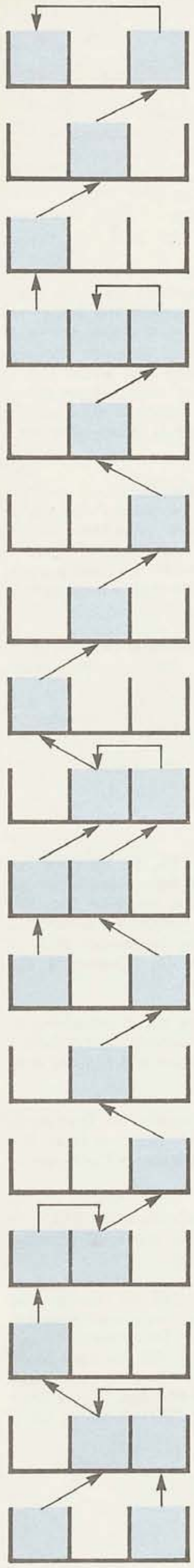


FORTH DIMENSIONS

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

Volume III
Number 4
Price $2.50

INSIDE

101 FORTH and the University

102 FORTH in Laser Fusion

104 Proceedings of the 1981 Rochester FORTH Standards Conference

105 Implementing FORTH-Based Microcomputers

110 Data Structures in a Telecommunications Front End

113 Mapped Memory Management Techniques

116 A High Level Interrupt Handler in FORTH

118 Optimized Data Structures for Hardware Control

121 The String Stack

125 Complex Analysis in FORTH

126 A FORTH-Based, Micro-sized Micro Assembler

FORTH DIMENSIONS

Published by Forth Interest Group

Volume III No. 4 November/December 1981

Publisher Roy C. Martens
Editor C. J. Street

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL MATERIAL PUBLISHED BY THE FORTH INTEREST GROUP IS IN THE PUBLIC DOMAIN. Information in FORTH DIMENSIONS may be reproduced with credit given to the author and the Forth Interest Group.

Subscription to FORTH DIMENSIONS is free with membership in the Forth Interest Group at $15.00 per year ($27.00 foreign air). For membership, change of address and/or to submit material, the address is:

Forth Interest Group
P.O. Box 1105
San Carlos, CA 94070

HISTORICAL PERSPECTIVE

FORTH was created by Mr. Charles H. Moore in 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of dissatisfaction with available programming tools, especially for observatory automation.

Mr. Moore and several associates formed FORTH, Inc. in 1973 for the purpose of licensing and support of the FORTH Operating System and Programming Language, and to supply application programming to meet customers' unique requirements.

The Forth Interest Group is centered in Northern California. Our membership is over 2,400 worldwide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

EDITOR'S COLUMN

A special thanks this month goes to Mr. Larry Forsley and the University of Rochester. The majority of this issue comes from his efforts and those of his asociates. While acting as guest editor for this issue of FORTH DIMENSIONS, Mr. Forsley was also compiling and editing the proceedings from this year's FORTH conference at the University of Rochester. Even with this "double duty," Mr. Forsley has done an excellent job.

The quality of material we have received from the University of Rochester is excellent and greatly encourages me in my plans to "de-Californize" FORTH DIMENSIONS through the use of regional guest editors. While Mr. Forsley and the University of Rochester may be a tough act to follow, I will welcome contacts from anyone else (person and/or organization) who would like to try guest editing an issue. For your peace of mind, let me assure you that production (typesetting, proofing, printing, etc.) will be handled for you. If you think you have what it takes, give me a call or drop me a line.

You may find that some of this issue's sections have been reduced is size and/or eliminated. This is a temporary concession because of the volume of material we have to publish in this issue. Postal costs prohibit expanding the size of FORTH DIMENSIONS to publish all we receive, so when we have a quantity of quality material we publish those items that would seem to have the greatest reader interest.

I hope to meet many of you at the FIG National Convention in Santa Clara, California on November 28th. Meanwhile, GO-FORTH and get additional members.

C. J. Street
Editor

PUBLISHER'S COLUMN

We are heading into some busy times for FIG. By the time you get this copy of FORTH DIMENSIONS we'll have completed the Mini-Micro Show in Southern California and be deep into the details of the FORML Conference and FIG National Convention. Remember that the Convention is Saturday, November 28th at the Marriott Hotel in Santa Clara, California. Expect to see many of you there.

We've sent out packets to FORTH vendors about exhibiting at the FIG National Convention. If you are interested in exhibiting and haven't received a packet, call the FIG line and request one: (415) 962-8653. Only $50 for a table!

This issue is the much awaited University of Rochester effort. Its packed with useful material. You ought to order the Proceedings of the 1981 Rochester FORTH Standards Conference. It has 378 pages of excellent papers

"Starting FORTH" by Leo Brodie is available from FIG ------------- and replaces "Using FORTH" as the book to have about the FORTH language.

Now, a little lecture. We have conducted an unscientific survey and found that in many locations there are people who are using FORTH and aren't members of the FORTH Interest Group. You as a member should work on them to join. All you have to do is make a copy of the Order Form ------------------ and have your associates fill in their name and address. If we each get one more person to join we'll have over 5,000 members. Let's do it.

Roy C. Martens

FORTH AND THE UNIVERSITY

Lawrence P. Forsley
Laboratory for Laser Energetics
University of Rochester

Welcome to the wonderful world of URTH, or, University of Rochester FORTH. URTH was developed several years ago and has been used for many applications, some of which are documented here. Beginning with the 1978 FORTH Internatinal Standards Conference, held on Catalina, we have followed the FORTH standardization effort. As a result, the majority of our systems are close to being FORTH-79 Standard, although not FIG model. Very few papers in this issue will refer to URTH.

The 1981 Rochester FORTH Standards Conference was held at the University. The major reason for this, aside from the delightful weather at that time of year, is the FORTH activity at the University. This work shows up in several divisions and departments including the University Computing Center; Optics; Physics and Astronomy; Chemical Engineering; Mechanical Engineering; Department of Radiology, Division of Diagnostic Ultrasound; Department of Cytopathology; Electrical Engineering and the Laboratory for Laser Energetics. Indeed, we are indebted to the original work by Dick Berg, who in 1976 was an assistant professor of Physics and Astronomy, for deriving the first URTH system; and to Ken Hardwick, who in 1977 was with the University Computing Center, for bringing up the IBM 360/65 TSO version based on Dick's work. At this time, Ken, Dick and I were the only FORTH users at the University. I believe the name URTH was coined by Ken, although Dick was partial to PARTH, for Mike Williams' multitasking Intel 8080 FORTH system. Unfortunately, Ken and Dick are no longer with the University; and Mike's commitments prevented his authoring a paper. However, their work is reflected in the material presented here.

This issue starts with three overview papers. The first paper is mine and covers the development of FORTH at the Laboratory for Laser Energetics, which remains the largest university FORTH user. The second paper, by Peter Helmers, reflects on the uses of FORTH in medical research and clinical applications. The third, by John Lefor, covers one of the more visible university FORTH systems: The IBM 3032 telecommunications front-end.

The next three papers demonstrate a variety of ways by which FORTH can be used to interact with hardware. The first paper, by Rosemary Leary and Carole Winkler, deals with three methods of using mapped memory. A second paper, by Bob Keck and me, demonstrates a high level interrupt handler used in plasma physics experiments. The third paper in this section is by Joe Sawicki, and suggests powerful structures for easily and efficiently interfacing hardware.

The last section illustrates the difficulty with defining the difference between systems and applications. The first paper is by Michael McCourt and Richad Marisa, and describes a transportable String Stack. The second paper is by Alfred Clark and covers a FORTH-based complex arithematic calculator. The last paper is by Greg Cholmondeley and documents a microprocessing tool similar to one supplied by Signetics.

These papers have many things in common. One example is the difficulty in discriminating between users and implementors. Bob Keck, a user, worked with me to develop a tool for high level interrupt handling. Likewise, Al Clark, also a user, has augmented a floating point package with words appropriate to the complex plane. The String Stack is clearly a system tool. Complex arithmetic is less so, and a microprogramming system is clearly an application. Or is it? In the context of <u>its user</u>, the microprogramming words are <u>a</u> system. We seem to be forever chasing our tail when determining a FORTH context. But I think that this is the power of FORTH.

Another facet is the use of defining words used throughout the papers. An extension of defining words, Paul Bartholdi's TO concept,[1] is used in both Joe Sawicki's and Greg Cholmondeley's code. Mike McCourt's "IN" concept[2] is used by Peter Helmer's to implement the TO concept. However, a student, Carole Winkler, thought that TO complicated things unnecessarily, so she doesn't use it.

This last comment illustrates one of the virtues of universities: freedom of dissent. Unfortunately, I have found that most groups, and many people, using FORTH are intolerant of different views. During my involvement with FORTH I have watched many groups rise to ascendency, tout the true way, and then be replaced by another group. This has been especially true of the FORTH Standards effort where Kitt Peak, FORTH, Inc., the European FORTH User's Groups and FIG have all played this role. But another view is possible, which is more in keeping with FORTH's nature.

Many of us see FORTH as being a system of controlled, or directed, anarchy. Since every man, or woman, can be for himself it is highly idiosyncratic and anarchistic in form. Anyone who has tried a team approach to FORTH programming is familiar with the tendency towards a Tower of Babel. On the other-hand, people comfortable with thie unstructured environment find both their productivity and creativity increased. But, some direction must be applied to share code among users. I suggest that this direction should be one of form, and not of content.

It is appropriate to define documentation standards which imply a form. But is is inappropriate to state that something can be done only <u>one</u> (with the implied right) way. However, people who learn something by doing it the wrong way understand much better than people who are <u>told</u> the right way.

I think an example of this can be found in a conversation I had with Kim Harris.[3] Kim took exception to an earlier paper by Peter Helmers on Userstacks.[4] I was told that the approach was wrong. Period. But on further discussion, I found that I agreed with Kim. The fault was that Peter had found only a partial solution to data typing, and in a multitasking system his technique might be very cumbersome. That's fine. Peter Helmers does not use multitasking systems, as his systems are all single user, interrupt/event driven. thus, it is worth remembering that each of us has different, and valid, viewpoints.[5]

As a major promoter of FORTH at the University of Rochester, I have tried to define an environment conducive to this type of interplay. This has resulted in a learning environment with many student opportunities; and with Leo Brodie's book, <u>Starting Forth</u>, and Don Colburn's study guide, <u>Going Forth</u>, we can begin teaching with FORTH. Not teaching FORTH, but teaching with it. Four of the authors in this issue are students and three other authors teach courses or seminars. If FORTH is ever to catch on like Pascal, or FORTRAN, then it must begin wtih university teaching as those two languages did. In five years my present students will be in industry, as my first student contacts already are. A univeristy environment coupled with its students' enthusiasm and their eventual employment will further FORTH more than any seminar series or interest group. But it will take time.

---

1. FORTH DIMENSIONS Vol. I No. 4 and Vol. I No. 5.
2. FORTH DIMENSIONS Vol. II No. 4
3. Personal conversation on May 10, 1981 prior to the Rochester Conference.
4. FORTH DIMENSIONS Vol. II, No. 2
5. Since that paper, Peter has published another one, entitled "Alternative Parameter Stacks," which can be found in the <u>Proceedings of the 1981 Rochester FORTH Standards Conference.</u>

FORTH IN LASER FUSION

Lawrence P. Forsley
Laboratory for Laser Energetics
University of Rochester

Abstract

Inertial confinement fusion research using lasers has resulted in the laboratory creation of extraordinary conditions of temperature and pressure, duplicating those found in the cores of white dwarf stars. The machines which create these conditions and the diagnostics that monitor them have become increasingly automated. The demands of this research have forced us to adopt new techniques, like FORTH, for enhancing interactions between engineers, physicists and their experiments.

Introduction

Lasers have been used to simulate plasma conditions of high density (approaching solid) and temperature (over 60 million degrees) for several years. The goal of these experiments has been either for weapons effect simulation, practiced at the national laboratories, or for the possible commercial generation of power. This latter program has been exclusively pursued by the Laboratory for Laser Energetics (LLE) for almost a decade. As can be expected, these experiments have resulted in the development of new diagnostics, and these diagnostics, in turn, have resulted in new fields of physics. Besides the Laser Fusion Feasibility Project, there are research programs in: sub-picosecond lasers, nanosecond X-Ray sources, X-Ray lasers, laboratory astrophysics, and materials damage testing.

These research programs, and the main supporting lasers, are highly automated. About one half of the computer systems on the 24 beam 13 terrawatt infrared Omega laser and all of the computers on the single beam Glass Development Laser (GDL) are implemented in FORTH. This paper will explore the development of FORTH-like languages at LLE.

The laboratory is also part of the College of Engineering of the University of Rochester. Thus, there is an important interplay between the staffs, and students, of LLE and the University. Most of our FORTH systems have been partially, or totally, implemented by students from chemistry, electrical engineering, physics and computer science. Four of the other papers in this journal issue have a student author who is also a member of LLE.

Standardization

LLE was one of the first Laser Fusion laboratories to automate its laser systems.[1] Whenever possible, we relied upon standard computers, interfaces and software. Originally, in 1971, we chose the Hewlett Packard 2100 series computer, and the RTE (Real Time Executive) Operating System with Fortran, Assembler and Algol. We used the HP backplane for our instrument interface. This system ran for over five years and 15,000 shots, but building a completely automated laser with 24 instead of 4 beams required a different approach.

The Hewlett Packard computer backplane was limited in the number and variety of devices which could be procured and attached to it. We overcame this difficulty by adopting CAMAC (5). CAMAC provided us with a large capacity, computer-independent backplane. It was also a widely used standard in the nuclear physics community with instrumentation and interfaces appropriate to our needs available from several sources.

The problems of computer and software standardization were more difficult. Some of our applications were real-time, and appeared to require a fast interrupt response. In other cases, we were interested in direct image digitization and needed a large address space. Other requirements suggested the need for a powerful multiprogramming operating system. Unfortunately, no one computer type and operating system supported all of our applications; and yet, with limited manpower, it was difficult to support a variety of hardware and software.

Computer languages, including FORTRAN, are different from one vendor to another, and especially when operating system calls were taken into account. The problem of software consistency and support was not limited to dissimilar computers. Ehrman (4:16,17) has shown that as many as 12 different languages may be encountered by a programmer when editors, linkers, and loaders are included in addition to the programming language. Therefore, a unifying software approach was needed among various operating system functions and languages on the same and different computers. We did not know of the unix System from Bell Laboratories (11:1905-1929) and the 'C' programming language of Richie and Stevens (12:1991-2019) in 1976. However, I had talked with people at Kitt Peak in 1976 and travelled there in the spring of 1977 to see FORTH being used.

FORTH

FORTH was originally developed as a small, real time operating system for telescope control and image processing by Moore (8:497-511), (9) and Rather (10:223-240) at the Kitt Peak and NRAO facilities which are funded by the National Science Foundation. I found three groups at these facilities using FORTH: scientists, computer engineers and technicians. In some cases, the scientists were very knowledgeable about FORTH, whereas in other cases, they only knew a few words. I was especially impressed by Dr. Mark Alcott, who was, at the time, with Cal Tech and was observing on NRAO's 36 foot radio telescope. He was pleased with his ability to change the graphics routines and other "systems" software while continuing to collect data. Similarly, I found many technicians programming and writing test programs. This appeared to make good use of their time, especially when they would be familiar with a device, like a Varian computer disk controller, and did not have to explain its function to a programmer. It also appeared that many of the computer group's staff enjoyed FORTH, although there were problems with standardization and change. I found out several years later, talking with Jeff Moler, who was then in operations at Kitt Peak and is now with the Livermore Tandem Mirror Experiment, how difficult it was to maintain programs in this environment.

FORTH seemed to have many desirable characteristics, and it provided the same programming environment on many machines. It allowed both very low level access to hardware and high level structures to shield users from that hardware. There was an assembler, a compiler, and an interpreter. What we did not know then was the care required in documenting it, and the tendency to create personalized applications and words. But, we needed a version of FORTH at the University.

Dick Berg, an assistant professor in physics and astronomy at the time,[2] decompiled a Kitt Peak Varian nucleus circa 1974. He recoded it for the National Semiconductor PACE microprocessor. Ken Hardwick, then with the Univerity Computing Center,[3] used this as a model for the IBM 360/65 under TSO and Mike Williams developed a multitasking version on the INTEL 8080. This was the birth of URTH.

We also procurred a version for the Zilog Development System from FORTH, Inc. at about the same time to demonstrate an automated X-Ray spectrometer. Although I had a system for the Hewlett Packard 2100 from Kitt Peak and a "diskless" version from Don Berrian at Princeton, I decided that we should develop our own version based upon the URTH model. Ken Hardwick and I did this in late 1977. Since then, other members of the University community and the Laboratory for Laser Energetics have worked on various versions of FORTH for Data General, Modcomp, PDP 212 and IBM 3032 computers. Through the efforts of Mike McCourt, originally with the Department of Cytopathology and then with LLE, we developed a FORTH-79 system. All of these were multitasking systems (2:314-

318).

Testbeds

The first FORTH applications at LLE were hardware testbeds. There are two distinct phases in dealing with hardware. The first occurs during its initial checkout and reoccurs when it fails, or you suspect it of failing. At this stage, one is concerned with device and interface implementation, and it is important to be able to interactively set and test data and address lines.

A testbed must be capable of exercising hardware at a rate of about 1 kilohertz. Devices which operate in a faster time domain will usually be buffered, as an example, with transient digitizers. Most other devices, such as relays, operate in a 10 Hz or slower time domain. At a 1 kHz rate, sufficient samples can be taken from A/D's and D/A's to quickly check their accuracy and range, and thereby checkout many parts of a system quickly.

Several language features are required for tests like these. A means must be provided to individually and collectively set address and data lines. There must also be a way of repetitively issuing data/ address patterns. Often, a hardware problem is intermittent, and a test and branch capability is necessary to allow loopiung until a failure occurs.

Thus, the specification for a testbed language grows quite large, with a major role occupied by the command processor, or text interpreter. Regardless of whether the testbed language is implemented in Fortran, Basic, Pascal or most other programming languages, a substantial effort will be spent on the text interpreter. One of the virtues of FORTH is that it comes with a generalized text interpreter, suitable for testbeds and other applications.

Our FORTH testbed applications included: power conditioning testbed for checking out laser amplifiers; alignment testbed for debugging and calibration of automated components; and, general CAMAC module testing. Other testbeds have been used to develop image processing hardware and software, and one-dimensional reticon arrays.

The laser amplifier testbed was developed along the following schedule:

1. October 1977-Ken Hardwick and I began writing a FORTH system for the HP 2114.
2. January 1978- The FORTH system was completed and CAMAC software started.
3. March 1978- A laser amplifier testbed was demonstrated.
4. April 1978- Single laser amplifier testbed was operational at laser hardware subcontractor's site, with a duplicate at LLE.

   By April, it was clear that the Omega Power Conditioning computer would not be available until August, 1978. Since the Department of Energy four-beam milestone was originally scheduled for early September, 1978, this left insufficient time for laser preparation.
5. April 1978- An LLE engineer, John Boles, and a consultant with the software subcontractor developing the power conditioning software, began coverting the single amplifier testbed to run 4 laser beams synchronized with the laser oscillator.
6. June 1978- A six beam laser system was operational.
7. August 1978- Preliminary delivery of full 24 beam system which was Fortran-based.
8. October 1978- Department of Energy Milestone passed.

There were substantial differences between the 24 beam Fortran based system and the 6 beam FORTH version. These included the lack of an error detecting command processor, a graphic display and error archiving on disk. However, whereas the FORTH version used 16K words of memory and a floppy disk, the Fortran based system required 196K words of memory and a 15 megabyte hard disk.

This application also made us aware of FORTH'S compactness and the speed with which applications could be developed. It is my feeling that this, and several other applications, were brought up in one half the time it would have taken in Fortran, including FORTH training time. Once good documentation is available, FORTH will prove even better.

Also, I have found FORTH systems to be more maintainable than comparable Fortran systems, because FORTH uses 10 times fewer source lines. Some care is needed when writing FORTH. Another advantage can be gained by the ease of using data base technology when building process control systems in FORTH.

Spatial and Temporal Relationships

The first phase of dealing with hardware is over when the hardware works. The relationships among devices then become important. One can hierarchically organize related devices into subsystems. This hierarchy consists of both spatial and temporal relationships among components (1), (3). The manipulation of these relationships requires the development of a data-base-like language. My initial work with Fortran and RTE, and discussions with Ray Helmke and Eric Knobil at the Wilson Synchrotron,[4] led me to develop such a language for process control called Maps, because it "maps" relationships 6:109,110.

A Map contained two types of structures, or Tags. A tag was either a collection of data, or a set of pointers to other Tags. The Map contained an inverted list of pointers to each tag, so that all tags were unique and accessible. Two specialized programs, SETUP and BUILD, were developed to manipulate and create the initial Maps from text files. About a dozen subroutines were developed to allow tags to be accessed. Data could then either be placed into one or more Tags, or retrieved from them. In the interest of speed, this system was recoded in assembly language and later microcoded on a Hewlett Packard 21MX-E computer. This computer currently runs the Omega 24 beam power conditioning, and was mentioned in the Testbed Section of this paper.

Alternatively, by using the text interpreter and FORTH's capability to define arbitrary data structures, several data-base-like systems have been developed. In its simplest form, everything in FORTH is an executable data structure. Thus, FORTH allows one to define spatial and temporal relationships in a simpler, and more concise fashion than Maps. In addition, it is internally consistent, whereas Maps had Fortran, assembler, microcode and operating system interface facets.

Production Systems

Once FORTH had proven viable for small systems, we decided to implement production systems in it. These systems included automated diagnostics as well as the laser control systems. The prototype Omega 24 beam calorimetry system was an example of an early production system. It used simple, vector like structures to contain the addresses, relationships and values associated with various calorimeters, analog to digital convertors and calibrators. It was capable of displaying beam energies and calculating exponential fits to the data.

