Vierte Dimension
Volume IV/Nr.1 März 1988

VIERTE DIMENSION

NC4000 INTERUPT REPARATUR
LOKALE VARIABLEN
ACCESSORIES
FORCE
M17

Jahrestreffen in München 14.-15.Mai

FORTH MAGAZIN

5.00 DM

CWT - Selbstverwalteter Computerwerbetausch

Kontaktadresse: Rainer Mertins - Antilopenstieg 6a - 2000 Hamburg 54

***** Sonderangebot fuer Mitglieder der Forth-Gesellschaft *******

Aus dem Kreise der volksFORTH-Autoren entstand die Idee zu einem selbstverwalteten Anzeigenblatt aller Interessierten vom Hobby-Computerfan (der seine Programme tauschen moechte), ueber den Halbprofessionellen (der seine Datenbank-Kenntnisse bezahlterweise anbieten moechte), bis hin zum Profi (der eine zusaetzliche -preiswerte- Werbemoeglichkeit sucht).

Die Nullnummer des SELBSTVERWALTETEN COMPUTERWERBETAUSCHES wird in einer Auflage von ca. 500 erscheinen und im wesentlichen an "Multiplikatoren" verschickt werden, also an Computerlaeden, Buchhandlungen, Computerclubs und -zeitschriften und an Personen, die selber halb- oder vollprofessionell in der Branche taetig sind.

Fuer die Nullnummer werden den Mitgliedern der FORTH-Gesellschaft e.V. Sonderkonditonen eingeraeumt(andere zahlen etwas mehr).

- Fliesstextanzeige: Bis 5 Zeilen a 35 Anschlaege kostenlos. Jede weitere Zeile 30 Pf (andere: jede Zeile 50 Pf)
- selbstgestaltete Anzeige: 30 Pf je qcm (andere: 50 PF)
- Ganzseitenanzeige: 20.-DM (andere:30.-DM)
- Eintrag ins Branchenverzeichnis: kostenlos
  Ein Eintrag enthaelt ausser dem Namen max. 5 Zeilen a 35 Anschlaege mit Beschreibung zur Taetigkeit. Jede weitere Zeile: Kleinanzeigenpreise.
- Belegexemplar: 1.10 DM (Porto)

Auftraege unter Angabe der Mitgliedsnummer (steht auf dem Adressaufkleber der 4.Dimension) an Rainer Mertins. Kleinere Betraege in Briefmarken, sonst Verrechnungsscheck beilegen!

Erscheinungstermin:Mai 88 Spenden und Mitarbeit erwuenscht.

Lesenswert

Wem die Kontrollstrukturen des Forth ein Dorn im Auge ist und bleibt, dem sei der Artikel "Readable Forth" von Carl Wenrich (FD IX/4) empfohlen. Dort wird das Thema 'Sprache in der Sprache' behandelt mit Bezug auf M.Stolowitz (FD IV/6) und A.Lindley (FD VII/1,2). Konstruktionen wie

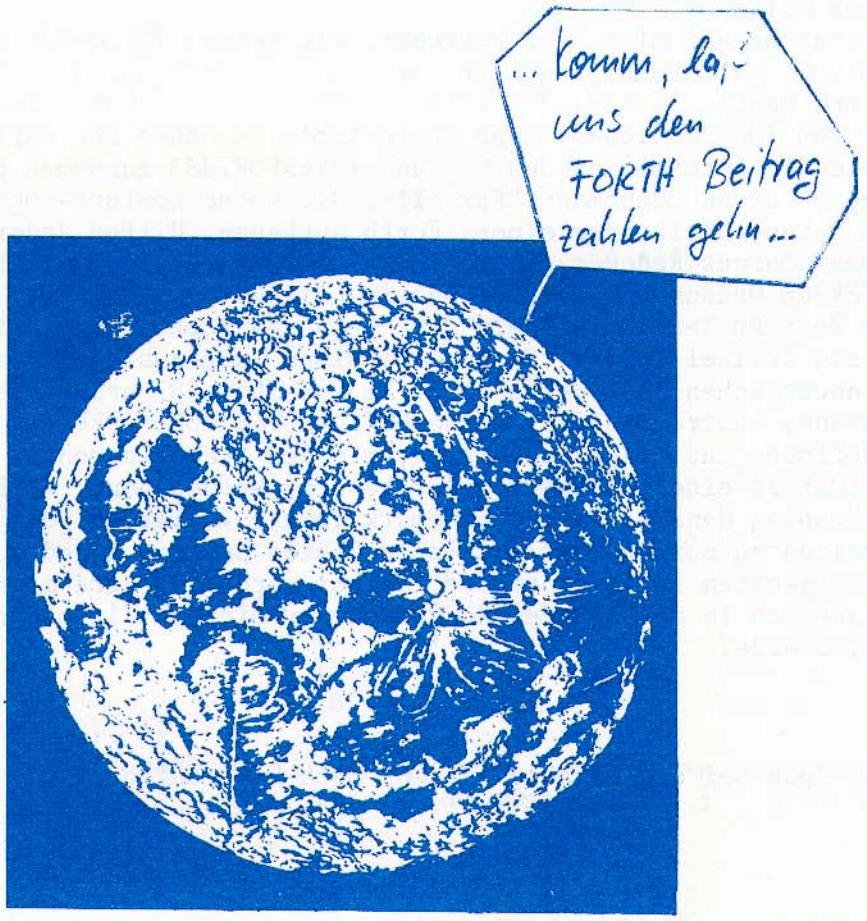
```
BEGIN  IF X ² 10 STAY  DISPLAY X  LET X = X + 1 NOW  END
```

kann man dort finden, also die hohe Kunst der Interpretation des Input-Stream mit viel COMPILE und TICKS und ³R und dergleichen mehr. Code für F83.

Ein ähnliches Thema ist die interaktive Benutzung von Kontrollstrukturen, nützlich zB für bedingtes oder alternatives Laden von Blocks. Lars Erik Svahn, Tyreso, Schweden, (FD IX/4) zeigt einen Weg, DO IF BEGIN etc im Interpreting Mode zu benutzen: Das Precompiling.

Die Transzendentalen Funktionen sind schwer zu implementieren - behauptet Phil Koopman (FD IX,4). In seinem Buch "MVP-FORTH Integer and Floating Point Math", Mountain View Press, 1985, sei es ihm jedoch gelungen. In diesem Artikel gibt er einige Kostproben. (Wer hat das Buch? Bitte melden beim Editor!)

INHALT



EDITORIAL

Im Mai werden wir in München sicherlich auch der Frage nachgehen, wie sich die FG und Forth in letzter Zeit entwickelt haben. Die Themen des Treffens werden noch bekannt gemacht. Fest steht schon, das der ganze Samstag und Sonntag Vormittag Forth getrieben werden wird und am Sonntag Abend der Verein auf der Tagesordnung steht. Neue Direktoren sollen gewählt werden und die Finanzen sind natürlich ebenfalls Thema.

Wie es den Forthvereinen in USA geht, und das es bezüglich ANS FORTH noch nichts neues gibt, berichtete kürzlich J.Shifrin. Ich finde das ja immer sehr interessant, aber zu viel Seiten Vereinsklüngel sollen im FORTH MAGAZIN auch nicht verschwendet werden. Daher erscheinen seine 5 Seiten aus der FD IX/5 nicht in der VD. Lest es bitte dort nach. Den schöne Leserbrief von Jose Betancourt in einer der letzten Forth Dimensions will ich Euch jedoch nicht vorenthalten. CISC, RISC, TIMS, WISK, FISC ... alles klar?

Und nun zu den Forthmaschinen. Nicht für jederman gedacht ist die "Processor Toolbox" von Harris zur Herstellung von 16-Bit Ein-chip-Micros. Der Prozessorkern ist eine Forth-Maschine, die "Forth Optimized Risc Computing Engine", kurz FORCE. Man könnte zwar, will aber der Welt keinen preiswerten Harris Forth-Micro schenken. Nehmen wir also den M17 Microprocessor. Dieser MISC ist angekündigt für Anfang 1988. Er hat nur noch 6 Instruktionen: CALL, RETURN, JUMP, SETFLAG, ACCESS und PROCESS. Software Optionen: Assembler, Forth-83 und C. Gemacht für Control-Applikationen. Ist das mein Zen-Forth? Um bei den FISCs zu bleiben: Der üble Fehler des NC4000 im Interupt-Eingang kann mit ein wenig Hardware umgangen werden. Wie die Reparatur erfolgen kann, zeigt Klaus Schleisiek auf.

Überrascht hat mich Marc Pertemann mit seinem TURBO-FORTH-83: Forth mit integrierter Textverarbeitung per Wordstar, Zugriff auf die Bibliotheken des FORTRAN und PASCAL, dBASE, MULTIPLAN etc. Das ideale Forth für alle glücklichen Besitzer von IBM-PC Clones? Eine komfortable Umgebung für Forth bietet auch Atari. Wie die Accessories des GEM und volksFORTH83 zusammen gebracht werden können zeigt Bernd Pennemann. Für alle, die keine Komfort-Umgebung in ihren Rechnern haben, bleibt nur eines: Forth ausbauen. Hilfen jeder Art sind da willkommen. Marcus Redeker, Herten, fand AROUND nützlich als Ergänzung zu COPY und CONVEY um Ordnung in den Screens zu halten.

Von Zeit zu Zeit entwickeln sich spontane Forth-Programmier-Wettbewerbe. Oft ist ein Artikel in der 'Forth Dimensions' der Auslöser - so wie kürzlich für den neuerlichen CASE Wettbewerb. Im November '87 präsentierte Peter Ross aus Brisbane, Australien dort seine LOKALEN VARIABLEN (FD IX/4). Jetzt zeigen Ulrich Hoffmann und Klaus Schleisiek, wie der Stack-Kommentar direkt als DECLARATIONS in einer Forth-Prozedur benutzt werden kann. Verblüffend einfach erzeugt Henning Hansen, Lynby, Dänemark lokale Variablen. Um die Bandbreite der Möglichkeiten zu zeigen, ist auch sein kurzer Beitrag abgedruckt.

Kleinigkeiten für die private Toolbox würde ich gern mehr veröffentlichen. Leider habe ich in den letzten Monaten keine einzige Einsendungen dazu bekommen. Schade.

Viel Spaß bei der Lektüre wünscht Euch Euer Editor.

Name That Architecture...
Dear Editor,

Articles now appearing on Forth-related processors have many ways of naming these items. For example, I have seen the terms RISC, Forth Engine, and Stack Machine. While these are descriptive, a better naming approach will have a few benefits. I propose that processors that run a Forth kernel, whether hard-wired or by programming in ROM or in microcode, have a standard, family name. There are many ways of doing this.

Following the style used by the mainstream, they can be labelled Forth Instruction Set Computers (FISC). This form parallels that used by other popular processor architectures: Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC).

A more technical term could be used: Threaded, Interpretive Stack Machines (TISM). This term is more precise and broadly applicable, and can even be used to describe processors related to Forth's *architecture*; for example, a Writable Instruction Set Processor. *[Or WISC Technologies' Writable Instruction Set Computers.—ed.]*

Another alternative is to honor Charles H. Moore, the creator of Forth and co-designer of a commercial Forth engine (the Novix NC4000), by naming the "two-stack, two-pointer, 4-space machine" after him. Unfortunately, the most direct term, Moore Machine, is already is use in connection with state-machine theory. Maybe someone can come up with something else.

The term "Forth engine," while it is still applicable to a FISC, does not seem correct, since Forth itself is undefined. Further, Forth's extensibility has not been translated to hardware extensibility. *[You has better look at those WISC machines, Jose.—ed.]* Perhaps, when someone puts a Xilinx logic cell array; a writable, threaded, interpretive stack machine; 8K EEPROM; and an LCD with nano-keyboard on one tiny chip, and this *anything chip* leisurely chugs along at 33 MIPS, we will have an interactive, real-time engine.

Sincerely,
Jose Betancourt
85 Arlo Road #1A
Staten Island, New York 10301

Nachrichten

*** FORTH GESELLSCHAFT, JAHRESTREFFEN Am 14./15.Mai 1988 wird in München die Jahresversammlung der Forth Gesellschaft eV stattfinden. Anreise ab Freitag, den 13.Mai, 15:00. (Hoffentlich ist keiner abergläubisch!!) Tagungsort: Kolpinghaus, Hanebergstraße 8, 8000 München 19. Wer sich bisher nicht angemeldet hat, kann voraussichtich nicht mehr im Kolpinghaus untergebracht werden und sollte sich selbst um eine Unterkunft bemühen. Die Tagungsgebühr mit Verpflegung ohne Übernachtung beträgt 60,- DM. Anfragen bitte an Heinz Schnitter, Nelkenstr.52, 8044 Unterschleißheim richten.

*** JAHRESBEITRAG Es ist höchste Zeit, den Jahresbeitrag zu bezahlen. Immer noch gilt ein ermäßigter Beitrag von 32,-DM für Schüler, Studenten, und andere Mittellose, der ordentliche Beitrag von 64,-DM sowie der Förderbeitrag für Spendenwillige in Höhe von 128,-DM

*** FORTH KURSE geben. Besondere Nachfrage bzw scheint es in und um Aachen, Dortmund, Hamburg und München, aber auch Düsseldorf zu geben. Zwei Beispiele aus jüngster Zeit: Aus Aachen kommt Prof. Dr.-Ing. K.W. Pleßmann. Er leitete im Dezember '87 im VDI Bildungswerk in Düsseldorf das Seminar "FORTH in Automatisierungs-Systemen" und benuzte Fig-Forth. Oberstudiendirektor Albert und Studiendirektor Sengpiel, Gewerbliche Schulen I der Stadt Dortmund, benutzen in der Ausbildung Forth-83. Die Schüler arbeiten in der Schule mit einem kommerziellen Forth-83 System, zu Hause in der Vor- und Nachbereitung auf ihren privaten Rechnern (Commodore, Atari, Amiga, IBM-pc-clones) mit public domain Forth-83. Gute Forthlehrer werden gesucht.

*** FORTH SYMPOSIUM in Australien vom 19.-20. Mai '88. Eine Guppe professioneller Forth Benutzer aus Industrie und Akademischen Organisationen hat es organisiert. Charles Moore ist der Eröffnungsredner. Das Symposium fällt in die Zeit der 200 Jahrfeier Australiens mit der World Expo 88 (30.4-30.10.88). Ob Gruppenreisen von Europa aus dorthin angeboten werden, konnte ich nicht erfahren. Von USA aus kann man jedenfalls mit der FIG fliegen.

FORTH DIMENSIONS

Vol. IX / 4

FORTH DIMENSIONS

Vol. IX / 5

Wie die Forth Gesellschaft am Telefon besser erreichbar sein könnte

Die Forth Gesellschaft ist zu klein, als daß jemand bezahlt werden kann, der tagtäglich am Telefon erreichbar ist und Forth-Fragen beantworten könnte.

Deshalb hatten wir in der Vergangenheit die Regelung, daß an einem Tag in der Woche abends jemand im Forth Büro erreichbar war. Nach dem Umzug des Büros in den Antilopenstieg ist diese Möglichkeit entfallen.

Einen Ausweg sehe ich darin, daß sich möglichst viele Mitglieder bereit erklären, Fragen zu Forth am Telefon zu beantworten. Das ist für Mitglieder einfach, da in der Mitgliederliste meistens eine Telefonnummer angegeben ist.

Wenn sich aber regional noch Ansprechpartner zur Verfügung stellen, deren Telefonnummer in der Vierten Dimension veröffentlicht wird, dann können Anfragen, die schriftlich bei der Forth Gesellschaft ankommen und sich nicht vom Büro beantworten lassen, auf den nächstgelegenen "Forthhelfer" und dessen Telefonnummer verwiesen werden.

Bei einer regionalen Begrenzung wird auch niemand übermäßig belastet, wie das der Fall wäre, wenn es nur einen zentralen Ansprechpartner gibt. Die Anfragen sind auch meistens so, daß dazu kein Expertenwissen notwendig ist und von jedem beantwortet werden können, der sich jemals ernsthaft mit Forth beschäftigt hat.

Wer weiß, vielleicht entwickelt sich dann daraus ein erstes privates Treffen und letztendlich eine regionale Gruppe mit regelmäßigem Treff?

Jeder, der dazu bereit ist, schicke bitte eine Postkarte mit Telefonnummer an das Forth-Büro, Antilopenstieg 6a, 2000 Hamburg 54.

