 'EVAL @EXECUTE is used in EVAL to process a token parsed out of a line of text, the contents in 'EVAL determines the behavior of the text interpreter. If $INTERPRET is stored in 'EVAL, as [does, the tokens are executed or interpreted. If we invoked] to store $COMPILE into 'EVAL, the token will not be executed, but added to the top of the code dictionary. This is exactly the behavior desired by the colon definition compiler in building a list of tokens in the code field of a new colon definition on the top of the code dictionary.

$COMPILE normally adds a token to the code dictionary. However, there are two exceptions it must handle. If the token parsed out of the input stream does not exist in the dictionary, the string will be converted to a number. If the string can be converted to an integer, the integer is then compiled into the code dictionary as an integer literal. The integer number is compiled into the code dictionary following the token doLIT. The other exception is that a token found in the dictionary could be an immediate word, which must be executed immediately, not compiled to the code dictionary. Immediate words are used to compile special structures in colon definitions.

```
: [ ( -- )
     [ ' $INTERPRET ] LITERAL
     'EVAL !             ( vector EVAL to $INTERPRET )
     ; IMMEDIATE         ( enter into text interpreter mode )

: ] ( -- )
     [ ' $COMPILE ] LITERAL
     'EVAL !             ( vector EVAL to $COMPILE )
     ;
```

Here is a group of words which support the compiler to build new words in the code dictionary.

' (tick) searches the next word in the input stream for a token in the dictionary. It returns the execution address of the token if successful. Otherwise, it displays an error message.

ALLOT allocates n bytes of memory on the top of the code dictionary. Once allocated, the compiler will not touch the memory locations.

, (comma) adds the execution address of a token on the top of the data stack to the code dictionary, and thus compiles a token to the growing token list of the word currently under construction.

COMPILE is used in a colon definition. It causes the next token after COMPILE to be added to the top of the code dictionary. It therefore forces the compilation of a token at the run time.

[COMPILE] acts similarly, except that it compiles the next word immediately. It causes the following word to be compiled, even if the following word is an immediate word which would otherwise be executed.

```
: '  ( -- ca )
     ( Search context vocabularies for the next word in input stream.)
     TOKEN NAME?
     IF EXIT THEN
     THROW ;

: ALLOT   ( n -- )
     ( Allocate n bytes to the code dictionary.)
     CP +1 ;             ( adjust code dictionary pointer)

: ,  ( w -- )
     ( Compile an integer into the code dictionary.)
     HERE DUP CELL+      ( align to cell boundary)
     CP ! ! ;            ( update code dictionary pointer)

: [COMPILE]    ( -- ; <string> )
     ( Compile the next immediate word into code dictionary.)
     ' , ; IMMEDIATE

: COMPILE ( -- )
     ( Compile the next address in colon list to code dictionary.)
     R> DUP @ ,          ( compile address )
     CELL+ >R ; COMPILE-ONLY  ( adjust return address )
```

LITERAL compiles an integer literal to the current colon definition under construction. The integer literal is taken from the data stack, and is preceded by the token doLIT. When this colon definition is executed, doLIT will extract the integer from the token list and push it back on the data stack. LITERAL compiles an address literal if the compiled integer happens to be an execution address of a token. The address will be pushed on the data stack at the run time by doLIT.

$," compiles a string literal. The string is taken from the input stream and is terminated by the double quote character. $," only copies the counted string to the code dictionary. A token which makes use of the counted string at the run time must be compiled before the string. It is used by ." and $".

