OAK RIDGE FORTH

April 23, 2009

This 8K version of Forth for the Zilog Z8 implements a microcontroller version of the 83 Forth standard. It features a screen editor to facilitate rapid code development and has a "native code" extension that permits testing "assembler" code in a Forth environment. The Forth code is commonly used as part of a 2K to 64K system, with a Forth kernel occupying the lower portion of ROM. The system is modular so that a 2K or 4K version of the Forth kernel can be developed easily.

Information available on the Internet includes Starting Forth by Leo Brodie (13 chapters):
http://home.iae.nl/users/mhx/sf0/sf0.html…
http://home.iae.nl/users/mhx/sf12/sf12.html

Manuals and technical information on the Z8 family:

Z8 (8681) User Manual
Z8 (CMOS series) Manual,
Z8 86C91 hardware description

Although the Forth system itself is in Z8 assembler code, most of the material consists of "precompiled" Forth words. That is, the assembler code is what Forth itself produces when it compiles a Forth word. In most cases, the Forth definitions are included along with their assembler counterparts.

Oak Ridge Forth contains drivers for an external I^2C EEPROM, for a PCF 8574 I^2C serial-to-parallel interface (useful for an implementing an LCD), and a PC keyboard interface. These driver routines operate via Port 3. Port 2 is not used by Forth and is available for use by an application program. The drivers increase the size of the system by about a 1,200 bytes.

Also included is a package of 4-byte (IEEE 754) floating point routines. As compiled, these routines take about 2,000 bytes.

Several routines are described that assume use of the Zilog 86C93, which has hardware implemented multiply/divide. These routine include integer sin/cos, and 128-point fast Fourier transform. Programming for the 86C93 can be found here

The default configuration Forth system assumes availability of serial input/output. The shareware HyperTerminal or TeraTerm are good options.

To use the screen editor with this Forth system, set the terminal emulator properties to VT220 emulation with a 50 ms character delay.

The Forth system assumes an 8681, 86C91 or 86C93 Z8 processor. Other Z8-compatible chips, such as the Sharp LU800, may also be used.

Architecture of the compiler

This implementation of Forth is designed to be as compact as possible and uses one byte tokens for most of its words. With the Z8, this implementation is nearly as fast as the Forth version available from Micromint. However, execution speed can be improved by optimizing critical words Oak Ridge Forth's using its assembly language capability.

Original Z8 architecture and eZ8 (Z8 with flash memory) compared

In the past few years Zilog has encouraged use of its eZ8 version of the Z8 (having strict Harvard architecture) for program development. The eZ8 has internal flash memory (from 2K to 64K) and an internal byte-addressable volatile memory (from 2K to 8K). With these versions of the Z8 users are encouraged to use flash memory memory for program development and use Zilog's version of "C" as the development language platform.

The problem with this mode of development is that when a new or altered version of a program is needed,
the entire code for the Z8 must be compiled and loaded into flash memory. And further, as internal memory can only be used for data (one cannot execute code located in the internal data store), this means that new or altered code can reside only in flash memory.

The original Z8 architecture (a hybrid of Harvard and van Neumann design) permitted a combination of not-easily-altered code (PROM) plus a large amount of volatile RAM. Thus, one is able to load PROM with code that is (mostly) checked out, and incrementally developing new or altered code in RAM. When the new or altered code is checked out, it can be

transferred to PROM and the development cycle repeated with the next code requirement.

One drawback of the original Z8 as compared to the eZ8 is that two (usually quite valuable) ports are tied up with external memory communication. However, with the advent of serial-to-parallel chips for I^2C protocol, it is relatively easy and inexpensive to add any number of ports to the original Z8 architecture.

One could argue that development in Forth with the eZ8 architecture does not suffer from the drawback of "C" with the eZ8, as Forth words, which are just double-byte data items, can be stored in local eZ8 RAM, then transferred to flash memory once the code is checked out. The problem with this tact is that Forth (at least most versions) give a user an ability to program both in Forth and in native code (Oak Ridge Forth has these options).

External memory utilization

RAM is assumed to follow ROM. Forth starts by examining memory to determine the end of ROM and the start/end of RAM. ROM is always assumed to start at address zero. If the two bytes at the end of ROM are –1 (FFFF), then Forth starts as if QUIT was just entered; otherwise, the word at end of ROM is assumed to be the address of a Forth word where execution is to begin. The Forth return stack is started at the end of RAM; the Forth data stack starts 64 bytes below the return stack. The Forth dictionary pointer is initialized to the low address of RAM.

As is usual in Forth, the text input buffer (64 bytes long plus a one byte sentinel) is located at the end of the return stack, adjacent to the start of the data stack. Thus, an underflow of the data stack is usually fatal, as this stack then starts to overwrite the text input buffer. The SETALT routine (described in the "utility routines" section) is available to check for a stack underflow in the Forth word in which it occurs.

If a system is configured as 32K of ROM followed by 32K of RAM, then the memory allocation (in hex) is as follows:

0000	- 1FFF	Forth kernel
2000	- 7FFF	Available for application code & drivers
8000	- …	Bottom of RAM – start of dictionary
…	- …	Screen page areas (1K blocks)
…	- FF7F	Data stack
FF80	- FFC0	Text input buffer (65 bytes long)
FFC2	- FFFF	Return stack (32 word capacity)

Both the data stack and return stack can be defined to reside in the register file for RAM-less systems.

Z8 Control Registers

The control registers are initialized as follows:

Z8 Control Register		Initialization
F0	SIO	--
F1	TMR	00
F2	T1	F0
F3	PRE1	03
F4	T0	01
F5	PRE0	17
F6	P2M	FF
F7	P3M	41
F8	P0IM	92
F9	IPR	2B
FA	IRQ	00
FB	IMR	00
FC	FLAGS	--
FD	RP	--
FE-FF	SP	(see above)

Note that the distribution version of Forth is configured for 19.2 KB serial I/O (12.288 MHZ clock). Also note that the initialization specifies the "normal bus timing" option (Register F8, bit 5). Setting F8 to $B2_h$ will set memory access to extended timing and thus decrease execution speed by about 1/2.

Register File Utilization

Bytes 4-16 of the register file are used for interrupt jumps (see below). The next 32 bytes (10h-2Fh) are used for Forth system variables. The standard Forth system register group is the next 16 bytes (30h-3Fh). Changing the Forth register group from the default is discussed in the example of an interrupt service routine (below).

Interrupts

Z8 hardware sends interrupts to one of six vectors at the first 12 bytes of program memory (ROM). Forth has addresses in those locations that cause a branch to indirect branch instructions that reference locations in

bytes 4-15 of the register file. Thus, a programmer can place addresses in these low register file locations that will cause branches to appropriate interrupt handling routines. As an example, suppose that the interrupt service routine for Interrupt 0 is located at 4562h. By placing the address 4562h at register file location 4, the interrupt proceeds from location 4 in ROM to 94h. This address contains a JP@ instruction with an indirect jump via location 4 in the register file. This indirect jump transfers control to the interrupt routine. A comprehensive example of writing Forth code for interrupts appears in the "native code" section.

Case

In general Forth words may be entered in either upper or lower case. This works because the dictionary is searched first for the word as entered then, if the word is not found, the dictionary is searched again with the entered word converted to upper case. Note, however, that several native-code words require a proper case for address mode specification and condition-testing codes.

The case of words created via a "colon" or "create" definition is preserved. For this reason, users should create new words in upper case so that they can be found whether referred to in upper or lower case.

The default is to retain the character count and the first three letters of new Forth words. Thus, new Forth words ABCD and ABCE (both four letters long and both starting "ABC" will be considered identical. The default can be overridden by specifying a new character count in the Forth variable WIDTH.

Loading and Storing

@ At
… X A → … X V
The number at the top of stack specifies an address which is replaced with the 16-bit number at that address. If the number is less than 100h, the address is interpreted as a register file address. The most significant byte is in the lower-numbered address.

! Store
… X V A → … X
The number second on the stack is store in memory at the location specified by the number on the top of stack. If the number is less than 100h, the address is interpreted as a register file address. The most significant byte is in the lower-numbered address. Remember that registers 80-EF are not defined in the 8681 version of the Z8, and registers F0-FF are system registers.

C@ Char-At
… X A → … X C
The number at the top of stack specifies an address. A 16-bit number - the lower 8-bits from the character at the specified address and zeros in the high order part - replaces the number specifying the address.

C! Char-Store
… X C A→ … X
The number at the top of stack specifies an address. The lower 8-bits of the number at second from top are stored at the specified address.

+! Plus-Store
… X I A → … X
The address specified at the top of stack is incremented by the second value on the stack.