The Omega 24 beam Alignment System is more complex. It has run on an LSI 11/2 with 5 CAMAC crates and 3 color displays, controlling over 1000 devices. Initially, the operators used the FORTH text interpreter for all commands and queries. One advantage was their ability to write new "macros" to setup complicated alignment procedures more quickly. However, there was a risk associated with letting operations' personnel directly program the system. Therefore, the new Alignment System has a more complete command processor implemented in FORTH, but which does more error detection than the simple text interpreter. This system also uses the defining words capability and has a large disk resident data base for describing components. With the advent of the command processor, the system was switched over to an LSI 11/23 with mapped memory.[5] This addition allowed approximately 20 tasks to handle various functions, communicating via a queue-based message protocol.

The laser beam quality is also important to us. We use streak cameras interfaced to Princeton Applied Research Optical Multichannel Analyzers for this purpose. The PAR OMA includes a FORTH-based LSI 11 for acquisition and reduction. As with the early Alignment and Calorimetry systems, it is programmed directly in FORTH.[6] Unlike those systems though, this was originally not a turnkey system provided by software engineers, but rather was incrementally developed by physicists and students.

We also use FORTH exclusively on the Glass Development Laser (GDL) with similar computer systems. A FORTH based HP 2100 is used for power conditioning and interlocks for the main bay and three surrounding laboratories. A DEC LSI 11/2 collects laser and target calorimetry data, reduces it, and also maintains a data base on disk. A second LSI 11 is used in a PAR OMA for processing streak camera data. This is especially significant since GDL is engaged in converting the infrared light to ultraviolet, and the first harmonic IR, a second harmonic green and the third harmonic, UV are observed with the same streak camera. This required a very flexible system to allow reduction in a quasi-two dimensional mode. Another Hewlett Packard 2100 has two video digitizers and a color graphics unit. It is used for determining absolute beam intensity and modulation for materials damage testing. This system is being converted to a DEC LSI 11/23 with an RL01 disk attached. A third LSI 11 has been used by a graduate student to observe target plasma produced X-rays.[7] Finally, an LSI 11/23 is used with the nanosecond X-Ray facility for the real time acquisition and reduction of 2D X-ray diffraction patterns. Recently, this system has had an array processor interfaced to it to allow real-time fast fourier transforms of sample diffraction rings. All of these systems are FORTH based, with the automated imaging diagnostics serving as prototypes for Omega diagnostics.

Conclusion

Although FORTH was relatively unknown, it has made a positive impact on the development of systems and instrumentation at LLE. It has allowed the computer sytems group to adopt the philosophy of providing tools to scientists and engineers, equipping them to do a job themselves. Sometimes, it was questioned whether this was the best use of their time: and, for some people, it wasn't. But, for the majority of people in GDL, and a fair number on the Omega systems and other laboratories at LLE, FORTH has been a success.

Acknowledgements

I would like to thank an almost endless list of people for their help over the past five years. Most important among them though, are Ken Hardwick, Dick Berg, Chip Nimick and Mike McCourt. Also, without the help of many students during this period, many of these sytems would never have been built.

This work was partially supported by the following sponsors: Exxon Research and Engineering Company, General Electric Company, New York State Energy Research and Development Authority, Northeast Utilities, The Standard Oil Company (Ohio), the University of Rochester, Empire State Electric Energy Research Corporation, and the U. S. Department of Energy inertial fusion program under contract number DE-AC08-80DP40124.

Lawrence P. Forsley is group leader of the Computer Systems Group at the Laboratory for Laser Energetics, University of Rochester, Rochester, N.Y.

Footnotes

1 The four-beam system, Delta, had computer control and monitoring in 1972. (6:101).

2 He is now with the Defense Mapping Agency in Washington, D.C.

3 Ken is now with Network Systems Inc., in Minneapolis, MN.

4 Cornell Univerity in the summer of 1977. This facility is now known as the Cornell Electron Storage Ring.

5 The mapped memory techniques are discussed by Leary and Winkler in the "Mapped Memory Techniques in FORTH" paper in this issue.

6 PAR purchased this system from FORTH, Inc.

7 This is mentioned in Bob Keck's and my paper, "A High Level Interrupt Handler in FORTH", which can be found in this issue.

PROCEEDINGS OF THE 1981 ROCHESTER FORTH STANDARDS CONFERENCE

Many have been waiting for this conference proceedings to come out, from what was a very interesting, and different conference. It was the first conference to address the FORTH Standard since the Catalina meeting of October 1979. Although it was suggested that the Rochester conference was only a regional meeting, attendees came from six countries and thirteen states. Also notable, we successfully divided papers into serial oral sessions one morning and had parallel poster sessions that afternoon. This way, almost everyone of the seventy participants presented something, and no one missed anything (we think).

In addition, we added travel sponsorship this year. The Standard Oil Company (Ohio), Friends Amis, Inc., Miller Microcomputer Services, and Software Ventures contributed over $5,000. This travel fund covered partial travel expenses for attendees from as far away as Hawaii, Chile, Germany and the Netherlands, and as close as California and Kentucky.

The original call for papers was in three major areas: the Standard, floating point and files management. These areas are well represented in the proceedings. In addition, there are sections on Philosophy, Vocabulary, Multi-tasking and Data Acquisition, Data Structures and the Future of FORTH. The organization we adopted combined poster sessions, oral sessions and some material not presented at the conference. There is an entire section devoted to working groups on areas like Standards clarification, FORTH techniques, Floating Point and Files Management. There are 378 pages covering the state of FORTH. The Proceedings are available for $25. See the FIG Order Form.

For those who are interested, there will be another Rochester FORTH Conference the third week of May, in 1982. The tentative subject area will be Process Control and Data Acquisition. We expect that there will be subareas dealing with microprogramming, FORTH machines, personal computing, and the Standard. For information, please contact the conference chairman:

Lawrence P. Forsley
Laboratory for Laser Energetics
250 East River Road
Rochester, NY 14623

IMPLEMENTING FORTH BASED MICROCOMPUTERS AT THE UNIVERSITY OF ROCHESTER MEDICAL CENTER

Peter H. Helmers

Introduction

"The micros are coming!" Everyone has heard this so that it is not unexpected that physicians and researchers at the University of Rochester Medical Center ask the question: "How can they be put to use?" Over the past four years I've been attempting to answer this question by assembling a series of microcomputers for both research and clinical applications. These systems are all similar in their use of an S-100 bus hardware architecture and a FORTH software environment. Yet they differ significantly when it comes to specific hardware interfaces, application software, and types of system users.

In this article, I am going to focus on both these similarities, and these differences in microcomputer systems. I am going to start out by discussing their common hardware foundation, and then explore peripheral devices unique to each system's design. Because the ultimate users of a system have a significant impact on application software, I am going to try to characterize the types of users I have dealt with, and their specific software capabilities and needs. From here I will discuss some common software packages that were written to transcend both variable hardware, and variable user, requirements. By discussing all of this in terms of how FORTH has aided system development, I hope to fully support my contention that FORTH is an ideal environment to meld many different types of users to just as diverse hardware configurations.

General Hardware Organization

So let's start out by considering the common architectural arrangement of these microcomputers. They are all Z-80 based machines with typical memory sizes of from 32K to 48K bytes of static read/write memory and 1K to 2K of EPROM memory used to contain machine specific implementations of commonly needed I/O routines such as console and disk drivers. Each microcomputer uses one or two eight inch single density floppy disk drives. The primary system console is comprised of a 16 line by 64 character memory mapped video display along with detached ASCII keyboard. Each machine also has an RS-232 serial port for printer hookup.

These computers are all organized around the S-100 (IEEE-696) bus with from ten to fifteen card slots available. With the basic setup described above using from four to six of these slots, the customization to specific system configurations is accomplished by a mixture of standard commercial and/or wire-wrapped peripheral interface cards. Let's consider some of these systems in greater detail, looking at special hardware and how this is reflected in the systems' software.

Ultrasound Diffraction Apparatus (UDA)

The UDA microcomputer is part of an experimental system to explore the scattering (diffraction) of medical ultrasound signals through tissue samples. The scattering is a function of both frequency of the ultrasound signal (2 to 8 Mhz) and the angular position of a receive transducer relative to the ultrasound transmitter. The UDA system thus must control three primary functions: analog carrier signal generation, tissue sample positioning, and received signal analog processing. At present, only sample positioning (using stepper motors) is not directly handled by the UDA microcomputer.

Carrier signal generation is controlled by means of a Hewlett-Packard 8165A programmable signal generator interfaced to the microcomputer by means of an IEEE-488 (GP-IB or HP-IB) instrumentation bus. An opto-isolated parallel TTL output port is used to control a programmable attenuator on the output of the 8165A. With a range of 0 to 130 db, the attenuator can be used to automatically adjust gains for maximum signal dynamic range.

The most critical aspect of the UDA hardware is the generation of gating signals used by the analog processing circuitry. This is accomplished by using high speed analog mixers driven by digital timing circuitry with a resolution of 100 nsec., and an accuracy of 0.01%.

Study of Vein Mechanics

The basis of this system is an experiment to measure axial force, diameter and transmural pressure in a blood vein (in vitro) while controlling axial strain and pressure. The system consists of a vertical chamber for the vein specimen, a prefusion and pressure clamping apparatus, force and pressure transducers, and a microprocessor for data acquisition.

The microprocessor contains a sixteen channel, twelve bit multiplexed analog to digital (A/D) converter to digitize the force and pressure signals under high level program control.

In conjunction with this A/D is a commercial video (TV) digitizer capable of programmed resolution up to 240 lines of 256 picture elements. The input to this digitizer is from a TV camera aimed at the blood vessel under study. A special code definition was written to analyze a programmable area of the TV image for an indication of vessel diameter. This works by first threshholding, then detecting vessel edges via a software algorithm. By using FORTH/Z-80 assembly language, the diameter determination executes in less than one second.

This data acquisition system also contains a dual mode graphics display capable of 128x128x4 grey scale images or 256x240 dot graphics. Digitized video images use the former mode while acquired pressure and force data use the dot graphics. In addition, the TV signal dynamic range can be studied by a dot graphic plot of TV signal amplitude versus time.

Also included in this system, to aid in data reduction, is an Advanced Micro Devices AM9511 high speed floating point processor IC. This circuit's speed, combined with the memory mapped graphics display, allows real-time analysis and display of acquired data, thus giving continuous feedback on the progress of the experiment.

Overall, this system replaced a manual strip chart and photographic recording setup that required several days for data collection and analysis. Now data can be automatically acquired and processed within a couple of hours.

Pulmonary Microcomputer

The pulmonary clinic uses a microcomputer identical to that just described except without the TV video data acquisition interface. Used in a clinical setting, this pulmonary microcomputer is integrated with a mass spectrometer and a breathing chamber to allow analysis of pulmonary tissue volume and capillary blood flow. The basic procedure requires keeping track of the patient's breathing (by monitoring volume within the flexible breathing chamber) while analyzing the decreasing concentration of two soluble gases: dimethyl ether (DME) and acetylene (C_2H_2), referenced to the concentration of an insoluble gas: helium (He).

The hardware floating point unit facilitates rapid (30 seconds) analysis of the acquired data, including several curve fitting operations, and analysis of signals for relative maxima/minima. The graphics interface allows immediate viewing of the acquired data to ascertain proper signal levels, and to compare raw data to the curve fit data.

X-Ray Scanning System

This experimental scanner uses a slotted wheel and two horizontal slots (mounted at 90° to the radial orientation of the wheel) to achieve a mechanically raster scanned X-ray source. The wheel and horizontal slots are controlled by means of three separate stepper motors pulsed under control of the microcomputer. X-ray exposure is also controlled by the computer as a function of measured patient X-ray attenuation.

The microcomputer contains a counter/timer chip which is used to control the stepper motors, a seven channel multiplexed eight bit A/D converter (used to measure patient X-ray attenuation and X-ray power), and an eight bit D/A converter to control the exposure time of each X-ray pulse. Several digital I/O lines are used to start the X-ray rotor, turn on the X-ray generator, and control stepper motor direction. Other lines are used to sense mechanical limit switches.

The software used in this machine is primarily concerned with controlling exposure time for each X-ray pulse in synchrony with the motor movement. The system ramps the motors up to speed from an initial stopped condition. In addition, it gradually increases speed to compensate for linear speed as the horizontal slots are moved radially towards the center of the wheels. The software also controls exposure time by sampling the attenuation of X-rays through the patient once each motor step, and using table look-up techniques to set the next pulse's exposure time. In addition, total x-ray power is sampled and accumulated to keep track of total patient dosage and X-ray tube usage.

How Users' Needs Impact These Systems

In my development of these systems, I have encountered three types of users: system developers, researchers, and physicians (and their clinical technicians). This grouping of users also roughly corresponds to levels of FORTH software utilization. The system developer--myself and presumably yourself--is expected to know all the in's and out's of system operation. If something is missing, it's generally easy to add it; this is a primary reason why many of us like FORTH. However we don't actually apply a system, we only set up the software foundation for the system. As users, we don't count!

A true end user, whether researcher or physician, cannot be sold on FORTH because missing capabilities can be easily filled in; they don't have the knowledge to do so. Nor do they really want to learn to do so. They have to be sold on other virtues of FORTH.

In my experience, researchers have been very receptive to FORTH. In general they have sophisticated technical backgrounds but little practical computer knowledge. This is a prime benefit: they may have used FORTRAN on a large machine for number crunching, but otherwise they have few preconceived notions about computer organization. They are less impressed with structured programming techniques or file systems than they are by the fact that they can physically, and interactively, control peripheral devices. A research scientist may not understand how a word like RAMP or SAMPLE works, but can readily learn what they do.

For example, the FORTH software written for the UDA system allows explicit user control of the hardware for setup purposes as well as automatic control during experimental data acquisition runs. Setup can be done through words such as:

```
OK 25 DB
( RPN's a natural here! )
OK " FRQ 2500 KHZ" TALK
(via the GP-IB )
OK 2.5 USEC CARRIER-OFF
```

A data acquisition experiment can be set up using words such as:

```
OK 100 2000 SWEPT-FREQUENCY
( define control of HP8165A )
OK FIXED-ATTENUATION
( define control of atten )
OK DON'T-SHOW-ATTENUATIONS
OK 1500 32 NOVA-CONTROL
( let the minicomputer take
over control of the micro.)
```

In addition, the researcher can build upon basic words to create custom application programs as needed. Thus the X-ray scanner system can be easily programmed by:

```
OK MOTOR WHEEL-MOTOR
( define a 'MOTOR' data type)
OK : ROTATE-EM
OK     DO
OK          WHEEL-MOTOR RAMP
( ramp stepping motors)
OK          LIMIT-SWITCHES?
( exit loop if motor limited)
OK          SYNCHRONIZE
( synchronize to motor pulse)
OK     LOOP
OK ;
```

A physician or clinical technician is much more of an end-user than the researcher. As such, they are less concerned with words that allow them flexibility in control of peripheral hardware; instead they want words that control hardware in specific ways towards some specified clinical objectives. Thus they need to implicitly use both basic FORTH words and peripheral driver words, but want to only explicitly know words that achieve specific aims. But even here FORTH can be appreciated. It allows a flexible, conceptual system with a non-confining syntax. With the pulmonary microcomputer, the physician might typically have the following dialog:

```
OK PULMONARY CALCULATIONS
( acquire data, and calc it )
OK PRINTER SHOW RESULTS
( print results )
OK DME VIEW
( view plots of gases on )
OK C2H2 VIEW
( ... graphics display )
```

By learning a limited, yet full, vocabulary of perhaps twenty to fifty well chosen words, these non-technical users can effectively use a FORTH based microcomputer with little training or understanding of programming. And without fail, they learn to use colon definitions to group these basic words to their own specific usage patterns.

Common Software Packages

As we have just seen, I group FORTH software in three coarse categories corresponding to types of users: basic FORTH system software, peripheral support extentions, and custom applications. The basic system software does not vary at all while custom application software is unique to each end-user system. Peripheral support software is in a hazy area. From the point of view of documentation and support, any given type of peripheral should appear uniform between systems; but at the hardware level, each type of peripheral varies in myriad details. By creating common software packages with this in mind I have been able to avoid constantly recreating software because of hardware variations.

Common software packages can do more than just ease support for similar systems. It can effectively hide hardware details from the user, thus making dissimilar A/D converters, for example. appear identical from the software point of view. And a well designed set of driver software also imparts increased capabilities to a system than just those of the "raw" hardware. Let's look at a few examples of software peripheral drivers to reinforce these points.

Many of these microcomputers are used for data acquisition purposes involving different types of A/D converters and real time clocks. From a hardware point of view, some of these A/D's have eight bit versus twelve bit resolutions. Some have seven or eight analog multiplexer channels while others have sixteen. Some of the real time clocks have fixed 60 Hz resolutions, others are programmable.

From a conceptual point of view, these data acquisition systems all operate identically: they can randomly sample multiple analog signals at some specified rate. The driver software implements these concepts using two words: SAMPLE and DELAY. SAMPLE takes an integer multiplexer channel number as an input argument, and returns an integer amplitude value. It works identically no matter what hardware is controlled by it; the multiplexer addressing and A/D digital output format are hidden from the user. Similarly, the real time clock works in a manner transparent to hardware specifics. DELAY requires only an input argument to specify the number of real time clock "ticks" to delay.

But the conceptual basis of the data acquisition package transcends just the A/D hardware; there must be some place to put the data. This may be on the parameter stack, in data arrays, or in disk based virtual arrays. When this capability is added, the data acquisition specific hardware creates a synergy with the fundamental system hardware such as read/write memory or floppy disk.

Another example of a peripheral driver package that I developed is a memory-mapped video graphics package. The typical hardware interfaces ranged from 240x256 resolution up to 512x480 resolution, with as many different methods of addressing specific dots on the display.

Conceptually, we want, first of all, to be able to plot physical X,Y points independent of hardware specifics. A word such as PLOT, using X and Y integer parameters on the stack top, can give us this capability very readily.

But to really use graphics effectively, it is nice to be able to specify different areas on the video screen to plot different data, as well as scaling functions to adopt logical coordinates to this specified graphics area. The GRAPH data type (built with a defining word) allows these different graphics areas and scaling functions to be associated, and invoked, by a common name. Further capabilities were added to allow easy creation of vectors, grids, tick marks, axes, and boxes. All of a sudden, a very proletarian graphics peripheral is transformed into a powerful tool. And because these new functions are all built on the PLOT word, they are readily tansferred between systems with different hardware interfaces.

A final software driver to consider is that of the hardware floating point unit. It is interesting to consider this from both a FORTH, and a conventional language point of view. In a language such as PASCAL, the system generally has built in software based operators for floating point. Because the system is not inherently extensible, the addition of a hardware floating point peripheral requires either a manufacturer rewrite of the PASCAL floating point routines, or else a user interface through PASCAL functions or procedures. The former requires manufacturer acceptance and support of a new hardware peripheral; unless a very popular device, such support will be reluctant at best. The latter requires a very awkward language syntax to invoke hardware floating point capabilities. Either way, the problem is that the hardware has to be forced to conform to the manufacturer's language standard.

At the Medical Center, a hardware floating point package was easily added as an extention to the basic FORTH system; the language adopted the hardware!

Anachronism or Portent?

At this juncture it is valid to ask if FORTH justified itself in its use at the University of Rochester Medical Center. Is it an anachronism of the past, or a philosophy portending the future?

Admittedly, FORTH is somewhat limited without such things as a file system or procedural name scoping of variables. Perhaps there should also be less explicit knowledge of addresses, and more system security. Perhaps. But if so, then these things will be evolved as FORTH matures.

It is what FORTH espouses, though, that justifies its use. It allows hardware components to dictate the software design, thus allowing rapid incorporation of technological advances. Other languages force conformance of hardware to language standards--a slow, expensive process.

FORTH allows isolation of users from hardware dependencies, and adds capabilities to the basic hardware. The result is a user environment that supersedes specific machine configurations with concept oriented, yet free syntax, computer operation. The FORTH system developer might need to know "how", but the system user need only know "what". Conventional systems, to the contrary, generally require everyone concerned to ask: "why?"

FORTH encourages an exploratory development technique. A user can choose between interactively trying concepts, writing full programs, editing programs. compiling programs, and/or debugging programs. He or she can do this in a single, consistent FORTH environment, utilizing any of these phases of development as required. The result is efficient use of all system resources.

The embodiment of the FORTH philosophy is that programming is not what it is often taught to be: the application of top-down programming techniques to a single problem. Instead, it involves a series of interrelated problems all related to system use. This might mean a set of words that allow a researcher to control a TV digitizer, or it may mean a series of words to calculate and graphically display the results of a mathematical analysis. While the series of capabilities needed will always vary between different systems, it is only by providing a rich enough vocabulary that a user can have a flexible, effective, and friendly system. FORTH is unique among languages in that it encourages the programming of solutions!

Peter Helmers is a senior laboratory engineer in the diagnostic ultrasound research laboratory within the Department of Radiology at the University of Rochester Medical Center.

Helmers' article continued
on next two pages

BUG FIXES

Correction to FEDIT

Sorry you had trouble with FEDIT. The listing was retyped at FIG and several typos creeped in. They are:

1. SCR 64 Line 10: compile should be COMPILE
2. SCR 65 Line 23: 1+ /MOD should be 1+ 16 /MOD
3. SCR 67 Line 48: B/BUD should be B/BUF
4. SCR 67 Line 49: : E should be : .E
5. SCR 67 Line 50: + ALIN should be +ALIN

You are perfectly right that source text should be loadable. I talked to some of the people at FIG about this and they were acutely aware of the problem but they are simply not set up to directly reproduce listings in FD at the present time. They do the best job they can with the resources available to them, and they work darn hard at it. I can't fault them.

REPL is a pseudonym for the fig-FORTH line editor definition, R . I used the pseudonym because FEDIT was the first program I wrote in FORTH and I wasn't really familiar enough with Vocabularies to comfortably use a word that was already used in the FORTH vocabulary.

Let me know how it works for you. If you would like a machine produced listing, I could run one for you from my current version. Let me know. Good luck.