RECURSE is an interesting word which allows eForth to compile recursive definitions. In a recursive definition, the execution address of the word under construction is compiled into its own token list. This is not allowed normally because the name field of the current word under construction is not yet linked to the current vocabulary and it cannot be referenced inside its own colon definition. RECURSE stores the address of the name field of the current word into CURRENT, thus enable it to be referenced inside its own definition. Recursive words are not used in everyday programming. RECURSE is defined here in eForth merely as a teaser to wet your appetite. It is not used in eForth.

```
: LITERAL ( w -- )
     ( Compile tos to code dictionary as an integer literal.)
     COMPILE doLIT       ( compile doLIT to head lit )
     , ; IMMEDIATE       ( compile literal itself )

: $,"     ( -- )
     ( Compile a literal string up to next " .)
     34 WORD             ( move string to code dictionary)
     COUNT + ALIGNED     ( calculate aligned end of string)
     CP ! ;              ( adjust the code pointer)

: RECURSE ( -- )
     ( Make the current word available for compilation.)
     LAST @              ( pointer to current word )
     CURRENT @ ! ; IMMEDIATE  ( link it to current vocab )
```

Control Structures

A set of immediate words are defined in eForth to help building control structures in colon definitions. The control structures used in eForth are the following:

```
Conditional branch  IF ... THEN
                    IF ... ELSE ... THEN
Finite loop         FOR ... NEXT
                    FOR ... AFT ... THEN... NEXT
Infinite loop       BEGIN ... AGAIN
Indefinite loop     BEGIN ... UNTIL
                    BEGIN ... WHILE ... REPEAT
```

This set of words is more powerful than the ones in figForth model because they permit multiple exits from a loop. Many examples are provide in the source code of eForth like NUMBER?, parse, find and >NAME.

A control structure contains one or more address literals, which causes execution to branch out of the normal sequence. The control structure words are immediate words which compile the address literals and resolve the branch address.

```
: FOR     ( -- a )
     ( Start a FOR-NEXT loop structure in a colon definition.)
     COMPILE >R               ( runtime code of FOR )
     HERE ; IMMEDIATE         ( push HERE on stack )

: BEGIN   ( -- a )
     ( Start an infinite or indefinite loop structure.)
     HERE ; IMMEDIATE         ( push HERE for later ref. )

: NEXT    ( a -- )
     ( Terminate a FOR-NEXT loop structure.)
     COMPILE next             ( runtime code of NEXT )
     , ; IMMEDIATE            ( compile address after FOR )

: UNTIL   ( a -- )
     ( Terminate a BEGIN-UNTIL indefinite loop structure.)
     COMPILE ?branch          ( runtime code of UNTIL )
     , ; IMMEDIATE            ( compile branch address )

: AGAIN   ( a -- )
     ( Terminate a BEGIN-AGAIN infinite loop structure.)
     COMPILE  branch          ( runtime code of AGAIN )
     , ; IMMEDIATE            ( compile branch address )
```

One should note that BEGIN and THEN do not compile any code. They executes during compilation to set up and to resolve the branch addresses in the address literals. IF, ELSE, WHILE, UNTIL, and AGAIN do compile address literals with branching tokens. Here are many excellent examples on the usage of COMPILE and [COMPILE], and they are worthy of careful study.

```
: IF ( -- A )
     ( Begin a conditional branch structure.)
     COMPILE ?branch          ( runtime code of IF )
     HERE                     ( push HERE to resolve addr )
     0 , ; IMMEDIATE          ( compile dummy address now )

: AHEAD   ( -- A )
     ( Compile a forward branch instruction.)
     COMPILE branch           ( compile uncondition jump )
     HERE                     ( push HERE to resolve addr )
     0 , ; IMMEDIATE          ( compile dummy address now )

: REPEAT  ( A a -- )
     ( Terminate a BEGIN-WHILE-REPEAT indefinite loop.)
     [COMPILE] AGAIN          ( compile uncondition jump )
     HERE SWAP ! ; IMMEDIATE  ( compile jump address )

: THEN    ( A -- )
     ( Terminate a conditional branch structure.)
     HERE SWAP ! ; IMMEDIATE  ( resolve address for IF/ELSE)

: AFT     ( a -- a A )
     ( Jump to THEN in a FOR-AFT-THEN-NEXT loop the first time through.)
     DROP                     ( discard address left by IF )
     [COMPILE] AHEAD          ( compile uncondition jump )
     [COMPILE] BEGIN          ( leave HERE on stack )
     SWAP ;                   ( realign jump addresses )

: ELSE    ( A -- A )
     ( Start the false clause in an IF-ELSE-THEN structure.)
     [COMPILE] AHEAD          ( compile uncondition jump )
     SWAP                     ( exchange address with IF )
     [COMPILE] THEN ; IMMEDIATE    ( resolve jump address )

: WHILE   ( a -- A a )
     ( Conditional branch out of a BEGIN-WHILE-REPEAT loop.)
     [COMPILE] IF             ( compile condition jump )
     SWAP ; IMMEDIATE         ( realign jump addresses )
```