CMOVE Character move
… X S D N → … X
N characters are moved from a source location (S) to a destination location (D) starting at the beginning of the source. Note that if the destination is one greater than the source, this instruction can be used to fill the source area by propagating N bytes.

CMOVE> Character move - higher
… X S D N → … X
N characters are moved from a source location (S) to a destination location (D) starting at the end of the source area.

Logical and Arithmetic

+ Plus
… X A B → … X S
The two 16-bit values on the stack are replaced by their sum.

- Minus
… X T B → … X D
The number at top of stack is subtracted from the number second from top and the result replaces them on the stack.

1+ One-plus
… X T B → … X D

The number at top of stack is incremented by one.

NEGATE Negate
… X A → … X R
The 16-bit value on the stack is replaced by its negative.

D+ D-Plus
… X AL AH BL BH → … X SL SH
The two 32-bit values on the stack are replaced by their sum.

DNEGATE D-Negate
… X L H → … X RL RH
The 32-bit value on the stack is replaced by its negative.

AND And
… X A B → … X R
A logical "and" is performed on the two numbers at the top of stack. They are then replaced with this result.

OR Or
… X A B → … X R
A logical "or" is performed on the two numbers at the top of stack. They are then replaced with this result.

U* Unsigned Multiply
… X A B → … X L H
The two numbers at top of stack are multiplied, assuming they are unsigned quantities. The 32-bit result replaces the arguments on the stack.

U/ Unsigned Divide
… X A B D → … X R Q
The 16-bit number at top of stack divides the double number at the second and third positions on the stack. Both quantities are interpreted as unsigned quantities. The remainder and quotient replace the arguments on the stack.

S* Signed Multiply
… X A B → … X L H
The two numbers at top of stack are multiplied, assuming they are signed quantities. The 32-bit result replaces the arguments on the stack.

S/ Signed Divide
… X L H D → … X R Q
The 16-bit number at top of stack divides the double number at the second and third positions on the stack. Both quantities are interpreted as signed quantities. The remainder and quotient replace the arguments on the stack. The results are a "floored division" as defined in the Forth 83 standards. For example:

```
 10  /  7 =  1    R:  3
-10  /  7 = -2    R:  4
 10  / -7 = -2    R: -4
-10  / -7 =  1    R: -3
```

Thus Q * D + R = N, where Q is the quotient, D is the denominator, R is the remainder and N is the numerator.

If a 86C91 processor, which has hardware multiply and divide, is used, these functions can be used in Forth via the fast multiply and fast divide definitions:

```
:A FU* FD 0E RIM OR, 4 R POP, 5 R POP,
 2 R POP, 3 R POP, 6 40 RIM LD, NOP,
 NOP, NOP, 3 R PUSH, 2 R PUSH, 1 R PUSH,
 0 R PUSH, FD F0 RIM AND, EXIT,

:A FU/ FD 0E RIM OR, 4 R POP, 5 R POP,
 0 R POP, 1 R POP, 2 R POP, 3 R POP,
 6 20 RIM LD, NOP, NOP, NOP, NOP,
 1 R PUSH, 0 R PUSH, 3 R PUSH, 2 R PUSH,
 FD F0 RIM AND, EXIT,
```

Stack Manipulaton

DROP
… X V → … X
This word at top of stack is removed.

DUP
… X V → … X V V
This word at top of stack is duplicated.

SWAP
… X A B → … X B A
The two words at top of stack are swapped.

ROT Rotate
… X A B C → … X B C A
The word third from the top of stack is placed at the top – the two words formerly at top of stack are pushed down.

PICK
… N → … V
The word at a depth of N in the stack is duplicated and place at the top of stack. Thus 1 PICK is equivalent to the word OVER.

TWIXT
… L H V → … R
Places the value "true" (-1) on the top of stack if the value at top of stack is between the lower and upper [inclusive] limits (3rd and 2nd on the stack), "false" (0) otherwise.

>R Onto-R
… X N → … X
The word at top of stack is removed and placed on the return stack.

R> R-Onto
… X → … X N
The word at top of the return stack is removed and placed on the stack.

I
… X N → … X
The word at top of the return stack is copied to the stack.

Input / Output

KEY
… X → … X V
This word causes the computer to wait until a value has been entered via the serial port. The value appears as the lower 8 bits in a 16 bit word at the top of stack..

EMIT
… X C → … X
This word writes a character to the serial port using the lower 8 bits of the 16 bit word at top of stack..

BASE
… X → … X A
This word places the address of the current number base on the stack. Base is a two-byte (word) variable.

H. H-Dot
… X V → … X
This word removes the number at top of stack and converts it to a string of four characters (in the current number base) which are then sent to the serial port. Note that the number is interpreted as an unsigned quantity.

D. D-Dot
… X V → … X
This word removes the signed double word at top of stack and converts it to a string of characters that are sent to the serial port. The Forth number base controls the characters used to represent of the number. The common Forth output word, "." (DOT), is simply defined as

```
: . S->D D. ;
```

To define a version of "DOT" that always produces a number in base 10

```
: .  BASE @  5 5 + BASE ! SWAP
     S->D D. BASE ! ;
```

Although this appears complicated, it simply saves the current number base on the stack, changes the base to 10, outputs the number, and then restores the base to its original value.

WORD
… X C → … X A
This word reads one word from the input stream to temporary area at the end of the dictionary. The count of the characters in the word (a byte) preceeds that word itself. The address of the preceeding count byte is left on the stack.

NUMBER
… X C → … X DL DH
This word converts the counted character string (its address is the argument on the stack) to a double word constant. If the character string contains an embedded decimal point, the count of the digits to the right of the decimal point is placed in the system (byte) variable, PRDP (places to the right of the decimal point).

Using # for formatted output

<# Into-Pound
Pound
#S Pound-S
SIGN Sign
#> Pound-Into
… X S L M → … X (If SIGN is used)
… X L M → … X (If SIGN is not used)

The "#" words are used to output a variable number of digits (n the current number base) of a double length argument. For example, the following will output the magnitude of a double length number as three digits:

<# # # # #> TYPE

To output a a signed number (rather than the magnitude), you must put the sign of the number on the stack underneath the double length number itself and use the "SIGN" word, as follows:

<# # # # SIGN #> TYPE

Remember that the current number base is used. To use base regardless of the currently set base, set the base to 10 and retore the previous base as is in the example shown in explanation of D.

TYPE Type
… X N A → … X
This word is generally used only with the "#" type of formatted output. It assumes that a string of N characters has been placed at the the address at top of stack and outputs the characters to the serial port.

." Dot-quote
This word defines a quoted string that begins after the space following the word, and ends with a quote. This word can only be used in word definitions. Thus the construct:

: msg ." A message " ;

when run, produces

msg A message OK

COUNT Count
This word assumes the address of a quoted string is on the stack. The quoted string must be in the form of a count, followed by the string itself. The address is placed by the count of the characters in the string

Condition Testing

Condition setting Forth words use –1 ($FFFF_h$) as true and 0 as false.
= Equals
… X A B → … X R
If the two numbers at top of stack are equal, they are replaced with true (-1) – otherwise they are replaced with false (0).

>= Greater Than or Equals
… X L R → … X R
If the number beneath the top of stack is greater than or equal to the number at top, they are replaced by true (-1) – otherwise they are replaced with false (0).

=0 Equals Zero
… X V → … X R
If the number at top of stack is zero, it is replaced with true (-1), otherwise it is replaced with false (0).

Program Flow Control

IF – THEN
IF – ELSE – THEN
… X V → … X

BEGIN – WHILE – REPEAT
The words within the begin/repeat sequence are repeated until the word on the stack tested by "while" is zero.

BEGIN – UNTIL
This words between the begin/until sequence are repeated until the word on the stack tested by "until" is true (not zero).

DO – LOOP
DO – +LOOP
… X H L → … X
These two constructs differ only in that +LOOP increments the loop count by the number found at the top of stack. At the start of a loop the high limit (next to top of stack) and start value (top of stack) are removed and placed on the return stack. Forth 83 conventions are used for controlling the looping process. Thus

10 0 DO … LOOP

will execute 10 times with indices 0, 1, …, 9. The construct

-10 0 DO .. –1 +LOOP

will execute 11 times with indices 0, -1, …, -9, -10.

I
… X → … X I
This word places a copy of the DO control (the top of the return stack) on the stack.

LEAVE
… X → … X

This word causes the loop to terminate at the next encounter with LOOP or +LOOP, regardless of the value of the loop index.

CASE
n1 OF ... ELSE
…
n2 OF ... ELSE
DROP ...
ENDCASE
… X V → … X

The "CASE" construct is a convenient way of writing a nested set of "IF" conditions. CASE assumes that a value to be tested is on the stack at the time CASE is encountered. Each "n OF ... ELSE"clause causes the initial value to be compared to n and the phrase following executed if equal. The DROP ... at the end is executed if no match occurs.