Edgar H. Fey
18 Calendar Court
La Grange, IL 60525

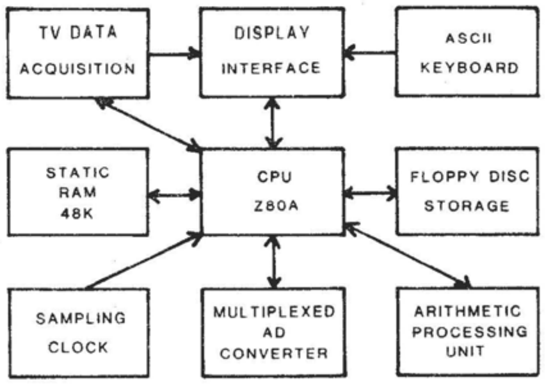


Fig. 1: Block diagram of a typical S-100 based microcomputer; this one is used to study blood vein mechanics.

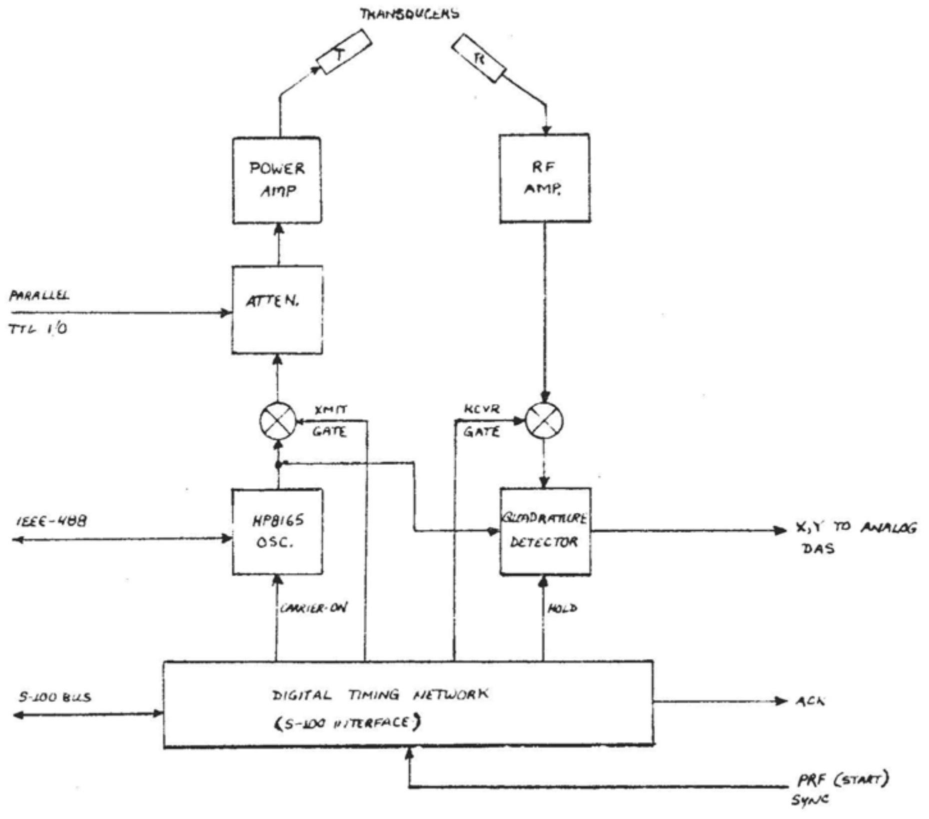


Fig. 2: Block diagram of UDA analog electronics timing control interface. Microcomputer sets up interface parameters, but timing then runs independently using PRFSYNC and ACK handshaking signals from Nova Minicomputer data acquisition system. Because the microcomputer can synchronize to timing hardware, other capabilities such as attenuator and frequency control can be utilized.

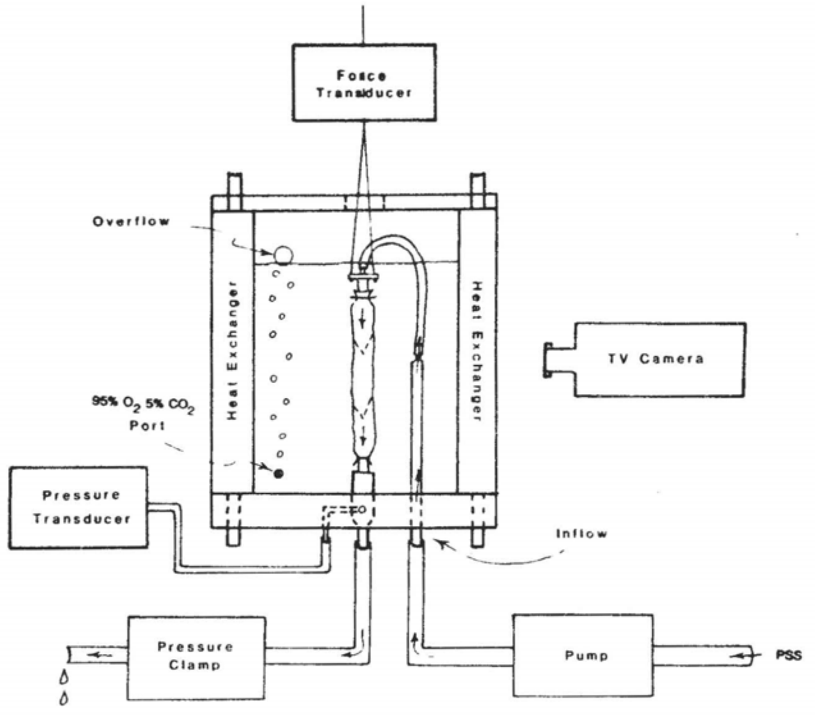


Fig. 3: Diagram of vein mechanics experimental chamber. Microcomputer samples pressure and force signals, and determines vein diameter from software analysis of TV image.

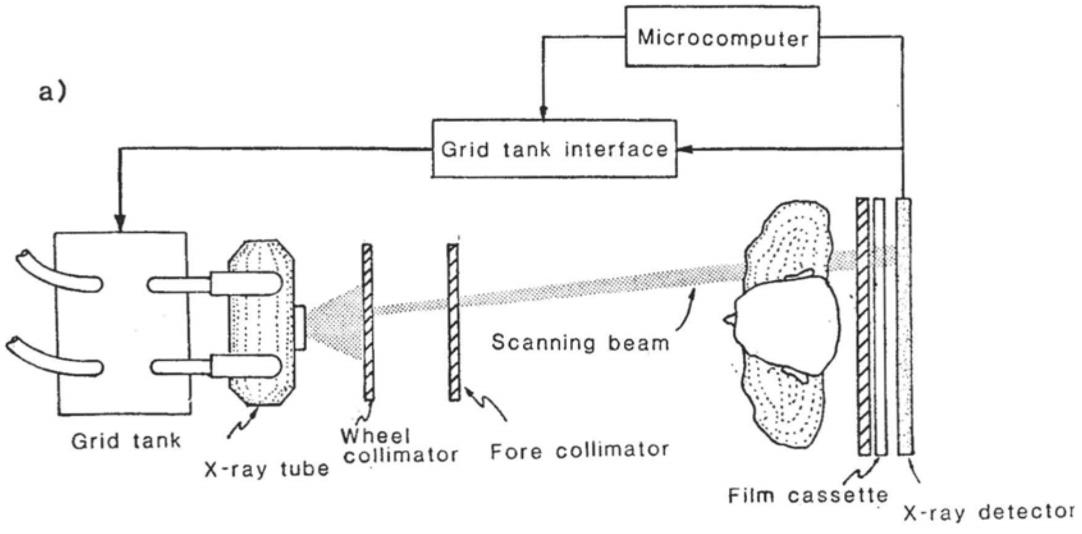


Fig. 4: Diagram of X-ray scanner apparatus showing how wheel collimator and fore and aft horizontal collimators, controlled by stepper motors, create a mechanically scanned X-ray raster. The microcomputer, with A/D and D/A interfaces, also monitors and controls X-ray exposures.

DATA STRUCTURES IN A TELECOMMUNICATIONS FRONT END

John A. Lefor
University of Rochester

Asbtract

URTH, the University of Rochester dialect of FORTH, was used to implement a telecommunications front end for an IBM 3032. This package provides access to the IBM 3032 from as many as 160 ASCII terminal at speeds up to 9.6Kb. Each of these terminals contend for 128 simultaneous connections at the IBM computer.

The reasons for choosing URTH as the development language and a review of the major advantages and disadvantages of using Urth for this project is discussed. Also, some conclusions as to the applicability of URTH, and the data structures used in this application is reviewed. The use of conventional data structures for providing information paths between the various components of the system is examined and the possible advantage of less conventional data structures more firmly based in URTH constructs is explored.

A plan for development of similar systems is presented which integrates some of these concerns and promises a better structured system.

Introduction

In 1977, the University of Rochester Computing Center first got involved with the FORTH language. The initial development in FORTH was the implementation of various flavors of the FORTH system known collectively as URTH. Most of the URTH systems developed have provided multitasking capability on a variety of micro-, mini-, and mainframe computers. During the development of the various URTH systems, a number of people within the Computing Center showed interest in using an URTH based system for development of real projects rather than viewing URTH as just another academic curiosity.

Concurrent with the development of the URTH system, was the growth of telecommunications in computing at the University. A need for additional telecommunications lines into the computer was fast becoming a necessity and the financial support for such a purchase was on the verge of becoming a reality.

In this environment, the design and implementation of a locally designed telecommunications front end was beginning to emerge. The front end had to exist in an academic computing center where the need for teleprocessing was growing. The front end had to communicate with an IBM host (it was generally believed that the IBM environment was at the University for many years to come). The front end had to provide access for the ever growing number of ASCII terminals being purchased for both computing and noncomputing environments. Importantly, the front end had to provide for access to the IBM host from more terminals than could be dedicated to the host at any one time. The only front end which could possibly meet these goals and be reasonably cost effective had to be one of local design. meeting local requirements.

Features Provided

The front end designed at the University of Rochester Computing Center does provide some unique features to the users of our IBM 3032 computer. To be sure, the features are not unique within the context of computing, but are not generally available in an IBM mainframe environment.

One of the major advantages provided by the locally designed front end is the ability to switch between systems from the same terminal. In a traditional (non-SNA) IBM mainframe, it is not always convenient to have a terminal switched between different software teleprocessing applications. Typically, a terminal either is connected to one application or another. With the locally designed front end, it is possible to choose the application ot which the terminal is attached. In effect, the front end is a port contender for various applications on the mainframe.

The second major feature arising from a local front end is the ability to support an XON/XOFF protocol. Since the IBM mainframe communicates with its terminals in a half duplex mode, XON/XOFF support is not traditionally available. The local front end is based on full duplex communication to the terminal so XON/XOFF can be supported in a fully effective fashion. Those terminals which have buffers which can overflow can turn off the input at will, a feature not available without special support in the IBM world.

The front end is today running at the University of Rochester Computing Center. It is supporting 160 ASCII terminals contending for 128 host computer ports. Each terminal can select connection speed between 110 and 9600 Baud as well as a few other tailored features. The fact that the implementation continues to run frequently appears to be a miracle but represents some faith that the concept is at least essentially sound.

Hardware Decisions

In order to implement the telecommunications front end to an IBM computer, the processor chosen for the implementation had to provide the capability to interface to an IBM byte multiplexor channel. Since the protocol for channel interfacing is non trivial, there are a limited number of vendors of minicomputers who were able to provide this interface capability. Another important consideration in the design of a telecommunications front end is the realization that if a failure should occur in the front end, there is a perception that the host computer failed. Because there is great need to access the host computer, it is undesirable to have hardware failures affecting the front end. To this end, the mini-computer chosen as the front end had to have both a history of reliable service and a maintenance team capable of repairing any difficulty with a minimum of fuss.

In evaluating the available minicomputers against these criteria, the processor which was finally chosen was a Digital Equipment Corporation PDP 11/34. The interface to the channel is via a DX-11B, and the ASCII terminals are supported by DZ-11's (actually many of the terminals are supported by a Digital Communications Associates 205, which emulates 32 lines of DZ-11 on a single quad height board).

In retrospect, we can see that though the PDP 11/34 does work in the required environment there are some deficiencies. The most notable is in the maintainability of the DX-11B (the channel interface which connects the PDP 11/34 processor to the IBM processor). There are so few DX-11B's in production throughout the United States that the DEC customer engineers are relatively unfamiliar with the details of its operation. When subtle problems have occurred, the repair of the problems has taken considerable time and talent. To be sure that the subtle difficulties were discovered and corrected is a tribute to the engineers dedication to the problem, but a more popular interface would probably have been repaired in a shorter time.

Software Decisions

In determining the nature of the software to run for this application, it was necessary to evaluate the probable structure of the end goal and to consider all the concerns of a project of this sort. After the major considerations are evaluated, the best software choice can be made based on the concerns and knowledge of what is available.

A telecommunications front end is a realtime device which must be able to handle a relatively large number of potential I/O devices. In particular. many terminals are expected to be connected to the front end. Also, there were considerations for attachment of synchronous lines for support of Hasp Bisync, Remote 3170's, and local area network communications. All these considered together, it was important to choose a software implementation which provides support for reltime device handling.

The wide variety of I/O devices which were contemplated for the front end also reuired that the software provide tools to help the designers of the system gain understanding of a wide variety of hardware devices. There were going to be asynchronous and synchronous devices as well as a channel interface which had no well defined characteristics (the best documentation of how the DX-11B worked was found in the diagnostic programs supplied for hardware maintenance). In addition, there was always the possibility of needing to support a new and different class of I/O device. Though the manuals documented how the hardware worked, any software which would allow interaction with the unfamiliar hardware would be beneficial in the debugging of the overall system.

Another area of debugging which was considered in the software choice was the software protocols. The connection to the channel of an IBM computer by asynchronous ASCII devices invokes a non-trivial set of software protocols. A simple example of the kinds of problems is in the transmission of any single ASCII character to the channel. In the IBM environment, the software running in the processor expects that any ASCII characters transmitted from a telecommunications front end are sent not as simple ASCII characters (as generated by the terminal), but rather demands that each ASCII character be bit reversed.[1] Though this is not a difficult feat to accomplish, it points out the nature of some of the software protocol issues which must be dealt with in a telecommunications front end. Suffice it to say the software used to design the front end would benefit the designer if it helped to identify, and resolve, software protocol issues.

In the development of any realtime software project, it is recognized that the throughput of the system is important. The telecommunications front end is no exception. Since there are to be a large number of I/O devices providing input to the software application asynchronous to the operation of the software, it is imperative that the application software be able to keep pace with the demand. On the other hand, the inability of the front end to keep pace with the demand is not critical. If a character destined to a terminal is lost, a human being will not die but a programmer may get upset. Keeping these priorities in mind the project had to be implemented in an environment which was not wasteful of processor time, but there was no need to be alarmed if there was the potential to loose data.

The hardware decision made specific features of the processor had to be considered in the software choice. Specifically, the PDP 11/34 had 64K bytes of memory. We had to have some degree of confidence that the entire system could be packaged in 64K bytes. If that was not possible, the development time could be slowed down waiting for shipment of additional memory. The speed of the 11/34 processor led us to believe we would have sufficient CPU to do the job, but not a lot to spare.

The final and perhaps major consideration which affected the choice of software was the perceived development time. The project was initiated at a time when there was an extra IBM processor at the University. It would be possible to design and debug the entire front end on a processor which was not in use. That was a real opportunity not to be passed up. However, the processor could not remain idle for too long a time. Any software package which could help to shorten the development time and thereby allow debugging of the front end on the unused processor would be of great benefit to the implementation.

Alternative Software Strategies

Examining the issues in making the software choice, there appear to be three alternative software strategies. The use of assembler language, the use of a high level language such as C or Fortran, or the use of URTH.

Assembler language provides a number of solutions to the problems outlined. It tends to be compact in memory usage, it certainly has the potential to make most efficient use of the limited CPU, and it is quite capable of handling the foreign devices needed for a front end. However, the assembler has a few drawbacks. Probably the major difficulty with assembly language is the extended development time. Debugging is slow and tedious and design of code and data structures to aid debugging is totally a responsibility of the programmer. Thus, development of a major application in assembly language is concerned both with the solution of the problem but also much effort is spent on good design and coding techniques. Another difficulty with the assembler is maintainability. Each programmer has an individual design style. The documentation rests largely in design of the code. If the original designer is no longer available for maintenance of the project, there is a long learning curve to train a new individual.

High level languages solve many of the difficulties with assembly language. If the language is well conceived for a realtime problem, it will support the difficult hardware issues and will provide a framework for data structure design which provides readability and maintainability of the software. A major difficulty with high level languages is their use of memory, and sophisticated operating system services. These two concerns may make a larger faster CPU needed for effective execution of the application. Another drawback of both the assembler and high level solution is the lack of inherent interactive develoment and debugging tools. They typically can be designed into the system, but they generally are not present in the basic environment.

Evaluation of URTH

URTH appears to meet many of the goals in the software choice. Though there are limitations, the advantages seem to outweigh the disadvantages especially when design time is so important a consideration.

When looking at URTH, a clear advantage afforded by URTH is implementation time.[2] Most of the other advantages provided by URTH can be directly tied to the speed of implementation. URTH provides easy access to any set of unusual devices, because the device handlers are ach tailored to the system and the hardware. Once a program is debugged in URTH, there is good reason to believe it will continue to work.[3] Another major advantage offered in the URTH environment is the enormous flexibility in design of both source codes and data structures. The ability to code both high level URTH and machine level code and to achieve a uniform interface provided many opportunities to speed up inefficient code. The ability to design new data strucutres to work in a large scale environment offers much flexibility in design.

The URTH environment is not without fault. The fact that URTH is an interpreter does mean the code is not as efficient in CPU speed as possible. Of course, the ease of generating assembly code helps alleviate this problem. However, a major drawback of the URTH environment stems from its flexibility in data structure design.

The very fact that it is possible to design any needed data structures coupled with the implementation of the traditional data structures of arrays, constants, and variables created some difficulties in the design of system which had so much pressure for development in a short time. There was not a lot of time spent on development of the best data structure for the problems encountered. Rather, traditional data structures were used to meet individual demands. In particular, many arrays were implemented for storing of information relating to specific I/O devices, and queues (obtained from a freepool) were used to buffer data between devices. The use of such data structures had two major impacts on the project. First, the queues were sufficiently difficult to handle as to have impact on the speed of the overall system.[4] The use of the arrays to hold information for later processing yielded much difficulty in debugging individual words and tended to leave side effects which had impact on words already debugged.

Thus, the use of URTH has many virtues but it is crucial to recognize the particular issues which may lead to difficulty in debugging. Using data structures such as arrays and variables to communicate information between tasks in the front end tended to leave open many portential pitfalls in the debugging and design of a system as complex and highly integrated as a front end.

Alternative Design Strategies

In examining the resulting front end for difficiencies, it becomes clear that there are some strategies for alternative design which could limit the difficulties encountered in any similar realtime project, and would make URTH a vehicle for well designed, well integrated, and effective systems design.

The issues of code design are well considered in URTH. The ability to switch between machine level code and high level URTH provides a classic tradeoff between speed of execution and memory utilization. The fact that the interface between both environments is standard allows all design in high level URTH, and conversion to machine code when and where appropriate. In this area, URTH provides sufffícent tools and a good set of options.

In the data design area, URTH provides so many options that the best data structure choice is very much at the control of the programmer. In the case of the front end design, the traditional data structures were not sufficient to effect the job but there was insufficient time to design an optimal data structure. In retrospect, it is possible to peruse the alternatives and choose a structure which provided the flexibility needed, and also limits the side effects from preventing effective debugging of words.

One of the major advantages URTH provides over alternative software approaches is the stack. Proper design of URTH words with parameter passing via the stack helps to insure that a debugged word will tend to continue to work, and will have no side effects Given this observation, it would be natural to use the stack to pass parameters in the telecommunications environment. Unfortunately, the stack is not useful in communication between tasks, and the stack is difficult to address and use when too much information is passed. In the front end, there are so many unrelated parameters which need to be passed between tasks that the stack is not useful. But, the concept of a stack does solve one of the major difficulties encountered in the front end design. Given this set of considerations, it seems like a good idea to define a "named object stack"[5] for each I/O entity defined in the telecommunication environment. When a particular I/O device needs some form of service, the named stack is invoked and all data relating to the I/O device is available. The stack can contain pointers to ring buffers as well as current status of the device. Using this strategy provides an environment that naturally fits within the basic strucutre of URTH programming, makes effective use of constructs within the URTH system, and promotes good URTH programming practices which minimize the side effect problems. Overall speed of the application is not significantly impacted and many old functions can take advantage of the data structure.

The stack will contain sufficient volumes of information about each I/O device that it may be advisable to create a "framing" of the stack. This would allow access to individual parts of the stack as if it were the current top of stack, thus allowing access to more data in a convenient notation.

Summary

The telecommunications front end designed and implemented at the University of Rochester Computing Center is a useful model of many realtime applications. In the design are found a number of flaws which are primarily related to the particular pressures present at the time of the design. The choice of URTH as the software vehicle appears to have been an excellent one however, the choice of data structures to use within the URTH environment was not as well conceived.

URTH provided a software environment which clearly effected time effective development of a complex system. It provided a comprehensive interactive debugging environment with the ability to address specific speed inefficiences in a uniform manner. The major drawbacks to the URTH environment resulted from the choice of data structures for intertask communication within the application.

URTH does provide tools to develop the optimal data structures for any particular application. In the case of realtime applications, the choice of data structures is particularly critical. From my experience, I believe that a data structure similar to the named object stack would benefit many realtime applications in URTH both function provided and in the limiting of side effects so prevelant in global data strucutres such as arrays.

A second feature which would be valuable in an URTH environment would be any useful stand-alone dump with indexing to help the programmer walk through the dictionary. When total application collapse occurs, URTH is not very informative as to the nature of the problem. A memory dump (with a good index for the dictionary) would help to debug some rather sticky timing problems.

Overall, URTH is a good choice for development of realtime applications, but care in the design of data structures should help to make the overall maintenance of the application a simpler chore.

Footnotes

1. This is not simple an example of a perverse IBM, but instead is another fact of IBM computing history. The standard device IBM used to connect ASCII terminals to the host (a 270x) was not designed using today's UARTS, rather it collected the bit serial data in a register. The data was collected in a register in such a way as to cause the characters to be captured in bit reverse order. Rather than correcting the problem in the front end, they transmitted the bit reversed ASCII to the host, and translated the bit reversed ASCII to EBCDIC for processing. The software stayed, so the need for bit reversed ASCII exists today.