String Literals

Character strings are very important devices for the program to communicate with the user. Error messages, appropriate warnings and suggestions must be displayed to help the use to use the system in a friendly way. Character strings are compiled in the colon definitions as string literals. Each string literal consists of a string token which will use the compiled string to do things, and a counted string. The first byte in a counted string is the length of the string. Thus a string may have 0 to 255 characters in it. A string is always null-filled to the cell boundary.

ABORT" compiles an error message. This error message is display when the top item on the stack is non-zero. The rest of the words in the definition is skipped and eForth re-enters the interpreter loop. This is the universal response to an error condition. More sophisticated programmer can use the CATCH-THROW mechanism to customize the responses to special error conditions.

." compiles a character string which will be printed which the word containing it is executed in the runtime. This is the best way to present messages to the user.

$" compiles a character string. When it is executed, only the address of the string is left on the data stack. The programmer will use this address to access the string and individual characters in the string as a string array.

```
: ABORT"  ( -- ; <string> )
     ( Conditional abort with an error message.)
     COMPILE abort"           ( compile runtime abort code )
     $," ; IMMEDIATE          ( compile abort message )

: $" ( -- ; <string> )
     ( Compile an inline string literal.)
     COMPILE $"|              ( compile string runtime code)
     $," ; IMMEDIATE          ( compile string itself )

: ." ( -- ; <string> )
     ( Compile an inline string literal to be typed out at run time.)
     COMPILE ."|              ( compile print string code )
     $," ; IMMEDIATE          ( compile print string )
```

Name Dictionary Compiler

We had discussed how the compiler compiles tokens and structures into the code field of a colon definition in the code dictionary. To build a new definition, we have to build its header in the name dictionary also. A header has a token pointer field, a link field, and a name field. Here are the tools to build these fields.

?UNIQUE is used to display a warning message to show that the name of a new token is a duplicate to a token already existing in the dictionary. eForth does not mind your reusing the same name for different tokens. However, giving many tokens the same name is a potential cause of problems in maintaining software projects. It is to be avoided if possible and ?UNIQUE reminds you of it.

$,n builds a new entry in the name dictionary using the name already moved to the bottom of the name dictionary by PACK$. It pads the token field with the address of the top of code dictionary where the new code is to be built, and link the link field to the current vocabulary. A new token can now be built in the code dictionary.

```
: ?UNIQUE ( a -- a )
     ( Display a warning message if the word already exists.)
     DUP NAME?                    ( name exists already?)
     IF ."  reDef "               ( if so, print warning )
          OVER COUNT TYPE         ( with the offending name )
     THEN DROP ;                  ( discard token address )

: $,n     ( na -- )
     ( Build a new dictionary name using the string at na.)
     DUP C@                       ( null input?)
     IF ?UNIQUE                   ( duplicate name? )
          ( na) DUP LAST !        ( save na for vocabulary link)
          ( na) HERE ALIGNED SWAP  ( align code address )
          ( cp na) CELL-          ( link address )
          ( cp la) CURRENT @ @    ( link to current vocab )
          ( cp la na') OVER !
          ( cp la) CELL- DUP NP !  ( adjust name pointer )
          ( ptr) ! EXIT           ( save code pointer and exit )
     THEN                         ( here if null input )
     $" name" THROW ;             ( this is an error return )
```

$COMPILE builds the body of a new colon definition. A complete colon definition also requires a header in the name dictionary, and its code field must start with a CALL doLIST instruction. These extra works are performed by :. Colon definitions are the most prevailing type of words in eForth. In addition, eForth has a few other defining words which create other types of new definitions in the dictionary.