Screen Editor

The screen editor assumes VT220 emulation. Loading a code page (up to 1024 characters) can be done with the terminal emulators "transfer file" function. With Hyperterminal, this is Transfer > SendTextFile. To save a code page, use the Forth word "LIST" to list a page, use the mouse to mark the block, then paste it in your word processor. For example, to list the page at 9000 (hex), use:

```
9000 400 LIST
```

SCR
… X → … X A
This word places the address of the screen variable on the stack. To set the screen address to 8000h, use the words:
8000 SCR !

EDIT
This word displays an editable screen of the 1024 (400h) characters at the screen address. The editor responds to the cursor movement arrow keys, backspace, and return. Five control characters are also available: ^b to blank the page and insert the terminator word "**;S"** at the end of the page; ^n to insert a new line; ^y to delete an entire line, ^o to toggle the "insert/type-over" mode; and ^x to exit the editor.
Because of the processing necessary to process an inputted character in "insert" mode, change to "overwrite" when loading characters from a file.

ENS Edit next screen
EPS Edit prior screen
Start an edit of the next screen and prior screen, respectively.

ES Edit screen
This word assumes a screen address follows. The screen address is loaded into the screen variable and that screen is displayed for editing.

LSCR
This word attempts to load the current screen (and possibly subsequent screens) continuing until a "**;S"** control word is encountered.

\ Back-slash
This word causes the compiler to ignore the remainder of the current line. Thus this symbol can be used as the start of a comment at the end of a line.

;S Semicolon-S
This word terminates a compilation initiated by LSCR.

Native Code Definitions

Although most of the native code words correspond to their Z8 assembler counterparts, there are a few significant differences. Chief among these are the words for Load (LD).

In this implementation, LD is reserved for versions of instruction having opcodes of the form En and Fn (E3, E4, E5, E6, E7, F3 and F5). The "indexed" versions of load (opcodes C7 and D7) are given the special identifiers LDX and STX. The "group register" versions of "load" (r8, r9) have the identifiers LDGR and STGR respectively. The "load group register immediate" (opcodes "rC") has the identifier LDGRIM.

Another minor difference is in the definition of INC. This identifier refers only to opcodes 20h and 21h. The "group register" version of INC (Opcode "nE") is given the special identifier INCGR.

The "load constant" and "load external" mnemonics are also different from their assembler counterparts. (This is mainly done to clearly differentiate between "load group register from" and

"store group register to" versions of these instructions. The code "LDC" and "LDE" load from memory to a group register; "STC" and "STE" store a a group register value in memory. The auto-increment versions of these instructions follow the same convention. Thus we have:

Forth	Assembler
r6 r8 LDC,	LDC R6,@RR8
r8 r5 STC,	LDC @RR8,R5
r6 r8 LDCI,	LDCI R6,@RR8
r8 r5 STCI,	LDCI @RR8,R5

For the "load indexed" opcodes C7 and D7, the index register is specified following the source or destination address modified by the index. STX means store a group register into an indexed destination location; LDX means load a group register from an indexed source location. Thus we have:

Forth	Assembler
F0 r1 rA STX,	LD 240(R1), R10
rA F0 r1 LDX,	LD R10, 240(R1)

These compile into the three-byte sequences D7 A1 F0 and C7 A1 F0.

The Forth operand sequences correspond to the Z8 assembler conventions: all definitions are in destination/source order (when both are required) followed by an address mode indicator (when required). Thus, the code for "load register 9 from register 6 is

Forth	Assembler
9 6 RR LD,	LD 9,6

This produces the three-byte result E4 06 09.

Oak Ridge Forth reserves the words R0..R9, RA.. RF and their lower case counterparts, r0 … r9, rA .. rF for use in native code definitions. These correspond to references to registers in the current register group. For example, for "load group register 9 from 46h", one can write

Forth	Assembler
r9 46 LDGR,	LD R9, 46h

This will compile into the word 9846. The complementary operation, "store group register 9 in 46h" is

Forth	Assembler
46 r9 STGR,	LD 46h, R9

This compiles into the word 9946. The convention usually adopted is to use lower case "rn" when a four-bit register designation is referred to, and a capital R - "Rn" - when an eight bit group register reference in the form "En" is used. Also, remember that STGR can not reference a group register (Ex) as a destination.

Several special words are predefined to indicate the address mode for native code definitions. These are

Address Mode	Example	Code Produced
R	R2 R DEC,	00E2
IR	R2 IR DEC,	01E2
rr	r2 r3 rr ADD,	0223
rIr	r2 r3 rIr ADD,	0323
RR	47 32 RR ADD,	043247
RIR	47 32 RIR ADD,	053247
Irr	r6 r5 Irr LD,	F365
RR	46 RR INCW,	A046
RIM	45 7F RIM LD,	E6457F
IRIM	32 7F IRIM LD,	E7327F
IRR	32 45 IRR LD,	F54532

Note that the address modes must be stated in proper case.

The following table shows the acceptable address modes for instructions

	rr	rIr	RR	RIR	RIM	IRIM	
x1 ADC	x2 SUB		x4 OR		x6 TCM		xA CP
x0 ADD	x3 SBC		x5 AND		x7 TM		xB XOR

	RR	RIR	RIM	IRIM	IRR
E4,E5,E6,E7 LD					

R	IR				
B0,B1 CLR	60,61 COM	40,41 DA	00,01 DEC		
20,21 INC	50,51 POP	70,71 PUSH	90,91 RL		
10,11 RLC	E0,E1 RR	C0,C1 RRC	D0,D1 SRA		
F0,F1 SWAP					

rIr	Irr		
C2,D2 LDC	C3,D3 LDCI	82,92 LDE	83,93 LDEI
C2,D2 STC	C3,D3 STCI	82,92 STE	83,93 STEI

RR	IR		
80,81 DECW	A0,A1 INCW		

(No address mode used)			
nC LDGRIM	n8 LDGR	n9 STGR	nE INCGR
8F DI	9F EI	BF IRET	FF NOP
EF CF	CF RCF	AF RET	DF SCF
31 SRP	D6,D4 CALL		

The following codes are used for condition testing:

always

never	
carry	
nc	(no carry)
z	(zero)
nz	(non zero)
pl	(plus)
mi	(minus)
ov	(overflow)
nov	(no overflow)
eq	(equal)
ge	(greater or equal to)
lt	(less than)
gt	(greater than)
le	(less than or equal to)
uge	(unsigned greater or equal to)
ult	(unsigned less than)
ule	(unsigned less or equal to)
ne	(no equal)

These condition codes are intended for use with the structured control statements (if, else, then, etc.) that are discussed below. Note that the condition test codes are all lower case.

These words indicate the beginning and end of a Z8 native-code definition:

:A	Colon-A
EXIT,	Exit

The following lists the assembler mnemonics recognized by this Forth implementation:

ADC,	Add with carry
ADD,	Add
AND,	And
CALL,	Call
CCF,	Complement carry flag
CLR,	Clear
COM,	Complement
CP,	Compare
DA,	Decimal adjust
DEC,	Decrement
DECW,	Decrement word
DJNZ,	Decrement & jump non-zero
DI,	Disable interrupts
INC,	Increment
INCW,	Increment word
INCGR,	Increment group register
IRET,	Interrupt return
JP,	Jump indirect
LD,	Load (three byte or indirect)
LCE,	Load constant
LCEI,	Load constant & auto increment
LDGR,	Load group register
LDGRIM,	Load group register immediate
LDX,	Load indexed
LDE,	Load external
LDEI,	Load external & auto increment
NOP,	No operation
OR,	Or
POP,	Pop from stack
PUSH,	Push to stack
RCF,	Reset carry flag
RET,	Return from call
RL,	Rotate bits left arithmetic
RLC,	Rotate bits left through carry
RR,	Rotate bits right arithmetic
RRC,	Rotate bits right through carry
SBC,	Subtract with carry
SCF,	Set carry flag
SBC,	Subtract with carry
SRA,	Shift right arithmetic
SRP,	Set register pointer
STC,	Store constant
STCI,	Store constant & auto increment
STE,	Store external
STEI,	Store external & auto increment
STGR,	Store group register
STX,	Store indexed
SUB,	Subtract
SWAP,	Swap nibbles
TCM,	Test complement under mask
TM,	Test under mask
XOR,	Exclusive or

Notice the absence of the conditional (and unconditional) “jump” instructions – opcodes cD (three-byte instruction - two-byte destination) and cB (two byte instruction - one byte relative address). These were omitted with the intent is that users will avail themselves of the structured conditional statements provided (if, else, then, etc.). The only “jump” instruction included is the “jump indirect” (opcode 30h). The instruction DJNZ (decrement & jump if non-zero) is provided, but it is used as part of a structured construct. DJNZ is discussed in the next section.