2. This advantage was certainly realized in the actual project. The basic system was operational within four months from beginning of the project.

3. This is dependent upon good URTH programming practices. But, in our project there became clear a self evident truth. We attempted to debug so many "words" which were already correct, we began to believe that it is very difficult to debug a working program.

4. Converting most of the queues to individually assigned ring buffers speeded up overall processing by 20% or more.

5. See Peter Helmers, "Userstack", FORTH DIMENSIONS, Vol. III, No. 1 and Peter Helmers, "Alternative Parameter Stacks", Proceedings of the 1981 Rochester FORTH Standards Conference.

Acknowledgements

I would like to thank Richard Marisa, Ken Hardwick, Mike Armstrong, and Mike Williams for their assistance.

J.A. Lefor was senior systems programmer at the University Computing Center at the University of Rochester and is now Data Communications Manager.

MAPPED MEMORY MANAGEMENT TECHNIQUES IN FORTH

Rosemary C. Leary
Carole A. Winkler
Laboratory for Laser Energetics
University of Rochester

Abstract

Three techniques for using memory management hardware in a FORTH system have been implemented at the Laboratory for Laser Energetics at the University of Rochester. One method uses mapped memory for data storage by creating a "data window" in the logical address space. A second method increases the available space for programs by mapping tasks in a multi-tasking system. The third uses mapped memory for data storage by taking advantage of special instructions and a second set of memory management registers.

Introduction

The problem of insufficient memory for programs or data is commonly encountered on computers with a 16 bit word size. Many manufacturers now offer hardware to alleviate this problem. At the University of Rochester's Laboratory for Laser Energetics we have devised solutions to three different aspects of the problem using FORTH on PDP-11/23 and PDP-11/34 computers.

Two applications at the Laboratory had a need for large image processing arrays (up to 100K words). We solved this by using a double precision array index which maps physical memory into a logical memory "data window" within the FORTH system.

On a different, very large FORTH application, we needed both more program space and more data space. We increased the amount of program space by implementing a multi-tasking system in which certain portions of memory contain the nucleus and common code, while other portions are task specific and are periodically switched in and out of active use.

To increase the available data space we are using special instructions and a second set of memory management registers on the PDP-11/23 and PDP-11/34 computers.

Additional material on these systems can be found in "FORTH in Laser Fusion," by Larry Forsley, in this issue of FORTH DIMENSIONS.

Hardware

The memory management hardware on the PDP-11/23 and PDP-11/34 computers consists of two sets of registers that map 16 bit logical addresses into 18 bit physical addresses. One set of registers is used when the processor is in "kernel" mode, the other when it is in "user" mode. The mode is determined by two bits of the processor status word.

Each set of registers contains eight 32-bit Active Page Registers (APR's). Each APR is actually two registers: the Page Address Register (PAR) which contains a base address, and the Page Descriptor Register (PDR) which contains the page length and the access control key.

The 16-bit logical address space is divided into eight "pages" shown in Table 1. When the memory management unit is enabled, any access to memory will be mapped through the APR for that address.

Page	Logical Address Range (octal)
0	0 - 17776
1	20000 - 37776
2	40000 - 57776
3	60000 - 77776
4	100000 - 117776
5	120000 - 137776
6	140000 - 157776
7	160000 - 177776

Table 1. Logical Address Space.

The physical memory address that will actually be accessed is a combination of the logical address and the PAR for that page. Figure 1 shows how the logical address is deriv d. Bits 15-13 of the logical address give the page (or APR) number. The PAR for that page gives the base address in 64 byte blocks. This value is added to the block number field of the logical address (bits 12-6) to find bits 17-6 of the physical address. Bits 5-0 of the physical address are the same as bits 5-0 of the logical address.

Figure 2 shows the logical address space.

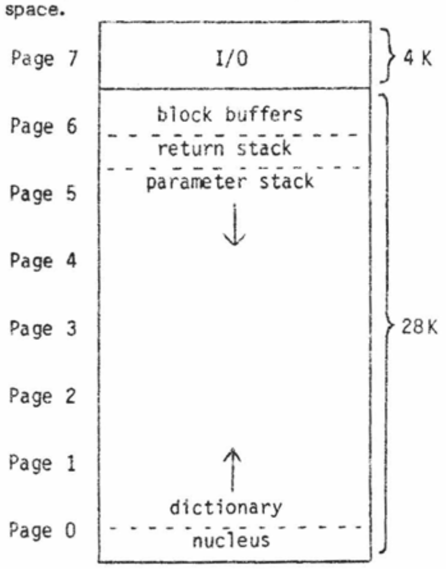


Figure 2. Logical address space for single task without mapped memory.

Additional information on the PDP-11 memory management unit can be found in the processor handbook[1].

Data Window and Memory Management

One way to utilize the memory management hardware and additional memory is to use it for data storage. Two of our applications at LLE require large data arrays (up to 100K words) for image processing. We solved this problem by creating a "data window" in our logical address space. Figure 3 shows the logical address layout of a system with a data window.

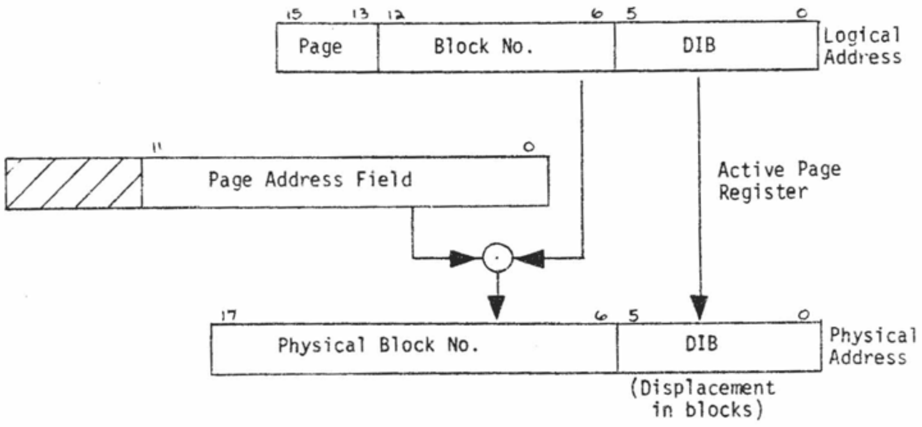
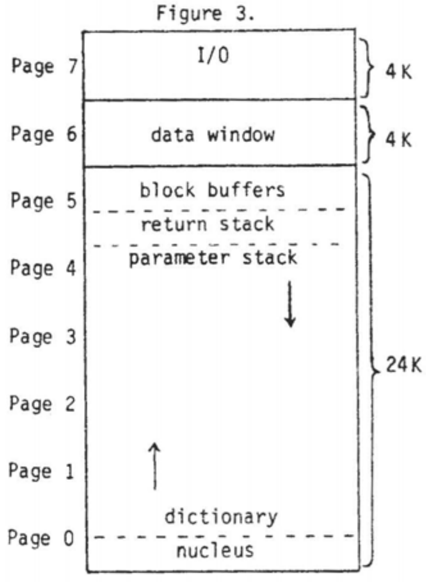


Figure 1. Construction of a Physical Address

(derived from figure 7-9 of [1] and reprinted with permission from DEC.)

Figure 3.



Logical Address Space With Data Window.

The block buffers, return stack, and parameter stack are moved down to the top of the next 4K word page of logical memory, leaving a 4K word gap in the logical address space. In a 128K word system, 100K words of physical memory are then accessed through this window.

The X and Y coordinates of the image array are converted to a double precision index. This is done by multiplying the Y coordinate by the number of pixels per line and adding the X coordinate. This index is divided by the number of pages per image. The quotient indicates which page the pixel is in, and the remainder will be the address offset of the pixel into the page.

The relocation constant for the needed page is set in the PAR so that it can be accessed through the data window. The logical address of the pixel is obtained by adding the address offset to the starting address of the data window.

Multi-tasking and Memory Management

Our version of FORTH implements multi-tasking in the following manner. Each task has a "state vector" which contains "user" variables that can differ from task to task. This includes:

- Dictionary and stack pointers
- Program counter and interpreter pointer
- Status flags and state indicators
- Terminal I/O routines and buffer pointers
- Vocabulary pointers
- Number base

The state vector for the master task is included in the nucleus.

Each task also has its own terminal buffer, dictionary, parameter stack, and return stack. New tasks are created with a routine called BLDTASK which allocates space for them in the master task's dictionary. Figure 4 shows the logical address space in an unmapped multi-tasking system.

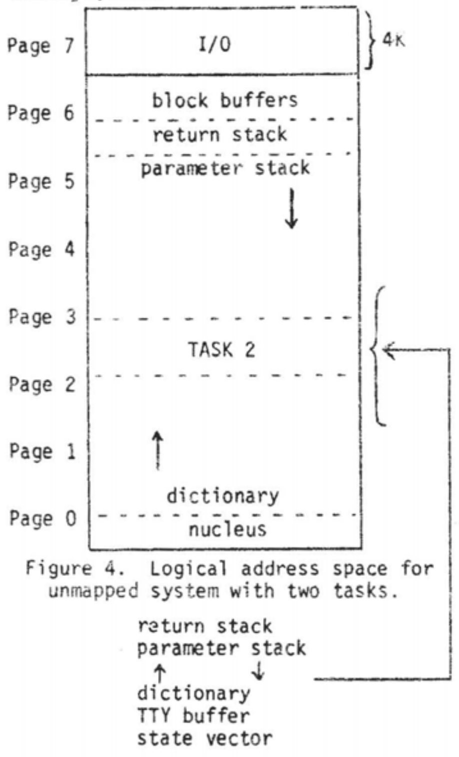


Figure 4. Logical address space for unmapped system with two tasks.

Task state vectors are linked to each other in a circular fashion, one pointing to the next and the last back to the first. A "round robin" scheduler starts running a new task when the current task executes a PAUSE. PAUSE stores the current machine state into the state vector of the existing task and sets the new machine state according to the new task's state vector.

Additional information on multi-tasking can be found in works by Forsley[2], McCourt[3], and Leary and McClimans[4]. Figure 2 shows the logical address space of a FORTH application with a single task and not using memory management.

To add program space to our multi-tasking system, we reserved a "task window" in the logical address space. The master task occupies the low five pages of address space. Code in this area is usable by all tasks.

Mapped tasks occupy pages 5 and 6 of the logical address space. Definitions and data within a mapped task are accessible only to itself. Each task must have a separate vocabulary. If definitions in a mapped task are entered into the FORTH vocabulary, the dictionary links will be gone when the next task becomes active. This usually results in a system crash. Figure 5 shows the logical address space in a mapped multi-tasking system.

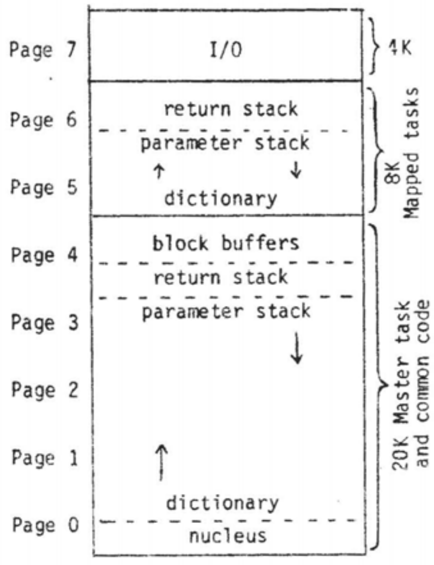


Figure 5. Logical Address Space for Mapped Multi-tasking System.

Implementing this technique required the following changes:

- Modify the scheduler PAUSE so that it sets the page 5 and 6 memory management registers, as well as swapping in the usual state vector information.
- Move the block buffers and master task stacks to the top of page 4.
- Change the routine BLDTASK to assign the new task's return stack, parameter stack, and dictionary to pages 5 and 6, instead of giving them space in the master task's dictionary.
- Change BLDTASK to assign physical memory to the task. It must calculate the appropriate settings for APR 5 and APR 6 and save them in the task's state vector so that they can be loaded into the memory management registers by PAUSE.

User Space for Data

The two approaches discussed previously both ran in processor "kernel" mode. To increase our memory resident data storage in the multi-tasking application described previously, we use the "user" mode memory management registers.

The processor status word has two mode fields: current mode and previous mode. The instruction MFPD moves a word from the "previous" mode address space to the "current" mode processor stack (the return stack in our FORTH implementation). The instruction MTPD moves a word from the "current" mode processor stack to the "previous" mode address space.

Using these instructions it is possible to retrieve and store data quickly and efficiently, and the data stored there is accessible to all kernel mode programs, whether they are mapped tasks or not. Data tables that would otherwise need to be disk resident because of their size can now be memory resident to speed response time.

The source listing of the user mode data storage code is included at the end of this article.

Conclusion

The first technique, the data window, has been used for two image processing applications. One is used to view infrared and ultraviolet laser beams in materials damage testing experiments. The system does circular averaging and calculates an absolute intensity within the 10 minute shot cycle.

The other image processing application observes X-ray diffraction patterns produced by a nanosecond X-ray source. A technique of radial averaging is also used here to enhance the diffraction pattern and study changes induced by sample stimulation.

The second and third techniques are used on the Omega Alignment System, which now has 17 tasks installed and uses about 140,000 bytes of memory for program space. The user mode data storage method is used by the data base software and for the intertask message queues.

Although this paper describes techniques used with DEC PDP-11 series computers, the techniques are similar to those used with any limited address system with logical/physical mapping hardware. Thus, they are applicable to minicomputers like the Hewlett-Packard 1000 series and the much newer 16 bit microcomputers like the Motorola 68000 and Zilog 8000. The techniques are especially appropriate in a FORTH-79 context where the FORTH machine is defined as having a 64K byte address space, carved out of an arbitrarily large physical address space.

Acknowledgements

The following people played a major role in the development of the software described in this article: Donald P. McClimans, Lawrence P. Forsley, Reade B. Nimick, Robert D. Frankel, Joseph A. Abate, and Robert L. Keck.

This work was partially supported by the following sponsors: Exxon Research and Engineering Company, General Electric Company, New York State Energy Research and Development Authority, Northeast Utilities, The Standard Oil Company (Ohio), the University of Rochester, Empire State Electric Energy Research Corporation, and the U.S. Department of Energy inertial fusion program under contract number DE-AC08-80DP40124.

```
**********************   BLOCK #   445   **********************

( MEMORY MANAGEMENT - U@, U! )
CODE U@  ( [ADRS]---[DATA] RETRIEVE FROM USER MODE MEMORY )
   7777760 @#  300000 #  MOV,  ( SET PROCESSOR STATUS WORD: )
                               (   CURRENT=KERNEL, PREV=USER )
               S @)+     FPD,  ( FROM ADRS ON STACK TO RP )
   7777760 @#  0 #       MOV,  ( PSW BACK TO NORMAL )
   S -)        RP )+     MOV,  ( RP TO STACK )
                         NEXT, ( RETURN )
CODE U!  ( [DATA][ADRS]---[] STORE IN USER MODE MEMORY )
   RP -)       2 S I)    MOV,  ( DATA FROM STACK TO RP )
   7777760 @#  300000 #  MOV,  ( SET PROCESSOR STATUS WORD: )
                               (   CURRENT=KERNEL, PREV=USER )
               S @)+     TPD,  ( FROM RP TO ADRS ON STACK )
   7777760 @#  0 #       MOV,  ( PSW BACK TO NORMAL )
               POP       J,    ( RETURN WITH CLEAN STACK )
-->

**********************   BLOCK #   446   **********************

( MEMORY MANAGEMENT - K>U )
CODE K>U  ( [K ADRS][U ADRS][COUNT]---[] COPIES 'COUNT' )
      ( WORDS FROM KERNEL SPACE TO USER SPACE )
      W          S )+      MOV,  ( W=COUNT )
      R0         S )+      MOV,  ( R0=USER SPACE ADDRESS )
      R1         S )+      MOV,  ( R1=KERNEL SPACE ADDRESS )
      7777760 @# 300000 # MOV,   ( SET PROCESSOR STATUS WORD: )
                                 (   CURRENT=KERNEL, PREV=USER )
      BEGIN,
        RP -)    R1 )+     MOV,  ( FROM KERNEL SPACE TO RP )
                 R0 )+     TPD,  ( FROM RP TO USER SPACE )
      W SOB,                     ( DEC W, BRANCH IF NOT ZERO )
      7777760 @# 0 #       MOV,  ( PSW BACK TO NORMAL )
                          NEXT,  ( RETURN )
-->

**********************   BLOCK #   447   **********************

( MEMORY MANAGEMENT - U>K )
CODE U>K  ( [U ADRS][K ADRS][COUNT]---[] COPIES 'COUNT' )
      ( WORDS FROM USER SPACE TO KERNEL SPACE )
      W          S )+      MOV,  ( W=COUNT )
      R0         S )+      MOV,  ( R0=KERNEL SPACE ADDRESS )
      R1         S )+      MOV,  ( R1=USER SPACE ADDRESS )
      7777760 @# 300000 # MOV,   ( SET PROCESSOR STATUS WORD: )
                                 (   CURRENT=KERNEL, PREV=USER )
      BEGIN,
                 R1 )+     FPD,  ( FROM USER SPACE TO RP )
        R0 )+    RP )+     MOV,  ( FROM RP TO KERNEL SPACE )
      W SOB,                     ( DEC W, LOOP IF NOT ZERO )
      7777760 @# 0 #       MOV,  ( CURRENT=KERNEL, PREV=KERNEL )
                          NEXT,  ( RETURN )
;S
```

R.C. Leary is a consultant employed by the Engineering Division of the Laboratory for Laser Energetics. C.A. Winkler is an undergraduate in the Department of Mathematics, University of Rochester.

References

1. Microcomputers and Memories, Digital Equipment Corporation, Maynard, MA 01754, 1981.
2. Lawrence P. Forsley, "FORTH Multitasking in URTH," Proceedings of the 4th West Coast Computer Faire, 1979.
3. Michael A. McCourt, PDP-11 FORTH-79 Implementation Guide, University of Rochester, Laboratory for Laser Energetics, 250 East River Road, Rochester, NY 14623, 1981.
4. Rosemary C. Leary and Donald P. McClimans, Omega Alignment System Software Maintenance Manual, University of Rochester, Laboratory for Laser Energetics, 250 East River Road, Rochester, NY 14623, 1981.

FORTH CLASSES

November 16-20
December 7-11
January 11-15

Call: Inner Access
(415) 591-8295

A HIGH LEVEL INTERRUPT HANDLER IN FORTH

R. L. Keck and L. P. Forsley
Laboratory for Laser Energetics
Unversity of Rochester

Abstract

A system for writing interrupt service routines in high level FORTH is described. An example of the utility of high level interrupt service in a dynamic data acquisition situation is provided.

Introduction

X-ray data from laser-plasma interaction experiments on the GDL laser system at LLE has in the past been acquired from photographs of oscilloscope traces. Because of the large number of detectors currently being employed, this method has become impractical and we have chosen to use 12 channel integrating A/D converters for data acquisition. These A/D converters are CAMAC[1] compatible modules and because of the extensive CAMAC vocabulary available in the UR FORTH-79 system, as well as the suitability of FORTH for use in a dynamic programming environment, FORTH is used for the acquisition software.

The A/D modules integrate the signal at each of their 12 inputs for the duration of a gate signal, which is derived from the laser oscillator. The oscillator is fired once every 10 seconds to keep it in stable operation, however, our data signal occurs only when the full system of laser amplifiers is fired as well, an event which occurs when a fire sequence is carried out by the laser system controller on command from the operator. We require a means of clearing the A/D modules just in advance of the oscillator pulse at which the full system will fire. This is accomplished by feeding a ready-to-fire signal, provided by the laser system controller 4 seconds in advance of fire-time, to a CAMAC contact sense input module. Our acquisition sequence then is: look for a ready-to-fire signal from the contact sense input module, clear the A/D module, wait for data available indication from the A/D module and read the data from the A/D module.

The above sequence could be implemented directly, using the available CAMAC vocabulary, by simply continuously interrogating a module until the desired condition occurs and then proceeding to the next step. This method needlessly ties up the computer executing loops and prevents it from handling any other task while the sequence is in progress. Since both the contact sense input module and A/D module will generate CAMAC Look At Me's (LAM's) when a signal occurs at their inputs and a CAMAC LAM can generate an interrupt, we can use an interrupt driven acquisition system which will avoid needless looping. This requires the writing of interrupt service routines in machine code, which is at best cumbersome. It would be nice to be able to write high level FORTH interrupt service routines which could be readily changed. This can, in fact, be done and our method for doing this is discussed below.

Implementation

Our system consists of UR FORTH-79 running on a Digital Equipment Corporation LSI-11 microcomputer under DEC's RT-11 operating system. While a complete description of the implementation of this system may be found in the implementation guide[2], we will briefly cover FORTH's usage of processor registers for reference in the following discussion.

Four of the processor's general purpose registers are dedicated FORTH registers. R6, the system stack pointer, serves as FORTH's return stack pointer (RP). R5 is used as the stack pointer (S). R4 is used as the FORTH interpreter pointer (IC); it contains the address of the compilation address (also referred to as the code field address or CFA) of the next word to be executed. Finally, R1 is the state vector pointer (SV); more will be said about the SV later.

The procedure for executing a FORTH word from code is essentially quite simple and is accomplished by the word XEQ.MACRO (a listing is included in the appendix). It accepts an address, into which will later be placed the compilation address of the interrupt service word, on the stack and generates code which will place the compilation address of the service word on the stack [MOV @#<ADDR> ,-(S)], loads the IC with the address of the compilation address of the return from interrupt code [MOV #<HERE+8>,IC] (note that <HERE+8> contains the compilation address of RTI (COMPILE RTI), the return from interrupt code word) and then jump to the executable code for EXECUTE to begin execution of the interrupt service word [JMP ' EXECUTE]. The net effect of this code sequence is to start execution of a high level interrupt service word and subsequently execute the return from interrupt code.