OVERT links a new definition to the current vocabulary and thus makes it available for dictionary searches.

; terminates a colon definition. It compiles an EXIT to the end of the token list, links this new word to the current vocabulary, and then reactivates the interpreter.

] turns the interpreter to a compiler.

```
: $COMPILE ( a -- )
     ( Compile next word to code dictionary as a token or literal.)
     NAME?                    ( parse the next word out )
     ?DUP                     ( successful? )
     IF   C@                  ( yes, get the lexicon )
          [ =IMED ] LITERAL AND    ( is it an immediate word? )
          IF   EXECUTE        ( yes.  execute it )
          ELSE ,              ( no.  compile it )
          THEN
          EXIT                ( done. exit )
     THEN                     ( not a valid word )
     'NUMBER @EXECUTE         ( convert it to a number )
     IF   [COMPILE] LITERAL   ( successful. compile a literal number )
          EXIT                ( done )
     THEN                     ( not a number either )
     THROW ;                  ( generate an error condition )

: OVERT ( -- )
     ( Link a successfully defined word into the current vocabulary. )
     LAST @                   ( name field address of last word )
     CURRENT @ !              ( link it to current vocabulary )
     ;

: ;  ( -- )
     ( Terminate a colon definition. )
     COMPILE EXIT             ( compile exit code )
     [COMPILE] [              ( return to interpreter state )
     OVERT ; COMPILE-ONLY     ( restore current vocabualry )
     IMMEDIATE

: ] ( -- )
     ( Start compiling the words in the input stream.)
     [ ' $COMPILE ] LITERAL   ( get code address of compiler )
     'EVAL ! ;                ( store it in 'EVAL )
```

:   creates a new header and start a new colon word.   It takes the following string in the input stream to be the name of the new colon definition, by building a new header with this name in the name dictionary.   It then compiles a CALL doLIST instruction at the beginning of the code field in the code dictionary.   Now, the code dictionary is ready to accept a token list.   ] is now invoked to turn the text interpreter into a compiler, which will compile the following words in the input stream to a token list in the code dictionary. The new colon definition is terminated by ;, which compiles an EXIT to terminate the token list, and executes [to turn the compiler back to text interpreter.

call,   compiles the CALL doLIST instruction as the first thing in the code field of a colon definition.

IMMEDIATE sets the immediate lexicon bit in the name field of the new definition just compiled.   When the compiler encounters a word with this bit set, it will not compile this words into the token list under construction, but execute the token immediately. This bit allows structure words to build special structures in the colon definitions, and to process special conditions when the compiler is running.

```
: call,   ( ca -- )
     ( Assemble a call instruction to ca.)
     [ =CALL ] LITERAL ,      ( assemble CALL machine code )
     HERE CELL+ - , ;         ( change ca to 8086 call offset )

: :  ( -- ; <string> )
     ( Start a new colon definition using next word as its name.)
     TOKEN $,n                ( compile new word with next name )
     [ ' doLIST ] LITERAL CALL,    ( compile CALL doLIST )
     ] ;                      ( switch to compiling state )

: IMMEDIATE    ( -- )
     ( Make the last compiled word an immediate word.)
     [ =IMED ] LITERAL        ( immediate bit in the name length)
     LAST @ C@ OR             ( OR it into the length byte )
     LAST @ C! ;              ( store back to name field )
```

Defining Words

Defining words are molds which can be used to define many words which share the same run time execution behavior. In eForth, we have : , USER, CREATE, and VARIABLE.