Einladung zur ordentlichen Mitgliederversammlung der Forth Gesellschaft eV

am Sonntag, den 15. Mai 1988 ab 14:00 im Kolping-Haus, Hanebergstr. 8, 8000 München 19

Ab 13. Mai findet das jährliche Forthtreffen statt. Dieses Jahr haben einige Firmen angekündigt, ihre Forthprozessoren zu präsentieren. Weitere Vorträge aus dem Wissenschafts- und Hobbyistenbereich sind zugesagt. Ein ausführliches Programm kann schriftlich bei der Forth Gesellschaft eV, Antilopenstieg 6a, 2000 Hamburg 54 angefordert werden und ist voraussichtlich ab Mitte April verfügbar.

Tagesordnung:

1.) Rechenschaftsbericht des Direktoriums

2.) Kassenbericht und Jahresbilanz 1986 und 1987

3.) Aussprache und Entlastung des Direktoriums

4.) Wahl des Direktoriums

5.) Berichte aus den lokalen - und Fachgruppen

6.) Verschiedenes

FORTH Processor Core for Integrated 16-Bit Systems

Peter S. Danile and Christopher W. Malinowski, Harris Corp. Semiconductor Division, Melbourne, FL

The development of a high-performance dedicated 16-bit processor using an industry-standard microcontroller or bit-slice processor calls not only for an extensive board-level design effort, but also for a long-term development program for software and firmware. It has been difficult to use semicustom techniques for such development because core processors have been scarce and microcode development for custom ICs is very difficult.

However, in a growing number of high-performance systems for digital signal processing, control, and arithmetic, application-specific processors are bringing forth the advantages of integration—including high throughput, low power dissipation, and much higher density than board-level implementations.

Harris Semiconductor now has a semicustom technology, called the Processor Toolbox, that eases the development of high-performance 16-bit integrated processors. Toolbox features include a very small core-processor cell, a highly parallel architecture for maximum throughput, easy programming and code development, code portability, a full set of core-compatible peripheral cells that can support processor clock frequencies as high as 15 MHz, and a set of high-speed arithmetic and logic cells, such as a 16-bit multiplier, for further customization.

The processor architecture derives from one conceived by Charles Moore, inventor of the FORTH language. This RISC-like, highly parallel architecture meets the size and throughput requirements. The combination of the processor's instruction set—a directly executable set of FORTH high-level primitives—and its reliance on two stacks that reflect a FORTH virtual machine results in a compact core processor with less than 2500 gates. This core is called the FORTH Optimized RISC Computing Engine, or Force.

Because each instruction comprises more than one FORTH primitive (opcode), data-manipulation throughput can exceed the processor's clock frequency, often by a factor of three. Consequently, for instruction sets rich in multiple-opcode instructions, peak processing throughput can exceed 30 MIPS, with a steady throughput of 10 to 20 MIPS.

Users of the Toolbox can develop application code in a high-level FORTH language working with an interactive and interpretive environment. FORTH is not only portable but also offers expeditious debugging tools and easy target-compilation from one environment to another. Therefore, a wealth of code written for DSP, artificial intelligence, control, number-crunching, and real-time data-processing applications can be ported to the Harris engine.

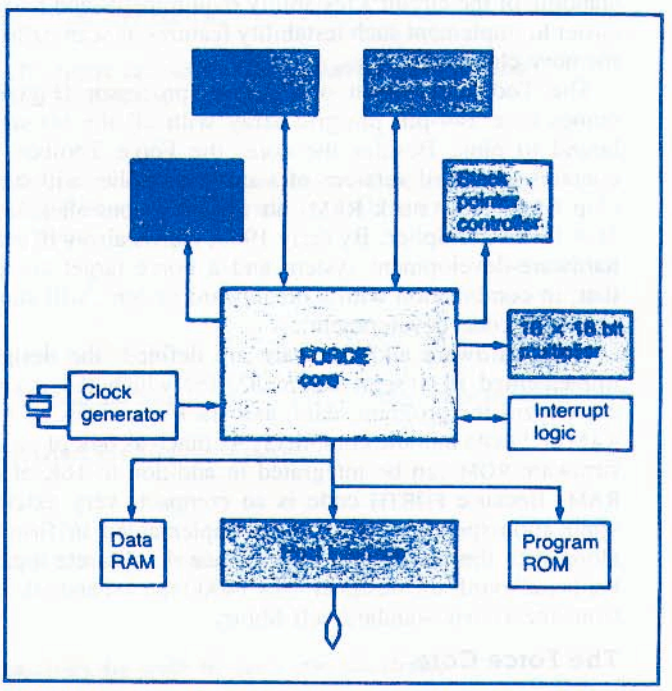


FIGURE 1. The Force Toolbox contains a FORTH processor and support peripherals, all available as cells and packaged parts.

The principal advantage to designers of dedicated processors stems from the host of proprietary LSI and VLSI cells being developed to support the FORTH core processor. The Toolbox provides designers of Force-based products with a set of packaged Force circuits identical to the cells available in Harris' standard-cell library (Figure 1). The designer can use these parts to create a breadboard and experiment with different configurations of the Force processor before moving to an ASIC. He gains greater confidence in the design's functionality, because it can be exercised in a real environment in addition to a CAE simulation environment. Moreover, the designer can use the prototype breadboard to generate a reliable and accurate set of functional test vectors, a task both formidable and error-prone otherwise.

Another advantage of creating a breadboard is that the designer can compile the applications code and run it before committing it to a ROM pattern. Running the application code lets the designer make trade-offs between implementing func-

Reprinted from VLSI Systems Design June 1987, Copyright 1987 C.M.P. Publications Inc.

tions in hardware and coding them in software. The core's execution speed allows many functions—such as memory swaps, arithmetic and logic functions, shifts, and masking—to be implemented in firmware without sacrificing performance, shrinking die size considerably. Such trade-offs can be investigated only if the designer has access to the processor's functional blocks and can exercise alternatives in real time—that is, on a breadboard.

Finally, the Toolbox approach makes it easier to design a testable circuit. During breadboarding, the designer can observe the timing of signal paths, such as the processor's data paths, which may not be directly accessible in an integrated system. Identifying buried trouble spots increases the understanding of the circuit's testability requirements and makes it easier to implement such testability features as scan paths and memory-check routines.

The Toolbox version of the core processor (Figure 2) comes in a 144-pin pin-grid array with all the I/O signals bound to pins. Besides the core, the Force Toolbox also contains packaged versions of a stack controller with an on-chip 64 × 16-bit stack RAM, an interrupt controller, and a 16 × 16-bit multiplier. By early 1988, Harris also will offer a hardware-development system and a Force target compiler that, in combination with a breadboard system, will support firmware-code development.

Once hardware and firmware are defined, the design is implemented in a semicustom IC, for which the designer customizes the program, data, and stack memories by using RAM and ROM module compilers. As much as 64K of on-chip firmware ROM can be integrated in addition to 16K of data RAM. Because FORTH code is so compact, very extensive application-specific code can be implemented in firmware along with the kernel code. To replace the discrete logic on the breadboard, the designer uses 7400-type SSI and MSI cells from the Harris standard-cell library.

The Force Core

The Toolbox's Force core processor is a bare control engine with 123 I/O lines. These signals include three parallel 16-bit data buses (two for stack memories and one for main memory), a 16-bit main-memory address bus, and a dual-purpose 5-bit address-extension bus. In addition, a general-purpose 16-bit bus (G-bus) acts as the processor's primary I/O signal path.

A principle of RISC philosophy is to bring execution speed as close as possible to the maximum memory-access speed. As a corollary, the number of multicycle instructions in the processor's instruction set is reduced to maximize the processor's data-bus throughput and bandwidth.

The processor's independent buses account for the Force core's high throughput. Because each instruction executes in no more than two clock cycles, at least three of the five buses are active during any clock cycle. The G-bus transfers data at up to 30 MB/s when the processor operates at 15 MHz; the main-memory bus transfers data at up to 30 MB/s when in a streamed-move mode.

The Force core processor's highly parallel architecture (Figure 3) reflects the structure of its horizontal instructions. Its eight main registers provide parallel storage and access to the parameter stack's top two locations (TOP and NEXT), the top location of the return stack (I), the instruction register

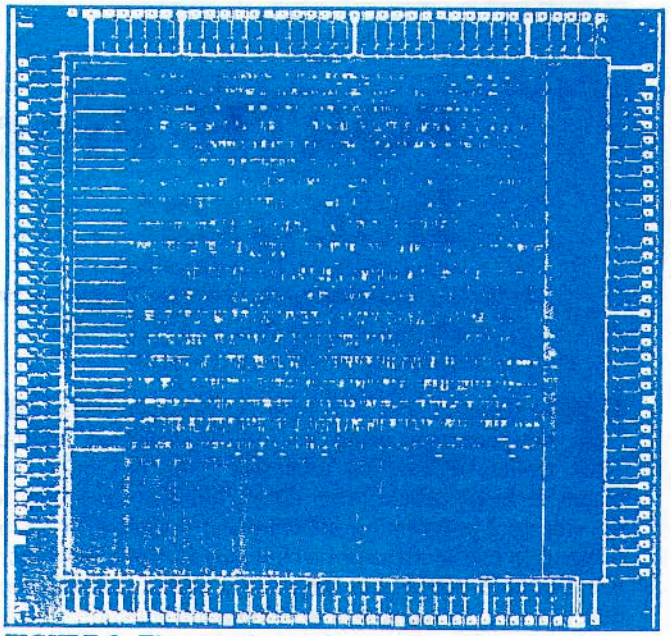


FIGURE 2. The packaged version of the Force core processor.

(IR), the program counter (PC), and two arithmetic-instruction registers (MD and SR). The MD register stores the partial results of step-multiply and step-divide instructions; the SR register stores the partial results of hardware-assisted square-root operations.

The minimal overhead to support subroutines also reflects the nature of the FORTH language, which is heavily oriented toward the use of subroutines. The core processor is optimized for the minimum number of cycles necessary to execute a subroutine call and return. Through instruction partitioning and architectural refinement, the execution of a subroutine call requires only a single clock cycle; the return requires no added clock cycles. All interrupts that are interpreted as subroutine calls therefore require only one clock cycle of overhead.

Stack Controller Cell

Because the stack-oriented FORTH instructions employ stacks in every command, the most critical peripheral used with the Force core processor is the stack controller. Controlling the stacks with software routines would degrade the system's performance. The core directs the stack controller through the RW and SA signals. The stack controller responds to the SA signal with either a push (data write) or pop (data read), according to the status of the RW signal.

In the packaged version of the stack controller are 64 words of memory with an access time of approximately 30 ns, so the designer can operate the core at 15 MHz without worrying about the access time of data in the stack. In an ASIC implementation, the stack's memory size can be altered and the access time improves to about 20 ns.

The stack controller operates with the core processor's parameter stack (through SAS and RWS signals) and the return stack (SAR and RWR), giving a typical system two stack controllers (Figure 4). To enhance system flexibility, the signals OVER and UNDER generate interrupts when the stacks are ready to overflow or underflow. UNDER occurs when the

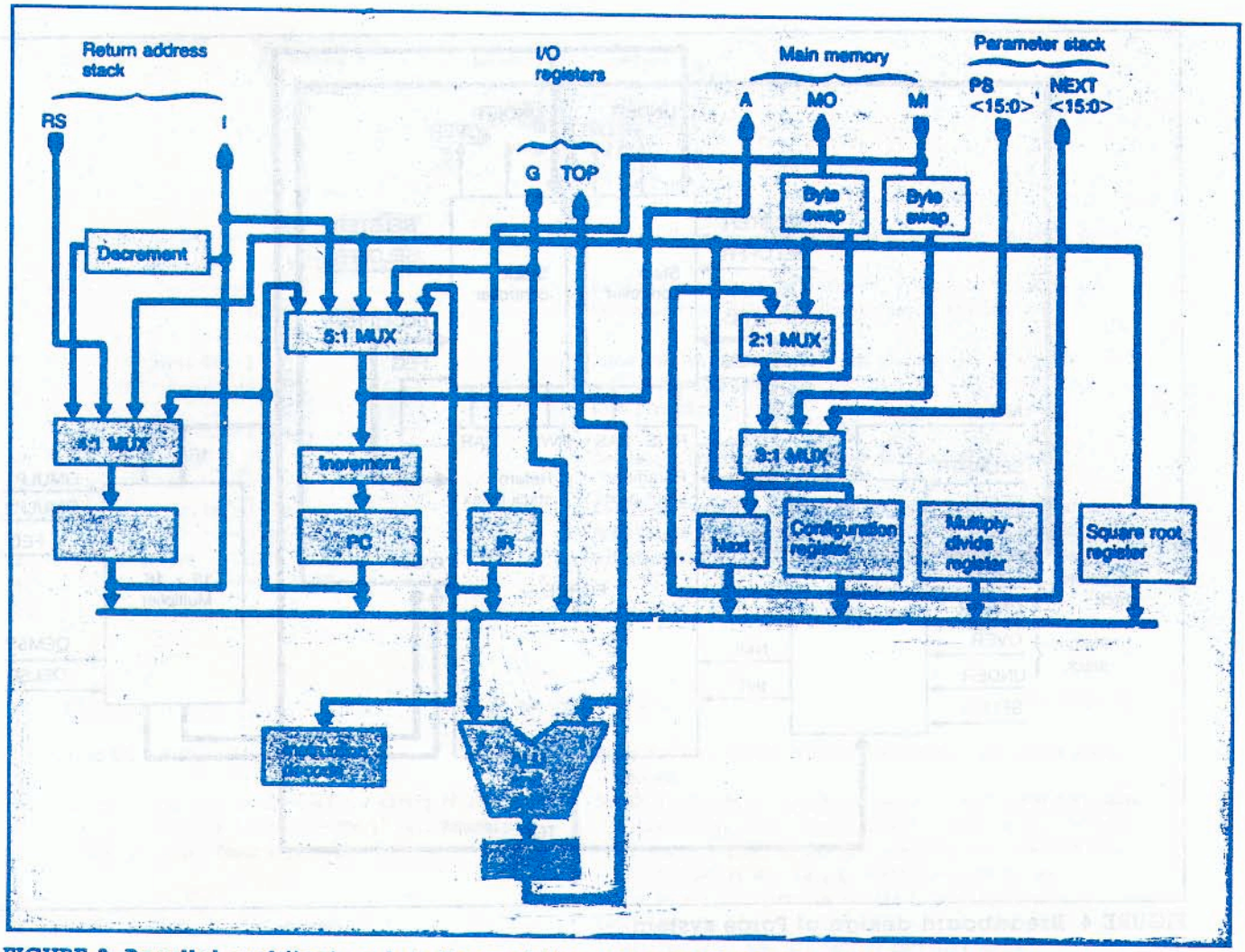


FIGURE 3. Parallel architecture lets three of five processor buses be active for all instructions.

stack is pushed more than popped and the stack address lines reach 00. The assertion should initiate a routine that either resets the controller (if more returns than routines were called) or tries to recover from other sources of underflow. The OVER signal occurs when the number of words pushed onto the stack exceeds a user-defined maximum. This maximum can be programmed into the ASIC implementation by writing into an offset register through the G-bus.

To implement multitasking versions of the Force processor, Harris is developing a multitasking stack controller (MSC). The MSC enables the user to partition the 256-word stack into eight separate stacks via a 3-bit address size register. For example, if the system must run two concurrent jobs, the size register is programmed to divide the stack RAM into two 128-word stacks. To switch between tasks, the processor enables a task-select register through the G-bus. When this register is written to, the stack pointer of the current task is saved and the stack pointer for the new task is restored at the new task's current address.

When the stack controllers and core processor are integrated on a single chip, options for increasing performance are available. Not only can the access times of the controller and the data RAMs be reduced merely by integrating, but access time can also be reduced further by separating the bidirectional data buses into read (POP) and write (PUSH) buses (Figure 5). In the discrete versions, the data buses are bidirectional to allow them to connect directly to standard RAMs with bidirectional data buses. Separating the buses adds some additional routing area; on the other hand, the time required to set up the buses for either a read or a write is eliminated, improving system response time substantially.