Before execution of the code generated by XEQ.MACRO can begin, the contents of the processor registers must be preserved by pushing them onto the system stack. Code to do this is generated by REG.SAVE.MACRO. We must additionally ensure that the S and SV registers point to valid memory areas. In the multitasking UR FORTH-79 system, this is most easily accomplished by having a separate interrupt task area. The task area contains return and parameter stack memory allocations as well as a state vector allocation. The SV register points to the state vector and the state variables contained in the state vector are addressed relative to the value of the SV register.

It should be noted that it is not necessary to have a multi-tasking system in order to implement high level interrupt routines. This is because the values of the state variables referenced by the interrupt routine are in general identical to those for the master task. On a non multitasking system we would simply reserve a parameter stack area for the interrupt routines and set S to point to it. It is necessary, however, that FORTH be coded reentrantly for this scheme to work.

The SV.SET.MACRO is used to generate code which will set the SV and S registers. Note that it also changes the return stack location. This would not be necessary, except for the fact that the FORTH stack checking routines require that the return stack be located in memory immediately above the parameter stack. The value of the interrupted task's return stack pointer is stored in a free vector location [52T(SV)].

SETUP.INT sets the interrupt vector, in this case specifically for CAMAC (the vector for the device in slot N for the CAMAC crate is located at 400+N*4). The processor is run at priority 7 during interrupt service to prevent further interrupts from occurring.

To make it simple to create interrupt service routines, the macros previously discussed are combined to produce a defining word called

CREATE.CAMAC. INT.WORD .

This word when executed, accepts a task area and CAMAC slot number on the stack and creates a word which contains the code sequences previously developed starting at the second parameter field location of the newly created word and sets the interrupt vector to point to this code. The first parameter field location is reserved to hold the compilation address of the word to be executed when an interrupt occurs. The DOES> part of the new word will load this reserved location with the compilation address of the desired interrupt service word.

An Example

The listing for blocks 3 and 4 illustrate how the interrupt handler is used in our acquisition system. A task area (1TASK) is created and initialized for the interrupt routines to use. It must be delinked from the multi-tasking system to make it transparent to the multi-tasking dispatcher. Then two interrupt service routines are defined (RDY.WORD and FIRE.WORD) each with an associated CAMAC slot (or device). They share the same task area since only one interrupt service routine can be active at a time.

In block 4, the high level service routines are defined. RDY.INT is used to clear the A/D module, enable A/D LAM's (XCLR XENLAM) and then clear and disable further LAM's from the contact sense input module, on occurrence of a LAM from the contact sense module. FIRE[collects the A/D data, disables further A/D LAM's (XCOLLECT XDISLAM) and activates another task which will print the results (2TASK DISPATCH) on occurrence of a LAM from the A/D module. These high level routines are installed as the interrupt service routines for the appropriate CAMAC devices with the sequences: RDY.WORD RDY.INT and FIRE.WORD FIRE[. Changing an interrupt service routined with this system requires only defining a new high level handler word and installing it as the handler word, e.g., FIRE.WORD FIRE2[will make the word FIRE2[the new interrupt service routine for the A/D module.

Conclusions

We have shown that it is possible to write high level interrupt service routines in FORTH. This makes it possible for programmers unfamiliar with interrupt programming to easily write interrupt service routines. In addition, the facility with which this system permits changes to be made to the interrupt handlers makes this an ideal way to handle data acquisition in a rapidly changing experimental environment.

Acknowledgement

The authors would like to thank Michael McCourt for assistance with details on the internal operation of UR FORTH-79.

R.L. Keck is a graduate student in Mechanical Engineering at the University of Rochester. L.P. Forsley is Group Leader of Computer Systems at the Laboratory for Laser Energetics, University of Rochester.

1. Modular instrument and digital interface system (CAMAC, IEEE STD. 583-1975)
2. McCourt, Michael, "University of Rochester PDP-11 FORTH-79 Implementation Guide," Release Number 1.0, May 1981, unpublished.

APPENDIX
WORD LISTINGS

```
BLOCK     1 ************************************************

( High level FORTH interrupt handler   rlk lpf   25-may-81      )

: REG.RESTORE.MACRO               ( <>-<>, restore registers 0-5 *)
  ASSEMBLER 0 5 DO I RP )+ MOV, -1 +LOOP FORTH ;
CODE RTI               ( restore registers, return from interrupt *)
  RP 52T SV I) MOV, REG.RESTORE.MACRO RTI, FORTH
: XEQ.MACRO                ( <addr of xeq word, assembly time>-<> *)
  ASSEMBLER S -) SWAP @# MOV, ( push handler word addr on stack)
           IC HERE 8 + # MOV, ( preset the IC )
             ' EXECUTE P JMP, ( Jump to execute )
                 COMPILE RTI  ( pointer to next instruction )
  FORTH ;
: REG.SAVE.MACRO                   ( <>-<>, save registers 0 - 5 *)
  ASSEMBLER 6 0 DO RP -) I MOV, LOOP FORTH ;

                                                         -->
BLOCK     2 ************************************************

( more interrupt stuff                        25-may-81   rlk )
: SETUP.INT           ( <slot#><code addr>-<> set camac vector *)
  SWAP 4 * 4000 + DUP ROT SWAP !
  2+ 3400 SWAP ! ;

: SV.SET.MACRO     ( <SV loc>-<> set SV for interrupt routines *)
  ASSEMBLER SV SWAP # MOV, S 14T SV I) MOV, 52T SV I) RP MOV,
  RP 16T SV I) MOV, FORTH ;

: CREATE.CAMAC.INT.WORD      ( <SV loc><slot#>-<>, create int. *)
                                               ( defin. word. *)
  <BUILDS 0 , HERE SETUP.INT HERE 2- REG.SAVE.MACRO
  SWAP SV.SET.MACRO XEQ.MACRO
  DOES> [COMPILE] INSTALL SWAP ! ;

                                                         -->
 BLOCK     3 ************************************************

 ( Interrupt task area initialization                rlk 16SEP81)

 20 30 0 47 BLDTASK 1TASK                    ( create a task area *)
 1TASK TCLEAR                            ( initialize task area *)
 1TASK DUP !       SV DUP !           ( delink task from task list *)
 1TASK DISPATCH                           ( mark task as active *)

        ( create a ready to fire handler word for CAMAC slot 6 *)
 1TASK 6 CREATE.CAMAC.INT.WORD RDY.WORD
                    ( create a fire time word for the A/D module *)
 1TASK XAD CREATE.CAMAC.INT.WORD FIRE.WORD



                                                         ;S
 BLOCK     4 ************************************************

 ( xray interrupt service         13-apr-81   rlk )
 40 120 0 47 BLDTASK 2TASK       ( task area for post fire word *)

 : RDY.INT                            ( rdy fire int handler *)
   XCLR XENLAM 6 N 0 A 2 F DROP 24 F ;

 : FIRE!                               ( fire time handler *)
   XCOLLECT XDISLAM 2TASK DISPATCH ;


  RDY.WORD RDY.INT              ( make RDY.INT the ready to fire *)
                                    ( interrupt service routine *)

 FIRE.WORD FIRE!  ( make FIRE! the fire time interrupt handler *)

                                                         -->
```

OPTIMIZED DATA STRUCTURES FOR HARDWARE CONTROL

Joseph D. Sawicki
Laboratory for Laser Energetics
University of Rochester

Abstract

Data structures have been developed to more easily control hardware. A disk driver is used as an example for exploring alternative FORTH data structures and ways of optimizing them. These examples show that FORTH data structures are well suited to minimizing programming time and increasing software efficiency.

Introduction

While working at the Laboratory for Laser Energetics this summer one of my projects was to write a general purpose backup routine for a DEC-like[1] RX02 mode floppy disk drive. In doing this certain commonly used FORTH tools became useful. This paper will serve to illustrate these tools, and the modifications necessary due to the nature of the project.

Data Structures

The TO concept was developed by Dr. Paul Bartholdi and was described in FORTH DIMENSIONS Vol. I No. 4 and Vol. I No. 5 concept[2] in variables. This could be implemented in high level as follows:

```
0 VARIABLE %TO
: TO 1 %TO ! ;
: VAL <BUILDS ( <#>-<> , ACCEPTS INITIAL VALUE )
      ,
      DOES> ( <#>-<>;<>-<#>, STORES OR GIVES "VAL" )
      %TO @
      IF !
         0 %TO !
      ELSE @
      THEN ;
```

It would be used like a variable. Entering 0 VAL<NAME>would define a variable with an initial value of zero. To change the value to a six one would say 6 TO<NAME>; saying<NAME>would now put a six on the stack.

This technique makes the code more readable by eliminating the use of @ and [with variables (and ' with constants) to access and modify them. The backup driver is no exception to this and in fact offers the opportunity to carry the concept one step further. In the DEC PDP-11 architecture, I/O is memory mapped so that, for instance, the Disk Control Status Register is at location 177170O[3] and the Data Buffer Register is at location 177172O. One way to communicate with these addresses is to define two constants:

```
177170O CONSTANT CSR
177172O CONSTANT DBR
```

but then the use of @ and [becomes necessary. A way around this problem is to define a data structure similar to VAL except that it contains an address in its parameter field instead of a value. It would also be useful to fetch the address as well as to send data to and from the address. An easy, though by no means optimal, implementation of such a structure is given below.

```
: TO ( SETS FLAG SO  THAT A NUM WILL BE STORED IN A REG.)
     1 %TO ! ;
: FROM ( SETS FLAG SO THAT A NUM WILL BE GOTTEN FROM A REG)
    -1 %TO ! ;                                 ;5


( TEST BED FOR BEGINING OF RX02 DRIVER   JDS    15JUN81          )
: REGISTER <BUILDS ,
           ( <ADD>-<>, BUILDS A DATA TYPE CALLED A REGISTER )
           DOES> ( GIVES REGISTER ADD, CONTENTS OR SENDS DATA
                TO THE REGISTER DEPENDING ON THE STATUS OF %TO
           @ %TO @ ( GET ADDRESS OF REG AND %TO
          DUP -1 = IF SWAP @ SWAP (GET CONTENT)
                   THEN
               1 = IF ! ( STORE VALUE IN REG )
                   THEN 0 %TO
```

Once these two structures are implemented it becomes very easy to talk to the disk drive. For example, if a VAL had been defined called IN-TRACK# which contained the track to be read, sending it to the DBR would simply consist of saying IN-TRACK# TO DBR.

In the RX02 mode there are eight disk commands. They are all similar in that they need to have a drive and density bit set and they are sent to the CSR. The first problem is solved by a VAL called DRIVE/DENSITY and the four words shown below:

```
: SINGLE-DENSITY ( <COM.>-<COM.>, SETS THE DENSITY BIT TO 0 )
   DRIVE/DENSITY 256 BIC TO DRIVE/DENSITY ;
: DOUBLE-DENSITY ( <COM.>-<COM.> , SETS THE DENSITY BIT TO 1 )
   DRIVE/DENSITY 256 BIS TO DRIVE/DENSITY ;
: 0DRIVE ( <COM.>-<COM.>, SETS THE DRIVE BIT TO 0 )
   DRIVE/DENSITY  16 BIC TO DRIVE/DENSITY ;
: 1DRIVE ( <COM.>-<COM.> , SET THE DRIVE BIT TO 1 )
   DRIVE/DENSITY  16 BIS TO DRIVE/DENSITY ;
```

After setting the drive and density as desired, the VAL DRIVE/DENSITY can then be ORed with the command to produce the desired results. There are two approaches that can be taken at this point. For example, take the command to format a disk in a single or double density; call it (SET-DEN). A word could be defined, along with seven others like it, as shown:

```
: (SET-DEN) 110 DRIVE/DENSITY OR TO CSR ;
```

The second approach would be to again use a defining word:

```
: DISK-COMMAND <BUILDS ( <CON>-<> TAKES THE CON FOR A DISK OP. )
                     ,
                     DOES> ( GET COM AND DRIVE DEN INFO OR, AND SEND )
                       @ DRIVE/DENSITY OR TO CSR ;

110 DISK-COMMAND (SET-DEN) ( USED TO FORMAT DISKS SING OR D DEN)
```

Optimization

As usual we have a classic FORTH space-time tradeoff. The second approach executes somewhat slower (see figure 1) because the constant needs to be fetched, but whereas the first approach takes 18 bytes per command or a total of 144 bytes, the second approach takes only 10 bytes per command plus 24 bytes for the defining word for a total of 104 bytes. Because of the space savings the philosophy that very similar things should be grouped together could override the execution speed losses and the second approach was used.

All of this would have been fine except that when doing the track to track backup a sector interleaving technique must be used to keep backup times down to a reasonable level. Since these VAL's and REG's have high level IF statements in them and they are used each time a sector is read or written, they require an overly large interleave step size. The solution to this problem is to use ;CODE instead of DOES>. Though this makes the word less transportable it isn't seen as a problem since this is a PDP-11 specific disk backup. The VAL word now can be defined as follows:

```
: VAL <BUILDS ( <#>-<>, TAKES THE INITIAL VALUE OFF THE STACK )
      ,
      ;CODE ( <#>-<> OR <>-<#>, GETS VALUE OR STORES VALUE )
          %TO P TST, ( SEE IF %TO POSITIVE )
           GT IF,
              WPARAM W I)  S )+ MOV, ( STORE VALUE )
              %TO P 0 # MOV, ( ZERO OUT %TO FLAG )
           ELSE,
               S -) WPARAM W I) MOV, ( FETCH VALUE OF VAL )
           THEN, NEXT,                          -->
```

where W is the PDP-11 register containing the CFA (code field address) of the word executing, WPARAM is a constant equal to the offset from the CFA to the PFA, and I) indicates indexed addressing. Not only is the coded VAL faster than the high level version, but it is also faster than a VAR at fetching and the same speed at storing (see figure 2). It was also necessary to code REG as shown below:

```
: REG <BUILDS ( BUILDS A DATA TYPE CALLED A REGISTER )
      ,
      ;CODE ( <#>-<>,<>-<#>, GETS ADD, VALUE OR STORES VAL )
      %TO P TST, ( CHECK IF %TO IS POS NEG OR ZERO )
      GT IF,
           WPARAM W @I) S )+ MOV, ( STORE VALUE IN REG )
         ELSE,
              LT IF,
                    S -) WPARAM W @I) MOV, ( GET VALUE )
                ELSE,
                       S -) WPARAM W I) MOV, ( PUT T.O.S. )
                THEN,
         THEN,
         %TO P 0 # MOV, NEXT,
                                             -->
OK TTv
```

To illustrate the use of these concepts the FORMAT-DISK word will be shown. But first to insure that the program doesn't try to do things before the disk controller is ready, two more words are needed that wait for the done and transfer request bit to be asserted in the CSR.

```
: TR.WAIT ( WAITS FOR THE DATA TRANSFER BIT TO BE SET )
   BEGIN 2000 FROM CSR AND END ;
: DONE.WAIT ( WAITS FOR THE DONE BIT TO BE ASSERTED )
   BEGIN  400 FROM CSR AND END ;
```

The disk command as shown before was called (SET-DEN). After receiving this command the disk controller waits for a "key" byte (111O, the letter I in ASCII) to be sent to the DBR, therefore the entire command is coded as shown:

```
: FORMAT-DISK ( <>-<>, SETS THE DENSITY OF A DISK )
      (SET-DEN) TR.WAIT
      1110 TO DBR ( SEND 'KEY' BYTE )
      DONE.WAIT ;
```

To format the disk in the drive one double density one would enter 1DRIVE DOUBLE-DENSITY FORMAT-DISK; to format the disk in drive zero single density one would enter 0DRIVE SINGLE-DENSITY FORMAT-DISK.

Timing

To show the effects of the different approaches timing tests were run. The first contrasts the difference between the two types of disk commands. In all tests the action was placed inside a double loop like:

```
: TEST 10 0 DO 30000 0 DO LOOP LOOP ;
```

This routine took 23 seconds which was then subtracted from the other results to give the time to do the operation 300,000 times. This was then divided by 300,000 to give the time per operation. These are the results on a DEC LSI 11/2:

	To Send Disk Command
Colon definition	.23 msec.
Defining word	.28 msec.

Then a high level VAL was compared to a coded VAL and a VAR:

	fetching (msec)	storing (msec)
high level VAL	.237	.39
coded VAL	.067	.11
VAR	.083	.093

Summary

This paper not only showed the usefulness of certain techniques in FORTH but also illustrates some general properties of the language. The first of these is the ease of implementation of new data structures. Through the use of BUILDS ... DOES or BUILDS ... ;CODE one can first build the structure to suit the needs of the application and then imbed in the executable code necessary operations for the structure. Also a structure can easily be given variable execution as in the case of VAL and REG. Another important benefit of FORTH is the ease of optimization of the word by the use of assembly code. Changing the VAL and REG words to ;CODE took less than a half hour.

Acknowledgements

I would like to thank Mike McCourt, Bob Keck, Lawrence Forsley and Peter Paulson for their help in getting the hardware running and for comments on the software.

This work was partially supported by the following sponsors: Exxon Research and Engineering Company, General Electric Company, New York State Energy Research and Development Authority, Northeast Utilities, The Standard Oil Company (Ohio), the University of Rochester, Empire State Electric Energy Research Corporation, and the U.S. Department of Energy inertial fusion program under contract number DE-AC08-80DP40124.

J. Sawicki is an undergraduate with the Electrical Engineering Department of the College of Engineering at the University of Rochester. He is a DJ in his spare time.

---

1. DEC and PDP-11 are trademarks
2. The TO concept by Paul Bartholdi FORTH DIMENSIONS Vol. I No. 4 and Vol. I No. 5.
3. Where an O suffix indicates octal

THE STRING STACK

Michael McCourt
Laboratory for Laser Energetics
University of Rochester

Richard A. Marisa
Production Automation Project
University of Rochester

Abstract

Applications which require a text data type are supported by a group of functions which operate with string variables and a string stack. The string stack is analogous to the parameter stack, however, the data type with which it operates is the string, containing length and character data.

String Defining Words

Two defining words are available for the creation of string data entities. The first is:

```
<maxlen> STRING-VAR <NAME>
```

which creates a varying length character string with maximum length **<maxlen>**. Invoking<NAME>places

```
<beginning address><maximum string length>
```

on the parameter stack. The first byte at<beginning address>is the current string length; the string text begins at the next byte.

The second string defining word is:

```
<number of elements> <maxlen> ()STRING <NAME>
```

which creates an array of variable length strings. Invoking

```
<i><NAME>
```

places `<address of the i-th string> <maxlen>`

on the parameter stack. Note that (number of elements) x (maxlen) bytes will be allocated to hold the string array.

String Stack Manipulation

A string stack, separate from the parameter stack, is maintained in memory for the purpose of manipulating string data. Several words which manipulate the string stack are defined in the string stack library which can be compiled by executing >STRINGS (which loads in the string stack package). Currently 200 (decimal) bytes are allocated for the string stack.

The quote word (") is available for placing a string on the string stack. To stack a string, type:

```
" <text>"
```

" is followed by exactly one space, then <text> delimited by a quotation mark.