USER creates a new user variable. The user variable contains an user area offset, which is added to the beginning address of the user area and to return the address of the user variable in the user area.

CREATE creates a new array without allocating memory. Memory is allocated using ALLOT.

VARIABLE creates a new variable, initialized to 0.

eForth does not use CONSTANT, because a integer literal is more economical than a constant. One can always use a variable for a constant.

```
: USER    ( u -- ; <string> )
     ( Compile a new user variable.)
     TOKEN $,n               ( compile user name )
     OVERT                   ( restore current vocabulary )
     [ ' doLIST ] LITERAL
     call,                   ( compile CALL doLIST )
     COMPILE doUSER          ( compile doUSER)
     , ;                     ( compiler user area offset )

: CREATE  ( -- ; <string> )
     ( Compile a new array entry without allocating code space.)
     TOKEN $,n OVERT         ( compile name for array )
     [ ' doLIST ] LITERAL
     call,                   ( compile CALL doLIST )
     COMPILE doVAR ;         ( compile doVAR )

: VARIABLE     ( -- ; <string> )
     ( Compile a new variable initialized to 0.)
     CREATE                  ( use CREATE to build name and code)
     0 , ;                   ( initialize data as 0 )
```

8. Utilities

eForth is a very small system and only a very small set of tools are provided in the system. Nevertheless, this set of tools is powerful enough to help the user debug new words he adds to the system. They are also very interesting programming examples on how to use the words in eForth to build applications.

Generally, the tools presents the information stored in different parts of the memory in the appropriate format to let the use inspect the results as he executes words in the eForth system and words he defined himself. The tools include memory dump, stack dump, dictionary dump, and a colon definition decompiler.

Memory Dump

DUMP dumps u bytes starting at address b to the terminal. It dumps 16 bytes to a line. A line begins with the address of the first byte, followed by 16 bytes shown in hex, 3 columns per bytes. At the end of a line are the 16 bytes shown in characters. The character display is generated by _TYPE, which substitutes non-printable characters by underscores. Typing a key on the keyboard halts the display. Another CR terminates the display. Any other key resumes the display.

dm+ displays u bytes from b1 in one line. It leave the address b1+u on the stack for the next dm+ command to use.

_TYPE is similar to TYPE. It displays u characters starting from b. Non-printable characters are replaced by underscores.

```
: _TYPE   ( b u -- )
     ( Display a string. Filter non-printing characters.)
     FOR                      ( repeat u+1 times )
          AFT                 ( skip to THEN the first time )
          COUNT               ( get one character from b )
          >CHAR EMIT          ( display only printable char )
          THEN
     NEXT                     ( repeat u times )
     DROP ;                   ( discard b address )

: dm+     ( a u -- a )
     ( Dump u bytes from , leaving a+u on the stack.)
     OVER 4 U.R SPACE         ( print address first )
     FOR  AFT                 ( repeat u times )
          COUNT 3 U.R         ( display bytes in 3 columns )
          THEN
     NEXT ;                   ( repeat u times )

: DUMP    ( a u -- )
     ( Dump u bytes from a, in a formatted manner.)
     BASE @ >R                ( save current radix )
     HEX                      ( always dump in hex )
     16 /                     ( dump 16 bytes at a time )
     FOR  CR                  ( new line for each 16 bytes )
          16 2DUP dm+         ( dump 16 bytes with address )
          -ROT 2 SPACES _TYPE ( display the ASCII characters )
          NUF? 0=             ( exit the loop if a key is hit )
     WHILE
     NEXT                     ( repeat until all bytes are dumped)
     ELSE R> DROP             ( key hit.  Discard loop count )
     THEN
     DROP  R> BASE ! ;        ( restore radix )
```

Stack Dump

Data stack is the working place of the Forth engine. It is where words receive their parameters and also where they left their results. In debugging a newly defined word which uses stack items and which leaves items on the stack, the best was to check its function is to inspect the data stack. The number output words may be used for this purpose, but they are destructive. You print out the number from the stack and it is gone. To inspect the data stack non-destructively, a special utility word .S is provided in most Forth systems. It is also implemented in eForth.