A special native-code word, GF (GoForth), provides a link to a Forth definition from within a native code definition. Thus, one can use the constructs:

```
: msg  .” A little message ” ;
:A test  GF msg exit,
```

When executed, the native-code word definition “test” will print out “A little message” using the Forth word “msg”.

Native Code Program Flow Control

IF, – THEN,
if, -- then,

IF, – ELSE, – THEN,
if, -- else, -- then,

These native-code constructs are analogous to their conventional Forth counterparts, except that the condition-test indicator controls branching when the "if" is encountered. Secondly, the option of a two-byte address or relative address jump corresponds to the upper and lower case versions of the code.

For example, suppose group register 6 is to be loaded from 42 or 43 depending on whether group register 5 is zero or one. This could be written:

```
r5 r5 rr OR,
z if,
  r5 42 LDGRIM,
else,
  r5 43 LDGRIM,
```

In this example, the lower case "if, … else, … then," compiles into the instructions:

```
4255   OR   R4,R4
EB04   JR   NZ,$+4
5842   LD   R8,42h
8B02   JR   $+2
5843   LD   R8,43h
```

If upper-case versions are used, the code generated (stating at location 2008h) is:

```
2008    4255      OR   R4,R4
200A    ED2012    JP   NZ,$+4
200D    5842      LD   R8,42h
200F    8D2014    JP   $+2
2012    5843      LD   R8,43h
2014     …
```

The "decrement and jump non-zero" instruction is provided, but this instruction is used only with a structure. Thus to build a loop using this instruction, one uses the structure:

```
BEGIN,
   …
rx  DJNZ,
```

Note that both of words are suffixed with a comma. The instruction DJNZ, assumes a register designation is on the stack when it is compiled.

Here is a simple example of a complete native code program using the structure just given. It shows that the sum of the number from 0 to n (n odd) is n*(n+1)/2. This example for n=63 returns the result 7E0 (in decimal, 2016 = 63*32)

```
:A test
  r6 0 LDGRIM, r7 0 LDGRIM, r8 3F
LDGRIM,
    begin,
      r7 r8 rr ADD, r6 0 rIM ADC,
    r8 DJNZ,
```

In this example, the number is accumulated in the word r6/r7; r8 is the counter initialized to 63, and the result is pushed onto the stack, low byte first.

Simple loops are constructed with the words UNTIL, (upper-case for loops longer than 128 bytes) and until, (lower-case for loops 128 bytes or shorter). Similar to other Forth flow constrol words, a loop is set up

begin,
…
(condition code) **until,**

The condition code is the same as those used with "if, … then, … else," Thus, a simple loop to implement a "move characters" function (source, target, count on the stack) could be implemented as:

```
:A MoveChars
  rC R POP,  rD R POP,
  rA R POP,  rB R POP,  r8 R POP,  r9 R
POP,
    begin,
       r6 r8 LDC,  rA r6 STC,
       r8 RR INCW, rA RR INCW,  rC RR DECW,
```

An Interrupt Service Routine Example

The following example shows how to write a simple seconds display using the native code facility to implement an interrupt service routine. The program uses the T1 timer and its prescaler to furnish interrupts.

```
: TOUT DUP A BASE ! 0 <# # # #> TYPE 10 BASE ! D
EMIT ;
: M 5D 40 C! 41 C@ 1+ DUP 3C >= IF 3C - THEN TOUT 41
C! ;
:A EI EI, EXIT, :A DI DI, EXIT,
:A ISR 40 R DEC, z if, 50 r0 RR LD, 51 r1 RR LD, 50
SRP,
 GF M then, 30 SRP, IRET,
: INITSR DI
   0 F2 C!          \ zero T1
   3 F3 C!          \ zero PT1, int clock, set mod n
   8 F9 C!          \ IPR to A>B>C
  20 FB C!          \ interrupt on T1
   A F1 C!          \ enable T0 and T1
```

When the initialization routine INITSR is run, it disables interrupts and initializes the T1 counter and it's prescaler to 256 and 64 (their maximum counts), respectively. After INITSR is run, entering the Forth word EI enables interrupts and starts the timer interrupts. The interrupt service routine is in "native code" so that the service routine starting address (which is entered into the jump vector location 0E in the register file) is two more that the execution address of a normal Forth word definition. Note that the interrupt service routine uses the GOForth word to transfer control to a Forth-coded display routine when the timer furnishes an interrupt.

Locations 40 and 41 are used for a count-down accumulator and the elapsed seconds. The initial value of the accumulator is 93 ≈ 12,228,000 / (8*256*64). The interrupt routine ISR uses group registers 50h-5Fh for Forth system registers for the Forth code to display the count. The only register that has to be initialized in this new area is the return stack register, R0/R1 (located at 50h-51h).

Some Utility Routines

FILL
… T c F → …
Fill the target address with c bytes of the fill character F.

?S
… → …
Display the current status of the stack.

DUMPHEX
… T c → …
Dump c bytes starting at the target address as an Intel hex format suitable for ROM programming.

DUMP T
… → …
Dump 256 bytes starting at the target address in hex and as characters.

LIST
… T c → …
List c bytes starting at the target address T. The display is in a format that may be selected and copied into a text editor.

HIBYTE
… n → … R
The high byte of the double byte value at top of stack is placed in the low byte of the result; the high byte of the result is zero. Equivalent to dividing the (unsigned) value at top of stack by 256.

Compilation and Related Topics

: Colon
; Semi-colon
These words indicate the beginning and end of a Forth word definition.

CONSTANT
This word compiles the word following as a constant. For example,

```
471B CONSTANT K4
```

creates the constant K4 with the value 471B. The constant uses only two bytes when compiled rather than the four bytes usually required for a constant. However, the definition itself requires nine bytes (5 bytes for the head, four bytes for the body), so the gain occurs only if the constant is used many times in a program.

′ TIC
… T → … T A (If not compiling)
If compiling, this word causes code to be generated that will place the execution location of the following word on the stack when TIC is executed. If not compiling, the address of the word following is placed on the stack. This word uses Forth 83 convention of producing the execution address of a Forth word.

, COMMA
… X A → … X
Store the word on the stack at HERE and advance HERE by 2.

C, C-COMMA
… X C → … X

Store the low byte of the word on the stack at HERE and advance HERE by 1.

COMPILE
When executed, this word will compile the following word into the Forth definition being constructed.

[COMPILE]
This word causes the following immediate word to be compiled (as if it was not immediate) rather than immediately executed.

[Into immediate
] Into compile
These words cause the compiler to temporarily return to immediate mode and return to compile mode.

CREATE
This word places the succeeding Forth word in the dictionary. The "smudge" bit is set, meaning that the word can't be executed.

SMUDGE
This routine simply resets the "smudge" bit of the most recently defined Forth word, thus making it executable.

<BUILDS
>DOES
These two words are used to create a special compiling sequence. It can be thought of as an "enhanced" version of the ": … ;" (colon .. semicolon) type of definition. An extensive discussion regarding this pair of words is in Brodie's *Starting Forth*.

ALLOT
… X V → … X
This word increments the dictionary pointer by the word at top of stack, thus reserving space in the dictionary.

I²C EEPROM Utilities

The EEPROM utilities assume a page identifier has been set before the calls are made. The identifier, located at the base system address (in the distribution version this variable is at location 10_h), contains the bits "ORed" into the poll byte (A0 or A1) to identify the page address. The value in the identifier is twice the page address.

POLL
… X → … X R
This routine polls the EEPROM as many as 256 times seeking a true (ACK) response. The result (true -1 or false - 0) is left on the stack.

PNWOS Place N words on stack
… X A N → … X Rn Rn-1 ... R1 F
This routine reads N words from the EEPROM starting with the address at next to top of stack. If successful, the resulting N words are placed on the stack. The validity of the result of the read (true [-1] or false [0]) is left at the top of the stack. The top-most result word is the lowest address read.

TNBTR Transfer N bytes to RAM
… X Ra Ea N → … X F
This routine transfers N bytes from EEPROM to RAM starting with the EEPROM address given underneath the top of stack. at third from top of stack. If successful, the resulting N words are placed in RAM starting at third from top of stack. The validity of the result of the read (true [-1] or false [0]) is left at the top of the stack.