A string print word .SS is used to print the top element of the string stack,

```
******************  BLOCK     96      *******************

( STRING STACK--FIXED LENGTH STRING COMPARISON  LAR 19-SEP-79 )
: SS ;    ( NOTE: PARAM ORDER NOW <ADR><LEN> MAM  11-JUN-80 )
   ( [ADD A, ADD B, LEN],---[ADDA, ADDB, = OR + OR - ] )
   ( COMPARES CHARS. IN STRINGS A & B PARIWISE; RETURNS 0 IF
   ( STRINGS ARE =, + IF A>B, - IF A<B )

: S?FDO 0 SWAP 0 DO DROP OVER C@ OVER C@ - ROT 1+
        ROT 1+ ROT DUP 0= NOT IF LEAVE THEN LOOP ;
   ( [ADD A, ADD B, LEN]---[= OR + OR -], SAME AS S?FDO )
   ( EXCEPT ADDRESSES NOT RETURNED )

: S?F S?FDO ROT ROT 2DROP ;
   ( [ADD A, LEN]---[= OR + OR - ], COMPARES STRING A TO )
   ( A STRING OF BLANKS--RETURNS 0 IF TWO ARE EQUAL )
: S?B 0 SWAP 0 DO DROP DUP C@ BL - SWAP 1+ SWAP DUP 0<>
       IF LEAVE THEN LOOP SWAP DROP ;              -->

******************  BLOCK     97      *******************

( STRING COMPARISON---VARYING LENGTHS )
   ( [ADD A, ADD B, LEN DIFF]---[= OR + OR -] FIRST TESTS )
   ( TO SEE IF LENGTH DIFF. BTWN. A & B IS 0 ; IF NOT, TESTS )
   ( THE LONGER STRING TO SEE IF THE EXTRA CHARS. ARE BLANKS )
   ( IN BOTH CASES 0 IS RETURNED, OTHERWISE + OR - )
: S?BLTEST DUP 0= IF DROP 2DROP 0 ELSE DUP 0<
    IF MINUS ROT DROP S?B MINUS ELSE SWAP DORP S?B THEN THEN ;
   ( [ADD A, ADD B]---[0 IF A=B, - IF A<B, + IF A>B ] TESTS )
   ( WHETHER 2 VARIABLE LENGTH STRIGNS HAVE BOTH THE SAME # )
   ( OF CHARS. AND THE SAME ORDER & TYPE )
: S? OVER C@ OVER C@ 2DUP - >R MIN ROT 1+ ROT 1+ ROT S?FDO
     DUP 0<> IF ROT ROT 2DROP R> DROP
               ELSE DROP R> S?BLTEST THEN ;


                                                 -->

******************  BLOCK     98      *******************

( STRING STACK WORDS                       LAR 19-SEP-79 )
0 SVAR SSO    0 SVAR SSM    0 SVAR SST
: SSTOP SST @ ;        : SSTOP! SST ! ;
: SSORG SSO @ ;        : SSMAX  SSM @ ;
   ( [FROM, TO, LEN ]---['] CHECKS FOR STACK BOUNDARIES )
: SOVCHECK OVER SSORG U<
  IF SSMAX SSTOP ! 14T TABORT THEN ;
   ( [ADD]---[] INSURES THAT ADDRESS POINTS TO STRING )
: SSVER DUP DUP C@ + SSMAX U>=
  IF SSMAX SSTOP ! 13T TABORT THEN ;
   ( ADD OF TOP STRING]---[AD OF NEXT STRING DOWN] )
: SSDOWN DUP C@ 1+ + ;
   ( [ADD]---[] PUSHES STRING AT ADDR. TO TOS )
: SSPUSH DUP C@ 1+ SSTOP OVER - DUP SSTOP! SWAP RMOVE ;
                                                 -->

******************  BLOCK     99      *******************

( STRING STACK WORDS                LAR  19-SEP-79 )
: 'DROP           ( []---[] REMOVES TOP STRING FROM STACK *)
  SSTOP SSVER SSDOWN SSTOP! ;
: 'LEN            ( []---[] RETURN LEN OF TOS STRING *)
  SSTOP SSVER C@ ;
: 'LOC            ( []---[] RETURN ADDR OF TOS STRING *)
  SSTOP 1+ ;
: 'DUP            ( []---[] COPY TOS STRING *)
  SSTOP SSVER SSPUSH ;
: 'SWAP           ( []---[] EXCHANGE TOP 2 STRINGS *)
  SSTOP DUP SSDOWN DUP SSPUSH SSDOWN SSTOP SWAP SSTOP!
  SWAP SSPUSH SSPUSH ;
: 'ROT            ( []---[] ROTATE TOP THREE STRINGS ABC->BCA *)
  SSTOP DUP SSDOWN DUP SSDOWN DUP SSPUSH SSDOWN SSTOP SWAP
  SSTOP! SWAP SSPUSH SWAP SSPUSH SSPUSH ;
                                                 -->

******************  BLOCK     100     *******************

( STRING STACK WORDS                       MAM  13-JUN-80 )

: 'OVER        ( []---[] PUSH 2ND STRING DOWN ONTO TOS *)
  SSTOP SSDOWN SSVER SSPUSH ;
: '2DUP        ( []---[] COPY TOP 2 STRINGS *)
  'OVER 'OVER ;
: '2DROP       ( []---[] DROP TOP 2 STRINGS *)
  'DROP 'DROP ;
: '2OVER       ( []---[] PUSH 3RD AND 4TH TO TOS *)
  SSTOP SSDOWN SSDOWN DUP SSDOWN SSVER SSPUSH SSPUSH ;
: '2SWAP       ( []---[] EXCHANGE 1ST & 2ND WITH 3RD AND 4TH *)
  DUP SSDOWN SSDOWN SSDOWN SSTOP! SSPUSH SSPUSH SSPUSH SSPUSH ;
: '@           ( <ADDR><LEN>---[]  PUSH STRING AT ADDR TO SS *)
  DROP SSPUSH  ;

                                                 -->
```

removing the top element in the process. For example,

```
OK " STACK THIS STRING "  <CR>

OK

.SS  <CR>

STACK THIS STRING OK
```

Notice that the functions .SS and . are similar. Several other functions operate on the string stack in a manner analogous to words which operate on the parameter stack. These are:

```
*******************  BLOCK    101      *******************

( STRING STACK WORDS CONT'D              MAM 13-JUN-80 )

: "!      ( [ADR][LEN]---[] STORE TOS AT ADDR. & DROP TOS *)
  SSTOP "DROP SWAP OVER C@ MIN 2DUP SWAP C! 1+
  ROT SWAP RMOVE ;
   ( [STRING]---[] STORES STRING IN PAD THEN MOVES IT FROM )
   ( THERE TO THE TOSS -- WORKS DURING EXECUTION TIME )
: X" 420 WORD 0 "@ ;
: $" R> DUP 0 "@ DUP C@ DUP 2 MOD
    IF 1+ ELSE 2+ THEN + >R ;
   ( [STRING]---[] STORES STRING AT TOP OF DICT. STACK )
   ( DURING COMPILATION )
: C" COMPILE $" 420 WORD C@ DUP 2 MOD
   IF 1+ ELSE 2+ THEN ALLOT ;
: " STATE @ IF C" ELSE X" THEN ; IMP "
                                                       -->
```

WORD	FUNCTION	BEFORE	AFTER
"DUP	copies top of stack	B A	B A A
"SWAP	reverses top two strings on the stack	B A	A B
"DROP	removes top of stack	B A	B
"OVER	copes 2nd string onto top	B A	B A B
"ROT	moves 3rd string to top	C B A	B A C
"2DUP	copies top 2 strings	B A	B A B A
"2DROP	removes top 2 strings	C B A	C
"2SWAP	reverses 1 & 2 with 3 & 4	D C B A	B A D C
"2OVER	copies 3 & 4 to top	D C B A	D C B A D C
"+	string addition (catenation)	B A	BA

String Relationals

Just as the parameter stack relational operators remove their arguments from the parameter stack, the following string stack relational operators remove their arguments from the string stack. The logical result of the string relation is placed on the parameter stack. The available relationals are:

"=	"<=
"<>	">=
"<	">

String Variable Storage and Retrieval

The string store word, "[, places the top of the string stack in the string variable described by the parameter stack, popping the string stack. The string retrieve word, "@, places the string referred to by the parameter stack onto the string stack.

```
*******************  BLOCK    102      *******************

( STRING STACK WORDS CONT'D      MAM 18-MAR-81 )
: .SS     ( []---[] TYPE OUT STRING AT TOSS *)
  SSTOP SSVER "DROP COUNT TYPE  ;
: ",  ( <>-<>, PUT STRING IN DICTIONARY, MAKE EVEN LENGTH )
  420 WORD COUNT DUP HERE SWAP 1+ -2 AND ALLOT SWAP CMOVE ;

 ( SOME  FIXED LENGTH STRING DEFINITIONS )
   ( [ADDR,MAX LEN]---[] PUSH STRING AT ADDR TO TOSS )
: "@F DUP SSTOP OVER - 1- SSTOP! SSTOP C! SSTOP
      1+ SWAP CMOVE ;
   ( [ADDR,MAX LEN]---[] COPY CHARS ONLY FROM TOSS TO ADDR )
: "!F 2DUP BLANK "LEN MIN SSTOP 1+ ROT ROT CMOVE "DROP ;

                                                       -->
```

```
*******************  BLOCK    103      *******************

( STRING STACK WORDS CONT'D                    LAR 19-SEP-79 )
: "+    ( []---[] ADD TOP 2 STRINGS ON STACK LEFT TO RIGHT *)
  "SWAP SSTOP SSDOWN SSVER C@ SSTOP C@ DUP ROT +
  SSTOP C! SSTOP DUP 1+ ROT 1+ RMOVE SSTOP 1+ SSTOP! ;
   ( [LEN, BEGINNING CHAR #]---[] REPLACE TOSS WITH )
   ( SUBSTRING OF LENGTH [LEN]. STARTOMG WITH SPECIFIED )
   ( CHAR OF ORIGINAL STRING )
: "SUBSTR 1- SSTOP SSVER "DROP + DUP ROT ROT C! SSPUSH ;
   ( [ADD OF 2ND STR+1, 1ST CHAR OF 1ST STR, LEN OF 2ND, 0 ] )
   ( ---[OFFSET OR 0 ]   SEARCHES 2ND STR. FOR 1ST CHAR OF )
   ( 1ST STR.; IF FOUND, COMPARES 2ND STR. FROM THAT POINT  )
   ( TO 1ST STR )
: "INDEXDO DO OVER I + C@ OVER =
    IF OVER I + SSTOP 1+ "LEN S?F 0=
       IF DROP I 1+ ROT ROT LEAVE THEN THEN LOOP ;
                                                       -->
```

```
OK 30 STRING-VAR MYSTRING  <CR>

OK

" string text " MYSTRING "!  <CR>

OK

 MYSTRING "@ MYSTRING "@ "+ .SS   <CR> string text string text

OK
```

Invoking the name of the string variable MYSTRING in the preceding example placed <address> <maxlen> on the parameter stack. String store and string retrieve check the maximum and current length of the string variable when moving string data.

When it is required to move fields of fixed length which do not contain an embedded current length in the first byte, fixed length string store and retrieved words may be used. The syntax is:

```
<address> <length> "!F

<address> <length> "@F
```

String Functions

"LEN returns on the parameter stack, the length of the string on top of the string stack. The string remains on the string stack. The address of the first byte of the string (one byte after the length field) is found by executing "LOC.

```
<length> <beginning character number>
            "SUBSTR
```

replaces the top of the string stack with a substring of length <length>, beginning with the specified character of the original string. For example,

```
OK
" abcde" 2 3 "substr .SS
cd OK
```

The "INDEX function searches for the first occurrence of the string in the second string. If an occurrence is found, its offset is returned on the parameter stack. If an occurrence is not found, -1 is returned. The top of the string stack is popped.

String Stack Errors

Two errors are reported by the string stack package: string stack underflow and overflow. As stated previously 200 bytes are initially allocated for the string stack. If repeated overflows are generated more space can be allocated for the string stack by changing the parameter passed to "INIT in the string stack library. String stack initialization is the last function performed when the string stack library is loaded.

Summary

This was the first major software package transported throughout the University URTH community. Originally, it had a few code routines which were machine specific to reduce execution time. However, these were removed on all the systems but the Intel 8080. The package has run, without change (except for the above mentioned machine-specific code) on Hewlett Packard 2100, DEC PDP-11, IBM 360 and the INTEL 8080.

```
******************  BLOCK    104      *******************

( STRING STACK WORDS CONT'D                      LAR  19-SEP-79 )
   ( []---[-1 OR OFFSET]  SEARCHES FOR 1ST OCCURENCE OF )
   ( TOP STR. IN 2ND STR.---IF FOUND OFFSET IS RETURNED )
   ( ON PARAM STACK ELSE -1 IS RETURNED. TOSS IS POPPED . )
: "INDEX -1 SSTOP DUP C@ 0<>
   IF DUP SSDOWN SSVER DUP C@ ROT 1+ C@ ROT 1+ SWAP
   ROT 0 "INDEXDO
   ELSE 0 ROT ROT THEN 2DROP "DROP ;
   ( []---[] COMPARE & DROP TOP 2 STRINGS, LEAVE 0,<0 OR >0 )
: "? SSTOP DUP SSDOWN S? "DROP "DROP ;
   ( []---[T/F] LOGICAL =, TESTS TOP 2 STRINGS )
: "= "? 0= ;
   ( []---[T/F] LOGICAL LESS THAN TESTS TOP 2 STRINGS )
: "< "? 0 > ;

                                                    -->

******************  BLOCK    105      *******************

( STRING STACK WORDS CONT'D        MAM   18-MAR-81 )
: ">      ( []---[T/F]  TESTS TOP 2 STRINGS FOR > *)
  "? 0<  ;
: "<=     ( []---[T/F]  TESTS TOP 2 STRINGS FOR <= *)
  "> NOT ;
: ">=     ( []---[T/F]  TESTS TOP 2 STRINGS FOR >= *)
  "< NOT ;
: "SPACES  ( <N>-<>, PUSH A STRING OF N SPACES ON SS *)
  DUP 0 DO SSTOP 1- BL SOVCHECK OVER C! SSTOP! LOOP
  SSTOP 1- DUP ROT SWAP C! SSTOP! ;

: "INIT ( <#CHARS TO ALLOCATE FOR SS>-<>, INIT SS INTO DICT )
  1 ?#  HERE SSO ! ALLOT HERE 2- DUP SSM ! SST ! ;

200T "INIT   ( ALLOCATE 200 CHARS FOR STRING STACK )
                                                    -->

******************  BLOCK    106      *******************

( STRING VARIABLE AND STRING ARRAY          MAM 13-JUN-80 )
   ( [MAX LEN]---[]  ALLOTS SPACE IN DICT FOR MAX LEN AND )
   ( MAX # OF CHARS. )
: STRING-SPACE DUP , 0 , 2/ DP+!  ;
   ( [MAX LEN] STRING <NAME>  --- BUILDS A STRING VARIABLE )
   ( WHEN <NAME> IS EXECUTED THE BYTE ADDR. OF THE STRING )
   ( START AND LENGTH ARE LEFT ON THE STACK )
: STRING-VAR <BUILDS STRING-SPACE
  ;CODE     S -)    W     MOV,  ( PUSH PARAM ADDR )
            S)       4 #  ADD,  ( POINT TO COUNT AND FIRST CHAR )
            S -) 2 W I) MOV,  ( PUSH MAX LENGTH )
                          NEXT,


                                                    -->

******************  BLOCK    107      *******************

( STRING ARRAY ROUTINE                     MAM 13-JUN-80 )
: ()STRING     ( [# OF ELEMENTS, MAX LEN] ---<NAME>    *)
  <BUILDS SWAP DUP ,        (  BUILD HEADER, STORE # OF STRINGS )
  0 DO DUP STRING-SPACE    ( ALLOT DIC SPACE, STORE MAX LEN )
    LOOP DROP

DOES> 2+ DUP @ ROT ROT 3 PICK             ( ADDR OF 1ST ELEMENT )
    DUP 2 MOD IF 3 + ELSE 4 + THEN  ( 1+ TO MAX LEN IF ODD )
                                  ( 2+ IF EVEN, 2+ FOR MAXLEN )
    ROT * 2+ + SWAP ;   ( STRING ADDR + ELEMENT OFFSET )
                        ( RETURNS COUNT AND ADDR )


                                                    -->

******************  BLOCK    108      *******************

( STRING EXECUTION ROUTINE             LPF,MAM   18-MAR-81 )

: "EXEC    ( <WORD NAME ON TOSS>-<>, EXECUTE WORD IF FOUND )
   HERE "LEN "!
   FIND ?DUP IF EXECUTE
             ELSE 0 TABORT THEN ;  ( UNDEFINED WORD ERROR )

: "FORGET ( <WORD NAME ON TOSS>-<>, FORGET WORD IF FOUND )
   HERE "LEN "!
   FIND ?DUP IF WPARAM + $FORGET
             ELSE 0 TABORT THEN ;   ( UNDEFINED WORD ERROR )

                                                   ;S
```

The first application was for a screen-oriented data entry system. Later applications included an ISAM data base, a menu-driven interface for flow cytometry and a word processing system. The package consists almost entirely of its original code written in 1977 by Mike Williams, of the University Computing Center. The major change has been the addition of comments.

Acknowledgements

We would like to thank the following people for their assistance: Mike Williams, of the University Computing Center, who developed the original String Stack Package for URTH on the IBM 360 and the Intel 8080; and two undergraduates who worked for Lawrence Forsley, Lynn Raymond and Dan Blumenthal, for documenting this package.

This work was partially supported by the following sponsors: Exxon Research and Engineering Company, General Electric Company, New York State Energy Research and Development Authority, Northeast Utilities, The Standard Oil Company (Ohio), the University of Rochester, Empire State Electric Energy Research Corporation, and the U.S. Department of Energy inertial fusion program under contract number DE-AC08-80DP40124.

R. Marisa is the manager of the computing facility of the Production Automation Project in the College of Engineering at the University of Rochester. M. McCourt was a senior laboratory engineer with the Laboratory for Laser Energetics at the University of Rochester and is now an applications engineer for Harvey Electronics.

HELP WANTED

Associate Systems Manager, Pulmonary Computer Systems

Primary responsibility for designing, debugging and implementing major software projects on the Pulmonary Computer System. Programming experience with PDP-11 Assembly language and FORTH desirable. Some hardware experience will be useful.

Salary range to $35,000. Superior benefits package, three weeks vacation first year.

Contact:

John Gilbert, Employment Officer
Cedars-Sinai Medical Center
8723 Alden Drive
P.O. Box 48750
Los Angeles, CA 90048
(213) 855-5529

NEW PRODUCTS

FORTH Application Modules Diskette

The diskette of FORTH application moduels, a new product by Timin Engineering, is a variety package of FORTH source code. It contains hundreds of FORTH definitions not previously published. Included on the diskette are data structures, software development aids, string manipulators, an expanded 32-bit vocabulary, a screen calculator, a typing practice program, and a menu generation/selection program. In addition, the diskette provides examples of recursion, <BUILDS...DOES> usage, output number formatting, assembler definitons, and conversational programs. One hundred screens of software and one hundred screens of instructional documentation are supplied on the diskette. Every screen is in exemplary FORTH programming style.

The FORTH screens, written by Scott Pickett, may be used with Timin FORTH or other fig-FORTH. The price for the diskette of FORTH application modules is $75 (if other than 8" standard disk, add $15). To order the FORTH modules, write Timin Engineering Company, 9575 Genesee Ave., Suite E-2, San Diego, CA 92121, or call (714) 455-9008.

INNER ACCESS FORTH SOFTWARE AND DOCUMENTATION

Fig-FORTH compiler/interpreter for PDP-11 for RT-11, RSX11M or stand-alone with source code in native assembler. Included in this package are an assembler and editor written in FORTH and installation documentation.

This is available on a one 8" single density diskette only. #20011-01 ($80)

Reference Manual for PDP-11 fig-FORTH above. #20011-99 ($20)

Fig-FORTH compiler/interpreter for CP/M or CROMEMCO CDOS system comes complete with source code written in native assembler. Included in this package are an assembler and editor written in FORTH and installation documentation.

All diskettes are single density, witr 5.25" diskettes in 128 byte, 18 sector/track format and 8" diskettes in 128 byte, 26 sector/track (IBM) format.

Released on two 5.25" diskettes with source in 8080 assembler #20080-85 ($80).

Released on one 8" diskette with source in 8080 assembler #20080-88 ($80).

Released on two 5.25" diskettes with source in Z80 assembler #20080-Z5 ($80).

Released on one 8" diskette with source in Z80 assembler #@0080-Z8 ($80).

Manual for CP/M (or CROMEMCO) fig-FORTH above #20080-99 ($20).

METAFORTH Cross-Compiler for CP/M or CROMEMCO CDOS to produce fig-FORTH on a target machine. The target can include an application without dictionary heads and link words. It is available on single density diskettes with 128 byte 26 sector/track format. Target compiles may be readily produced for any of the following machines.

CROMEMCO--all models
TRS80 Model II under CP/M
Northstar Horizon
Prolog Z80

Released on two 5.25" diskettes #20100-85 ($1,000).

Released on one 8" diskette #20100-88 ($1,000).

Complete Zilog (AMD) Z8002 develoment system that can be run under CP/M or CROMEMCO CDOS. System includes a METAFORTH Cross-Compiler which produces a Z8002 fig-FORTH compiler/interpreter for the Zilog Z8000 Development Module. Package includes a Z8002 assembler, a Tektronix download program and a number of utilities.

Released on two 5.25" diskettes #29102-85 ($4,000).

Released on one 8" diskette #29102-88 ($4,000).

Zilog Z8002 Develoment Module fig-FORTH ROM set. Contains fig-FORTH with Z8002 assembler and editor in 4 (2716) PROMS. #38002-00 ($850).

For orders and further information, contact:

INNER ACCESS CORPORATION
Software Division
Box 888
Belmont, CA 94002
(415) 591-8295

ANNOUNCEMENTS

Sym-FORTH Newsletter now available, contact: Saturn Software Ltd., PO Box 397, New Westminister, British Columbia, V3L 4Y7, CANADA.

COMPLEX ANALYSIS IN FORTH

Alfred Clark, Jr.
Department of
Mechanical Engineering
University of Rochester

During my years as an engineering educator and a researcher in theoretical fluid mechanics, I have often wished for the perfect calculator--a compact machine which would perform intricate and useful mathematical tasks in response to a few keystrokes. The pocket scientific calculators, amazing as they are, never seemed to have quite the power and flexibility (and certainly not the graphics ability) that I hoped for. I always supposed that my hopes were unreasonable until I discovered FORTH two years ago. Having been a FORTRAN programmer for 20 years, I found the transition to FORTH somewhat difficult and even painful at times. Originally, I took up FORTH out of curiosity, but gradually I realized that the quest for the perfect calculator was over--it is FORTH plus a microcomputer.

Perhaps I should say a little more about what a perfect calculator is supposed to do. Among other features, it should have (1) standard trigonometric and exponential functions, (2) other common special functions (e.g., Bessel functions), (3) graphics and automated plotting of functions, (4) numerical integration, (5) a root-finder, (6) special purpose applications, such as a direction field plotter for first order differential equations, and (7) complex arithmetic, including complex transcendental functions. Further, all procedures should be executable with a few keystrokes.

The last item in the list--complex--is in some ways the most stringent test of any would-be perfect calculator. It's certainly not available on any pocket calculator. Although it can be implemented in BASIC, it is cumbersome and requires a large package of subroutines. The versions of FORTRAN available for small machines generally omit the complex arithmetic and complex functions which are available on large machines. With FORTH, however, the extension to complex from real floating point is simple to implement, easy to use, and powerful. Since complex arithmetic is not yet very common in FORTH on small machines, I thought it would be worthwhile to sketch briefly my implementation.

The most fundamental question in introducing complex analysis is how to represent complex numbers. Here it turns out that the pure mathematician's definition of a complex number as an ordered pair of real numbers is exactly what we need. Thus the complex number 3.5 + 7.2i is regarded as an ordered pair, and is pushed on the stack by typing 3.5 7.2 . With this convention established, it is easy to define all of the important stack manipulations such as ZDROP, ZDUP, ZOVER, ZROT, and ZSWAP, which perform exactly like their integer and floating point counterparts. The basic load and store operators, Z@ and Z[, can be defined in terms of @ and [.

There are many single number operations which are useful. These include the real part REZ, the imaginary part IMZ, the complex conjugate CONJ, the modulus /Z/, the square of the modulus /Z/2, the reciprocal 1/Z, and the phase ARGZ (radians). Most of these are quite simple to define. IMZ, for example, is just
: IMZ FSWAP FDROP ; where FSWAP and FDROP are floating point stack operations. As another example, consider 1/Z defined by

```
: 1/Z  ZDUP /Z/2 FROT FOVER F/
       FROT FROT F/ CONJ ;
```

For ARGZ it is very important to establish a precise range and to implement it carefully. The conventional range, which I have used, is -PI < ARGZ <= PI. Any carelessness in the definition of ARGZ will lend to disasters later when multi-valued functions are introduced. Many engineering applications require the phase in degrees, and it is convenient to build in a function DARGZ which supplies this.