.S dumps the contents of the data stack on the screen in the free format. The bottom of the stack is aligned to the left margin. The top item is shown towards the left and followed by the characters '<sp'. .S does not change the data stack so it can be used to inspect the data stack non-destructively at any time.

One important discipline in learning Forth is to learn how to use the data stack effectively. All words must consume their input parameters on the stack and leave only their intended results on the stack. Sloppy usage of the data stack is often the cause of bugs which are very difficult to detect later as unexpected items left on the stack could result in unpredictable behavior. .S should be used liberally during Forth programming and debugging to ensure that the correct data are left on the data stack.

```
: .S ( ... -- ... )
     ( Display the contents of the data stack.)
     CR                       ( start stack dump on a new line)
     DEPTH                    ( get stack depth )
     FOR  AFT                 ( repeat that many times )
          R@ PICK .           ( print one item in stack )
          THEN
     NEXT                     ( repeat until done )
     ."  <sp" ;               ( print stack pointer )
```

Stack Checking

.S is useful in checking the stack interactively during the programming and debugging. It is not appropriate for checking the data stack at the run time. For run time stack checking, eForth provides !CSP and ?CSP. They are not used in the eForth system itself, but are very useful for the user in developing serious applications.

To do run time stack checking, at some point the program should execute !CSP to mark the depth of the data stack at that point. Later, the program would execute ?CSP to see if the stack depth was changed. Normally, the stack depth should be the same at these two points. If the stack depth is changed, ?CSP would abort the execution.

One application of stack checking is to ensure compiler security. Normally, compiling a colon definition does not change the depth of the data stack, if all the structure building immediate words in a colon definition are properly paired. If they are not paired, like IF without a THEN, FOR without a NEXT, BEGIN without an AGAIN or REPEAT, etc., the data stack will not be balanced and ?CSP is very useful in catching these compilation errors. This stack check is a very simple but powerful tool to check the compiler. !CSP and CSP are the words to monitor the stack depth.

!CSP stores the current data stack pointer into tan user variable CSP. The stack pointer saved will be used by ?CSP for error checking.

?CSP compares the current stack pointer with that saved in CSP. If they are different, abort and display the error message 'stack depth'.

```
: !CSP    ( -- )
     ( Save stack pointer in CSP for error checking.)
     SP@                     ( get the current stack pointer )
     CSP ! ;                 ( store it in a variable )

: ?CSP    ( -- )
     ( Abort if stack pointer differs from that saved in CSP.)
     SP@                     ( get the current stack pointer )
     CSP @ XOR               ( compare it with that in CSP )
     ABORT" stack depth" ;   ( abort if they are different )
```

Dictionary Dump

The Forth dictionary contains all the words defined in the system, ready for execution and compilation. WORDS allows you to examine the dictionary and to look for the correct names of words in case you are not sure of their spellings. WORDS follows the vocabulary thread in the user variable CONTEXT and displays the names of each entry in the name dictionary. The vocabulary thread can be traced easily because the link field in the header of a word points to the name field of the previous word. The link field of the next word is one cell below its name field.

WORDS displays all the names in the context vocabulary. The order of words is reversed from the compiled order. The last defined words is shown first.