Interrupt Controller and Host Interface

The interrupt controller and the host interface help the core processor interact efficiently with the surrounding system. First, the interrupt controller contains 15 prioritized interrupt-request inputs and a separate input for nonmaskable interrupts (NMI). The interrupt priorities are fixed (to decrease response time) but can be defeated by writing in the interrupt controller's mask register (which has a discrete address on the G-bus). The interrupt controller samples the request inputs on opposite edges of the system clock. When two consecutive samples confirm that an interrupt is present, the INT line is asserted. The core processor responds with the INTA signal, which directs the interrupt controller to generate the appropriate vector for the interrupt. This vector comprises a 7-bit user-defined field for the location of the interrupt-vector table and a 5-bit field that designates the appropriate interrupt.

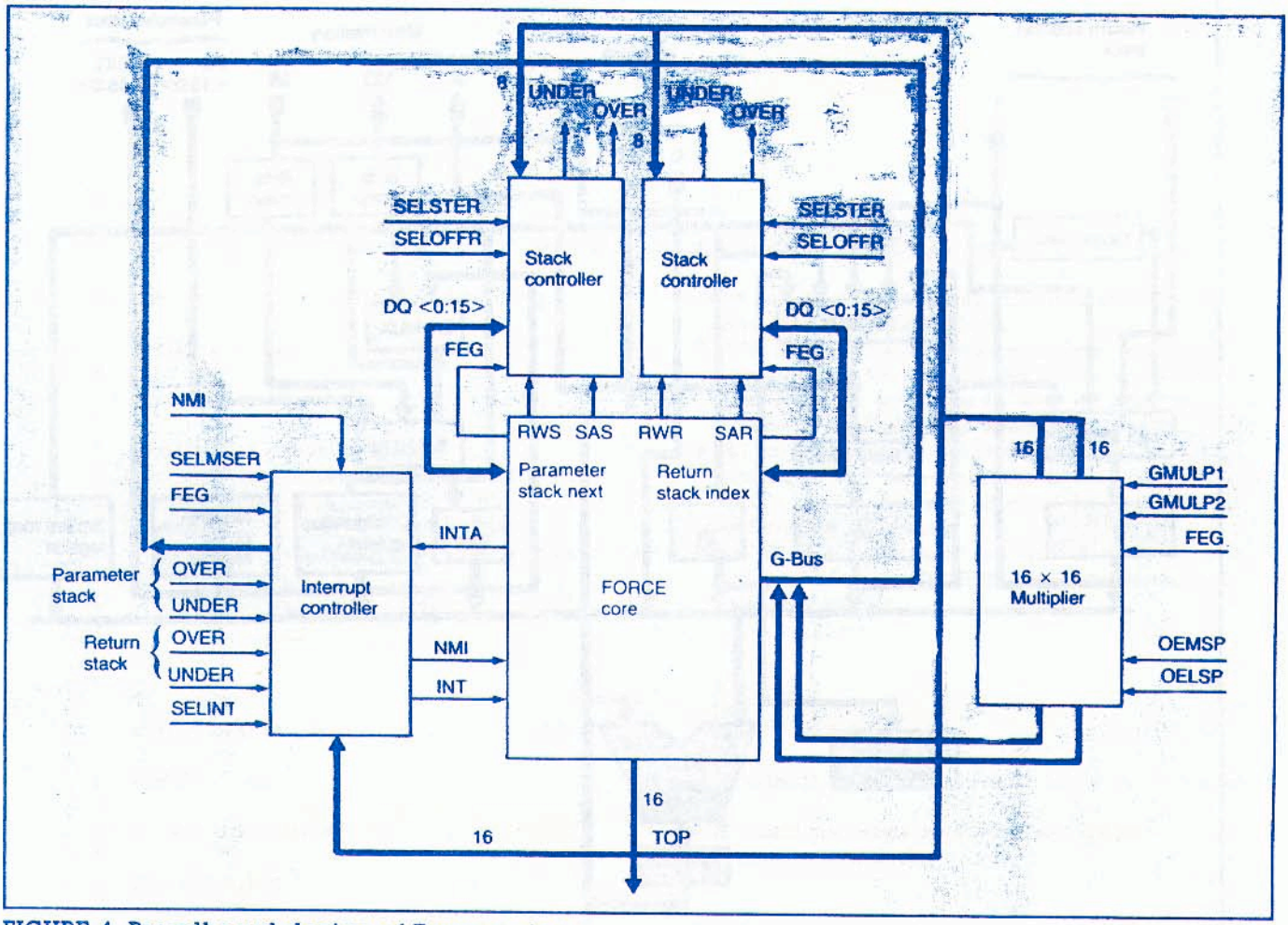


FIGURE 4. Breadboard design of Force system.

In the packaged version of the interrupt controller, the interrupt vector for the valid highest-priority interrupt is presented to the core processor within 40 ns of the INTA pulse. Thus the controller can operate with the core processor at system frequencies greater than 15 MHz. Once the system has entered an interrupt routine, the core's INTE signal inhibits any new interrupt from generating a new INTA. During INTE assertion, the system can clear the interrupt source and rewrite the Force configuration register to re-enable the interrupts. The interrupt controller does not automatically nest interrupts, so the system does not need to modify the interrupt controller until it must mask or unmask any interrupt line.

To create an interface between the Force core and a host controller that does not degrade the core's performance, the Toolbox includes a host interface cell. The interface allows another processor to read and write data in the Force core's memory-address space; it receives as inputs the address and data lines of the shared memory, the command lines from the processors, the core's data signals, and a MEMORY_READY signal that indicates when the data in the shared memory is valid. It provides a READY signal to the host, to indicate when it can read or write data, and a clock signal to the core. Figure 6 shows a typical configuration of the interface, the core processor, and the shared memory.

The host interface suspends the core processor when it attempts to read invalid data. When the data is not in the shared memory, the MEMORY_READY input to the host interface is de-asserted until the data in the memory becomes valid. While the signal is low, the interface suspends execution by the core processor; the processor continues when MEMORY_READY is re-asserted.

When the host processor wants to write to the shared memory, the host interface checks the FORCE_LOCK signal to determine if the Force processor has priority on the memory bus. If not, the interface suspends the core processor and hands control over to the host. If wait states are necessary, MEMORY_READY becomes low and the host interface stalls the host with the HOST_READY signal. When HOST_READY is asserted, the host can relinquish priority on the bus.

Writing host-processor data into the shared memory is simpler than reading from it. The host writes directly to the host interface, which holds the data in a buffer. When the memory bus becomes free, the interface writes the data into the memory. A HOST_READY signal is set when the buffer is full to prevent the host from writing over new data.

If the host processor wants to do some house-cleaning in the shared memory, it requests priority on the memory bus by asserting the HOST_LOCK pin. This signal suspends the core processor so the host can have exclusive access to the memory. In normal operation, the use of LOCK signals should be minimized so performance is not degraded.

The host interface is designed to work with an asynchro-

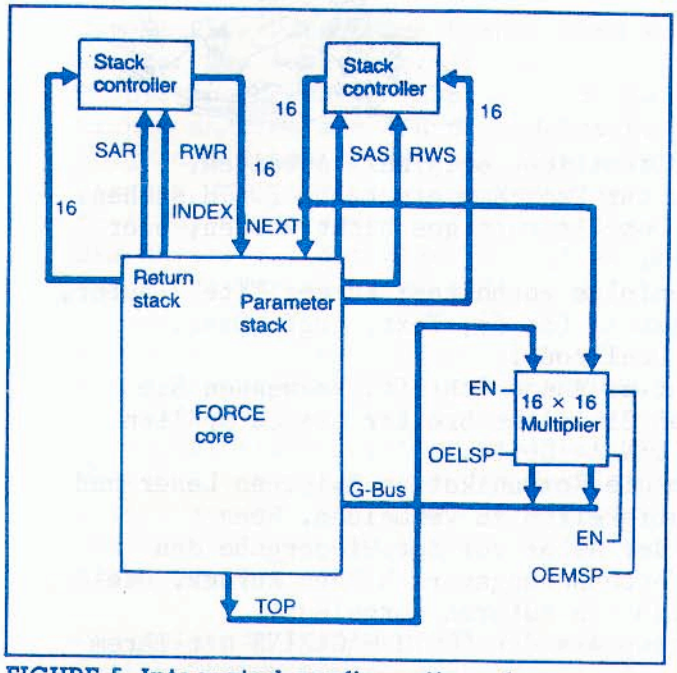


FIGURE 5. Integrated configuration of core, stack controller, and multiplier.

nous host by synchronizing all commands from the host with the Force clock signal. The core processor can work with synchronous or asynchronous RAMs, even if the host interface is used in a completely synchronous system. If the interface is integrated with the core, minor modifications can improve response time in the host read/write cycle. Also, if the shared memory is integrated with the other components, the MEMORY_READY line is not necessary because the on-chip compiled RAM is fast enough to always contain valid data.

Multiplier

Harris has designed a 16×16-bit multiplier, using its proprietary MPS algorithm, for use with the Force core processor. The peripheral performs full 16×16 multiplication, generating a 32-bit product in as little as two clock cycles when the peripheral is integrated with the core; a breadboard system can complete a multiply in five clock cycles.

The multiplier operates either clocked or unclocked and provides tristate signals at the output control and data (MSP and LSP) lines. It has data latches at its inputs to allow clocked operation independent of the core. On-chip results are generated in less than 50 ns from the edge of the clocking signal, and the two 16-bit words in the product can be read simultaneously or in sequence.

The designer can incorporate the multiplier into the Force architecture in several ways. First, he could designate several G-bus addresses to identify the multiplier, multiplicand, and the product's most-significant word (MSP) and least-significant word (LSP). Using this configuration, the core processor would write the addresses and read back the results to receive the result in four clock cycles.

He also could designate one G-bus address to identify either the multiplier or the multiplicand. The second operand can attach directly to the processor's TOP bus. To perform a multiplication, the multiplier would be written to the G-bus and the multiplicand placed on the TOP bus. Upon comple-

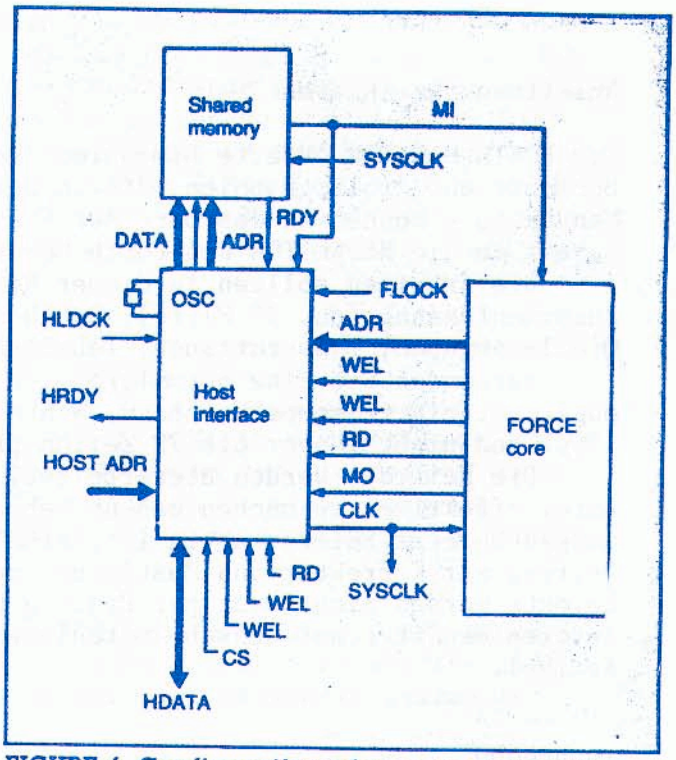


FIGURE 6. Configuration of core and shared memory with host interface.

tion, the processor executes a FETCH_SWAP from the G-bus to receive either the MSP or LSP of the result (depending on the multiplier's configuration), placing it in the NEXT register. A G-bus FETCH then retrieves the remaining product word. This operation requires only three clock cycles to multiply two 16-bit numbers, and when many numbers need to be scaled by a constant (the multiplicand), the multiply takes only two clock cycles once the initial multiplier is written to the G-bus.

When the multiplier is integrated with the Force processor, the multiplier and multiplicand can be placed directly on the TOP and NEXT buses. The multiplier would be configured in its feedthrough mode, and a 32-bit product would be available every clock cycle. To execute a multiplication, the processor needs to read only the appropriate G-bus addresses. □

About the Authors

Peter S. Danile is currently a section head of semicustom design at Harris Semiconductor. Prior to joining Harris in 1981, he worked for both Northern Telecom and Motorola Communications. A graduate of the University of South Florida in 1976, Peter went on to receive his MSE with honors from Florida Atlantic University in 1980.

Christopher W. Malinowski is a senior scientist for Harris' semiconductor research and development department, and a program manager for the Force project. He holds an MS degree in nuclear electronics and a PhD in solid-state physics from Warsaw Technical University.

Anleitung für Autoren

Das FORTH MAGAZIN 'Vierte Dimension' veröffentlicht originale Arbeiten, Berichte und Bibliographien, die in Bezug zur Programmiersprache FORTH stehen. Manuskripte können an das Büro der Forth Gesellschaft geschickt werden, oder direkt an die REDAKTION des Forth Magazins.

Die Arbeiten sollten in dieser Reihenfolge enthalten: Kurzer Titel, Autor, Zusammenfassung (ca. 50 Worte), Schlüsselworte (ca 5), Text, gegf. Dank, Quellenangaben, Ilustrationen, Tabellen, Quellcode.

Verwenden Sie eine normalgroße, nicht zu dünne Schrift. Verwenden Sie möglichst ein frisches Farbband. Schreiben Sie nicht breiter als 80 Spalten 12cpi und nicht länger als 72 Zeilen pro DIN A4 Seite.

Die Beiträge werden überarbeitet, um die Kommunikation zwischen Leser und Autor effektiver zu machen und um Mehrdeutigkeiten zu vermeiden. Wenn ausgedehnteres Edieren nötig ist, erhält der Autor vor der Wiedergabe den Beitrag zur Korrektur und Zustimmung zu Verbesserungsvorschlägen zurück. Die Layouts werden nicht mehr zur Prüfung durch die Autoren vorgelegt. Autoren erhalten auf Wunsch kostenlose Exemplare des FORTH MAGAZINS mit ihrem Artikel.

IMPRESSUM

Titel: FORTH MAGAZIN 'Vierte Dimension'.
Zeitschrift der Mitglieder der Forth Gesellschaft eV.
Herausgeber: Forth Gesellschaft eV.
Forth: Klaus Schleisiek und die Mitglieder des Review-Boards sowie alle namentlich genannten Autoren.
Redaktion: Michael Kalus, Tel: 0202-736591, Dasnöckel 92, 5600 Wuppertal 11
Erscheinungsweise: Ein Heft je Quartal.
Redaktionsschluß: Der mittlere Quartalsmonat.
Auflage: 500 Stück europaweit.
Druck: NN

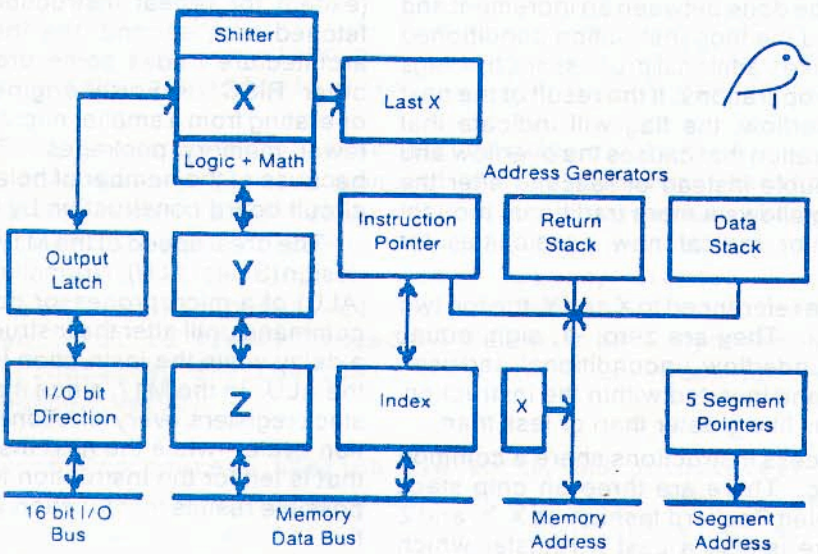
Kontaktaufnahme bitte über das Forth Büro:
Forth Gesellschaft eV, Antilopenstieg 6a, D 2000 Hamburg 54
Bankverbindung: Postgiroamt Hamburg, Kto: 563211-208, BLZ 20010020

Minimum Instruction Set Computer, Inc., announces the

M17 MICROPROCESSOR

The NEXT STEP in FORTH ENGINES

- floating point normalization step
- multiply and divide steps
- byte shift and byte store
- ROTATE on chip
- ROLL on chip
- 10 conditionals with OR and NOT
- 16 logic operations with shifts
- only two memory packages required
- auto sync to any peripheral speed
- repeating instructions
- 14 word length registers
- three layer stack
- last X register (like HP)
- 24 bit addressing
- call over a whole 64k segment
- bit programmable 16 bit I/O bus
- any memory speed—25ns to 450ns
- mix memory speed on same board



The M17 from MISC is the latest advance in performance and cost reduction for machine control and dedicated systems. Stripped of the complex architecture demanded by the desk top computer market, the M17 is optimized for the operations that take 80-90% of processor time.

