68xxx/683xx Target Code v6.2

MicroProcessor Engineering

December 16, 2003

Contents

Chapter 1

Generic start up

The file 68K\HARDWARE\FS332\INIT332.FTH contains the generic vector table, Forth start up code and Forth initialisation tables. It must be the first file compiled that generates code unless you are compiling operating system specific code, such as with an O/S that requires a header.

This file is an example for a 68332. See the CTL files in CONFIGS folder to find examples of other initialisation files.

1.1 Exception Vectors

At power up, the CPU branches by loading the stack pointer and PC from absolute address 0. By default, the PC is loaded with the address START332, which contains CPU specific startup code for chip selects and the external bus. This code falls through to label ECLD, which is the start of the Forth initialisation code.

By default a 1kb area at START-VECTOR is reserved for the exception table.

Chapter 2

68xxx code definitions

2.1 Register usage

On 68xxx CPUs the following register usage is the default:

```
A7 = return stack
A6 = data stack
A5 = user pointer
A4 = locals pointer
A3 = 0
A2 = scratch
A1 = scratch
A0 = scratch

D7 = TOS
D6..D2 = scratch
D1, D0 = scratch and optimiser temporary
```

The VFX optimiser reserves D0 and D1 for internal operations. CODE definitions must use D7 as TOS with NOS pointed to by A6 as a full descending stack. D2..D6 and A0..A2 are free for use by CODE definitions and need not be preserved or restored. You should assume that any register can be affected by other words.

2.2 Logical and relational operators

`CODE AND \ x1 x2 -- x3`
: Perform a logical AND between the top two stack items and retain the result in top of stack.

`CODE OR \ x1 x2 -- x3`
: Perform a logical OR between the top two stack items and retain the result in top of stack.

`CODE XOR \ x1 x2 -- x3`

Perform a logical XOR between the top two stack items and retain the result in top of stack.

`CODE INVERT \ x -- x'`

Perform a bitwise NOT on the top stack item and retain result.

`CODE MIN \ n1 n2 -- n1|n2`

Given two data stack items preserve only the smaller.

`CODE MAX \ n1 n2 -- n1|n2`

Given two data stack items preserve only the larger.

`CODE 0= \ x -- flag`

Compare the top stack item with 0 and return TRUE if equals.

`CODE 0<> \ x -- flag`

Compare the top stack item with 0 and return TRUE if not-equal.

`CODE 0< \ x -- flag`

Return TRUE if the top of stack is less-than-zero.

`CODE 0> \ x -- flag`

Return TRUE if the top of stack is greater-than-zero.

`CODE = \ x1 x2 -- flag`

Return TRUE if the two topmost stack items are equal.

`CODE <> \ x1 x2 -- flag`

Return TRUE if the two topmost stack items are different.

`CODE < \ x1 x2 -- flag`

Return TRUE if n1 is less than n2.

`CODE > \ x1 x2 -- flag`

Return TRUE if n1 is greater than n2.

`CODE <= \ x1 x2 -- flag`

Return TRUE if n1 is less than or equal to n2.

`CODE >= \ x1 x2 -- flag`

Return TRUE if n1 is greater than or equal to n2.

`CODE U> \ u2 u2 -- flag`

An UNSIGNED version of >.

`CODE U< \ u1 u2 -- flag`

An UNSIGNED version of <.

`CODE DU< \ ud1 ud2 -- flag`

Returns true if ud1 (unsigned double) is less than ud2.

`CODE D0< \ d -- flag`

Returns true if signed double d is less than zero.

`CODE D0= \ xd -- flag`

Returns true if xd is 0.

`CODE D= \ xd1 xd2 -- flag`

Return TRUE if the two double numbers are equal.

`CODE D< \ d1 d2 -- flag`

Return TRUE if the double number d1 is < the double number d2.

`CODE DMAX \ d1 d2 -- d3 ; d3=max of d1/d2`

Return the maximum double number from the two supplied.

`CODE DMIN \ d1 d2 -- d3 ; d3=min of d1/d2`

Return the minimum double number from the two supplied.

`CODE WITHIN? \ n1 n2 n3 -- flag`

Return TRUE if N1 is within the range N2..N3. This word uses signed arithmetic.

`CODE WITHIN \ n1|u1 n2|u2 n3|u3 -- flag`

The ANS version of WITHIN?. This word uses unsigned arithmetic, so that signed compares are treated as existing on a number circle.