TNBFR Transfer N bytes from RAM
… X Ra Ea N→ … X F
This routine transfers N bytes to the EEPROM from RAM starting with the RAM address at third from top of stack. If successful, the resulting N words are placed in EEPROM memory starting at the address given underneath the top of stack. The validity of the result of the write (true [-1] or false [0]) is left at the top of the stack. The writing assumes the EEPROM can be written in blocks up to 128 bytes long..

PCF 8574 I²C/Parallel Utility

PIA Read/Write a byte to 8574 interface
… X B → … X F
This routine is used to read or write a byte to a PCF 8574 serial/parallel interface. The routine uses the I²C device identifier located at the base system address (in the distribution version this variable is at location 10_h). This identifier should be 40, 42,...4E for the PCF8574 interface and 70, 72,...7E for the PCF8574A.

If writing, the byte to be written is in the low byte of the argument on the stack. The high byte of the argument is FF. The result word on the stack is FFFF is an ACK was received, 0 otherwise.

If reading, the argument byte must contain FE in

the high byte of the argument on the stack. The result will be in the low byte of the word returned on the stack. The high byte of the result is FF is an ACK was received, 0 otherwise.

An argument of 0 (as a word on the stack) is used to simply poll the device. A return of FFFF means an ACK was received, 0 otherwise.

LCD Implementation

This example uses the 8574A to provide an interface to a HD44780-based LCD character display. The interface uses a 4-bit mode method of communication. Lines A0-A3 of the 8574A are connected to the four high LCD addresses, LCD pins 11-14. 8574A lines A4-A6 are connected to the LCD Enable, R/W and RS (Register select) pins. The address of the 8574 is taken from the I^2C device identifier located at the base system address (in the distribution version this variable is at location 10_h). In the following example, the 8574A address of the LCD interface is 70 (hex).

```
:A S r8 R POP, r9 R POP, r9 R SWAP, r9 r PUSH, r8 R
PUSH, EXIT,
: PY PIO DROP ; : PX HERE C@ OR PY ;
: 1LSET 70 10 C! FE82 PY FE92 PY FE82 PY ;
: NOUT FE80 OR DUP DUP PX 10 OR PX PX ;
: NIN FEAF PY FEBF PY FFFF PIO FEAF PY ;
: WLCD SWAP HERE C! DUP S 0F AND NOUT 0F AND NOUT ;
: LCDON 0 0F WLCD ; : 2LSET 1LSET 0 28 WLCD ;
: SETADD 0 SWAP 80 OR WLCD ; : CLEAR 0 1 WLCD ;
: RBF NIN NIN 0F AND SWAP S F0 AND OR ;
: X1 0 4 WLCD RBF OVER + 1 - DUP >R SETADD
   0 DO 40 SWAP WLCD LOOP R> 1+ SETADD ;
:A PT r6 R CLR, r9 r2 LDC, rA r9 STGR, r2 R INCW, begin,
  r7 r2 LDC, r7 R PUSH, r6 R PUSH, r2 R INCW, r9 DJNZ,
  rA R PUSH, r6 R PUSH, EXIT,
: OUTTOLCD 0 DO I OVER + C@ 40 SWAP WLCD LOOP DROP ;
: H.LCD RBF SWAP 0 <# # # # # #> OUTTOLCD 4 + SETADD ;
```

In this implementation the error flag returned by the PIO interface (the result of the call to PIO in the word PY) is simply ignored.

1LSET 2LSET Initialize a 1-line or 2-line LCD

These routines initialize a one line or two line mode of the LCD. The initialization will blank the LCD, but will not turn it on. One of these routines must be the first call to the LCD interface and may not be used again after that first call.

LCDON Turn on the LCD

This routine turns on the LCD and displays a blinking cursor at position 0.

SETADD Move the LCD cursor

… X L → … X

The LCD cursor is moved to the location given in the low byte of the word at top-of-stack. The first line of the display starts at location 0; the second line (if available) starts at location 40 (hex).

CLEAR Clear the LCD display

The LCD display is cleared and the cursor is returned to address 0.

WLCD Write byte to an LCD

… X C B → … X

This routine writes the low byte of the word on the stack to the character LCD via an 8574. The low byte in the word at next-to-top of stack is specifies the Register Select (RS) and R/W signals:

RS	R/W	Argument
0	0	0
0	1	2
1	0	4
1	1	6

Thus, to write a "#" character (23 hex) on the LCD at the current character position, use

```
40 23 WLCD
```

RBF Read the Busy Flag and current address

… X → … X B

This routine reads the busy flag of the LCD. The high bit of the low byte of the word on the stack is the busy flag. The remainder of the byte is the current character address of the LCD.

PT ... X1 Write a character string to the LCD

When executed, these routines write a space-delimited string to the LCD at the LCD's current cursor position. This routine is the LCD equivalent of **."** word for the serial output. To write the string "Z8 Forth driver" starting at location 0 use:

```
: Test 0 SETADD
    PTX STRING Z8 Forth Driver" X1 ;
```

The cursor is positioned at the end of the displayed string. The word STRING compiles the succeeding characters up to a quote; the routine PTX places the compiled characters on the stack; and the routine X1 sends the characters on the stack to the display.

PC Keyboard Utility

GAL Get a line from the keyboard

… X A → … X

This routine accepts lower and upper case

characters from the keyboard and terminates when ENTER is keyed. The functions associated with the ALT and CTRL key, the keys on the alternate keypad, and the cursor control keys (home, end, arrow keys, etc.) are not recognized. The interface assumes that the data and clock lines are brought into ports 3,1 and 3,2 and that each line has a 10K pullup. This use of Port 3 input lines is compatible with the EEPROM and 8574 interface routine implementations of the I^2C interface. Locations 40-4F of the register file are used by the keyboard interrupt processor of the GAL routine. The result of the line of input is stored at HERE.

The routine requires one argument on the stack. This argument is the address of a routine that will echo input to some display device as characters are typed on the PC keyboard. A simple example to echo characters to the serial line is to provide the address of EMIT. Thus to supply the address of EMIT use
' EMIT.

A Forth word to collect a line of input and simply print it out (again) after return from the GAL routines is:

```
: X ' EMIT GAL CR HERE C@ 1
    DO HERE I + C@ EMIT LOOP CR ;
```

Fast Multiply/Divide in the Z86C93

The Z86C93 provides a fast 16x16 unsigned multiply and 32x16 divide. To show how these functions as used, below is an implementation of a 16-bit sin approximation originally given for the Analog Devices ADSP-2100. This algorithm produces a 16 bit sine for a 16-bit argument. The argument represents each quadrant as a 4000 bit quantity. Thus the first quadrant (0-90º) is 0-4000h. The second is 4000h-7FFFh; the third 8000h-C000h, and the fourth is C000-0. Similarly, the result is scaled by 7FFFh, so the sine of 4000h is 7FFFh. The approximation is accurate ±2.

The Forth ISIN program, which is shown below, is an example of how to code mainly in assembler using Forth words.

```
:A MS rE R POP, rF R POP, FD 0E RIM OR,
 4 28 RR LD, 5 29 RR LD, 2 2A RR LD, 3 2B RR LD,
 6 40 RIM LD, NOP, NOP, NOP, 0 R PUSH, 1 R PUSH,
 2 R PUSH, FD F0 RIM AND, rE JP,

:A MF FD 0E RIM OR, 4 22 RR LD, 5 23 RR LD,
 2 28 RR LD, 3 29 RR LD, 6 40 RIM LD, NOP, NOP,
 NOP, 2 80 RIM TM, nz if, 0 R INCW, then,
 29 1 RR LD, 28 0 RR LD, FD F0 RIM AND, RET,
: 'MS ' MS 2 + CALL, ; : 'MF ' MF 2 + CALL, ;

:A AX3 rE R POP, rF R POP,  rA R POP,
 rD rA rr ADD, rA R POP, rC rA rr ADC, rA R POP,
 rB rA rr ADC,  rE JP, : 'AX3 ' AX3 2 + CALL, ;

:A ISIN 22 R POP, 23 R POP, rA 22 LDGR, rA R RL,
 FC rA STGR, 22 40 RIM TM, nz if, 22 R COM,
 23 R COM, 22 R INCW, then, 23 23 RR ADD,
 22 22 RR ADC, 22 R DECW, mi if, 22 R INCW, then,
 28 22 RR LD, 29 23 RR LD, 2A 32 RIM LD,
 2B 40 RIM LD, 'MS rD R POP, rC R POP, rB R POP,
 'MF 2A 00 RIM LD, 2B 53 RIM LD, 'MS 'AX3 'MF
 2A 55 RIM LD, 2B 34 RIM LD, 'MS rA R POP,
 rD rA rr SUB, rA R POP, rC rA rr SBC, rA R POP,
 rB rA rr SBC, 'MF 2A 08 RIM LD, 2B B7 RIM LD,
 'MS 'AX3 'MF 2A 1C RIM LD, 2B CE RIM LD, 'MS
 'AX3 rD 10 RIM ADD, rC 0 RIM ADC, rB 0 RIM ADC,
 rA 5 LDGRIM, begin, rB R SRA, rC R RRC, rD R RRC,
 rA DJNZ, rC 80 RIM TM, nz if, rC 7F RIM LD,
 rD FF RIM LD, then,  FC 1 RIM TM, nz if,
 rD R COM, rC R COM, rC R INCW, then, rD R PUSH,
 rC R PUSH, EXIT,
```

On a 12 Mhz Z8, this approximation calculates a sine in about 412μsec.