The M17 is very cost effective, unlike previous Forth Engines or RISC processors. The chip itself is less than half their price. The package does not require a multilayered board or lots of expensive memory or elaborate chip sets. Just two 120ns static RAMs will perform at over 1 MIPS.

Because of delays beyond MISC's control, we are unable to bring samples to the FIG convention. The delay may work to the customer's advantage since it may allow us to include an oscillator pin on the M17 and eliminate an external clock chip (the oscillator pad was released to us on November 3, 1987). The design is completed and fully tested in simulation (except for the oscillator pad). MISC has enough money on deposit to make the M17 and is committed to full production in the first quarter of 1988.

MISC will support you at whatever level YOU choose. Because a nominal charge is made for support, MISC personnel are kept trained and numerous for your needs. Industry standard warranty, documentation, and initial phone support are included with every purchase.

The M17 has three software options: a simple, elegant assembler (now); Forth 83 modified for 16 bits (early 1988); and a C package (early 1988).

The M17 will be available during the first half of 1988 as a minimum system with a serial link to a desk top computer, a drop-in card for your clone, a complete development system, or as unsupported chips in sample or large quantities.

Minimum Instruction Set Computer, Inc. • 19704 East Loyola Circle • Aurora, Colorado 80013-3904 • 303-680-9749

The M17 microprocessor has very few instructions but a rich set of options for each instruction. The instructions are call, return, jump, setflag, access, and process.

The call instruction needs no special code. Merely place any even address in the next instruction and that address will be called anywhere in the current program segment—all 64k words of it! Only one address is wasted every other subroutine, or less than 1k words per 64k segment of a Forth dictionary.

The return instruction is conditional to any M17 flag. The return address can be dropped or kept, allowing a very fast conditional loop or a traditional return from subroutine or interrupt.

Jump uses the address in the next program counter location and is conditional to any M17 flag. Both jump and return can branch to an even or odd address, allowing all code except the first instruction of a subroutine to be in any address of a 64k segment.

Setflag is unique to the M17. While any M17 flag can be used immediately by the conditional instructions, setflag allows a previous flag condition to be frozen into the user flag, which can be the condition of a later instruction. For example, arithmetic can be done between an increment and test on the data stack and the loop instruction conditioned on that previous test. Unlike traditional processors, the flags of the M17 anticipate the operations. If the result of the next operation will be an overflow, the flag will indicate that condition before the operation that causes the overflow and a test can anticipate trouble instead of reacting after the damage is done. Setflag allows a more traditional thought process (latched flags) or radical new possibilities for control structures.

The flags available are referenced to X and Y, the top two words on the data stack. They are zero, -1, sign, equal, carry, overflow, borrow, underflow, unconditional, and user. The flags can be ORed and inverted within the instruction, allowing many conditions like greater than or less than.

Both access and process instructions share a common stack manipulation logic. There are three on chip stack registers, named in Hewlett Packard fashion as X, Y, and Z with X at the top. There is also a Last X register which contains the word in X prior to the last process instruction. Z can be updated with Z (no change), shifted Z, memory data bus, X, or Y. Y can be updated with Y (no change), Z, X (input to the SALU, for example: DUP INCREMENT in one step), or next X (output of the SALU, for example: INCREMENT DUP in one step). The stack address can be updated or frozen. X can be updated or frozen.

The access instruction moves data between X and internal or external locations. The internal locations are return data, return stack pointer, return stack pointer extension, program counter, program counter extension, data stack Z register, last X register, data stack pointer, data stack pointer extension, A and B buffer access extensions, context register (contains shift control and interrupt enable flags), and the repeat instruction counter (write only). Input/output locations (all 16 bit) are input data bus (read only), output data latch, and the bit direction latch. Input instructions generate a read signal on one pin and output instructions generate a separate write signal on a different pin. Memory access instructions include fetch and store to the next program counter (literal @ and !), fetch and store to and from X address with optional address extensions to the data stack segment, A buffer segment or B buffer segment. All memory accesses generate separate memory request and read/write signals and allow the whole word or the upper or lower byte to be stored. Most access instructions can swap data between X and the named location.

The process instructions are pass X, sum, invert X, or, pass Y, minus, and, exclusive or, multiply step, divide step, increment, decrement, two's compliment of X, clear X, clear sign bit (absolute), propagate sign bit, and floating point normalization step. In addition the results of any process instruction can be shifted one bit right, one bit left, or 8 bits swaped (byte swap) within the same instruction. The input for the bit abandonded by a shift can be 0, 1, Z, the sign bit or the least significant bit.

The repeat instruction counter will repeat the next instruction until it has decremented itself to zero. This allows streaming of process or access instructions without refetching the instruction.

Some additional features of the processor are separate reset and interrupt pins; a clocking system that can be crystal controlled or asynchronous; a single, low cost memory interface; and the unique SALU design.

The M17 uses two clock cycles for every instruction (except for repeat instructions). First, an instruction is fetched and, second, the instruction is executed. This architecture trades some processor speed compared to other RISC or Forth engines for the cost savings of operating from a smaller microprocessor package and from fewer memory packages. These saving are multiplied because of the number of holes and layers saved in printed circuit board construction by the M17.

The great speed of the M17 comes from its unique SALU design (Select ALU). Normally the Arithmetic and Logic Unit (ALU) of a microprocessor does not receive the data and command until after the instruction is latched. There is then a delay while the instruction is decoded and performed by the ALU. In the M17, since it always operates on the same stack registers, every function is precalculated every instruction cycle—while the next instruction is being fetched! All that is left for the instruction to do is to select out of all the possible results that function which the programmer specifies.

The M17 is packaged in an 84 pin plastic leadless chip carrier, the latest in cost effective package design. The 84 pin PLCC can be surface mounted or used in an inexpensive socket—without driving a car over it to seat it or cracking it to remove it. The process is a 2 micron HCMOS gate array. The pins are 18 power and ground, 24 memory address pins, 16 bidirectional memory data pins, 16 programmable input/output data pins, reset, interrupt, clock (may be changed to use the new oscillator pad), and 7 status and control output pins: write i/o, read i/o, read/write lo byte memory, read/write hi byte memory, memory request, interrupt acknowledge, and instruction fetch/execute phase.

Minimum Instruction Set Computer, Inc., will introduce a full product line over the space of a year. Chips, boards, systems, software, services, licenses, and other MISC processors will be included. The first set of products will be chips at $75 each in sample quantities, a single board computer with a serial link to a PC for $750, an assembler that runs under MSDOS at no charge with a chip purchase, a Forth 83 system modified for the 16 bit environment at $150, and a small C system for $150. A special preproduction discount of 50% and full refund policy is offered for all of these products.

TURBO-FORTH made in France

Marc Petremann

(Gekürzt übersetzt aus JEDI Heft 40, Dezember '87 von M.Kalus und versehen mit einer Einleitung.)

Seit gut zwei Jahren gibt es JEDI in Paris - wo auch sonst in Frankreich? Die Zeitung "Jedi" wird von sehr engagierten Leuten gemacht. Das Blatt befasst sich mit neuen Programmiersprachen. Seit einiger Zeit nimmt darin Forth - besonders das Forth-83 Modell von Laxen und Perry, das F83 - einen zunehmend breiteren Raum ein.
Der Name Marc Petremann tauchte immer wieder auf. Inzwischen hat er, zusammen mit Jean-Marie Premesnil und Michel Zupan, ein Buch herausgebracht: "Forth 83-Standard, pour tous Systems CP/M et MSDOS". Es enthält das Vokabular des F83 alpabetisch geordnet und ins Französiche übersetzt mit Kommentaren zum System selbst sowie den Betriebssystem-Unterschieden. Im Dezember '87 erschien nun Jedi Heft 40 (!) mit dem Aufmacher "Turbo-Forth". Dabei die Disketten und freundliche Grüße von M.Petremann.

Was ist das?!! Es ist eine gründlich überarbeitete Fassung des F83. Dabei wurde Wert auf eine gute Anpassung an das MSDOS gelegt und der Kern um 2K gestrafft! Verschwunden ist das Block-Konzept der alten Forth-Tage. Turbo-Forth kann andere Programme aufrufen und arbeitet daher mit dem guten alten Wordstar als Editor. Mit EDIT kommt man direkt ins Textfile, bearbeitet mit Wordstar seine Forthquellen mit allem Komfort einer richtigen Textverarbeitung und kehrt danach wieder ins Forth zurück als sei nichts, gewesen. Gut gelöst! Die verschachtelbare Funktion INCLUDE (filename) ist natürlich dabei. Interessante Befehle auch bei der String-Verarbeitung. Ein Beispiel:

```
255 STRING A$
" 110.30 125.05 45.00 17.55 " A$ $!     " 44.35 135.70 " A$ APPEND$
A$ 1 ITEM TYPE   liefert 110.30

Umwandlung in Zahlen:   A$ 3 ITEM $EXECUTE D.   liefert 4500
Aber auch so etwas geht:   " DARK WORDS BYE " A$ $!
A$ 1 ITEM $EXECUTE    A$ 2 ITEM $EXECUTE   ( etc.)
```

Das Turbo-Forth wird in zwei Ausführungen angeboten:
- als Probier-Diskette.
Sie enthält das TURBO.COM file ungekürzt, unlimitiert, ungeschützt, voll funktionsfähig, frei verwendbar und kostet nur einige Briefmarken. (Es gibt sie bereits in Wuppertal: M.Kalus 0202-736591 ...aber jetzt bitte nicht alle gleichzeitig hier anrufen, woll?)

- als Version für professionelle Anwender.
Sie soll dann an komprimierten Quellen enthalten: Meta-Compilers, Turbo-Forth Quellcode, File-Editor, Erklärung aller Worte und Beispiele für den Gebrauch. Erhältlich voraussichtlich März/April 1988. Dabei werden folgende Extensionen sein:

  - FORTRAN Bibliotheken ausführen
  - PASCAL Bibliotheken ausführen
  - dBASE III/II+ Dateizugriff
  - MULTIPLAN Funktionszgriff
  - HERCULES Graphik Unterstützung
  - MATH.8087 Coprocessor Unterstützung

Ein Vertriebsweg ist noch nicht bekannt. Wer sich schon jetzt um seine Profi-Version kümmern möchte, wende sich bitte an Marc selbst:

M. Petremann, 17 allee de la Noiseraie, F-93160 Noisy le Grand
oder über
Association Jedi, 17 rue de la Lancette, 75012 Paris,
Tel: (1) 43.40.96.53 oder (1) 46.56.33.67

Herr Petremann spricht übrigens ausgezeichnet deutsch.

2 interessante Screens vom FIG-83

```
 Scr 12 Dr 0
 0 \ kill
 1
 2 : kill  ( name ( -- )
 3  singletask
 4  ['] 1541r/w Is r/w
 5  ['] noop Is 'cold
 6  ['] noop Is 'restart
 7  ['] (quit   Is 'quit
 8  ' >name 4 - (forget  save  ;
 9
10 \\
11
12 Kann benutzt werden um geschuetzte
13 Worte zu vergessen.
14 Damit kann man eine Applikation wieder
15 auf den Kern zurueck stutzen, auch
16 nachdem sie schon geSAVEed wurde.
17 Denn FORGET weigert sich bekannter
18 Weise so etwas zu tun. Mit KILL  und
19 BUFFERS ist man in der Lage, vom Kern
20 an aufwaerts ein eigenes System zusammen
21 zu stellen.
22
23
24
```

```
Scr 13 Dr 0
 0 \ relocating the system        20oct87re
 1
 2 : relocate-tasks  ( newUP -- )
 3  up@ dup
 4  BEGIN  1+ under @ 2dup -
 5  WHILE  rot drop  REPEAT  2drop ! ;
 6
 7 : relocate  ( stacklen rstacklen -- )
 8  swap  origin +
 9  2dup + b/buf + 2+  limit u>
10   abort" buffers?"
11  dup   pad  $100 +  u< abort" stack?"
12  over  udp @ $40 +  u< abort" rstack?"
13  flush empty
14  under  +   origin $A + !        \ r0
15  dup relocate-tasks
16  up@ 1+ @   origin   1+ !        \ task
17       6 -  origin  8 + ! cold ; \ s0
18
19 : bytes.more  ( n -- )
20  up@ origin -  +  r0 @ up@ - relocate ;
21
22 : buffers  ( +n -- )
23  b/buf * 2+  limit r0 @ -
24  swap - bytes.more ;
```

Die Reparatur des Interrupt im NC 4000

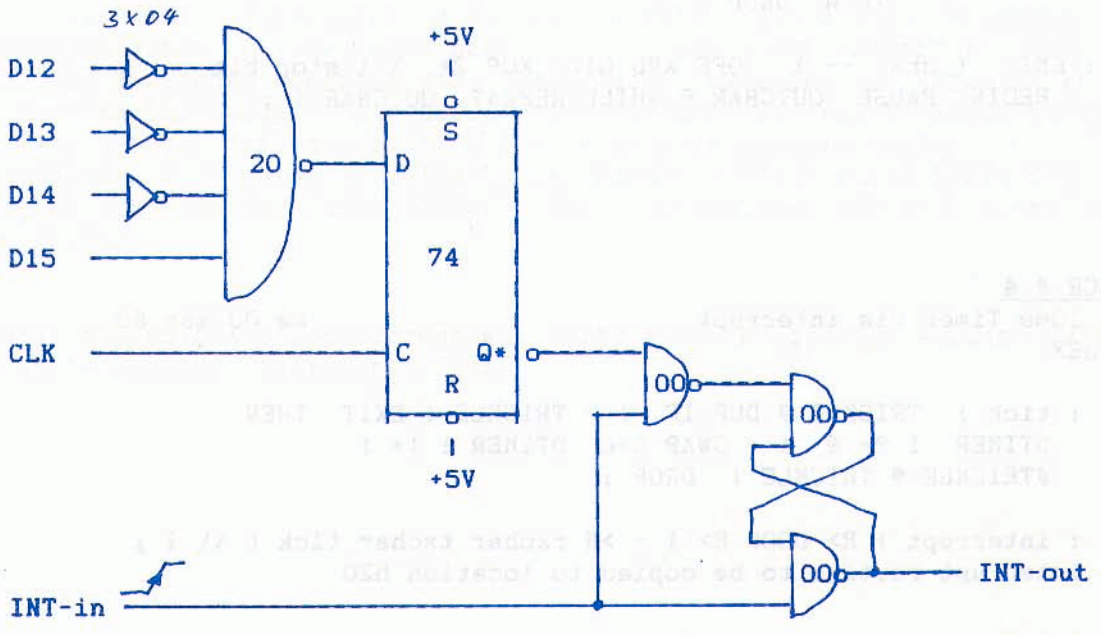
Klaus Schleisiek
Roter Hahn 42 / 2 Hamburg 72 / 040 644 9412

Der INT* Eingang des Novix-Prozessors NC 4016 ist wegen eines Fehlers auf dem Chip eigentlich nicht als Interrupt Eingang zu benutzen. Das liegt daran, daß sehr oft beim Aufruf der Interrupt Routine eine falsche Rückkehradresse auf den Returnstack gelegt wird. Dieser Fehler ist jedoch abhängig von der Instruktion, die während des Interrupts gerade ausgeführt wird.