Conversion words between rectangular and polar forms are also very useful. To go from retangular to polar, with the phase (in radians) on top of the stack and the modulus just below, we have

```
: POLAR  ZDUP /Z/ FROT FROT ARGZ ;
```

A similar word, DPOLAR, leaves the argument in degrees. For conversion from polar to rectangular, we have RECT (angle in radians)

```
: RECT  FOVER FOVER COS F* FROT FROT SIN F* ;
```

and a word DRECT for the angle in degrees. A very useful application of these is a rotation operator ROTZ, defined so that the sequence Z F ROTZ rotates Z by F radians and leaves the result on the stack. The definition is

```
: ROTZ  FROT FROT POLAR FROT F+ RECT ;  .
```

There are several different useful formats for complex output. (My system has 8 different formats, which is handy but a little extreme.) The word Z. prints the number as an ordered pair -- 3.5 7.2 , for example. The conventional mathematical notations is obtained by ZI. -- (3.5) + (7.2)I. Words to print in polar form are also useful. For example, ZP. is defined so that the sequence 3.5 7.2 ZP. gives

```
MOD = 8.00562303  ARG = 1.11832144 (RAD) .
```

All of these output words are defined in terms of the basic floating point print word F. . For example, Z. is defined by

```
: Z.  FSWAP F. 2 SPACES F. ;
```

The binary complex operations are Z+, Z-, Z*, and Z/. These are quite easy to define. For example, Z+ is defined by

```
: Z+  FROT F+ FROT FROT F+ FSWAP ;
```

where FROT is a floating point ROT, and F+ is a floating point add.

Higher functions can be defined, provided the underlying real floating point has the standard real functions SIN, COS, ATN, and EXP. The complex exponential, for example, is then defined by

```
: ZEXP  FSWAP EXP FDUP FROT FDUP COS FROT F*
        FROT F* FROT FROT SIN F*
```

Other useful functions such as ZSIN, ZCOS, ZTAN, ZSINH, ZCOSH, and ZTANH are defined similarly.

Of the multi-valued functions, the most useful are the square root ZSQR, the logarithm ZLOG, and the power Z**. As an example of the definitions, consider the principal value of the square root:

```
: ZSQR  POLAR 2. F/ FSWAP SQR FSWAP RECT ;
```

The basic words described above can be the building blocks for substantial applications. One such application, which is particularly useful pedagogically, is conformal mapping. I have defined a word MAP such that the sequence

```
MAP  <curve>  <function>
```

will take any previously defined curve in the Z-plane and any previously defined complex function, and produce a graph showing the curve and its image under the transformation. This tool allows students (and the instructor!) to improve their understanding of the geometry of complex functions.

Notes on Implementation

The code described above runs on the author's 48K Apple II. The underlying integer FORTH is the excellent version written by William Graves and distributed by SOFTAPE. The real floating point arithmetic and functions have been implemented by interfacing the SOFTAPE FORTH with the Applesoft ROM routines. The same data stack is used for integers (2 bytes), reals (6 bytes), and complex numbers (12 bytes). The code for the complex routines was written entirely in FORTH, and, in compiled form, occupies about 2K. The conformal mapping code compiles to about 1K additional.

A FORTH BASED MICRO-SIZED MICRO ASSEMBLER

Gregory E. Cholmondeley
Laboratory for Laser Energetics
University of Rochester

Abstract

The FORTH programming language can be used to implement a very small and useful micro assembler. Functions ranging from automatic field alignment to user definable macros can be written and altered easily, permitting a flexible and easy to use microcoding technique. This paper also serves to illustrate several of the many programming features found in FORTH.

Introduction

Computer central processors often contain an iternal data form called "microcode." This code defines the instruction set of the processor. The creation of this internal code is called "microcoding."

Microcoding by hand is at best a tedious and wasteful undertaking where a significiant portion of a programmer's time is spent aligning fields, formatting output and correcting typographical errors. Understanding (let alone debugging) a microcode program is difficult due to the lack of readability from a human point of view. Through the use of comments, automatic field positioning, labels and other such tools, a good micro assembler should minimize the above problems making microcoding a much more agreeable form of programming.

There already are micro assemblers written which handle these along with other problems associated with microcoding, but most of them share one rather serious drawback: they are large programs. The micro assembler presented here is based heavily upon the Signetics[1] micro assembler but requires only a few "blocks" of FORTH code. Thus it is possible to have a micro assembler on a small home computer[Such an assembler could be used as a design tool as well as an inexpensive and effective teaching aid. It would allow even wide instruction words to be built in a simple to use, high level form.

Usage

There are two main phases associated with this micro assembler: instruction definition and actual programming. A third phase will be implemented shortly to allow the user to explicitly and easily define output formats. The first of these phases to be explored is the instruction definition phase. This is the time when the various instruction word formats are defined. A simple example of such a definition would be as follows:

INSTRUCTION WIDTH 8
Define an 8-bit instruction.

FIELD A WIDTH 4 DEFAULT 3
Define field A as the 4 most significant bit positions in the instruction, having a default value of 3.

FIELD B WIDTH 2
Define field B as the next 2 bit positions, having a default value of 0.

FIELD C WIDTH 2 DEFAULT 1
Define field C as the 2 least significant bits, having a default value of 1.

END.INSTRUCTION
Close the instruction definition.

The resulting instruction word would appear in the following form:

```
:7          4:3    2:1    0:
|     A     |   B  |   C  |
```

From this point on the field names A, B, and C will be unique and may not be used to define other fields.

While the preceding example is rather trivial an instruction definition may become quite complex. It is, for instance, possible to define multiple formats for every field, with each of these containing multiple sub-fields. This is useful when it is deemed that fields should have different meanings depending upon the context of the rest of the instruction word (vertical versus horizontal programming). Subfields are treated in the same manner as fields so that they too may have multiple formats and sub-fields. This feature is implemented as a tree structure allowing an unlimited nesting of fields, formats and sub-fields. Figures (1) and (2) should clarify this concept.

This part of the micro assembler has error checking capabilities which prevent unintentional overwriting of fields. For example, if field EE of figure (1) is filled, then fields BB, DD and GG (and of course EE) could not be used. Automatic field defaulting uses the same mechanism so that if field EE is the only field filled (using the format from the previous example) then fields AA, CC, FF and HH will be defaulted.

```
INSTRUCTION WIDTH 32
  FIELD AA WIDTH 8           DEFAULT 255
  FIELD BB WIDTH 16          DEFAULT 65535
    FORMAT
      FIELD CC WIDTH 4       DEFAULT 15
      FIELD DD WIDTH 12      DEFAULT 4095
        FORMAT
          FIELD EE WIDTH 10  DEFAULT 1023
          FIELD FF WIDTH 2   DEFAULT 3
        FORMAT.END
    FORMAT.END
    FORMAT
      FIELD GG WIDTH 16 DEFAULT 65535
    FORMAT.END
  FIELD HH WIDTH 8       DEFAULT 255
END.INSTRUCTION
```

Figure (1) : Sample Instructon Definition

```
 instruction
     /
    /
   /
AA-->BB-->HH fields AA, BB and HH
      |
      |
      *---->* field BB has 2 alternate
     /      | formats
    /       |
   /        |
CC-->DD     GG format 1 contains fields CC
      |        and DD format 2 contains field
      |        GG
      *
     /         field DD has 1 alternate
    /          format
   /
EE-->FF        fields EE and FF
```

Figure (2) : Structure of Figure (1)

The programming phase of the micro assembler is where the actual microcoding takes place. An instruction is created by typing the name of a field followed by a number or expression representing the value that that field should take. This is continued for as many fields as needed in the instruction word. When the instruction is complete a "$" (dollar sign) is typed and the computer readies itself for another word. At this point any undefined fields are set to their default values, the instruction and other related information is stored in memory, and the location counter is incremented. Figures (3) and (4) demonstrate a simple microcoded program which merely sets one field at a time equal to a zero.

```
PROGRAM 1EXAMPLE WIDTH 32

ORG 512

  AA 0 $
  BB 0 $
  CC 0 $
  DD 0 $
  EE 0 $
  FF 0 $
  GG 0 $
  HH 0 $

END.PROGRAM
```

Figure (3) : Sample Program

```
0000000011111111 1111111111111111    AA used  BB & HH defaulted
1111111100000000 1111111111111111    BB used  AA & HH defaulted
1111111100001111 1111111111111111    CC used  AA, DD & HH defaulted
1111111111110000 0000000011111111    DD used  AA, CC & HH defaulted
1111111111110000 0000001111111111    EE used  AA, CC, FF & HH defaulted
1111111111111111 1111110011111111    FF used  AA, CC, EE & HH defaulted
1111111100000000 0000000011111111    GG used  AA & HH defaulted
1111111111111111 1111111100000000    HH used  AA & BB defaulted
```

Figure (4) : Sample Output

While automatic field alignment is in itself a vast improvement over hand coding, there are a few other tools available to the programmer which make microcoding even easier. A "(." denotes a comment allowing anything up to and including a ".)" to be ignored. Typing ORG and a number or an expression will set the location counter (LC) to that value. Typing

```
SET <new variable name>
    TO <number or expression>
```

will declare and initialize a variable, while typing

```
EQU <old variable name>
    WITH <number or expression>
```

will store a new value into a previously declared variable. These variables return their value when they are typed (similar to a constant in FORTH) and can be used in expressions at any time and in any phase of the micro assembler.

One of the most versatile tools available in this micro assembler is the MICROP function. Microps are user-definable functions designed to eliminate a large part of the repetitious programming associated with microcoding. For example there may be times when several fields will always take on constant or relative values. Rather than cluttering the program by having to set all of these fields every time, a microp can be written to do this automatically. A program written using well named microps can in turn be quite a bit easier to read and understand than one which merely sets the fields.

The definition of a microp requires a unique name and a set of commands which will be executed whenever its name is called. Any FORTH programmer will soon realize that a microp definition is nothing other than a colon definition, thus allowing the full power of FORTH to be easily accessed directly from the micro assembler[An example of a simple microp that sets a few fields to zero would be:

```
MICROP EX1  (. set fields CC, FF,
  CC 0         and HH to 0 .)
  FF 0
  HH 0
END.MICROP
```

An example of this microp in use would be found in the programming phase and might look like:

```
      .
      .
AA 7 HH ( LC ) $
AA 8 EX1 $
      .
      .
```

NOTE: LC in the preceding example is a variable, the "(" and ")" are required for its proper execution. They do not denote a comment in the MICRO vocabulary context. This is also true when building microps. In the MICRO vocabulary comments are delimited by "(." and ".)".

Being simple colon definitions, microps can do internal testing, looping and everything else offered in FORTH. Microps can expect parameters on the stack as well as numbers or expressions from the input buffer via a function called GET#. For example:

```
MICROP ?GT    (. <exprl> ?GT <expr2> -- tests if exprl is > expr2 .)
   GET# >
    IF AA 0 BB 0 CC 0
    ELSE HH ( LC )
    THEN
END.MICROP
```

This could be used like:

```
      .
      .
AA 19 $
<variable.name> ?GT 1024 $
      .
      .
```

Finally, microps have macro capabilities in that they can be nested and may even create several lines of code in one call (as may be needed in a test and branch, or jump substitute routine).

```
MICROP EX3
  LC 100 >
    IF EX1 $
       LC ?GT 1000 $
    ELSE AA 0 $
         CC 0 HH 0 $
    THEN
END.MICROP
```

Another way to increase readability in the micro assembler is through the use of labels. This feature is only partially implemented at this time but will work as follows. Labels must have unique names and must be declared via LABEL statements before they are used. When a label is found immediately preceding a new instruction word (or in other words; immediately following a "$") the current value of the location counter (LC) is stored as the value of the label. Multiple labels may be used to represent the same line of code. When a label is used inside an instruction definition after its value has been set, it will be treated as any other variable. If the label has not been set to a value (i.e., forward referencing) a zero will be returned and all information necessary to resolve the reference will be stored in memory for the second pass. During the second pass the micro assembler will shift the correct value(s) of the label(s) into the proper place(s) and then add the resulting number to the rest of the word. This allows labels to be referenced more than once in a single instruction. It also allows addition and subtraction of other non-label expressions to labels (i.e., AA (1LABEL + 2) or AA (1LABEL - 1) but not AA (1024 - 1LABEL)). When this is implemented another extended precision function (E+) will be needed to perform the extended precision addition.

The last major feature of the micro assembler concerns output formatting. This has not been developed at all but will consist of a basic instruction set for programmers to use to define specific output formats (i.e., hex, insertion of special delimiting characters, etc.). The programmer will define a function (similar to a microp or colon definition) for each type of output format. The executable code field address of the current formatting function is stored along with the other instruction word information on the first pass. On the second pass the formatting function will be executed to produce the desired result. It will be possible to change the current format function between instruction words by using a command of the form:

```
SET.FORMAT <format function name>
```

allowing multiple output formats within a single program. By installing different formats in currently existing ones, it will be possible to view the code in punched card format as well as a format suitable for blowing PROMs!

Implementing Techniques

The first problem that I addressed was how to align the fields in an instruction word definition. For words that are 32 or fewer bits wide the solution is simple, merely do logical shifting and ORing. Since 32 bits is a rather stringent limit on the word width, I have kept the same basic strategy but have defined a set of functions which can do logical operations upon extended precision words. The precision (in terms of 16-bit words) is stored in a variable called PRECISION and is set at the PROGRAM WIDTH statement. These are the extended precision functions which I needed:

1. EXT.PREC - This is a defining word that creates an extended precision variable which uses the Bartholdi "TO concept" to store and fetch extended precision numbers. EXT.PREC expects the desired precision of the new variable on the stack.

2. E.FILL - E.FILL expects a number and the precision of that number in terms of 16-bit words on the stack. It uses this to fill in the most significant places with zeros until the number has a precision equal to the current value of PRECISION. Notice that the value of PRECISION must be larger or equal to the length of the given number.

3. E.DROP - This function drops an extended precision number from the top of the stack.

4. ESL - The ESL function performs a logical shift to the left on an extended precision number. It expects the extended precision number and the number of shifts on the stack and returns the shifted number.

5. EOR - This takes two extended precision numbers off of the stack, logically ORs them together and returns the resulting number.

6. EXOR - This executes an exclusive OR operation between two extended precision numbers. It expects two extended precision numbers and returns the result.

7. ECOM - ECOM does a 1's complement of the given extended precision number.

One extended arithmetic function will be needed to implement forward referencing of labels. This function has already been mentioned and will be called E+.

```
******************  BLOCK    160     ******************

( algebraic notation              GEC               15-JUL-81 )

: GET# ( [<>--<input expression's value>] )
  32 WORD NUMBER NOT                 ( get next input char/num )
    IF R> R> SWAP >R >R THEN ;  ( if char then treat as '(' )

: (. [COMPILE] ( ; IMMEDIATE ( define (. as comment delimiter )

VOCABULARY MICRO                  MICRO DEFINITIONS
: + GET# + ;              (. [<#1>--<#1 + #2>] redefine + .)
: - GET# - ;              (. [<#1>--<#1 - #2>] redefine - .)
: * GET# * ;              (. [<#1>--<#1 * #2>] redefine * .)
: / GET# / ;              (. [<#1>--<#1 / #2>] redefine / .)
: ) R> R> SWAP >R >R ;    (. [<>--<>] end expression      .)
: ( ) ;                   (. [<>--<>] start expression    .)
FORTH DEFINITIONS                                   -->

******************  BLOCK    161     ******************

( value and flipflop types        GEC               10-JUN-81 )
0 VAR %TO ( flag )           : TO 1 %TO ! ;

: VAL ( returns value of variable [ not address ] )
    <BUILDS , DOES>
         %TO @ IF ! 0 %TO !     ( store value )
         ELSE @                 ( push value  )
         THEN ;

: FLIPFLOP ( returns 0/1 and stores 1/0 )
  <BUILDS 0 ,                     ( [<>--<>] initialize F.F )
  DOES> %TO @
    IF ! 0 %TO !                  ( [<1/0>--<>] set F.F.     )
    ELSE DUP @ DUP NOT ROT !      ( [<>--<1/0>] flip F.F.    )
    THEN ;
                                                    -->

******************  BLOCK    162     ******************

( variable definitions            GEC               19-JUN-81 )
0 VAL CUR.ADDR                                ( current address )
0 VAL C.FIELD                                   ( current field )
0 VAL C.FORM                                   ( current format )
0 VAL C.INSTR                       ( current instruction word )
0 VAL F.LENGTH                                   ( field length )
0 VAL F.POS                                    ( field position )
0 VAL LC                                     ( location counter )
0 VAL INSTRWIDTH                          ( instruction width )
0 VAL L.FIELD                                      ( last field )
0 VAL L.FORM                                      ( last format )
0 VAL L.INSTR                                ( last instruction )
0 VAL MEM               ( current memory addr for print routines )
0 VAL NEW.WORD            ( flag set at start of new instr. word )
0 VAL OFFSET                       ( offset of shift {used in ESL} )
                                                    -->

******************  BLOCK    163     ******************

( variable definitions - 2        GEC               19-JUN-81 )
0 VAL OVFLG                                     ( overflow flag )
0 VAL PLACE      ( addr of temp storage in extended operations )
0 VAL PRECISION            ( precision of word in 16 bit units )
0 VAL TEST.FLAG  ( flag used in error checking and defaulting )
0 VAL TSHIFT            ( intermediate number of shifts {ESL} )
0 VAL %DEF       ( default phase {0. use/1. set/2. initialize} )
0 VAL %FLAG                   ( value to store in flags {0/1} )
0 VAL %PRINT.FORMAT                ( addr of output format code )
FLIPFLOP FLD.FF    ( field F.F. for error checking & defaulting)
0 XEQ BROTHER     ( brother of current field/format )
0 XEQ PARENT        ( parent of C.FIELD )
0 XEQ SELF                          ( C.FIELD )
0 XEQ UNCLE                                  ( uncle of C.FIELD )

                                                    -->

******************  BLOCK    164     ******************

( extended precision functions  GEC              12-JUN-81 )

: EXT.PREC ( <precision>-<> builds an extended precision # )
  <BUILDS DUP 2* , 0 DO 0 , LOOP
  DOES>    ( <>-<low-order ... high-order> or reversed if %TO )
     DUP DUP @ + 2 + SWAP 2 +
     %TO @ IF DO I ! 2 +LOOP 0 %TO !        ( stores #  )
            ELSE SWAP 2 - DO I @ -2 +LOOP   ( fetches # )
            THEN ;

: E.FILL ( <# len>-<# 0 ... 0> puts 0's in high order places )
  PRECISION SWAP 2DUP > IF DO 0 LOOP ELSE 2DROP THEN ;

: EDROP ( <low-order ... high-order>-<> drops ext.precision # )
  PRECISION 0 DO DROP LOOP ;
                                    -->
```

When a field is assigned a value and is aligned, the following process occurs. An extended precision number with a precision equal to PRECISION is on the stack. This is the value of the current line of microcode. After the field-name is typed, an extended precision number with a precision equal to the width of the field is accepted. E.FILL is used on this number to make it the same precision as the instruction word, ESL is used to shift it over the proper number of bits, and EOR is used to update the micro-instruction. This is repeated until a "$" is encountered which will clear the flags, set any defaulted fields, store the extended precision instruction word in memory and leave an extended precision number equal to zero on the stack (for the next micro-instruction).

The second main problem that I faced dealt with how to handle multiple formats. I implemented a tree structure where the instruction is the root with the list of fields as its children. Each field has a list of formats or a zero for its children. Every format has a list of fields as its children and the cycle continues. Each node in this tree has pointers to its parent, "oldest" child, and next youngest brother. Each node also contains a flag denoting whether it is a valid field or not, a value corresponding to its starting position in the instruction word, its field length and its default value. Thus when a field is accessed a test is executed to determine whether it is valid or not. This is accomplished by traversing up the tree and checking the validity flag. If the first set flag is found in a field, then the programmer is trying to overwrite another format in the same field. If no flag is set and this is not a new line of microcode, then this field is not defined in the same instruction word as the previous one(s) and another error condition is found. If, however, the field is determined to be valid, then the flag bit of that field will be set along with the flag of its parent, and its parent, continuing up to the root. When a "$" is encountered, the tree is traversed in the same manner but from the root down and all flags are reset. At the same time any unused brothers of the lowest level fields used will be assigned their default values.