Fast Fourier Transform – A More Elaborate Example Using the Z86C93 Fast Multiply/Divide

Oak Ridge Forth includes a fast Fourier transform routine(FFT) that uses the fast multiply of the Z86C93. This routine is mostly written in the Forth version of the Z8 assembly language and incorporates a version of the integer sine routine just described. The FFT calculates transforms for $3 \le p \le 7$, where the number of points, n, is $n = 2^p$ (n = 8, 16, 32, 64, 128). As written, the data is located at page A0 (A000-A0FF) and the imaginary part (the result) at page A1 (A100-A1FF). The lower half of page A2 (A200-A27F) is temporary storage for sines and cosines. In addition to the standard Forth user area (the default is locations 30-3F of the register file), the FFT uses register file locations D0-DF.

The version of the integer sine routine in the FFT assumes an argument $0 \le arg \le FFFF$ that corresponds to the range $0 \le angle \le \pi$.

The results in page A1 appear in the conventional FFT order. Beginning in A100, the words in the result correspond to the frequencies f=0, 1/NΔ, 2/NΔ, 3/NΔ, ... (N/2-1)/NΔ,
±1/NΔ, -(N/2-1)/NΔ, ..., -1/NΔ.

On a 12 Mhz Z86C93 the FFT calculates a 128-point transform in about 500 ms. The FFT program itself requires 1400 bytes.

Floating Point Package

The floating point package is analogous to the integer routines. In general, the name for a floating point function is the same as the integer routine except for an appended leading “F”:

F+	Floating Add
F-	Floating Subtractions
F*	Floating Multiply
F/	Floating Divide
FDUP	Floating DUP
FOVER	Floating OVER
FDROP	Floating DROP
FSWAP	Floating SWAP
FABS	Floating ABS
FNEG	Floating Negate
F>=	Floating >=
D->F	Double integer to float
F->D	Float to double integer
FNAN?	Floating NotANumber?
F.	Output in ±nnn.nnn format
E.	Output in ±n.nnnnn±n format
FCON	Floating constant definition

The routines assume floating point variables are on the stack. If the routine requires two arguments, the argument at top of stack is the second argument.

The output word F. requires a format word on the stack. The format word is in the form mmnn where mm is the width of the output field and nn is the number of places to the right of a decimal. Thus the format word 0702 specifies a field 7 characters wide and two places to the right of the decimal. For example, the following sequence

FCON 3.14159 0702 F.

produces the result: 3.14 in a field seven characters wide. The F. routine produces results with up to 8 decimal digits. If a number requires more than 8 digits, the result field is filled with asterisks. Note that the floating output routines always produce a base 10 result.

The IEEE 754 floating point format defines an eight bit exponent (offset by 128) and 24 bit mantissa. This definition provides 6 to 7 significant digits in a range of approximately $10^{\pm 38}$. Examples of numbers in this format are:

1.0	8140 0000
-1.0	81C0 0000
10.0	8450 0000
Pi	8264 87EE

Underflows are set to zero; overflows are set to NotANumber, which is FF80 0000. An attempt to divide by zero also results in a NAN result.

The routine FCON is used to convert a following (base 10) number to floating point format. This routine is similar to LITERAL: In compile mode FCON compiles a floating constant definition into the current word; if in immediate mode the floating constant is left on the stack. For example, the definition:

```
: T FCON 3.14159 ;
```