Mit ein bißchen externer Hardware, die den Interrupt nur dann an den Prozessor weiterleitet, wenn dieser als nächstes eine "ungefährliche" Instruktion ausführen wird, läßt sich das Problem praktisch lösen. Sogenannte ALU-Instruktionen sind problemlos unterbrechbar und auch einfach auf dem Datenbus aufzufinden: Die Bits D12 .. D15 zeigen den Wert 8 - dieser muß dekodiert werden und ein Interrupt, auf den der Prozessor reagieren soll, muß nur solange verzögert werden, bis der Prozessor anfängt, eine solche "harmlose" Instruktion auszuführen. Dies leistet die unten gezeigte Schaltung. Sie ist sehr einfach in einem PAL oder GAL zu realisieren, das zusätzlich noch als Adreßdekodierer für den angeschlossenen Hauptspeicher dienen kann.

Seit einigen Wochen betreibe ich mit diesem Schema die serielle Schnittstelle, mit der der NC 4016 mit dem Host verbunden ist. Als übertragungsrate benutze ich 19.2 kbaud, so daß alle 17,4 ysec ein Interrupt erfolgt. Mysteriöse Systemabstürze habe ich bisher nicht erlebt. Dazu ist INT-in mit einem Ausgang des 8253 Timer Chips verbunden. Im folgenden ist auf vier Screens der Code für die serielle Schnittstelle per Interrupt dargestellt. Jede Bitzelle dauert 3 Interrupts; mit diesem Schema ergibt sich eine saubere Synchronisation auf das Startbit.

Da die Serialisierung als Interruptprozeß stattfindet, kann endlich das Wort KEY? realisiert werden. Mit dem Worth tick wird außerdem noch ein 32bit Zähler alle 10ms hochgezählt, so daß sich auch exakte Verzögerungszeiten realisieren lassen. Zusätzlich kann in diesem Wort bei einem Multitaskingsystem noch für jede Task ein Zähler geführt werden, der nach voreinstellbaren Zeiten Hintergrundprozesse wieder aufweckt. Sind alle diese Möglichkeiten in der Interruptroutine realisiert, so dauert der Interrupt ca. 8 ysec bei einem System mit einer linearen 4 MHz Clock. Damit verbleiben selbst bei 19,2 kbaud noch über 50% der Maschinenleistung für die Applikation.



SCR # 1

```
\ System Variables for serial I/O via INT          ks 03 mar 88
\ Variables below h20, 2 cycle access
\ accessed on almost every interrupt
 VARIABLE TRICKLE       \ counter for 10ms intervals
 VARIABLE OUTPHASE      \ counter for int. transmit routine
 VARIABLE OUTCHAR       \ register for int. transmit routine
 VARIABLE INPHASE       \ counter for int. receive routine
 VARIABLE INCHAR        \ register for int. receive routine

\ Variables above h20, 3 cycle access
\ initialized
 VARIABLE #TRICKLE  \ constant for 10ms interval trickle ctr.
 VARIABLE RXED      \ result register rx-serial
\ non initialized
 VARIABLE DTIMER    \ 10ms 32-bit Timer
 VARIABLE TIMER     \ 10ms 16-bit Timer
```

SCR # 2

```
\ receive serial via interrupt                     ks 03 mar 88
 HEX

( : rxchar )  INPHASE @ DUP IF  1 - INPHASE ! EXIT  THEN
    INCHAR @ IF  INCHAR @ 2/
                 Xport I@ 010 AND IF  0100 OR  THEN
                 DUP 1 AND IF  RXED !  0  THEN  INCHAR !
                 2 INPHASE !  DROP EXIT
    THEN  Xport I@ 010 AND IF  DROP EXIT  THEN
    0100 INCHAR !   3 INPHASE ! DROP ;

 : KEY?     ( -- f )   PAUSE RXED @ 1 AND ;

 : KEY      ( -- char )
    BEGIN  KEY? UNTIL  RXED @ 2/  RXED OFF ;
```

SCR # 3

```
\ send serial via interrupt                        ks 03 mar 88
 HEX

( : txchar ) OUTPHASE @ DUP IF  1 - OUTPHASE ! EXIT  THEN
    OUTCHAR @ IF  Xmask I@  01E Xmask I!
                  OUTCHAR @  DUP Xport I!
                  2/ OUTCHAR !   Xmask I!   2 OUTPHASE !
              THEN  DROP ;

 : EMIT  ( char -- )   OFF AND 0100 XOR 2*  \ 1 stop bit
    BEGIN  PAUSE  OUTCHAR @ WHILE REPEAT  OUTCHAR ! ;
```

SCR # 4

```
\ 10ms Timer via interrupt                         ks 03 mar 88
 HEX

( : tick )  TRICKLE @ DUP IF  1 - TRICKLE ! EXIT  THEN
    DTIMER  1 @+ @  1 + SWAP 0+c  DTIMER 1 !+ !
    #TRICKLE @ TRICKLE !  DROP ;

( : interrupt ) R> DROP R> 1 - >R rxchar txchar tick [ \\ ] ;
\ interrupt routine to be copied to location h20
```

Accessories und das volksFORTH83

Bernd Pennemann, Berlin

In dem folgenden Artikel wird gezeigt, wie Accessories für den Atari ST mit dem volksFORTH83 programmiert werden können. Hierzu wird ein einfaches Beispiel vorgestellt.

Schlüsselworte : Accessory, Alertbox, Atari ST, volksFORTH83

Accessories stellen eine der Annehmlichkeiten dar, die das GEM auf dem Atari ST bietet. Darunter versteht man Programme, die nach dem Einschalten des Rechners geladen werden und anschließend in jedem (GEM-unterstützen) Programm zur Verfügung stehen. Für die IBM-kompatiblen Rechner existiert etwas Ähnliches, so z.B. "Side-kick".
Accessories unterscheiden sich in mehreren Punkten von normalen GEM-Programmen, wie z.B. vom volksFORTH83. Daher können mit dem volksFORTH83 geschriebene Applikationen nicht ohne weiteres als Accessories verwendet werden.
Im folgenden wird gezeigt, welche Änderungen am volksFORTH83 selbst erforderlich sind. Anschließend wird anhand eines Beispiels gezeigt, wie eine Applikation, die als Accessory dienen soll, aufgebaut werden muß.

Nach dem Einschalten erwartet der Atari ST im Diskettenlaufwerk A: eine Diskette, von der zunächst alle (sic!) Files mit der Extension ".ACC" (z.B. CONTROL.ACC) geladen werden. Anschließend werden diese Programme der Reihe nach gestartet. Jedes Accessory muß nun folgendes erledigen :

1) Der Stack des Prozessors muß auf einen definierten Wert gesetzt werden.
2) Anschließend müssen eine Reihe von GEM-Aufrufen getätigt werden, die in dem Wort GRINIT des volksFORTH83 zusammengefaßt sind [1].
3) Um den Namen des Accessories in der Menüleiste festzulegen, muß MENU_REGISTER aufgerufen werden.
4) Schließlich muß der Event-Handler gestartet werden.

Damit ist die Initialisierung der Accessories beendet (sie werden nun gewöhnlich auf's Anklicken warten). Der Punkt 1) wird automatisch vom volksFORTH83 durchgeführt. Dabei muß allerdings eine Besonderheit beachtet werden:
Das volksFORTH83 reserviert sich gleich nach dem Laden ausreichend Speicherplatz oberhalb von HERE . Dort werden z.B. die Blockpuffer untergebracht. Normalerweise geschieht das durch Aufruf der GEM-Dos-Routine MALLOC . Dieser Weg funktioniert bei Accessories jedoch nicht, da der so reservierte Speicherbereich nicht an HERE anschließen würde. Statt dessen muß dem GEM-Dos durch Ändern des sog. Fileheaders mitgeteilt werden, daß dieses Programm einen uninitialisierten Datenbereich benötigt. Dessen Länge entspricht gerade der Länge des Bereiches oberhalb von HERE plus 1 Kbyte für den Stack. Er wird dann vom GEM-Dos bereits beim Laden des Programms reserviert. Anschließend muß dann (wie gehabt) der Stackpointer des Prozessors gesetzt werden.
Die erwähnten Änderungen am volksFORTH83 können einfach durch Laden des Files "MAKE_ACC.SCR" durchgeführt werden - der Programmierer muß sich nicht weiter darum kümmern.

Nun soll anhand des Beispielprogramms "TEST_ACC.SCR" gezeigt werden, wie man die Punkte 2 bis 4 realisieren kann.

Nach dem Starten des volksFORTH wird zunächst ACC_MAIN aufgerufen. Es erledigt zunächst Punkt 2 durch Aufruf von GRINIT . Dadurch wird eine GEM-Anwendung initialisiert. Sodann trägt sich das Programm mit Hilfe des Wortes ACC_INIT in die Menüleiste ein (Punkt 3). ACC_INIT ist, da es immer benötigt wird und die Sache etwas übersichtlicher macht, gleich in MAKE_ACC.SCR mit definiert worden. Schließlich tritt ACC_MAIN in die für Accessories typische Endlosschleife ein. Da einmal geladene Accessories bis zum Neustart des Rechners vorhanden sind, müssen sie natürlich endlos etwas tun. In unserem Beispiel wird (mit WAIT) darauf gewartet, daß das Accessory angeklickt wurde, um dann (mit ALERT) als Reaktion eine Alertbox [1] zu malen.
Das Wort WAIT ist nun recht interessant. Es ruft die komplizierteste GEM-Routine überhaupt, den Multi-Event-Handler EVNT_MULTI , auf. Dabei kann angegeben werden, auf welche Ereignisse reagiert werden soll. In unserem Beispiel ist es das das Auftreten einer Nachricht vom GEM. Zusätzlich soll sich der Handler jedoch nach 50 Millisekunden wieder zurückmelden (falls er nichts anderes zu tun hat ! Es kann auch sehr viel länger dauern !). Hat der Handler keine Nachricht für unser Programm, so wird nur einmal kurz der Multitasker des volksFORTH83 (mittels PAUSE) angeworfen. Erhält das Accessory jedoch die Nachricht, daß es angeklickt wurde, so kehrt es augenblicklich zu ACC_MAIN zurück.
Man kann nun statt ALERT etwas sinnvolleres programmieren. So ist es z.B. möglich, den Kommandointerpreter des volksFORTH83 zu starten. Dadurch stünde dem Benutzer das gesamte volksFORTH83 zur Verfügung. Er könnte als Entspannung während der anstrengenden Arbeit mit einem Textprogramm beispielsweise Forth-programme kompilieren, ausdrucken, ausführen oder debuggen.
Allerdings gibt's dabei eine Einschränkung : Accessories können keine eigene Menüleiste besitzen; der GEM-Editor kann also nicht unverändert benutzt werden.

Noch ein paar Tips:

Zum Programmieren von simplen Accessories vom Typ "zeige die Uhrzeit an" sollte man das volksFORTH83 besser nicht verwenden, denn es verbraucht dann (un-)verhältnismäßig viel Speicherplatz. Besser ist es dann, eine ganze Reihe von Funktionen zusammenzufassen. Das ist aufgrund der modularen Struktur von Forth sehr leicht und man spart zusätzlich Einträge im Desk-Menü.

Ist ein Accessory fertig programmiert, so sollte für die endgültige Erstellung des ".ACC"-Files ein volksFORTH83 ohne Editor verwendet werden. Dadurch wird das Programmfile verkürzt und verhindert, daß das Resource-File des Editors nachgeladen wird.

Um Speicherplatz zu sparen (wer muß das heutzutage noch ?), kann LIMIT heruntergesetzt und die Zahl der Blockpuffer auf 1 reduziert werden.
Soll der Multitasker nicht benutzt werden, so kann in WAIT anstelle von

```
BEGIN    pause   acc_prepare   evnt_multi
                 :mu_mesag and  UNTIL
```

einfacher evnt_mesag
geschrieben werden. Dadurch wird das Programm kürzer und schneller.

[1] "Alert-Boxen unter volksFORTH83 auf dem Atari ST", VD II/4 p.19-23

volksFORTH-83 FORTH-Gesellschaft eV (c) 1985/86 we/bp/re/ks MAKE_ACC.SCR Seite 1

1

```
0 \ Loadscreen für das Umwandeln des Forth in ACC       bp 22mar87 \\
1
2 Onlyforth
3
4 \needs Gem                    include gem\aes.scr
5 \needs :ac_open               &40 Constant :ac_open
6
7 \needs Code                   2 loadfrom assemble.scr
8
9 Onlyforth Gem also
10
11 1 2 +thru
12
13
14
15
```

2

```
0 \                                             bp 22mar87 \\
1
2 : savesystem    ( -- )        \ modifiziert für ACCs
3   limit here -   0            \ Rest bis 64k als Double
4   $0400.       d+             \  sowie Platz für den 68k-Stack
5   $0A 2!                      \  als BSS eintragen
6   savesystem ;
7
8
9 Variable acc_id               \ trage Accessory ID hier ein
10
11 \ : ap_id        ( -- n )   global 4+ @ ;
12
13 : acc_init      ( 0$ -- )   \ Komfort für den Anwender
14   ap_id  swap )absaddr  menu_register  acc_id ! ;
15
```

3

```
0 \ Neuer Code für Systemstart                          bp 22mar87
1
2 label bootcode                                        \ Anfang der Basepage
3   .1 bootcode $100 - pcrel) A5 lea
4   .1 $18 A5 D) D0 move                                \ BSS Anfang
5      $1C A5 D) D0 add                                 \ zeigt hinter BSS Ende
6      $4. #     D0 sub                                 \ 1 Wort Platz lassen
7      D0 A7 move                                       \ Setze 68k-Stack
8      bootcode $1C - pcrel) FP lea                     \ FP auf Forth-Anfang
9      ' cold  )body 2+ r#) jmp                         \ Kaltstart ohne $A00A
10                                                      \   (schaltet Maus ab) !
11
12 bootcode  $1C  here bootcode -  cmove                \ ersetze alten Bootcode
13
14 bootcode dp !                                        \ Vernichte bootcode
15
```

4

Loadscreen : versammelt alles, was benötigt wird.

AES	für MENU_REGISTER etc., außerdem ist ein Accessory ohne GEM selten sinnvoll.
Code	für neuen Bootcode - Assembler wird auf den Heap geladen.

5

SAVESYSTEM	ist so modifiziert, daß alles oberhalb von HERE in das BSS (uninitialisierter Datenbereich) paßt. Zusätzlich gibt's noch 1Kbyte für den 68000-Stack.
ACC_ID	merke die Identifikationsnummer des Accessories
ACC_INIT	nimmt einen String (durch 0 begrenzt) und trägt das Accessory unter diesem Namen ein.

0

\ Speicherlayout bp 22mar87

Origin (FP) zeigt auf die Kopie des Filenanfangs im Speicher :

0	$601A	magic number
2	here-$1C	size of text segment excl. header
6	0.	size of initialized data
$A	limit-(here-$1C)+1000	size of stack segment
$E	0.	size of symbol table
	rest is unused	
$1C	lea bootcode-100(PC),A5	Programmcode (text segment)
	:	

FP zeigt auf $E4 Bytes hinter der TPA; an dieser Stelle wird das erste Wort des Files ($601A) abgelegt. Der ausgeführte Code beginnt aber $100 Bytes hinter der TPA !

volksFORTH-83 FORTH-Gesellschaft eV (c) 1985/86 we/bp/re/ks TEST_ACC.SCR Seite 1

1

```
0 \ Testaccessory, das nur eine Alertbox malt          bp 22mar87
1
2 \needs acc_init               include make_acc.scr
3 \needs :mu_mesag              include gem\gemdefs.scr
4
5 Onlyforth Gem also
6
7 1 2 +thru
8
9 savesystem alertbox.acc
10
11
12
13
14
15
```

2

```
0 \ initialisiere ein Accessory                        bp 22mar87
1
2 : Create acc_events
3   %110000  ,                  \ Timer und Mesag Event
4   0 , 0 ,                     \ keine mouse-buttons
5   here $14 allot $14 erase    \ rectangles nicht vorh.
6   &50 , 0 ,                   \ 50 Millisekunden timer-delay
7
8 : acc_prepare     acc_events intin $20 cmove
9                   message )absaddr   addrin 2! ;
10
11 : wait                       \ Warte auf Anklicken des Acc.
12   BEGIN  BEGIN  pause   acc_prepare  evnt_multi
13                 :mu_mesag and UNTIL
14          message @ :ac_open =  message 8 + @ acc_id @ =  and
15   UNTIL ;
```

3

```
0 \ male alert-box                                     bp 22mar87
1
2 : alert
3   BEGIN 1 0" [1][volksFORTH83|Accessory !][OK]"
4         form_alert drop  wait  REPEAT ;
5
6 : acc_main
7     grinit
8     0"  volksFORTH83"  acc_init
9     ['] noop Is 'cold
10    BEGIN  wait  alert  REPEAT  ;
11
12 ' acc_main is  'cold
13
14
15
```

4

```
\                                                      bp 22mar87
```

Ein Beispiel für ein Accessory, das nur eine simple Alertbox malt. In einer wirklichen Anwendung müßte noch erheblich mehr geschehen, z.B. Retten des Bildschirms und Unterdrücken der Einschaltmeldung beim Starten des volksFORTH83

Der Loadscreen lädt die nötigen Bestandteile und erzeugt ein File ALERTBOX.ACC.