`CODE LSHIFT \ x1 u -- x1<<u`

Logically shift X1 by U bits left.

`CODE RSHIFT \ x1 u -- x1>>u`

Logically shift X1 by U bits right.

2.3 Control flow

`CODE EXECUTE \ xt --`

Execute the code described by the XT. This is a Forth equivalent to an assembler JSR/CALL instruction.

`CODE BRANCH \ --`

The run time action of unconditional branches compiled on the target.

`CODE ?BRANCH \ n --`

The run time action of conditional branches compiled on the target.

`CODE (OF) \ n1 n2 -- n1|n1 n2`

The run time action of OF compiled on the target.

`CODE (LOOP) \ --`

The run time action of LOOP compiled on the target.

`CODE (+LOOP) \ n --`

The run time action of +LOOP compiled on the target.

`CODE (DO) \ limit index --`

The run time action of DO compiled on the target.

`CODE (?DO) \ limit index --`

The run time action of ?DO compiled on the target.

`CODE LEAVE \ -- ; N.B. now non-immediate`

Remove the current DO..LOOP parameters and jump to the end of the DO..LOOP structure.

`CODE ?LEAVE \ flag -- ; N.B. now non-immediate`

If flag is non-zero, remove the current DO..LOOP parameters and jump to the end of the DO..LOOP structure.

`CODE I \ -- n`

Return the current index of the inner-most DO..LOOP.

`CODE J \ -- n`

Return the current index of the second DO..LOOP.

`CODE UNLOOP \ -- ; R: loop-sys --`

Remove the DO..LOOP control parameters from the return stack.

2.4 Arithmetic

Note that two sets of the words UM* M* and * exist which are selected by the equate CPU32?.

`CODE UM* \ u1 u2 -- ud`

Perform unsigned-multiply between two numbers and return double result.

`CODE * \ n1 n2 -- n3`

Standard signed multiply. N3 = n1 * n2.

`CODE M* \ n1 n2 -- d3`

Signed multiply yielding double result.

`CODE UM* \ u1 u2 -- ud ; MUST preserve D5`

The unsigned 32 * 32 -> 64 bit multiply primitive for CPUs without the long multiply instructions.

`: * \ n1 n2 -- n3`

Standard signed multiply. N3 = n1 * n2.

`CODE M* \ n1 n2 -- d ; relies on UM* leaving D5 alone`

Signed multiply yielding double result.

The division words select different forms according to the setting of CPU32?.

`CODE UM/MOD \ ud u -- urem uquot`

Perform unsigned division of double number UD by single number U and return remainder and quotient.

`CODE FM/MOD \ d1 n2 -- rem quot ; floored division`

Perform a signed division of double number D1 by single number N2 and return remainder and quotient using floored division. See the ANS Forth specification for more details of floored division.

`CODE SM/REM \ d1 n2 -- rem quot ; symmetric division`

Perform a signed division of double number D1 by single number N2 and return remainder and quotient using symmetric (normal) division.

`code /mod \ n1 n2 -- rem quot ; 6.1.0240`

Signed division of N1 by N2 single-precision yielding remainder and quotient.

`code / \ n1 n2 -- quot ; 6.1.0230`

Standard signed division operator. n3 = n1/n2.

`code mod \ n1 n2 -- rem ; 6.1.1890`

Return remainder of division of N1 by N2. n3 = n1 mod n2.

`: */MOD \ n1 n2 n3 -- rem quot`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the remainder and quotient. The point of this operation is to avoid loss of precision.

`: */ \ n1 n2 n3 -- n4`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the quotient. The point of this operation is to avoid loss of precision.

`: M/ \ d n1 -- n2`

Signed divide of a double by a single integer.

`: MU/MOD \ d n -- rem d#quot`

Perform an unsigned divide of a double by a single, returning a single remainder and a double quotient.

`CODE TNEGATE \ t1 -- -t1 ; negate value of triple cell number`

Negate a triple (3 cell) number.

`: M*/ \ d1 n1 +n2 -- d2`

Multiply double d1 by single n1 to give a triple precision result, and then divide it by n2 returning the quotient. The point of this operation is to avoid loss of precision.

`CODE M+ \ d1|ud1 n -- d2|ud2`

Add double d1 to sign extended single n to form double d2.

`CODE D+ \ d1 d2 -- d3`

Add two double precision integers.