compiles a constant definition into the definition of T so that when the word is executed it loads the floating equivalent of 3.14159 onto the stack. Numbers defined by this routine are limited to those expressible in 6 hex digits, that is ±8388607, regardless of the placement of the decimal point. For numbers outside this range, use the E. output word.

```
 \ FFT  9am  5 February 2005

 \ Auxiliary routines for inner loop
 \ Add X*Y to an accumulator.
 \ X and Y are assumed already loaded in multiply registers
 \ the Accumulator is three bytes: D0, D1, and D2.
:A MABCD r0 r2 LDGR, r0 r4 rr XOR, r0 R RL, FC 0 STGR,          \ Set UF1=1 if different signs, else 0
 r2 80 RIM TM, nz if, r2 R COM, r3 R COM, r2 R INCW,           \ Negate X if necessary
  r2 80 RIM TM, nz if, r2 R DECW, then, then,
 r4 80 RIM TM, nz if, r4 R COM, r5 R COM, r4 R INCW,           \ Negate Y if necessary
  r4 80 RIM TM, nz if, r4 R DECW, then, then,
 r6 40 LDGRIM, NOP, NOP, NOP, FC 1 RIM TM, z if,               \ UF1 will control whether to add to or
 D3 r3 RR ADD, D2 r2 RR ADC, D1 r1 RR ADC, D0 r0 RR ADC, else, \  subtract from the accumulator
 D3 r3 RR SUB, D2 r2 RR SBC, D1 r1 RR SBC, D0 r0 RR SBC, then,
 RET, : 'M ' MABCD 2 + CALL, ;

 \ Negates the sin stored in D8,D9
:A SS D8 R COM, D9 R COM, D8 R INCW, RET, : 'SS ' SS 2 + CALL, ;

 \ Clears the accumulator
:A ZD D0 R CLR, D1 R CLR, D2 R CLR, D3 R CLR, RET, : 'ZD  ' ZD 2 + CALL,

 \ Inner code
 \ D0-D3 cross product and ti:D0-D1, f:D2-D3, frj,qr:D4-D5, fij,qi:D6-D7,
 \ sin:D8-D9, cos:DA-DB, tr:DC-DD, i:DE, j:DF, rErF: vector
 \ r9 temp, i:rA-rB, j:rC-rD, A000: fr[]; A100: fi[]
 \ STACK (not changed)
:A IC FD R PUSH, D0 SRP,                                       \save RP, set RP=D0
 rF rF rr ADD, rF R PUSH, rE rE rr ADD, rE R PUSH,             \save 2j, save 2i

 rE A0 LDGRIM, r4 rE LDC, rF R INC, r5 rE LDC, rE A1 LDGRIM,   \frj=fr[j];
 r7 rE LDC, rE R DEC, r6 rE LDC,                               \fij=fi[j]

 0E SRP, 'ZD                                                   \RP=0E, tr=0
 r2 DA LDGR, r3 DB LDGR, r4 D4 LDGR, r5 D5 LDGR, 'M            \tr=c*fr
 r2 D8 LDGR, r3 D9 LDGR, r4 D6 LDGR, r5 D7 LDGR, 'M            \tr=tr+s*fi
 DC D0 RR LD, DD D1 RR LD, 'SS 'ZD                             \save tr, sin=-sin, ti=0
 r2 DA LDGR, r3 DB LDGR, r4 D6 LDGR, r5 D7 LDGR, 'M            \ti=c*fi
 r2 D8 LDGR, r3 D9 LDGR, r4 D4 LDGR, r5 D5 LDGR, 'M 'SS        \ti=ti-s*fr, sin=-sin

 D0 SRP, rF R POP,                                             \RP=D0, rrEF=A1+2*i
 r6 rE LDC, rF R INC, r7 rE LDC, rE A0 LDGRIM,                 \qi=fi[i]
 r5 rE LDC, rF R DEC, r4 rE LDC,                               \qr=fr[i]

 r4 R SRA, r5 R RRC, r6 R SRA, r7 R RRC,                       \qr=qr>>1,qi=qi>>1

 r2 r4 LDGR, r3 r5 LDGR, r3 rD rr ADD, r2 rC rr ADC,           \f=qr+tr
 rE r2 STC, rF R INC, rE r3 STC, rE A1 LDGRIM,                 \store f in fr[i], point to A1+2i+1
 r2 r6 LDGR, r3 r7 LDGR, r3 r1 rr ADD, r2 r0 rr ADC,           \f=qi+ti in fi[i]
 rE r3 STC, rE R DEC, rE r2 STC, rF R POP,                     \store f, rrEF=A1+2*j

 r7 r1 rr SUB, r6 r0 rr SBC, rE r6 STC, rF R INC, rE r7 STC,   \qi=qi-ti, store in fi[j]
 rE A0 LDGRIM,                                                 \point to A0+2j
 r5 rD rr SUB, r4 rC rr SBC, rE r5 STC, rF R DEC, rE r4 STC,   \qr=qr-tr, store in fr[j]
 FD R POP, RET, : 'IC ' IC 2 + CALL, ;                         \restore RP

 \ Middle code
 \ Default RP
 \ stack: k, istep, L, n=2^p, m <top> --> stack: k, istep, L, n=2^p
 \ rErF: vector & i,j, rD:L,  rC:istep,  rB:n,  rA:k,  r9:m,
:A MC rE R POP, r9 R POP, rE R POP, rB R POP,                  \r9=m, rB=n=2^p, rD=L, rC=istep, rA=n,
 rE R POP, rD R POP, rE R POP, rC R POP, rE R POP, rA R POP,   \rB=k, r9=m, rD=L, rC=istep, rA=k,
 FF 8 RIM SUB, FE 0 RIM SBC,                                   \ preserve k, istep, L, n=2^p
 rF r9 LDGR, rA R INC,                                         \rF=m
 begin, rF rF rr ADD, rA DJNZ,                                 \2j=m<<2k,
 DE rF STGR, FD R PUSH, D0 SRP,                                \save RP, set RP to D0
 rE A2 LDGRIM,  r8 rE LDC, rF R INC, r9 rE LDC,                \load sin(j) and cos(j)
 rF 80 RIM ADD, rB rE LDC, rF R DEC, rA rE LDC, FD R POP,      \restore RP
 begin, DE r9 STGR, DF r9 STGR, DF rD RR ADD, 'IC r9 rC ADD,   \i=m, start inner loop, DE:i=m, DF:j=i+L
 nc if, r9 rB rr CP, SWAP uge until, then, EXIT,               \do IC, i=i+istep, i<n?

  \ Auxiliary routines for isin:
:A MS r4 D6 LDGR, r5 D7 LDGR, r6 40 LDGRIM, NOP, NOP, RET,
:A MF r2 D4 LDGR, r3 D5 LDGR, r4 D6 LDGR, r5 D7 LDGR,
 r6 40 LDGRIM, NOP, NOP, NOP, D6 r0 STGR, D7 r1 STGR, RET,
: 'MS ' MS 2 + CALL, ; : 'MF ' MF 2 + CALL, ;
:A AX3 D2 r2 RR ADD, D1 r1 RR ADC, D0 r0 RR ADC, RET,
: 'AX3 ' AX3 2 + CALL, ;
: PP SWAP r2 SWAP LDGRIM, r3 SWAP LDGRIM, ;

 \ isin assumes the argument 0-FFFF (0<=arg<pi/2).
 \  argument is in D4D5, result is left in D0D1
 :A isin D4 80 RIM TM, nz if, D4 R COM, D5 R COM, D4 R INCW, mi if,
 D4 R DECW, then, then, D6 D4 RR LD, D7 D5 RR LD, FD R PUSH,
```

```
 FD 0E RIM LD, 32 40 PP 'MS D0 r0 STGR, D1 r1 STGR, D2 r2 STGR,
 'MF 00 53 PP 'MS 'AX3 'MF 55 34 PP 'MS D2 r2 RR SUB,
 D1 r1 RR SBC, D0 r0 RR SBC, 'MF 08 B7 PP 'MS 'AX3 'MF 1C CE PP
 'MS 'AX3 FD R POP, D2 10 RIM ADD, D1 0 RIM ADC, D0 0 RIM ADC,
 rA 3 LDGRIM, begin, D2 D2 RR ADD, D1 R RLC, D0 R RLC, rA DJNZ,
 D0 80 RIM TM, nz if, D0 7F RIM LD, D1 FF RIM LD, then, RET,
: 'isin ' isin 2 + CALL, ;

 \ Interface to isin for FORTH
:A asin D4 R POP, D5 R POP, 'isin D1 R PUSH, D0 R PUSH, EXIT,

 \ icos assumes the argument 0-FFFF (0<=arg<pi/2)
:A icos D4 80 RIM TM, z if, D4 R COM, D5 R COM, D4 7F RIM AND,
 'isin else, D4 7F RIM AND, 'isin D1 R COM, D0 R COM, D0 R INCW,
 then, RET, : 'icos ' icos 2 + CALL, ;

 \ Initialize sin/cos values in A200 & A280
 \ Stack: ..., p, 2^p <TOS> --> ...
:A INIT DE R POP, DF R POP, DE R POP, DE R POP, FD R PUSH,
 D0 SRP, rF R SRA, rE R DEC, r9 R CLR, SCF, begin,
 r9 R RRC, rE DJNZ, rC A2 LDGRIM, rD R CLR, rE R CLR,
 r4 R CLR, r5 R CLR, begin r4 rE LDGR, r5 R CLR, 'isin
 rC r0 STC, rD R INC, rC r1 STC, r4 rE LDGR, r5 R CLR, 'icos
 rD 80 RIM ADD, rC r1 STC, rD R DEC, rC r0 STC, rD 7E RIM SUB,
 rE r9 rr ADD, rF DJNZ, FD R POP, EXIT,

 \ Re-order data A000-A0FF. A100-A1FF (imaginary part) is zero
 \ STACK: ..., n=2^p --> ...
 \ rF: nn=n-1, rCrD:mvec, rArB:mrvec, r8r9: temps, r7:mr, r6:m
 \ r9:L, r8: cntr  Imaginary part assumed all zero
 \ rF:n=2^m,  rE:n-1,  rCrD:kvec,  rB:kr,  rArB:krvec,