5

```
\\                                                     bp 22mar87
```

Accessories besitzen eine Endlosschleife, in der sie auf ein sog. Event warten müssen. In diesem Fall ist es die Nachricht, daß dieses Accessory angeklickt und dadurch aktiviert worden ist. Dazu müssen drei Bedingungen eintreten:

1) Es muß ein sog. Message-Event auftreten, d.h. GEM sendet einer Applikation eine Nachricht.
2) Die Nachricht muß bedeuten, daß ein Accessory angeklickt wurde.
3) Der Adressat dieser Nachricht ist unser Accessory

WAIT wartet nun auf die Kombination dieser drei Ereignisse. Zusätzlich jedoch wird alle 50 msec. ein Timerevent zugelassen der nur dazu dient, PAUSE in Minimal 50 msec Abstand aufzurufen. Dadurch bleibt so etwas wie Multitasking (jedoch nur bei GEM-Applikationen) möglich.

0

```
0 \ male alert-box                                     bp 22mar87
```

ALERT malt eine Alertbox

ACC_MAIN tut alles für das Accessory...
Gem wird initialisiert (falls kein Editor vorhanden ist)
Das Accessory wird unter "volksFORTH83" eingetragen
ACC_MAIN wird aus der Kaltstartroutine entfernt
Endlosschleife, die im Prinzip alle Accessories aufweisen müssen.

Sorge dafür, daß ACC_MAIN nach dem Kaltstart ausgeführt wird.

Nocheinmal: Parameter und lokale Variable in Forth

Ulrich Hoffmann, Kiel
Klaus Schleisiek, Hamburg

Stichworte:
Parameter, lokale Variablen, Stack-Kommentare, Rekursion

Dieser Artikel beschreibt ein prinzipielles Vorgehen bei der Realisierung von lokalen Variablen in Forth. Das vorgestellte Konzept zeichnet sich durch seine Flexibilität und Erweiterbarkeit aus. Eine beispielhafte Implementation für volksFORTH-83 wird angegeben.

Zahlreiche Vorschläge der Implementation von lokalen Variablen in Forth sind schon veröffentlicht worden (u.a. /1/,/2/,/3/). Hier soll nun ein Verfahren vorgestellt werden, das zum einen einfach zu implementieren, zum anderen erweiterbar und damit potentiell leistungsfähiger ist als andere Konzepte.

Herkömliche Programmiersprachen (Pascal, Modula, C, Fortran, ...) enthalten mehr oder weniger ausgeprägt das Konzept der Prozeduren: benannte Programmteile, die an mehreren Stellen innerhalb eines Programmes aktiviert (aufgerufen) werden können. Damit Prozeduren vernünftig benutzt werden können, kennen die meisten Sprachen auch Paramater (Daten, die der Prozedur bei Aufruf übergeben werden). Daten, die nur innerhalb einer Prozedur gebraucht werden, heißen lokal zu dieser Prozedur. Für sie kann dynamisch, d.h. zur Laufzeit des Programms, Speicherplatz reserviert und nach Ende der Prozedur wieder freigegeben werden.

Auch Forth kennt Prozeduren, nämlich die Worte, und prinzipiell auch Parameter. Diese sind allerdings ohne Namen, da in Forth die übergabe der Parameter auf dem Datenstack stattfindet. Damit deutlich wird, wieviele und welche Parameter das Wort benötigt, haben Stack-Kommentare allgemeine Verwendung gefunden: Der öffnenden runden Klammer folgt eine Aufzählung der Parameter. Dabei steht der Parameter, der als oberstes Stackelement erwartet wird, ganz rechts. Den Parametern folgt ein "--", das die Ausführung des Wortes symbolisieren soll. Dann wird der Zustand des Datenstacks nach Ausführung des Wortes dargestellt: Wieder steht das oberste Stackelement ganz rechts. Die schließende runde Klammer beendet den Stack-Kommentar. Für das FORTH-83 Standard-Wort COUNT sieht er dann z.B. so aus:

```
COUNT ( StringAdresse -- Adressel.Zeichen Länge )
```

COUNT nimmt die Adresse eines "counted strings" und berechnet daraus die Adresse des ersten Zeichen sowie die Länge des Strings.

Forth kennt in der Grundausstattung normalerweise keine lokalen Daten, also Daten, die nur während der Laufzeit eines Wortes gültig sind und von anderen Worten nicht erreicht werden können. Dies wird im allgemeinen durch Programmierer-Disziplin gelöst. Daten, die lokal zu einem Wort sein sollen, dürfen in anderen Worten eben nicht benutzt werden. Nachteilig bei dieser Lösung ist, daß die lokalen Daten auch dann Speicherplatz beanspruchen, wenn das Wort gar nicht aktiv ist.

Folgende Lösung des Problems bietet sich an:

Es soll in Forth möglich sein, Parameter zu benennen, ihre Namen sollen dabei am Ende der Definition wieder verschwinden, so daß sie von anderen Worten gar nicht mehr benutzt werden können. Es soll eine bequeme Möglichkeit geben, lokale Daten zu vereinbaren, die dann erst bei Aufruf des Wortes Speicherplatz beanspruchen. Eine Definition für ein Wort, das die Summe der ganzen Zahlen zwischen m und n berechnet, könnte dann so aussehen:

```
: nsum   { ( m n -- s )
           Variable sum }
   0 sum !
   n @ 1+  m @  DO  I sum +!  LOOP
   sum @ s ! ;
```

Das Wort NSUM hat die zwei Eingabeparameter M und N (angegeben in Stack-Kommentar Notation) und liefert S als Ausgabeparameter. Zusätzlich wurde die lokale Variable SUM deklariert. Bei Aufruf von NSUM werden M, N und SUM als lokale Variable angelegt und zusätzlich M und N mit den beiden obersten Werten auf dem Stack vorbesetzt.

Dann wird SUM auf Null gesetzt, der Inhalt von N und M als Schleifenzähler auf den Stack gelegt und SUM in der Schleife berechnet. Nach Verlassen der Schleife wird der Inhalt von SUM in die lokale Ergebnisvariable S gespeichert. Dann wird das Wort NSUM verlassen und die lokalen Variablen M, N, S und SUM vernichtet, nachdem S als Ergebnis auf dem Stack abgelegt wurde.

Das so erdachte Konzept hat weitere Konsequenzen:

- es lassen sich nun rekursive Worte mit lokalen Daten schreiben, die bei jedem Aufruf des Wortes erneut angelegt werden.
  Etwa:

```
: fak recursive { ( n -- f ) }
    n @ 2 < IF 1 f ! exit THEN
    n @ 1-  fak  n @ *  f ! ;
```

- beliebige Stackoperatoren sind definierbar. z.B.

```
: 3dup { ( a b c -- a b c a b c ) } ;
```

Und so kann dieses Konzept verwirklicht werden:

Alle Befehle, die zur Definition der lokalen Daten benutzt werden sollen, werden in einem speziellen Vokabular LOCAL definiert. Die öffnende geschweifte Klammer, die "Declarations" ausgesprochen wird, sorgt dafür, daß zuerst dieses Vocabular durchsucht wird. In LOCAL sind zumindest

```
(  --  )  CREATE  VARIABLE  ALLOT
```

definiert. Diese Worte sorgen dafür, daß Namen für lokale Daten erzeugt werden und der Platzbedarf verwaltet wird. Die schließende geschweifte Klammer, die "Body" ausgesprochen wird, beendet schließlich die Definition von lokalen Daten, schaltet wieder auf die normale Suchreihenfolge zurück und sorgt dafür, daß beim Ausführen des Wortes genügend Platz für die lokalen Daten reserviert wird. Im Anhang ist am Beispiel des Wortes NSUM gezeigt, wie die lokalen Variablen im Dictionary kompiliert werden.

Tritt in der "{" folgenden Definition der Name eines lokalen Datums auf, so erzeugt dies entsprechenden Code, um später bei Ausführung des Wortes dieses lokale Datum zu erreichen. Als einzig sinnvoller Platz für die lokalen Daten hat sich der Returnstack erwiesen. Die dort sowieso abgelegten Daten (Rückkehradressen) werden wie die lokalen Daten beim Ein- und Austritt des Wortes angelegt, bzw. vernichtet.

Neue lokale Datentypen können ohne weitere Schwierigkeiten in LOCAL definiert werden. Das Konzept bleibt also erweiterbar und kann sich daher auch an die Problemgegebenheiten anpassen. So ist auch die öffnende runde Klammer in LOCAL nichts anderes als ein besonderes Wort zum Definieren von Parametern. Die folgenden Namen werden als lokale Variablen erzeugt und zusätzlich wird Code erzeugt, der später diese Variablen mit Werten vom Stack füllt. Die nach dem "--" folgenden Namen definieren lokale Variablen, deren Inhalt nach Ausführung des Wortes auf den Stack gelegt werden sollen. Entsprechender Code wird auch für diese Variablen erzeugt.

Die Definition der lokalen Namen wirft eine Schwierigkeit auf: Die Namen müssen zu einer Zeit erzeugt werden, zu der gerade eine Definition stattfindet, sie dürfen deshalb nicht einfach ab HERE abgelegt werden. Fortschrittliche Systeme stellen hierfür besondere Worte zur Verfügung (etwa >LABEL), ansonsten kann das Problem durch Veränderung des DictionaryPointers gelöst werden (/4/), so daß die lokalen Definitionen in einen anderen Speicherbereich compiliert werden und dadurch die gerade laufende Definition nicht gestört wird.

Die folgende Implementation wurde mit volksFORTH-83 realisiert. Bei anderen Forthsystemen können sich erhebliche Abweichungen ergeben. Es stellte sich heraus, daß die Implementation der Compiler für die lokalen Variablen in volksFORTH besonders einfach war.

/1/ "Stack Frames and Local Variables", George B. Lyons,
Journal of Forth Application and Research Vol 3 #1

/2/ "Local Variables", J.R.Hart, J.Perona,
Journal of Forth Application and Research Vol 3 #2

/3/ "Error Trapping and Local Variables", K.Schleisiek, FORML 1984

/4/ "Pascal-ähnliche lokale Variablen in Forth", Bernd Pennemann,
Vortrag auf dem Treffen der lokalen Gruppe Hamburg

Anhang: Der durch { und } compilierte Code

Als Beispiel soll die Definition für das Wort NSUM dienen.

```
: nsum  { ( m n -- s )  Variable sum }
   0 sum !
   n @ 1+  m @  DO  I sum +!  LOOP
   sum @ s ! ;
```

Zunächst wird - wie bei jeder :-Definition - der Name NSUM im Dictionary eingetragen und in den Compilationszustand geschaltet.

{ ist immediate, so daß es unmittelbar ausgeführt wird. Dadurch wird LOCAL als zuerst zu durchsuchendes Vokabular in die Suchreihenfolge gelegt.
Der Wert von LAST wird auf den Stack gelegt, einer Systemvariablen, die auf das Namensfeld (NFA) des Wortes zeigt, das gerade bzw. als letztes definiert worden ist, in diesem Falle NSUM.
Das Wort ALLOCATE wird im Dictionary abgelegt (COMPILE ALLOCATE) und, da ALLOCATE zur Laufzeit ein INLINE-Argument auf der folgenden Dictionary Adresse erwartet, wird die Adresse dieser Stelle auf den Stack gelegt und zunächst eine Null (0 ,) compiliert. An dieser Stelle wird später durch } die Anzahl der zu reservierenden lokalen Speicherstellen eingetragen.
Dann wird [([COMPILE] [) ausgeführt und das System befindet sich wieder im Interpreterzustand.

(beginnt mit der Compilation der Eingangsparameter. Die Situation wird dadurch verkompliziert, daß die Namen, die vor dem "--" stehen, in umgekehrter Reihenfolge abgearbeitet werden müssen. Dies wird dadurch erreicht, daß zunächst - rekursiv - nach dem Ende der Eingangsparameter gesucht wird, wobei jeweils der Zeiger >in gemerkt wird, und danach >in jeweils wieder zurückgestellt und die Namen als lokale Variable compiliert werden. Diese Umkehrung der Reihenfolge ist deshalb sinnvoll, da sonst das Wort PARAMETERS die Vorbesetzung der Werte nicht mit einem einfachen CMOVE machen könnte. Am Ende sind nun also M und N als lokale Variable definiert und nun compiliert

-- das Wort PARAMETERS (COMPILE PARAMETERS) und trägt dahinter die Anzahl der Bytes ein, die der Anzahl der durch (erzeugten Variablen entspricht. PARAMETERS kopiert zur Laufzeit diese Anzahl von Bytes vom Stack in die durch ALLOCATE geschaffene Stackframe.
Dann kümmert sich RESULTS um die Ausgangsparameter. Wieder muß - wie bei den Eingangsparametern - die Reihenfolge umgekehrt werden. Wenn die schließende) gefunden ist, wird für jede lokale Ergebnisvariable RESULT gefolgt vom Byteoffset in die zur Laufzeit aufgebaute Stackframe ins Dictionary compiliert.
Das Wort RESULT funktioniert ähnlich wie das Wort PUSH in volksFORTH. Wenn es ausgeführt wird, legt es Daten auf dem Returnstack ab, die ausgeführt werden, wenn das Wort zuende ist und mit EXIT verlassen wird. Resultat ist dann, daß der Inhalt der zugeordneten lokalen Speicherstellen auf den Stack gelegt werden.
Das System ist noch immer im interpretierenden Zustand und als nächstes wird

VARIABLE SUM verarbeitet. Da durch (das Vokabular LOCAL ganz nach oben in die Suchreihenfolge gelegt wurde, wird die VARIABLE im Vokabular LOCAL gefunden - und damit die lokale Variable SUM erzeugt. Nun wird endlich

) gefunden und, da durch alle lokalen Datentypen SIZE hochgezählt wurde, kann nun hinter das ALLOCATE, das von (compiliert worden war, die Anzahl der Bytes eingetragen werden, die für lokalen Daten reserviert werden muß. Nun wird lediglich noch LAST wieder auf den Namen NSUM zurückgestellt und LOCAL wieder aus der Suchreihenfolge durch TOSS entfernt und mit] findet die Rückkehr in den compilierenden Zustand statt.

Danach folgt normaler Forthcode. In der Definition können aber nun die zwischen (und) erzeugten Namen benutzt werden, die, da es sich dabei um IMMEDIATE Worte handelt, jeweils ausgeführt werden und dabei das Wort RELATIVE und den jeweiligen Offset in die Stackframe compilieren. (Siehe die Definition von CREATE in LOCAL). Wird nun die Definition von NSUM mit ; beendet, so wird nun - wie gewöhnlich - NSUM im Dictionary sichtbar gemacht. Dadurch, daß dieser Mechanismus in volksFORTH mit Hilfe der Variablen LAST und dem Wort VISIBLE verwirklicht ist, werden alle Namen der lokalen Datentypen aus dem Dictionary ausgehängt und sind nach dem ; nicht mehr vorhanden.