.ID displays the name of a token, given the token's name field address. It also replaces non-printable characters in a name by under-scores.

```
: .ID ( na -- )
     ( Display the name at address.)
     ?DUP                    ( not valid name if na=0 )
     IF   COUNT              ( get length by mask lexicon bits )
          $001F AND          ( limit length to 31 characters )
          TYPE               ( print the name string )
          EXIT
     THEN ." {noName}" ;     ( error if na is not valid )

: WORDS   ( -- )
     ( Display the names in the context vocabulary.)
     CR  CONTEXT @           ( search only the context vocab )
     BEGIN @ ?DUP            ( continue if not end of vocab )
     WHILE DUP SPACE
          .ID                ( print a name )
          CELL-              ( get the link field )
          NUF?               ( exit if a key is hit )
     UNTIL                   ( repeat next name )
          DROP
     THEN ;                  ( end of vocab exit )
```

Search Token Names

Since the name fields are linked into a list in the name dictionary, it is fairly easy to locate a token by searching its name in the name dictionary. However, finding the name of a token from the execution address of the token is more difficult, because the execution addresses of tokens are not organized in any systematic way.

It is necessary to find the name of a token from its execution address, if we wanted to decompile the contents of a token list in the code dictionary. This reversed search is accomplished by the word >NAME.

>NAME finds the name field address of a token from the execution address of the token. If the token does not exist in the CURRENT vocabulary, it returns a false flag. It is the mirror image of the word NAME>, which returns the execution address of a token from its name address. Since the execution address of a token is stored in the token field, two cells below the name, NAME> is trivial. >NAME is more complicated because the entire name dictionary must be searched to locate the token. >NAME only searches the CURRENT vocabulary.

```
: >NAME   ( ca -- na | F )
     ( Convert code address to a name address. )
     CURRENT                  ( search only the current vocab )
     BEGIN CELL+ @ ?DUP       ( end of vocabulary? )
     WHILE 2DUP
          BEGIN @ DUP
          WHILE 2DUP NAME> XOR     ( code pointer=ca? )
          WHILE CELL-
          REPEAT              ( ca not found, repeat next word)
          THEN NIP ?DUP
     UNTIL NIP NIP EXIT       ( found.  return name address )
     THEN 0 NIP ;             ( end of vocabulary, failure )
```

The Simplest Decompiler

Bill Muench and I spent much of our spare time in July, 1991 to build and polish the eForth Model and the first implementation on 8086/MS-DOS. One evening he called me and told me about this smallest and greatest Forth decompiler, only three lines of source code. I was very skeptical because I knew how to build a Forth decompiler. If a Forth colon definition contains only a simple list of execution addresses, it is a trivial task to decompile it. However, there are many different data and control structures in a colon definition. To deal with all these structures, it is logically impossible to have a three line decompiler.

I told Bill that I had to see it to believe. The next time we met, he read the source code in assembly and I entered it into the eForth model. The decompiler had 24 words and worked the first time after we reassemble the source code.

SEE searches the dictionary for the next word in the input stream and returns its code field address. Then it scans the list of execution addresses (tokens) in the colon definition. If the token fetched out of the list matches the execution address of a word in the name dictionary, the name will be displayed by the command '.ID'. If the token does not match any execution address in the name dictionary, it must be part of a structure and it is displayed by 'U.'. This way, the decompiler ignores all the data structures and control structures in the colon definition, and only displays valid tokens in the token list.

```
: SEE     ( -- ; <string> )
     ( A simple decompiler. )
     '                   ( find the next word in context voc)
     CR CELL+            ( skip the CALL instruction )
     BEGIN     CELL+     ( skip doLIST address offset )
          DUP @ DUP      ( get the next token )
          IF   >NAME     ( find its name field address )
          THEN
          ?DUP           ( if name is valid )
          IF   SPACE .ID ( print the name of token )
          ELSE DUP @ U.  ( else print the value as literal )
          THEN
          NUF?           ( exit if a key is hit )
     UNTIL               ( continue to the next token )
     DROP ;              ( discard the token address )
```

Sign-on Message

Since we expect eForth to evolve as experience is accumulated with usage, and as it has to track the ANS Forth Standard under development, version control becomes an important issue. To assure compatibility at different stages of development, the user can always inquire the version number of the eForth he is running. With the version number, corrective actions can be taken to put an overlay on the system to force it to be compatible with another eForth of a different version.