`CODE D- \ d1 d2 -- d3`

Subtract two double precision integers. D3=D1-D2.

`CODE 1+ \ n1|u1 -- n2|u2`

Add one to top-of stack.

`CODE 2+ \ n1|u1 -- n2|u2`

Add two to top-of stack.

`CODE 4+ \ n1|u1 -- n2|u2`

Add four to top-of stack.

`CODE 1- \ n1|u1 -- n2|u2`

Subtract one from top-of stack.

`CODE 2- \ n1|u1 -- n2|u2`

Subtract two from top-of stack.

`CODE 4- \ n1|u1 -- n2|u2`

Subtract four from top-of stack.

`CODE 2* \ x1 -- x2`

Signed multiply top of stack by 2.

`CODE 4* \ x1 -- x2`

Signed multiply top of stack by 4.

`code 2/ \ x1 -- x2`

Signed divide top of stack by 2 by arithmetic right shift.

`code U2/ \ x1 -- x2`

Unsigned divide top of stack by 2 by logical right shift.

`code 4/ \ x1 -- x2`

Signed divide top of stack by 4 by arithmetic right shift.

`code U4/ \ x1 -- x2`

Unsigned divide top of stack by 4 by logical right shift.

`CODE + \ n1|u1 n2|u2 -- n3|u3`

Add two single precision integer numbers.

`CODE - \ n1|u1 n2|u2 -- n3|u3`

Subtract two single precision integer numbers. N3—u3=n1—u1-n2—u2.

`CODE NEGATE \ n1 -- n2`

Negate a single precision integer number.

`CODE DNEGATE \ d1 -- d2`

Negate a double number.

`CODE ?NEGATE \ n1 flag -- n1|n2`

If flag is negative, then negate n1.

`CODE ?DNEGATE \ d1 flag -- d1|d2`

If flag is negative, then negate d1.

`CODE ABS \ n -- u`

If n is negative, return its positive equivalent (absolute value).

`CODE DABS \ d -- ud`

If d is negative, return its positive equivalent (absolute value).

`CODE D2* \ xd1 -- xd2`

Multiply the given double number by two.

`CODE D2/ \ xd1 -- xd2`

Divide the given double number by two.

2.5 Stack manipulation

`CODE NIP \ x1 x2 -- x2`

Dispose of the second item on the data stack.

`CODE TUCK \ x1 x2 -- x2 x1 x2`

Insert a copy of the top data stack item underneath the current second item.

`CODE PICK \ xu .. x0 u -- xu .. x0 xu`

Get a copy of the Nth data stack item and place on top of stack. 0 PICK is equivalent to DUP.

`CODE ROLL \ xu xu-1 .. x0 u -- xu-1 .. x0 xu`

Rotate the order of the top N stack items by one place such that the current top of stack becomes the second item and the Nth item becomes TOS. See also ROT.

`CODE ROT \ x1 x2 x3 -- x2 x3 x1`

ROTate the positions of the top three stack items such that the current top of stack becomes the second item. See also ROLL.

`CODE -ROT \ x1 x2 x3 -- x3 x1 x2`

The inverse of ROT.

`CODE >R \ x -- ; R: -- x`

Push the current top item of the data stack onto the top of the return stack.

`CODE R> \ -- x ; R: x --`

Pop the top item from the return stack to the data stack.

`CODE R@ \ -- x ; R: x -- x`
Copy the top item from the return stack to the data stack.

`CODE 2>R \ x1 x2 -- ; R: -- x1 x2`
Transfer the two top data stack items to the return stack.

`CODE 2R> \ -- x1 x2 ; R: x1 x2 --`
Transfer the top two return stack items to the data stack.

`CODE 2R@ \ -- x1 x2 ; R: x1 x2 -- x1 x2`
Copy the top two return stack items to the data stack.

`CODE 2ROT \ x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2`
Perform ROT operation on three double numbers.

`CODE DROP \ x --`
Lose the top data stack item and promote NOS to TOS.

`CODE 2DROP \ x1 x2 --`
Discard the top two data stack items.

`CODE SWAP \ x1 x2 -- x2 x1`
Exchange the top two data stack items.

`CODE 2SWAP \ x1 x2 x3 x4 -- x3 x4 x1 x2`
Exchange the top two cell-pairs on the data stack.

`CODE DUP \ x -- x x`
DUPlicate the top stack item.

`CODE ?DUP \ x -- | x`
DUPplicate the top stack item only if it non-zero.