Im folgenden noch der Code, der vom Compiler ins Dictionary compiliert wird:

```
|link|c|NSUM| (:) | allocate | 8 | parameters | 4 | result | 4 |...
                ...| 0 | relative | 6 | ! | relative | 0 | @ | 1+ |...
                ...| relative | 2 | @ | (do | 14 | I | relative | 6 |...
                ...| +! | (loop | endloop | relative | 6 | @ |...
                ...| relative | 4 | ! | unnest |
```

```
0
```

```
 0          +-------------+    Stack frame used        UH 23Jan88 \ lokale Variablen, Stackkommentare, Parameter uh/ks 16 mär 88
 1          !  local 1b   !    in this approach
 2  +-------) +-------------+
 3  !       :  .....      :                    Dieses File enthaelt Definitionen, die lokale Variablen und
 4  !    +-) +-------------+                    Parameter fuer volksFORTH zur Verfuegung stellen.
 5  !    !  !  local 2a   !
 6  !    !  +-------------+                    Diese lokalen Variablen sind vom gleichen Typus, wie sie
 7  !    !  !  local 1a   !                      auch von der Sprache C zur Verfügung gestellt werden - nur
 8  !    !  +-------------+ memory               noch flexibler.
 9  !    +---+-x  oldRP    !
10  !       +-------------+                    Der letzte Screen enthält einige Beispieldefinitionen.
11  +-------+-x  oldframe  !
12          +-------------+ (-- frame          Es ist zu beachten, daß der Aufbau der "Stackframes" auf dem
13          !    free     !                     Returnstack zur Laufzeit einiges an Zeit braucht, so daß jetzt
14          +-------------+   +- RP             auf einmal auch in Forth der Aufruf von "Prozeduren" teuer wird.
15          :             :   V
```

```
1
```

```
 0 \ LOAD-Screen                                uh/ks 16 mär 88  \ LOAD-Screen                                uh/ks 16 mär 88
 1
 2  ' find $22 + @ Alias found                                    Mache das Wort FOUND (im Kernel headerless) dem System wieder
 3                                                                  bekannt.
 4  : search   ( string 'vocab -- acf n / string ff )           SEARCH durchsucht ein gegebenes Vokabular nach einem bestimmten
 5     dup @ [ ' Forth @ ] Literal - Abort" kein Vokabular"        String durch, liefert dessen Code-Feld-Adresse und ein Wert,
 6     )body (find IF  found exit  THEN  false ;                   der Auskunft gibt, ob das Wort immediate oder restrict ist.
 7                                                                  Kann SEARCH das Wort im geg. Vokabular nicht finden, so
 8                                                                  gibt es die Stringadresse und ein False-Flag zurueck.
 9  : last'  ( -- cfa )   last @ name) ;                        LAST' legt die cfa des Wortes auf den Stack, das gerade
10                                                                  definiert wird bzw. als letztes definiert wurde.
11  2 7 thru clear
12
13 \ 8 load  \ Beispiele
14
15
```

```
2
```

```
 0 \ local Memory allocation                    uh/ks 16 mär 88  \ local Memory allocation                    uh/ks 16 mär 88
 1
 2  Variable frame   frame off                                    Ein Zeiger auf den aktuellen Stack-Frame (activation-record)
 3
 4  : memory ( -- addr ) frame @   4+ ;                          MEMORY liefert die Anfangsadresse des Stack-Frames.
 5
 6
 7 ! Create: free  r) frame !  r) rp! ;                          Aufraeumroutine, die den Frame-Zeiger auf seinen vorhergehenden
 8                                                                  Wert zuruecksetzt, und den momentanen Frame wieder freigibt.
 9 ! : allocate ( -- )  \ inline size                          ALLOCATE reserviert auf dem Returnstack die Zahl der Bytes, die
10     r) dup @  rp@ under swap - rp! )r                           in der dem Aufruf folgenden 16-Bit Konstanten angegeben ist.
11     frame @ )r  rp@ frame !  free )r 2+ )r ;                    Diese Bytes werden beim Verlassen des Wortes wieder
12                                                                  freigegeben.
13
14
15
```

3

```
 0 \ local storage adressing and results          uh/ks 16 mär 88
 1
 2 | : inline ( -- n )  r) r) dup 2+ )r @ swap )r ;
 3
 4 | : relative ( -- addr )  inline  memory + ;
 5
 6 | Create: pushresult   r) @ ;
 7
 8 | : result  ( -- )  \ inline offset
 9     r) dup 2+ swap @  memory + )r  pushresult )r )r ;
10
11 | : parameters ( ... -- ) \ inline # of bytes
12     sp@. memory  over r) dup 2+ )r @ under + )r cmove r) sp! ;
13
14
15
```

```
 0 \ local storage adressing and results          UH 23Jan88
 1
 2 INLINE holt die 16-Bit Konstante, die hinter dem Aufruf des
     Wortes steht, das INLINE aufruft.
 3
 4 RELATIVE Berechnet die Adresse des n-ten Bytes im Stackframe
     n ist die 16-Bit Konstante, die dem Aufruf von RELATIVE folgt.
 5
 6 Routine, die dafuer sorgt, dass die Werte von vorher bestimmten
     Adressen auf den Stack gelegt werden.
 7
 8 RESULT veranlasst, das beim Verlassen des Wortes, das RESULT
     enthaelt, der Wert der n-ten lokalen Variablen auf dem Stack
 9   liegt. n steht in Verbindung mit der 16-Bit Konstanten, die
10   dem Aufruf von RESULT folgt.
11
12 PARAMETERS kopiert eine Anzahl (sie steht als 16-Bit Konstante
     hinter dem Aufruf von PARAMETERS) von Stackelementen in den
13   aktuellen Stackframe.
14
15
```

4

```
 0 \ primitve local data-types                    uh/ks 16 mär 88
 1
 2  Vocabulary Local
 3
 4 | Variable size   size off
 5
 6  Local definitions
 7
 8  : Create  size @ )Label immediate restrict
 9  Does) compile relative @ , ;
10
11  : allot ( n -- )   size +! ;
12
13  : Variable   Create  2 allot ;
14
15
```

```
 0 \ primitve local data-types                    uh/ks 16 mär 88
 1
 2 LOCAL enthaelt die bekannten Worte der lokalen Definitionsphase.
 3
 4 SIZE entaelt die Groesse des Stack-Frames, die zur
     Uebersetzungszeit berechnet wird.
 5
 6
 7
 8 CREATE erzeugt eine lokales Element, das spaeter seine Adresse
     auf den Stack legt.
 9
10
11 ALLOT reserviert die angegebene Anzahl von Bytes im Stack-Frame.
12
13 VARIABLE erzeugt eine lokale 16-Bit Variable.
14
15
```

5

```
 0 \ Start and end of local defintion area        uh/ks 16 mär 88
 1  Forth definitions
 2
 3  : {  ( -- last addr err# )  \ pronounced "declarations"
 4     also Local  size off   last @
 5     compile allocate   here 0 ,  [compile] [   4 ; immediate
 6
 7  Local also definitions
 8
 9  : }  ( last addr err# -- )  \ pronounced "body"
10     4 ?pairs   size @ swap !  last !  toss ] ;
11
12
13
14
15
```

```
 0 \ Start and end of local defintion area        uh/ks 16 mär 88
 1
 2 { markiert den Anfang der lokalen Definitionen.
 3
 4
 5
 6
 7
 8
 9 } markiert das Ende der lokalen Definitionen.
10
11
12
13
14
15
```

6

```
0 \ Stack diagramm compiler                          uh/ks 16 mär 88
1
2 | : pname   ( -- string )   name nullstring? Abort" ( fehlt" ;
3
4 | : finis?  ( -- f )   pname ['] Local search nip ;
5
6 | Create loc
7
8 | : -exists? ( -- addr tf | cfa ff )   pname find
9     IF  dup @ [ ' loc @ ] Literal - exit  THEN  true ;
10
11 | : islocal  ( -- acf )  )in @
12     -exists? IF  swap )in !  Variable last'  THEN  nip ;
13
14
15
```

```
\ Stack diagramm compiler                          uh/ks 16 mär 88

PNAME holt das nächste Wort aus dem Quelltext und bricht mit
  einer Fehlermeldung ab, wenn der Quelltext erschöpft ist.
FINIS? testet, ob der nächste Abschnitt in der Stack-Kommentar-
  Compilation erreicht ist.

-EXISTS? testet das nächste Wort im Quelltext, ob es keine
  lokale Variable ist.

ISLOCAL liefert die cfa der lokalen Variablen, deren Namen
  im Quelltext steht und erzeugt sie, falls sie nicht schon
  existierte.
```

7

```
0 \ Stack diagramm compiler                          uh/ks 16 mär 88
1
2   : (  Recursive
3     )in @   finis? IF  )in ! exit  THEN  (
4     )in push   )in !  Variable ;
5
6 | : results   Recursive
7     )in @   finis? IF  drop exit  THEN  results
8     )in push   )in !  islocal  compile result  )body @ , ;
9
10    : )      compile parameters  size @ , ;
11
12    : --     )  results ;
13
14  Onlyforth
15
```

```
\ Stack diagramm compiler                          uh/ks 16 mär 88

( compiliert die Eingangsparameter.

RESULTS ( bzw. -- ) compiliert die Ausgangsparameter.

) compiliert die Größe der benötigten Stackframe.
```

8

```
0 \ sample-code                                    uh/ks 16 mär 88
1
2   : sqr ( ( n -- s ) )   n @ s !
3     30 0 DO  n @ s @ /  s @ +   2/  s !
4            s @ dup * n @ = IF LEAVE THEN LOOP ;
5
6   : pythagoras ( ( a b -- result ) )
7       a @ dup *    b @ dup *  +  sqr  result ! ;
8
9   : nsum  ( ( m n -- s )  Variable sum )  0 sum !
10    n @ 1+  m @  DO  I sum +!  LOOP     sum @ s ! ;
11
12  : fak  Recursive ( ( n -- f ) )
13    n @ 2 ( IF  1  ELSE   n @ 1- fak n @ * THEN  f ! ;
14
15
```

```
\ sample-code                                      uh/ks 16 mär 88

Iterative Berechnung der Wurzel nach Newton.

Berechnung der Hypothenusenlaenge aus den Kathetenlaengen im
  rechtwinkligen Dreieck.

Berechnung der Summe der natuerlichen Zahlen  zwischen
  m und n-1.

Berechnung der Fakultaet auf rekursive Weise.
```

Local Variables

Dear Editor,

Local variables have been the subject of many earlier contributions, but I have not seen the following, simple implementation.

My version of Forth local variables makes use of ordinary Forth variables declared, and probably used, outside the colon definition. The only word to be used, following the name of the variable, is LOCAL. This must be in the beginning of a colon definition, and not inside control structures or some other kind of return-stack manipulation. The variable can then be used freely inside the colon definition, and will be restored to its original value on exit.

The code for LMI PC/FORTH+ is shown in Figure One. (For PC/FORTH, * omit ADDR>S&O.)

Since the natural scope for a local variable in Forth is a colon definition, local variables can be managed on the return stack. LOCAL first saves the address and the value of the variable on the return stack, then arranges an exit through (LOCAL) by placing the address of (LOCAL) on the return stack. On exit from the colon definition, which was the local scope of the variable, (LOCAL) will then restore the old value of the variable from the return stack. There can be more than one local variable in a definition.

For a simple example, see Figure Two's rather foolish implementation of the factorial function N! with recursion and a local variable.

This is a simple and easy-to-use, high-level implementation of local variables. An assembler-coded version would probably provide very fast local variables in Forth.

Yours,
Henning Hansen
116, Technical University of Denmark
2800 Lynby, Denmark

AUS FD 1/87

```
: (LOCAL)
   R> R> ! ;                          \ restore variable address and value
from return stack

: LOCAL ( adr — )
   R> SWAP                            \ save top return address
   DUP @ SWAP >R >R                   \ put variable address and value on
                                        return stack
   ['] (LOCAL) >BODY ADDR>S&O >R      \ exit via (LOCAL)
   >R ;                               \ restore top return address to con-
                                        tinue current definition
```

* ADDR>S&O weglassen in F83 und volksFORTH83

Figure One. Hansen's local variables for PC/FORTH+.

```
VARIABLE VAR
: N! ( n — n! )
      VAR LOCAL
      DUP VAR !
      1- DUP 0> IF RECURSE ELSE DROP 1 THEN
      VAR @ * ;
\ 10000 times 12 N! in 45 sec, with PC/FORTH+.
```

The simpler word n! is much faster, without the local variable:

```
: n! ( n — n! )
      1 SWAP 1+ 1 ?DO I * LOOP ;
\ 10000 times 12 n! in 10 sec.
```

Figure Two. Sample use of LOCAL.

Screens verschieben mit AROUND

Markus Redeker

Es ist wichtig, ein Programm in einer sinvollen Ordnung zu haben, also zusammenhängende Definitionen auch beieinander. Nun stellt es sich aber oft erst während der Programmentwicklung heraus, welche Definitionen eng zusammengehören und welche weniger eng. Im Interesse der Übersichtlichkeit muß also der Programmtext ab und zu einmal umgeordnet werden.

Einzelne Definitionen kann man mit dem Editor versetzen, aber wennn es um den Inhalt ganzer Screens geht, beginnen die Scwierigkeiten: Das einzige Wort, das Screens zu mehreren versetzt, CONVEY, ist besser für den Transport in andere Dateien geeignet als zur Umordnung innerhalb einer Datei. Wenn man es trotzdem versucht, dann wartet ein kompliziertes und fehlerträchtiges Verfahren auf einen. Man braucht mehrere Arbeitsschritte, Zwischenspeicher für Screens und muß außerdem aufpassen, daß man keine Screens irrtümlich überschreibt. Notwendig ist überhaupt keine Verschiebung, sondern eine Rotation.

Als Antwort auf diese Misere ist AROUND entstanden. Das Grundprinzip von CONVEY wird beibehalten: Alle Screens in einem festgelegten Bereich werden um die gleiche Distanz verschoben - nur wird jetz der Bereich als ringförmig geschlossen angesehen. Auf den Screen am Ende des Verschiebungsbereichs folgt also wieder der erste, und alle Screens befinden sich nach der Verschiebung wieder im ursprünglichen Bereich. Es ist auf der Diskette kein zwischenspeicher mehr nötig und es könnnen auch keine Screens mehr überschrieben werden.

Der Algorithmus

Grundlage ist eine sehr einfache Idee: Man beginnt mit irgendeinem Screeen und vertauscht ihn mit dem Screen, der am Ende an seiner Stelle stehen soll. Dannn verfährt man an seiner Startposition genauso usw. Mit jedem Schritt kommt ein Screen an seine Zielposition, und der Screeen, mit dessen Ausgangsposition begonnen wurd, wandert in Gegenrichtung mit. Dieser "Rückläufer" spielt noch eine wichtige Rolle für die Effizienz des Programms. Irgendein Tausch bringt auch ihn an seine Zielposition und man muß aufhören. Dieser Zyklus muß eventuell noch mit anderen Startpositionen wiederholt werden, um alle Screens zu erfassen.

Wie oft, und mit welchen Startpositionen, das hängt in unserem Fall ab von der Distanz d, um die verschoben wird, und von der Anzahl n der verschobenen Blöcke. Meist erfaßt ein Zykls nicht alle Screens, aber sie folgen in einem gleichen Abstand aufeinander. Offensichtlich ist d ein Vielfaches davon, denn Start- und Zielposition eines Screens liegen im gleichen Zyklus, ebenso n. Daraus folgt, daß bei teilerfremden d und n der Abstand 1 ist und alle Screens von einem Zyklus erfaßt werden.

Und was macht man in den anderen Fällen? Dann existiert ein größter gemeinsamer Teiler t von n und d, und man teilt Gruppen zu je t Screens ein, deren Anzahl n/t beträgt und mit der neuen Schrittweise d/t wieder teilerfremd ist.

Nun müßte man die Screens eigentlich auch gruppenweise vertauschen, aber ea gibt ein besseres Verfahren. Beim Tausch von Screengruppen bleibt nämlich die Reihenfolge der Screens innerhalb der Gruppe erhalten; es gibt unabhängige Zyklen, die z.B. nur aus Screeens bestehen, die in ihrer Gruppe an zweiter Stelle stehen. Insgesamt sind es t verschiedene Zyklen. Wenn man nacheinander t aufeinanderfolgende Screens als Startscreens für einen Zyklus nimmt, erfaßt man auch alle Screens.

Dieses "versetzte" Verfahren hat den Vorteil, daß in jedem Zyklus der "Rückläufer"-Screen im Speicher bleiben kann - das halbiert die Zahl der notwendigen Diskettenzugriffe. Aber es verlangt auch, daß Anfang und Ende des verschobenen Bereichs gleich zu Beginn auf Zulässigkeit geprüft werden, da ein Programmabbruch mitten in einem Zyklus die Datei in ziemlicher Unordnung hinterlassen kann.