INSTRUCTION FORMAT FIELD

	INSTRUCTION	FORMAT	FIELD
Parent	0	field	format
Brother	0	format	field
Used Flag	0/1	0/1	0/1
Child	field	field	format
Field Starting Position			
Field Length			
Default Value or Zeros			- - - -

```
******************  BLOCK    165     ******************

( extended prec. functions - 2  GEC                12-JUN-81 )
: ESL ( <low-ord ... high-ord #-shifts>-<low-ord ... high.ord>
    shifts #-shifts to left {drops high ov & shifts in 0's }
  0 TO OVFLG HERE PRECISION 2 * + DUP TO PLACE HERE
     DO 0 I ! 2 +LOOP                        ( create workspace )
    0 PRECISION 1 - 2* DO I TO OFFSET DUP TO TSHIFT SWAP
                                ( for byte from high to low do )
    BEGIN TSHIFT 16 >=
     IF                                       ( #-shift >= 16 )
        OFFSET 2 + TO OFFSET
          TSHIFT 16 - TO TSHIFT
          1 TO OVFLG                         ( set overflow flag )
     ELSE                   ( #-shift < 16 { shift normally } ;
       DUP TSHIFT <-L      OFFSET HERE +
       DUP @ ROT   OR SWAP !
                                  -->

******************  BLOCK    166     ******************

( extended prec. functions - 3  GEC                12-JUN-81 )

       OFFSET 2 + HERE  + DUP @  ( handles #s that are split )
       ROT 16 TSHIFT - ->L OR SWAP ! ( into 2 bytes by shift )
      THEN OVFLG NOT 0 TO OVFLG
     END
  -2 +LOOP DROP
  PLACE HERE DO I @  2 +LOOP ;  ( fetch # from temp workspace )

: ?PRECISION                  ( [# of bits]--[# of 16-bit words] )
  0 17 M/MOD DROP SWAP DROP 1+ ;

: EGET ( [<addr of variable>--<ext.pre.#>] )
  DUP PRECISION 1 - 2* + DO I @ -2 +LOOP ;

                                                    -->

******************  BLOCK    167     ******************

( extended prec. functions - 4  GEC                15-JUN-81 )
: EOR      ( <ext.pre.# ext.pre.#>-<ext.pre.#> OR 2 ext.pre #s )
  HERE PRECISION 2* + 1 - DUP TO PLACE HERE DO 0 I ! 2 +LOOP
  1 PRECISION DO
     I PRECISION + PRECISION I - + PICK
     PRECISION 1 + PICK OR -1 +LOOP
  HERE PLACE DO I ! -2 +LOOP
  PRECISION 2* 0 DO DROP LOOP
  PLACE HERE DO I @ 2 +LOOP ;

: ECOM ( [<ext.#>--<NOT ext.#>] one complements ext.pre.# )
  HERE PRECISION 2* + 1- DUP TO PLACE HERE
  SWAP DO I ! -2 +LOOP
  PLACE HERE DO I @ COM 2 +LOOP ;

: ERROR.FUNCT .' ERROR CODE: ' . CR ;            -->

******************  BLOCK    168     ******************

( extended prec. functions - 5  GEC                15-JUN-81 )
: EXOR     ( <ext.pre.# ext.pre.#>-<ext.pre.#> OR 2 ext.pre #s )
  HERE PRECISION 2* + 1 - DUP TO PLACE HERE DO 0 I ! 2 +LOOP
  1 PRECISION DO
     I PRECISION + PRECISION I - + PICK
     PRECISION 1 + PICK XOR -1 +LOOP
  HERE PLACE DO I ! -2 +LOOP
  PRECISION 2* 0 DO DROP LOOP
  PLACE HERE DO I @ 2 +LOOP ;


                                                    -->

******************  BLOCK    169     ******************

( offsets in field structure    GEC                3-JUL-81 )
: OFF.VAL
  + %TO @  IF ! 0 %TO ! ELSE DUP 0<> IF @ THEN THEN ;

: ?PARENT 0 OFF.VAL ;               : ?BROTHER 2 OFF.VAL ;
: ?FLAG 4 OFF.VAL ;                 : ?CHILD 6 OFF.VAL ;
: ?ANCESTOR ?PARENT ?PARENT ;
: ?INSTRUCTION.WIDTH 8 OFF.VAL ;                ( INSTRUCTION )
: ?FIELD.START C.FIELD 8 OFF.VAL ;              (    FIELD    )
: ?FIELD.LENGTH C.FIELD 10 OFF.VAL ;            (    FIELD    )
: ?DEFAULT C.FIELD 12 + ;                       (    FIELD    )
: NEW.SON
  DUP ?CHILD DUP ROT AND
    IF 0 SWAP BEGIN DUP ?BROTHER ROT DROP DUP NOT END DROP
    ELSE DROP 0
    THEN TO BROTHER ;                                  -->
```

With the structures defined, the task of creating a program comes to light. An explanation has already been given describing how the words are constructed. The following diagram should help clarify how a "program" is actually stored in memory in its first pass form.

General First Pass Structure for Microcode Programs

```
                  _______
::::::::::::::::|___|___|       Forth
Forth           |___|___|        Name
Header          |___*--|----- Link
::::::::::::::::|___*--|----- Description
Program         |___*--|----- Instruction Word Width
Header          |___0___|
::::::::::::::::|___*--|----- Address of Label
                |___*--|----- Field (ie. # of shifts)
                |   :   |
Complete        |___:___|
First Pass      |___*--|----- Address of Label
Data For        |___*--|----- Field
One             |___0___|
Instruction     |___*--|----- Output Format
Word            |___*--|----- LC
                |_ _ _ _|       Instruction
                |  _ _  |           Word
::::::::::::::::|_______|
                |___*--|----- Address of Label
                |___*--|----- Field
                |   :   |
                |___:___|
                |  _ _  |       Instruction
                |_______|           Word
                |___1___|       End of Program
```

Each program has a unique name which defines a FORTH header. When this name is typed, the program is listed in a basic binary and hex form along with the format address, LC, and any unresolved labels.

One of the primary objectives of this micro assembler is to make microcoding easier by making it more readable, and there are quite a few places where the reverse polish notation found in FORTH does not appear quite as nice as an infix or prefix form. Hence, I have written a few short functions to allow FORTH functions to accept numbers and expressions from the input buffer as well as from the parameter stack.

This method uses the return stack via a function GET# which accepts input from the input buffer. If the input is a number GET# places it on the stack and returns. If the input is not a number then GET# assumes that the programmer typed a left parentheses "(" meaning that there is an expression or a variable in the input buffer. If this is the case then GET# will swap the last two values on the return stack and return. When a right parentheses is found, the top two values of the return stack are again swapped and the system is back to normal. This is simple and fast, although it has no method of checking whether a set of parentheses is properly closed. However, a variable could be used which would be incremented

```
*******************  BLOCK    170     *******************

( headers of fields & formats   GEC                3-JUL-81 )
: ?NAME DUP 0<> IF CFA TNAME ELSE DROP THEN ;
: IGNORE 32 WORD DROP ;

: HEADER             ( creates 1st 4 fields in FIELD and FORMAT )
  0 TO UNCLE   HERE TO SELF
  BROTHER 0<> IF SELF BROTHER TO ?BROTHER
              ELSE SELF PARENT TO ?CHILD
              THEN SELF TO BROTHER
  PARENT , 0 , 0 , 0 , ;              ( parent/brother/flag/child )

: FORMAT.HEADER  ( defines FORMAT relatives & executes HEADER )
  INSTALL L.FIELD IN UNCLE       INSTALL C.FIELD IN PARENT
  INSTALL L.FORM  IN BROTHER     INSTALL C.FORM  IN SELF
  C.FIELD NEW.SON HEADER 0 TO C.FIELD ; -->

*******************  BLOCK    171     *******************

( instruction and format defs.  GEC                3-JUL-81 )

: INSTRUCTION             ( INSTRUCTION <name> WIDTH <width> )
  0 TO C.FIELD FORMAT.HEADER
  IGNORE  GET#
  DUP ,                                      ( instruction width )
  DUP TO F.LENGTH  TO F.POS ;   ( field length/field position )
: FORMAT                                              ( FORMAT )
  ?FIELD.LENGTH TO F.LENGTH                    ( field length   )
  ?FIELD.START F.LENGTH + TO F.POS             ( field position )
  FORMAT.HEADER ;
: SET.FLAGS          ( <#>-<> sets flags from C.FIELD up to # )
  TO %FLAG
  C.FIELD
  BEGIN ?PARENT %FLAG TO OVER ?FLAG DUP NOT END DROP
  %FLAG C.FIELD TO ?FLAG ;                                  -->


*******************  BLOCK    172     *******************

( format.end and field header   GEC                3-JUL-81 )
: FORMAT.END                                     ( END.FORMAT )
  C.FIELD ?ANCESTOR DUP TO L.FIELD TO C.FIELD
  C.FIELD ?PARENT IF ?FIELD.START ELSE 0 THEN F.POS <>
    IF 2 ERROR.FUNCT RESTART
    ELSE ?FIELD.LENGTH TO F.LENGTH
         ?FIELD.START TO F.POS
    THEN ;

: FIELD.HEADER
    INSTALL L.FORM  IN UNCLE      INSTALL C.FORM  IN PARENT
    INSTALL L.FIELD IN BROTHER    INSTALL C.FIELD IN SELF
    SELF 0<> IF SELF ?PARENT ELSE C.FORM THEN
    DUP TO PARENT NEW.SON
  HEADER ;                                                  -->


*******************  BLOCK    173     *******************

( error checking for used fields GEC               3-JUL-81 )
: ER.CHECK              ( check to see if field is permitted )
  0 TO FLD.FF C.FIELD
  BEGIN
    DUP ?FLAG TO TEST.FLAG                ( set TEST.FLAG=FLAG )
    FLD.FF DROP                          ( flip field.flip.flop )
    ?PARENT                                       ( go to parent )
    DUP NOT TEST.FLAG OR         ( if flag found or root reached)
  END DROP
  TEST.FLAG FLD.FF AND
    IF 4 ERROR.FUNCT RESTART              ( field defined twice )
    ELSE TEST.FLAG NOT
      IF 5 ERROR.FUNCT RESTART        ( not proper instruction )
      THEN
    THEN 0 TO TEST.FLAG ;
                                                            -->

*******************  BLOCK    174     *******************

( defaults                      GEC                8-JUL-81 )
: DO.DEFAULT
  ?FIELD.LENGTH ?PRECISION
  %DEF SEL
      << 2 ==> DROP 0 DO 0 , LOOP 0 TO %DEF 0 >>
      << 1 ==> DROP E.FILL ?DEFAULT ?FIELD.LENGTH
               ?PRECISION OVER + 2* SWAP
               DO I ! 2 +LOOP 0 TO %DEF 0 >>
      << 0 ==> DROP ?DEFAULT SWAP 1 - 2* OVER +
               DO I @ -2 +LOOP ?FIELD.LENGTH ?PRECISION
               E.FILL ?FIELD.START ESL EOR 0 >>
      ENDSEL ;
: TO.DEF 1 TO %DEF ;                 : INIT.DEF 2 TO %DEF ;

: DEFAULT
  GET# TO.DEF DO.DEFAULT ;                                  -->
```

```
******************  BLOCK    175     ******************

( field structure                  GEC              3-JUL-81 )

: FIELD                          ( FIELD <name> WIDTH <width> )
  <BUILDS IGNORE GET#
    DUP F.LENGTH <=
      IF FIELD.HEADER
         F.LENGTH OVER - TO F.LENGTH
         F.POS OVER - DUP TO F.POS
         , ,                       ( field start/field length )
         INIT.DEF DO.DEFAULT
      ELSE 1 ERROR.FUNCT RESTART
      THEN
  DOES> TO C.FIELD
    NEW.WORD IF 0 TO NEW.WORD ELSE ER.CHECK THEN 1 SET.FLAGS
    GET# ?FIELD.LENGTH ?PRECISION E.FILL
    ?FIELD.START ESL EOR ;                                  -->
```

when a "(" is encountered and decremented when a ")" is found. This would catch any errors involving too many closing parentheses. A "]" function could be written which would behave in the same manner as the UCI LISP function of the same name. It would use the variable mentioned above to close all open parentheses for a successful evaluation of the expression.

GET# and its related algebraic functions have some interesting features in that there is no hierarchial ordering of functions (i.e., 2 + 3 * 5 = 25 while 5 * 3 + 2 = 17), however, expressions enclosed in parentheses will be solved before others (i.e., 2 + (3 * 5) = 17). The entire code for this is only a few lines long and is as follows:

```
: GET# 32 WORD NUMBER                              gets number
  NOT IF R> R> SWAP >R >R                          swap if not a number
  THEN ;

VOCABULARY ALGEBRAIC   ALGEBRAIC DEFINITIONS   redefine functions


: + GET# + ;                : - GET# - ;
: * GET# * ;                : / GET# / ;           re-swap return stack
: ) R> R> SWAP >R >R ;                             swap return stack
: ( ) ;

FORTH DEFINITIONS
```

A typical usage of this function could be:

```
          : {+} GET# + ;

          3 {+} ( 4 {+} 5 ) .

          12
```

Current Function	Command	Parameter Stack	Return Stack	
main	3	3	-	input a 3
{+}	{+}	3	main	call function {+}
GET#	GET#	3	main {+}	call function GET#
	(	3	{+} main	swap return stack
main	4	3 4	{+}	return and input a 4
{+}	{+}	3 4	{+} main	call {+} again
GET#	5	3 4 5	{+} main {+}	input a 5
{+}	+	3 9	{+} main	return and add
main		3 9	{+}	return to main
)	)	3 9	{+} main	call function)
		3 9	main {+}	swap return stack
{+}	+	12	main	return and add
main	.	-	-	return and print

There are a few general concepts which are used throughout this micro assembler, one of which is the "TO concept" (see Joe Sawicki's paper entitled Optimized Data Structures for Hardware Control). This concept allows the use of variables without the programmer having to deal directly with the address. While this may be thought of as being a bit un-FORTH-like, it does result in much cleaner code. I adapted the concept in one place to build a flip-flop function. This function creates a data type which alternately returns zeros and ones whenever it is called and makes use of the "TO concept" to allow itself to be initialized to either state. The micro assembler also makes use of multiple vocabularies to allow the same function to have different meanings in different contexts. While this is not absolutely essential for the assembler to run, it again makes the code cleaner and easier to use.

Conclusion

The reason why I have chosen to write this micro assembler in FORTH is simplicity. As I mentioned earlier, this "program" is based largely upon a very lengthy micro assembler written by Signetics and yet the FORTH code is only a few pages long. The time spent programming was equally short. It took roughly half of my time at work from around June 10 through July 15 to complete the micro assembler to this point (although I have occasionally gone back to add or change a feature or two). Two of the features that I did change, labels and forward referencing through the first pass, brought up another quality of FORTH: its modular nature. These are rather major additions and yet they only required one new "block" of code, a few minor changes in the old code and took only a few hours to implement[

Once the forward referencing is completed and the output formatting is implemented, this code will be a micro assembler by itself as well as a kernel for more extended versions. An example of an extended feature is the compilation of a symbol table at the end of a program. A further extension would involve tying this symbol table to other symbol tables to allow external references. Through the use of external symbol tables the microcode could be maintained in the first pass format so that the external references could be resolved several times for labels with differing values. This could result in a modular microcoding technique. Another extension could be a FORTH program which would be used, in much the same manner as the micro assembler, and similar to Hardware Description Languages, to describe a simulator for the microcode. These two programs would constitute a powerful yet inexpensive teaching aid as well as an effective design tool. Programmers and students would not need to waste their time punching cards or blowing PROMs in order to discover the errors in their code[A dozen other "nice" features can be imagined (i.e., prohibiting forward referencing to allow interactive microcoding, or the development of intrinsic microps to define commercial chips, etc.), but the point is that they could all be based around the small "kernel" micro assembler presented here.

Acknowledgements

I would like to thank Lawrence Forsley for the time and effort he expended helping to direct and complete this project. I would also like to extend thanks to Dr. Charles Merriam for his useful comments and suggestions.

This work was partially supported by the following sponsors: Exxon Research and Engineering Company, General Electric Company, New York State Energy Research and Development Authority, Northeast Utilities, The Standard Oil Company (Ohio), the University of Rochester, Empire State Electric Energy Research Corporation, the Center for Naval Analysis, under grant number CNA SUB N00014-76-C-0001 and the U.S. Department of Energy inertial fusion program under contract number DE-AC08-80DP40124.

G.E. Cholmondeley is currently an undergraduate student in the department of Electrical Engineering at the University of Rochester. His interests lie in computer software and hardware design.

1. Signetics Micro Assembler Reference Manual

HELP WANTED

FORTH Software Engineer

Program, edit and maintain files for 8080. Ability to troubleshoot the software-hardware interface.

Call:

Wendy Palmer
1-800-225-4040
Instrumentation Laboratory, Inc.
Analytical Instrument Division
Jonapin Road
Wilmington, MA 01887

```
******************  BLOCK     176     ******************

( end.instr & find root & brother GEC           11-JUN-81 )

: END_INSTRUCTION           ( checks for any undefined fields )
   BEGIN FORMAT.END C.FIELD ?ANCESTOR NOT END ;

: ROOT 0 SWAP                               ( finds instruction )
   BEGIN                           ( [* self]--[self parent] )
     DUP ?PARENT ROT DROP DUP NOT
   END DROP ;

: FIND.BROTHER 0 SWAP             ( finds brother with flag set )
   BEGIN
     DUP ?BROTHER ROT DROP          ( [* self]--[self brother] )
     OVER ?FLAG OVER NOT OR              ( flag OR not brother )
   END DROP DUP
   ?FLAG NOT IF DROP 0 THEN ;       ( [brother OR 0 ] ) -->

******************  BLOCK     177     ******************

( default - 2                    GEC                8-JUL-81 )
: DEFAULT!
   C.FIELD ROOT 0 OVER TO ?FLAG ?CHILD
   BEGIN DUP TO C.FIELD ?FLAG NOT
       IF DO.DEFAULT C.FIELD DUP ?BROTHER ( no flag set-default)
          C.FIELD CR ?NAME .' DEFAULTED '
       ELSE C.FIELD 0 OVER TO ?FLAG       ( flag set-reset to 0  )
         DUP ?CHILD FIND.BROTHER DUP      ( find sub-format used )
           IF 0 OVER TO ?FLAG       ( reset format flag to 0     )
             ?CHILD                 ( check sub-fields           )
           ELSE DROP DUP ?BROTHER ( no format used-find brother)
             C.FIELD CR ?NAME .' USED'
       THEN THEN DUP NOT
       IF BEGIN DROP ?ANCESTOR DUP ?BROTHER OVER NOT OVER OR END
       THEN SWAP NOT
   END DROP CR ;                                            -->

******************  BLOCK     178     ******************

( micro-assembler: forward ref. GEC             17-JUL-81 )
: LABEL                          ( LABEL <name> )
  <BUILDS
    0 , 0 , ( def.flag / val )
  DOES> NEW.WORD
    IF DUP @ IF .' Label previously defined' CR RESTART THEN
      1 OVER !                    ( set flag )
      2+ LC SWAP !                 ( set value )
    ELSE DUP @
      IF 2+ @
      ELSE ?FIELD.START SWAP , , 0
    THEN  THEN ;



                                                            -->

******************  BLOCK     179     ******************

( end of word & origin           GEC             11-JUN-81 )

: $                              ( ends word in program mode )
  C.FIELD ROOT IF DEFAULT! THEN
  0 ,                           ( end of labels )
  %PRINT.FORMAT , LC DUP , 1+ TO LC
  PRECISION 0 DO DUP B. , LOOP CR
  0 1 E.FILL
  1 TO NEW.WORD ;

: ORG
  GET# TO LC ;



                                                            -->

******************  BLOCK     180     ******************

( printing routine               GEC             18-JUN-81 )
: U.ZERO
  DUP 4096T U>= IF 0T
     ELSE DUP 256T U>= IF 1T
            ELSE DUP 16T U>= IF 2T
                  ELSE 3T
     THEN THEN THEN
  DUP IF DUP 0T DO 0T 1T U.R LOOP THEN
  4T SWAP - U.R ;




                                  -->
```

```
******************  BLOCK    181      *******************

( printing routines - 2          GEC              16-JUN-81 )

: #PRINT ( <ext.pre.#.addr>-<> print ext.pre.# in binary & hex )
  DUP PRECISION 2T * + SWAP 2DUP DO I @ B. 2T +LOOP
     ." :  " DO I @ U.ZERO 2T +LOOP ;

: MEM.INC MEM DUP 2+ TO MEM @ ;

 -->

******************  BLOCK    182      *******************

( printing routines - 3          GEC              16-JUN-81 )
: 1.PASS.PRINT
  DUP TO MEM @ 1 AND
    IF ." ERROR - PROGRAM LENGTH 0 " CR
    ELSE 16 BASE ! CR BEGIN MEM @
      IF BEGIN
          ." LABEL  : " MEM DUP @ CFA TNAME CR 2+ TO MEM
          ." SHIFTED: " MEM DUP @ . CR CR 2+ DUP TO MEM @ NOT
         END
      THEN MEM 2+ TO MEM
    ." FORMAT: " MEM DUP @ . CR 2+ TO MEM
    ." IC    : " MEM DUP @ . CR 2+ TO MEM
    MEM #PRINT CR MEM PRECISION 2* + TO MEM
    CR CR CR MEM @ 1 = END CR 10T BASE !
    THEN ;                                              -->

******************  BLOCK    183      *******************

( program statement              GEC              16-JUN-81 )

: PROGRAM
  <BUILDS IGNORE GET# DUP , ?PRECISION TO PRECISION 0 ,
    1 TO NEW.WORD
    0 1 E.FILL
  DOES> DUP @ ?PRECISION TO PRECISION 4 + 1.PASS.PRINT ; -->

******************  BLOCK    184      *******************

( end program & Microp commands GEC               17-JUN-81 )
: END.PROGRAM
   EDROP 1 , ;

: MICROP [COMPILE] : ;

: END.MICROP [COMPILE] ; ; IMMEDIATE

: SET ( defines a variable data type )
  <BUILDS IGNORE GET# ,        ( SET <var.name> TO <expression> )
  DOES>   @ ;                        ( <var.name> returns value )

: EQU ( EQU <var.name> WITH <expression> )
  ]'[ IGNORE GET# SWAP ! ;

MICRO  DEFINITIONS                                  ;S
```

INDUSTRY NEWS

FORTH-Based Savvy Lets User Talk to Computer

FORTH, Inc. is working with its parent company, Technology Industries, Inc. of Santa Clara, California, to develop a new software package for the Apple II, using a Z80 processor. With it, the Apple will offer the kind of casual and efficient man-computer interface that until now, existed only in movies like 2001 and Star Wars.

The project calls for Savvy--the trade name for Excalibur Technology Corporation's Adaptive Pattern Recognition Processor--to be used as a unique language interpreter. Savvy permits a user to communicate with a computer in the user's native language and normal praseology--no special language and formm are needed. Specifically, Savvy:

o Recognizes written words strung together in idiomatic phrases. (Future versions will understand spoken words and respond to Spanish commands as well as English. Other languages will follow.)

o Translates these imprecise patterns into precise computer commands.

Savvy's unique interactive approach to dealing with computers is an important development for the 80s. The powerful combination of FORTH and Savvy will be significant in realizing the system's full potential and demonstrating the power of FORTH. A special development team has been formed for this project, including Art Gravina, Chuck More, Dean Sanderson, and another programmer who has not been identified.

NO ROOM FOR THE ORDER FORM THIS TIME!
ORDER - Proceedings 1981 Rochester FORTH Standards Conference. Send check or MO to FIG in US funds on US bank, $25.00 US, $35.00 Foreign.

BULK RATE
U.S. POSTAGE
P A I D
Permit No. 261
Mtn. View, CA

DON'T MISS IT!

FORTH INTEREST GROUP

NATIONAL CONVENTION

NOVEMBER 28, 1981

Marriott Hotel
Santa Clara, CA

MAKE YOUR RESERVATIONS NOW!

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

TIME DATED MATERIAL
DELIVER BEFORE
NOV. 6

Address Correction Requested