:A ROD rF R POP, rF R POP, rF R DEC, r7 R CLR, r6 1 LDGRIM,
 rA A0 LDGRIM, rC A0 LDGRIM, begin, r9 rF LDGR, begin,
 r9 R SRA, r8 r7 LDGR, r8 r9 rr ADD, r8 rF rr CP,
 ult until, r7 r9 rr AND, r7 r9 rr ADD, r7 R INC,
 r7 r6 rr CP, gt if, rB r7 LDGR, rB rB rr ADD, rD r6 LDGR,
 rD rD rr ADD, r8 rC LDC, r9 rA LDC, rC r9 STC, rA r8 STC,
 rD R INC, rB R INC, r8 rC LDC, r9 rA LDC, rC r9 STC,
 rA r8 STC, then, r6 R INC, r6 rF rr CP, eq until, EXIT, ;S

 \ Use sin for real (A000) data. Zero the imaginary (A100) part
 \ Stack: p --> p
: DATA 1 OVER 0 DO DUP + LOOP OVER 8000 SWAP 2 DO 0 2 U/ SWAP
 DROP LOOP OVER 0 SWAP 0 DO DUP asin A000 I + I + ! OVER + LOOP
 DROP DROP DUP DUP 0 DO A000 I + I + @ NEGATE OVER A000 + I +
 I + ! LOOP DUP + 0 DO 0 A100 I + ! 2 +LOOP DROP ; ;S

 \ Main FFT routine
 \ Stack: p where there are n=2^p data points in A000
: FFT DUP 0 1 ROT ROT DO DUP + LOOP          \ stack: p, n
 OVER OVER INIT DUP ROD HERE !               \ initialize sin/cos, generate data, re-order data
 1 - 1 BEGIN DUP DUP + SWAP HERE @ OVER 0    \ stack: k=k-1 -> k, L=1 -> k, L, istep=2*L ->
                                             \           k, istep, L, 0  Begin middle loop
 DO I MC LOOP DROP                           \ stack: k, istep, L, n=2^p, m; drop n=2^p
 DROP SWAP 1 - SWAP DUP HERE @ >=            \ drop old L, istep is new L, k=k-1, copy L, L>=n?
 UNTIL ;                                     \ k, L=istep, test -> k, istep
```

```
  Floating Point Package  2pm 16 April 2005

: NAN? 0 0080 D+ OR 0= ; : FDROP DROP DROP ; : FNAN 0 FF80 ;
: -ROT SWAP >R SWAP R> ; : FSWAP >R -ROT R> -ROT ; : FDUP OVER
 OVER ; :A FNEG rC R POP, rD R POP, rE R POP, rF R POP,
 rF R COM, rE R COM, rD R COM, rE R INCW, z if, rD R INC, then,
 rF R PUSH, rE R PUSH, rD R PUSH, rC R PUSH, EXIT, : FOVER >R
 >R FDUP R> -ROT R> -ROT ; : BV? FOVER FOVER NAN? -ROT NAN? OR
 IF FDROP FDROP FNAN 0 ELSE -1 THEN ; :A FN? rC R POP,
 rD R POP, rD R PUSH, rC R PUSH, rD 80 RIM AND, rD R PUSH,
 rD R PUSH, EXIT, : SABS FN? IF FNEG 1 ELSE 0 THEN ; : MBP SABS
 >R FSWAP SABS >R FSWAP R> R> - ; :A FNORMX rC R POP, rD R POP,
 rE R POP, rF R POP, rB 19 LDGRIM, begin, rD 40 RIM CP, ult if,
 RCF, rF R RLC, rE R RLC, rD R RLC, rC R DEC, SWAP rB DJNZ,
 then, rB rB rr OR, z if, rC R CLR, then, rF R PUSH, rE R PUSH,
 rD R PUSH, rC R PUSH, EXIT,
: FNORM FN? IF FNEG FNORMX FNEG ELSE FNORMX THEN ;

:A MDP r4 R POP, r5 R POP, rB R CLR, rC R POP, rD R POP,
 rE R POP, rF R POP, r6 R POP, r7 R POP, r8 R POP, r9 R POP,
 r6 R PUSH, rC R PUSH, rF rF rr ADD, rE rE rr ADC, rD rD rr ADC,
 r9 r9 rr ADD, r8 r8 rr ADC, r7 r7 rr ADC, rA R CLR, rC R CLR,
 R4 JP, : 'P ' MDP 2 + CALL, ;
:A MDE r4 R POP, r5 R POP, rA 80 RIM TM, nz if, RCF, rA R RRC,
 rB R RRC, rC R RRC, r8 R INCW, then,
 r8 r8 rr OR, nz if, r6 R PUSH, r6 R PUSH, r8 80 RIM TM,
 nz if, r6 R PUSH, r6 R PUSH, else, r6 80 LDGRIM, r6 R PUSH,
 r6 FF LDGRIM, r6 R PUSH, then, else, rC R PUSH, rB R PUSH,
 rA R PUSH, r9 R PUSH, then, EXIT, : 'E ' MDE 2 + CALL, ;

: D->F DUP HIBYTE 0= OVER HIBYTE FF = + 1+ IF
 FDROP FNAN ELSE FF AND 9700 OR FNORM THEN ;
: FABS FDUP NAN? 0= IF SABS DROP THEN ;
:A FMULX 'P r6 18 LDGRIM, begin, RCF,
 r9 1 RIM TM, nz if, rC rF rr ADD, rB rE rr ADC, rA rD rr ADC,
 then, rA R RRC, rB R RRC, rC R RRC, r7 R RRC, r8 R RRC,
 r9 R RRC, r6 DJNZ, r8 R CLR, r9 R POP, r7 R POP, r9 r7 rr ADD,
 r8 r6 rr ADC, r9 81 RIM SUB, r8 r6 rr SBC, 'E
: F* BV? IF MBP >R FMULX R> IF FNEG THEN THEN ;
:A DBY2 r6 R POP, r7 R POP, r8 R POP, r9 R POP, RCF,
 r6 R RRC, r7 R RRC, r8 R RRC, r9 R RRC, r9 R PUSH, r8 R PUSH,
 r7 R PUSH, r6 R PUSH, EXIT,

:A FDIVX 'P r5 R CLR, r6 19 LDGRIM, begin, r9 rF rr SUB,
 r8 rE rr SBC, r7 rD rr SBC, CCF, r5 1 RIM TM, nz if, SCF,
 else, nc if, r9 rF rr ADD, r8 rE rr ADC, r7 rD rr ADC, RCF,
 then, then, r4 FC LDGR, r6 1 RIM CP, ne if, FC r4 STGR,
 rC R RLC, rB R RLC, rA R RLC, r9 R RLC, r8 R RLC, r7 R RLC,
 r5 R RLC, SWAP r6 DJNZ, then, r8 R CLR, r6 R CLR, FC r4 STGR,
 carry if, rC 1 RIM ADD, rB r8 rr ADC, rA r8 rr ADC, then,
 r7 R POP, r9 R POP, r9 r7 rr SUB, r8 r6 rr SBC,
 r9 80 RIM ADD, r8 r6 rr ADC, 'E : F/ BV? IF FDUP OR IF
 MBP >R FDIVX R> IF FNEG THEN ELSE FDROP FDROP FNAN THEN THEN ;
: MB10 FDUP D+ FDUP >R >R FDUP D+ FDUP D+ R> R> D+ ;
: F->D FDUP NAN? IF DROP 0 8000 ELSE DUP FF AND DUP 80
 AND IF FF00 OR THEN SWAP HIBYTE 97 SWAP - DUP 0 >= IF
 DUP 1 >= IF 0 DO DBY2 LOOP THEN ELSE FDROP 0 0 THEN THEN ;

:A FADDX r8 R POP, r9 R POP, rA R POP, rB R POP, rC R POP,
 rD R POP, rE R POP, rF R POP, r6 FD LDGR, r6 8 RIM ADD,
 r8 rC rr CP, ult if, r7 rC LDGR, r7 r8 rr SUB, r7 16 RIM CP,
 ugt if, rF R PUSH, rE R PUSH, rD R PUSH, rC R PUSH, EXIT, then,
 else, r6 4 RIM ADD, r7 r8 LDGR, r7 rC rr SUB, r7 16 RIM CP,
 ugt if, rB R PUSH, rA R PUSH, r9 R PUSH, r8 R PUSH, EXIT,
 then, then, r4 r6 LDGR, r6 R INC, r5 r6 LDGR, r6 R INC,
 r7 r6 LDGR, r7 R INC, begin, r8 rC rr CP, ne if, r4 IR INC,
 r5 IR SRA, r6 IR RRC, r7 IR RRC, SWAP never until, then,
 r4 rD LDGR, r5 r9 LDGR, r4 R RL, r5 R RL, rF rB rr ADD,
 rE rA rr ADC, rD r9 rr ADC, r5 r4 rr ADC, r5 R RRC, rD R RRC,
 rE R RRC, rF R RRC, rC R INC,
 rF R PUSH, rE R PUSH, rD R PUSH, rC R PUSH, EXIT,
: F+ BV? IF FADDX FN? IF FNEG FNORMX FNEG ELSE
 FNORMX THEN THEN ; : F- FNEG F+ ;

: F>= BV? IF FNEG F+ SWAP DROP 80 AND 0= ELSE FDROP
 0 THEN ;
: F. -ROT FDUP NAN? IF ." NAN" DROP FDROP ELSE ROT DUP
 HIBYTE >R OVER 80 AND 7F81 + >R FF AND >R FABS 0 8140 I IF I
 0 DO 0 8450 FMULX LOOP THEN FMULX DUP HIBYTE 97 >= IF R>
 DROP R> DROP R> 0 DO 2A EMIT LOOP FDROP ELSE 80 OVER HIBYTE
 >= IF FDROP 0 0 ELSE DUP FF AND SWAP HIBYTE 96 SWAP - DUP IF
 0 DO DBY2 LOOP 1 0 D+ ELSE DROP THEN DBY2 THEN BASE @ ROT ROT
 A BASE ! <# I IF R> 0 DO # LOOP ELSE R> DROP THEN 2E HOLD #S
 ROT BASE ! R> ROT ROT SIGN #> R> OVER - DUP 1 >= IF 0 DO 20
```

```
 EMIT LOOP ELSE DROP THEN TYPE THEN THEN ;

: E. FDUP NAN? IF ." NAN" FDROP ELSE DUP 80 AND IF ." -" FNEG
 THEN 6 >R 0 8140 BEGIN FOVER FOVER FDIVX 977A1201. F>= WHILE
 0 8450 FMULX R> 1+ >R REPEAT FDIVX 0 8140 BEGIN FOVER FOVER
 FMULX 947A11FF. FSWAP F>= WHILE 0 8450 FMULX  R> 1 - >R REPEAT
 FMULX 98 OVER HIBYTE - >R FF AND MB10 FDUP D+ R> DUP 1 >= IF
 0 DO DBY2 LOOP ELSE DROP THEN R> BASE @ >R >R A BASE !
 <# 7 0 DO # LOOP 2E HOLD # #> TYPE R> DUP 0 >= IF ." +" THEN
 S->D D. R> BASE ! THEN ; : PRDP 11 C@ DUP FF = IF DROP -1
 THEN ; : FCON BASE @ A BASE ! 20 WORD NUMBER DUP HIBYTE >R
 00FF AND 9700 OR R> IF FDROP FNAN ELSE FNORM PRDP 1 >= IF
 0000 8140 PRDP 0 DO 0000 8450 FMULX LOOP F/
 THEN THEN ROT BASE ! 1C C@ IF ' DLITERAL C + @ , , , THEN ;
 IMMEDIATE ;S
```