Realisation von SWAPSCR

Das Vertauschen von Screens ist im normalen Blockkonzept nicht vorgesehen, ich mußte deshalb auf etwas ungewöhnliche Weise vorgehen. SWAPSCR arbeitet also mit "Tricks", denn viele Definitionen, die eigentlich gebraucht werden, sind in volksFORTH headerlos kompiliert und daher nicht zugänglich, sodaß ich hier zu einer Notlösung greifen mußte. Der direkte Tausch der Blocknummern im Speicher schien mir dabei noch der eleganteste Weg zu sein. Wer anders denkt oder den genauen Aufbau seines Systems nicht kennt, kann statt dessen auch die Inhalte der Screens vertauschen.

Allerdings bringt es jede Benutzung von BLOCK mit sich, daß SWAPSCR nicht multitaskingfähig ist. BLOCK ruft PAUSE auf, sodaß beim zweiten Mal die erste Bufferadresse nicht mehr unbedingt gültig ist.

Hat man nur zwei Screenbufffer, dann muß man bei einem selbstgeschriebenen SWAPSCR unbedingt darauf achten, daß die Screens in der richtigen Reihenfolge auf die Diskettte zurückgeschrieben werden; nur so bleibt der "Rückläufer" während des ganzen Zyklus im Speicher.

Und wer nur einen Buffer hat, kann SWAPSCR auch ganz anders realisieren: Am Anfang von CYCLE wird der Startscreen zwischengespeichert und am Ende an seine Zielposition zurückgeschrieben (die normalerweise von DROP geschluckt wird). SWAPSCR wird dann durch COPY ersetzt.

Page No 1 volksFORTH83 der FORTH-Gesellschaft eV AROUND.SCR

1

```
0 \ Ladeblock mit Hilfsdefinitionen                    cep 30Nov87
1 OnlyForth decimal
2
3 : gcd  ( n1 n2 -- ggt )  BEGIN  under mod  ?dup 0= UNTIL ;
4
5 : ?outside  ( scr# -- scr# )  dup  [ Dos ] isfile@ in-range ;
6
7 : 'swap  ( addr1 addr2 -- )
8   over @ >r  dup @  rot !  r> swap ! ;
9
10 1 2 +thru
11
12
13
14
15
```

5

```
% Ladeblock mit Hilfsdefinitionen                    cep 27Dez87

GCD         Groesster gemeinsamer Teiler
            ( greatest common divisor )
?OUTSIDE    Existiert dieser Screen in der aktiven Datei?

'SWAP       Vertausche Zahlen im Speicher
```

Marcus Redeker, 4352 Herten

2

```
0 \ SWAPSCR, relative Screennummern                    cep 22Jan88
1 : >blk#  ( blkaddr -- 'blk# )  4- ;
2 : get#  ( blk# -- 'blk# )  block update  >blk# ;
3 : swapscr  ( scr#1 scr#2 -- )  get# swap get# 'swap ;
4   \ SCR#2 wird bei Platzbedarf zuerst zurueckgeschrieben.
5 : ?swapping  limit first @ -  [ b/buf 2* ] Literal -
6   0< Abort" needs 2 block buffers" ;
7
8 Variable start          \ Erster Block, der verschoben wird
9
10 : >relative  ( start end newstart -- length step )
11   >r  over ?outside start !
12   ?outside over - 1+  swap r> - ;
13 : relswap  ( scr#1 scr#2 -- )
14   start @ under  + >r  + r>  swapscr ;
15
```

6

```
% around                                              cep 22Jan88

>BLK#       berechnet die Adresse der Blocknummer.
GET#        Vorbereitung,um die Blocknummer zu aendern.
SWAPSCR     Screentausch im Speicher ( volksFORTH-spezifisch ).

?SWAPPING   Sind genug Screenbuffer fuer SWAPSCR vorhanden?

>RELATIVE   prueft die Parameter, initialisiert START und
            berechnet die Parameter fuer RELCYCLE.

RELSWAP     SWAPSCR relativ zu START.
```

3

```
0 \ around                                             cep 22Jan88
1 Variable length         \ Laenge des verschobenen Bereichs
2 Variable step           \ Schrittweite der Verschiebung
3
4 : nextscr  ( rel# -- rel#' )  step @ - length @ mod ;
5 : scr/cycle  ( length step -- scr/cycle #cycles )
6   over gcd  under /  swap ;
7 : cycle  ( rel# scr/cycle -- )
8   1 ?DO  dup nextscr  dup rot relswap  LOOP drop ;
9
10 : relcycle  ( length step -- )  2dup  step ! length !
11   scr/cycle @ DO i over cycle LOOP  drop ;
12
13 : around  ( start end newstart -- )
14   ?swapping  >relative relcycle  save-buffers ;
15
```

7

```
% around                                              cep 24Jan88

STEP        kann jeden beliebigen Wert gefahrlos annehmen.

NEXTSCR     berechnet den naechsten Screen im Zyklus
            relativ zu START.
SCR/CYCLE   Groesse eines Zyklus und Anzahl der Zyklen.
CYCLE       ein einzelner Tauschzyklus.

RELCYCLE    vereinfachte Version von AROUND, berechnet die
            Screens relativ zu START.

AROUND      Rotation des Bereichs von <start> bis <end>,
            sodass <newstart> an den Anfang kommt.
```

Zufallszahlen

M.Kalus

Eine Methode, Zufallszahlen zu gewinnen, besteht darin, ein rückgekoppeltes Schieberegister als Generator zu benutzen. Eine Konstante wird mit den oberen Bits AND-verknüpft und die danach noch gesetzten Bits werden XOR-verknüpft. Das Ergebnisbit wird als Eingabewert in das Schieberegister zurückgefüttert (Bild1).

Im Algorithmus RND wird die 16Bit breite Variable SEED als Schieberegister benutzt. Der angegebene Code ist für eine 65xx CPU geschrieben. Da diese 8bit breite Register hat, bedurfte es jeweils zweier Operationen um SEED zu bearbeiten.

Zunächst wird der Fall SEED = 0 abgefangen. Denn sonst würden immer nur wieder Nullen generiert. Der Gernerator versagt in diesem Fall. Dies ist wahr für $B4 als Konstante. Andere Werte verursachen ein anderes Verhalten des Generators - z.B. 'rastet' er mit $BA bei -1 ein.

Als nächstes wird das High-Byte von SEED mit der Konstanten $B4 AND-verknüpft und das Ergebnis aus dem Akkumulator über das Carry-Bit heausgeschoben. Dabei werden die gesetzten Bits gezählt. Die Zähl-Schleife läuft, bis der Akkumulator Null geworden ist. Das Y-Register dient als Zähler der gesetzten Bits.

Das erste Bit des Zählers wird über den Akkumulator durch Schiebe-Operationen als 'Ergebnisbit' in das Register SEED zurückgefüttert. SEED wird anschließend noch auf den Stack geholt.

Wird der Generator mit SEED=0 gestartet, beginnt er nach ca. 16 Aufrufen 'random' zu erzeugen. (Das Generatorverhalten habe ich u.a. mit dem Wort RNDTEST beobachtet.) Die 16Bit-Zufallszahl x kann nun weiter verarbeitet werden. RANDOM hinterläßt eine Zufallszahl aus einem bestimmten Bereich. Die Integer-Division von x durch max liefert als Rest immer eine Zahl, die kleiner als max ist. MOD läßt diesen Rest auf dem Stack. ABS sorgt schließlich noch für positive Zahlen.

Eingebürgert hat sich RANDOM (max -- +n), d.h +n liegt im Bereich 0<n<max. Das Intervall min<n<max' erhält man durch Addition von min. Diese Art RANDOM sollte für viele Anwendungsfälle bereits genügen. Typische Verwendung:

```
max RANDOM min +  ( -- adr )
```

Da aber keine Methode ein so echtes 'random' erzeugt wie Regentropfen auf dem Straßenpflaster, stellt sich die Frage nach der 'random'-Güte des x aus RND. In Abhängigkeit von der verwendeten Konstanten liefert dieser Generator periodisch wiederkehrende Zahlenfolge. Dabei soll $B4 mehr 'random' sein als andere Werte. Um eine gleichmäßigere und noch zufälligere Verteilung zu bekommen, bedient man sich einer Methode ähnlich dem Kartenmischen: Aus einem Stapel Karten irgendwo eine herausnehmen und obenauf legen (engl. shuffel=mischen). In (1) wurde dies für polyFORTH gezeigt. Ich benutzte das C64 ultraFORTH-83.

Ein Array wird mit Zufallszahlen gefüllt. Die letzte Zahl wird genommen, um aus ihren höheren Bits einen Index in dieses Array zu bilden. Damit wird eine Zufallszahl x aus dem Array auf den Stack 'gezogen' und an ihre Stelle eine neu erzeugte Zufallszahl 'gesteckt'. Außerdem wird x nun Ausgangswert für den nächsten Index und dazu in die letzten Stell des Arrays kopiert.

Dieses Mischen - irgendwo herausnehmen und obenauflegen - kombiniert damit, eine gezogene Zahlen gleich durch eine neue zu ersetzen, ergibt ein höhere Zufälligkeit von x. Dazu kann das Array noch in zufälligen Abständen komplett regeneriert werden.

Quelle:
Zettel, L.
Shuffled Random Numbers, Forth Dimensions, 1986, Vol.8, Nr.3 S.31

Anmerkung: Der Assembler des ultraForth83 verwendet die üblichen Kontrolstrukturen - sie heißen nur anders! Übersetzung:

```
cc ?[ ... [] ... ]?    = cc IF ... ELSE ... THEN
[[ ... ]]              = BEGIN ... REPEAT
[[ ... cc ?]]          = BEGIN ... cc UNTIL
[[ ... cc ?[[ ... ]]? = BEGIN ... cc WHILE ... REPEAT
```

cc steht für "conditional code". Siehe auch Handbuch volksForth83.

Glossar:

SEED (-- adr) Variable. Keimzelle neuer Zufallszahlen.
RND (-- x) Hinterläßt eine zufällige 16bit-Zahl auf dem Stack.
RANDOM (max -- +n) Hinterläßt eine zufällige 16bit-Zahl aus dem Bereich 0<+n<max auf dem Stack. Benutzt RND.
RNDS (-- pfa) Array für Zufallszahlen. Hinterlegt seine PFA.
SHSEED (-- adr) Variable. Keimzelle neuer Indizes für das Array RNDS.
NEWRNDS (--) Füllt das Array RNDS mit Zufallszahlen. Benutzt dazu RND.
SHRND (-- x) Hinterläßt eine zufällige 16bit-Zahl auf dem Stack. Methode des gemischten Arrays (engl. SHuffled RaNDom) voller RND Zahlen.
SHRANDOM (max -- +n) Hinterläßt eine zufällige 16bit-Zahl aus dem Bereich 0<+n<max auf dem Stack. Benutzt SHRND.

```
Scr 16 Dr 1
 0 \ random
 1
 2 decimal
 3
 4 variable seed
 5
 6 \ n = 16bit random number
 7 \ jeweils letztes n liegt in SEED
 8
 9 code rnd ( -- n )
10   seed lda  seed 1+ ora
11   0= ?[ 1 # lda  seed sta
12         0 # lda  seed 1+ sta  ]?
13   $B4 # lda  seed 1+ and  clc  0 # ldy
14   [[ .a lsr   cs ?[ iny ]?
15   0 # cmp  0= ?]
16   tya  1 # ldy
17   .a lsr  seed rol  seed 1+ rol
18   seed lda pha  seed 1+ lda  push jmp
19 end-code
20
21 : random ( max -- +n ) \ n ist random
22   rnd swap mod abs ;   \ 0<n<max
23
24 -->
```

```
Scr 17 Dr 1
 0 \ shuffled-random
 1
 2 create rnds   64 allot
 3 \ letzter wert = shuffled seed
 4 here dup , constant shseed
 5
 6 : newrnds ( )
 7   66 0 do rnd rnds i + !  2 +loop ;
 8
 9 : shrnd  ( -- x ) \ shuffeln
10   shseed @              \ -- n
11   32 um* swap drop 2* \ -- i
12   rnds +                \ -- adri
13   dup @ ( xi ziehen)  \ -- adri xi
14   rnd                   \ -- adri xi n
15   rot ! ( n stecken)  \ -- xi
16   dup shseed ! ;        \ -- x
17
18 : shrandom ( max -- +n ) \ n ist random
19   shrnd swap mod abs ;   \ 0<n<max
20
21 : rndtest ( -- )
22   base @ 2 base ! page
23   begin rnd 0 0 at 20 u.r key? until
24   base ! ;
```

Forth Interessengruppen die sich regelmäßig treffen:
- Darmstadt: Andreas Soeder, 06257-2744. Treffen an der VHS an einem Mittwoch in der Mitte des Monats.
- Hamburg: Peter Vollmann, 040-6440221. Treffen jeden vierten Samstag im Monat ab 16:00Uhr in der Berufsfachschule für Radio- und Fernsehtechnik, Eimsbüttelerstr.64-66.
- München: Heinz Schnitter, 089-3103385 und Christoph Krinninger 089-7259382. Treffen jeden vierten Mittwoch im Monat 19:30Uhr im Vereinsraum 1 im Bürgerhaus Unterschleißheim am Rathausplatz (S-Bahnhaltepunkt S1 Unterschleißheim)
- Wuppertal: Michael Kalus, 0202-736591. Treffen jeden vierten Freitag im Monat ab 20:00Uhr im Bahnhof Ottenbruch, Funckstrasse, W'tal-Elberfeld.

Es möchten in ihrer Region eine Gruppe gründen:
- Lüneburg: Peter Sommerfeld, 04134-7035, Lehrer
- Erkelenz: Sven Lütkemeier, 02431-2344, Schüler
- CH Büren: Thomas Müller, Solothurnstr.46, Büren a.a, Schweiz, Creativ-Direktor, Tel: ---

Außerdem kann man hier um Rat fragen:
- Berlin: Bernd Pennemann, 030-7923923
- Düsseldorf: Ralf Hohmann, 0211-289929
- Rhein/Neckar: Thomas Prinz, 06271-2830
- Karlsruhe: Siegfried Hirsch, 0721-886664
- Pfalz: Rudolf Marich, 07275-5116

Ihre Interessengebiete bekannt machen möchten:
- Volksforth und Ultraforth: Bernd Pennemann, 030-7923923 und Klaus Schleisiek, 040-6449412.
- Graphik und Animation: Marco Pauck, 040-3900139
- 32Bit Systeme: Robert Jones, 02434-4579
- Forthchips, -maschinen und RISC: Roland Steck, 06151-661192
- Künstliche Intelligenz: Ulrich Hoffmann, 04307-6869
- NC4000 Novix Chip: Klaus Schleisiek, 040-6449412
- Realtime, relationale Netze: Wigand Gawenda, 040-446941
- Gleitkomma Arithmetik: Andreas Döring, 02631-52786

Die FORTH INTEREST GROUPS der Welt sind in jeder Forth Dimensions verzeichnet. In der BRD ist die Hamburger Gruppe bis jetzt einzige offizielle FIG.

Forth in 1988 lokale Gruppe Hamburg den 19.1.88

Wir wollen uns im Jahr 1988, zur Entlastung des Wochenendes, jeweils am vierten Mittwoch um 19 Uhr in neuen Räumen treffen.
Die Adresse ist

Klaus Schleisiek
Roter Hahn 42c (Laden) Nebenstraße vom Pezolddamm
2000 Hamburg 72 Bus 277 von U-Bahn Berne oder Barmbek

Als Hauptprogramm wird dieses Jahr ein Volksforth-Kursus durchgeführt. Dabei wird aber auch auf andere Dialekte (F83) eingegangen.

Datum	Programm
27. Januar	Klaus Schleisiek Bericht über die FORML-Konferenz USA Einführung zum Volksforth-Kursus
24. Februar	Volksforth Daten- und Returnstack
23. März	Lutz Wellhausen Digitale Motorsteuerungen (Roboter)
27. April	Volksforth Dictionary
25. Mai	Volksforth Interpreter
29. Juni	Volksforth Compiler
Juli	Sommerpause
31. August	Volksforth Kontroll-Strukturen
28. September	Volksforth Fehlerbehandlung
26. Oktober	Volksforth Definierende Worte
30. November	Volksforth Multitasking
Dezember	Winterruhe

Lokaler Koordinator
Peter Vollmann Telefon 644 02 21