VER returns the version number of this eForth system. The version number contains two bytes: the most significant byte is the major revision number, and the least significant byte is the minor release number.

'hi' is he default start-up routine in eForth. It initializes the serial I/O device and then displays a sign-on message. This is where the user can customized his application. From here one can initialize the system to start his customized application.

```
$101 CONSTANT VER
     ( Return the version number of this implementation.)

: hi ( -- )
     !XIO                     ( initialize terminal I/O )
     CR                       ( start a new line )
     ." eForth v"             ( display sign-on text )
     BASE @                   ( save current radix on stack )
     HEX                      ( change radix to hexadecimal )
     VER                      ( get the version number )
     <#                       ( start converting )
     # #                      ( convert the minor version number )
     46 HOLD                  ( insert a period )
     #                        ( convert the major version number )
     #>                       ( terminate number conversion )
     TYPE                     ( display the version number )
     BASE !                   ( restore current radix )
     CR                       ( add a new line )
     ;
```

Hardware Reset

Because all the system variable in eForth are implemented as user variables and the name dictionary is separated from the code dictionary, eForth dictionary is eminently ROMmable and most suitable for embedded applications. To be useful as a generic model for many different processors and applications, a flexible mechanism is designed to help booting eForth up in different environments. Before falling into the QUIT loop, the COLD routine executes a boot routine whose code address is stored in 'BOOT. This code address can be vectored to an application routine which defines the proper behavior of the system.

After the computer is turned on, it executes some native machine code to set up the CPU hardware so that it emulates a virtual Forth engine. Then it jumps to COLD to initialize the eForth system. It finally jumps to QUIT which is the operating system in eForth. COLD and QUIT are the topmost layers of an eForth system.

'BOOT is an variable vectored to 'hi'.

COLD is a high level word executed upon power-up. Its most important function is to initialize the user area and execute the boot-up routine vectored through 'BOOT, and then falls into the text interpreter loop QUIT.

```
VARIABLE 'BOOT ( -- a )       ( The application startup vector. )

: COLD ( -- )
     BEGIN
          U0 UP 74 CMOVE      ( initialize user variable )
          PRESET              ( initialize stack and terminal buffer )
          'BOOT @EXECUTE      ( execute user bootup procedure )
          FORTH               ( make FORTH the context vocabulary )
          CONTEXT @ DUP       ( make FORTH the current vocabulary )
          CURRENT 2!
          OVERT               ( link new words to FORTH vocabulary )
          QUIT                ( invoke the Forth operating system )
     AGAIN                    ( safeguard the Forth interpreter )
     ;

LASTN     EQU  _NAME+4        ;last name address
NTOP EQU  _NAME-0             ;next available memory in name dictionary
CTOP EQU  $+0                 ;next available memory in code dictionary
MAIN      ENDS
END  ORIG
```

9. Some Final Thoughts

Congratulations if you reach this point the first time. As you can see, we have traversed a complete Forth system from the beginning to the end, and it is not as difficult as you might have thought before you began. But, think again what we have accomplished. It is a complete operating system with an integrated interpreter and an integrated compiler all together. If you look in the memory, the whole system is less than 7 Kbytes. What else can you do with 7 Kbytes these days?

Forth is like Zen. It is simple, it is accessible, and it can be understood in its entirety without devoting your whole life to it.

Is this the end? Not really. There are many topics important in Forth but we had chose to ignore in this simple model. They include multitasking, virtual memory, interrupt control, programming style, source code management, and yes, metacompilation. However, these topics can be considered advanced applications of Forth. Once the fundamental principles in Forth are understood, these topics can be subject for further investigations at your leisure.

Forth is not an end to itself. It is only a tool, as useful as the user intends it to be. The most important thing is how the user can use it to solve his problems and build useful applications. What eForth gives you is the understanding of this tool. It is up to you to make use of it.