`CODE 2DUP \ x1 x2 -- x1 x2 x1 x2`
DUPlicate the top cell-pair on the data stack.

`CODE OVER \ x1 x2 -- x1 x2 x1`
Copy NOS to a new top-of-stack item.

`CODE 2OVER \ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2`
Similar to OVER but works with cell-pairs rather than cell items.

`CODE SP@ \ -- x`
Get the current address value of the data-stack pointer.

`CODE SP! \ x --`
Set the current address value of the data-stack pointer.

`CODE RP@ \ -- x`
Get the current address value of the return-stack pointer.

`CODE RP! \ x --`
Set the current address value of the return-stack pointer.

2.6 Strings

`CODE COUNT \ c-addr1 -- c-addr2' u`

Given the address of a counted string in memory this word will return the address of the first character and the length in characters of the string.

`CODE /STRING \ c-addr1 u1 n -- c-addr2 u2`

Modify a string address and length to remove the first N characters from the string.

`CODE SKIP \ c-addr1 u1 char -- c-addr2 u2`

Modify the string description by skipping over leading occurrences of 'char'.

`CODE SCAN \ c-addr1 u1 char -- c-addr2 u2`

Look for first occurrence of CHAR in string and return new string. C-addr2/u2 describe the string with CHAR as the first character.

`CODE S= \ c-addr1 c-addr2 u -- flag`

Compare two same-length strings/memory blocks, returning TRUE if they are identical.

`CODE COMPARE \ c-addr1 len1 c-addr2 len2 -- n`

Compare two strings. The return result is 0 for a match or can be -ve/+ve indicating string differences. If the two strings are identical, n is zero. If the two strings are identical up to the length of the shorter string, n is minus-one (-1) if u1 is less than u2 and one (1) otherwise. If the two strings are not identical up to the length of the shorter string, n is minus-one (-1) if the first non-matching character in the string specified by c-addr1 u1 has a lesser numeric value than the corresponding character in the string specified by c-addr2 u2 and one (1) otherwise.

`: SEARCH \ c-addr1 u1 c-addr2 u2 -- c-addr3 u3 flag`

Search the string c-addr1/u1 for the string c-addr2/u2. If a match is found return c-addr3/u3, the address of the start of the match and the number of characters remaining in c-addr1/u1, plus flag f set to true. If no match was found return c-addr1/u1 and f=0.

`CODE CMOVE \ c-addr1 c-addr2 u --`

Copy U bytes of memory forwards from C-ADDR1 to C-ADDR2.

`CODE CMOVE> \ c-addr1 c-addr2 u --`

As CMOVE but working in the opposite direction, copying the last character in the string first.

`CODE (") \ -- a-addr ; return address of string, skip over it`

Return the address of a counted string that is inline after the CALLING word, and adjust the CALLING word's return address to step over the inline string. See the definition of (.") for an example.

`CODE UPPER \ c-addr u --`

Convert the ASCII string described to upper-case. This operation happens in place.

2.7 Memory operators

`CODE ON \ a-addr --`

Given the address of a CELL this will set its contents to TRUE (-1).

`CODE OFF \ a-addr --`

Given the address of a CELL this will set its contents to FALSE (0).

`CODE +! \ n|u a-addr --`

Add N to the CELL at memory address ADDR.

`CODE c+! \ b addr --`

Add N to the character (byte) at memory address ADDR.

`CODE INCR \ a-addr --`

Increment the data cell at a-addr by one.

`CODE DECR \ a-addr --`

Decrement the data cell at a-addr by one.

`CODE 2@ \ a-addr -- x1 x2`

Fetch and return the two CELLS from memory ADDR and ADDR+sizeof(CELL). The cell at the lower address is on the top of the stack.

`CODE @ \ a-addr -- x`

Fetch and return the CELL at memory ADDR.

`CODE W@ \ a-addr -- w`

Fetch and 0 extend the word (16 bit) at memory ADDR.

`CODE C@ \ c-addr -- char`

Fetch and 0 extend the character at memory ADDR and return.

`CODE 2! \ x1 x2 a-addr --`

Store the two CELLS x1 and x2 at memory ADDR. X2 is stored at ADDR and X1 is stored at ADDR+CELL.

`CODE ! \ x a-addr --`

Store the CELL quantity N at memory ADDR.

`CODE W! \ w a-addr --`

Store the word (16 bit) quantity w at memory ADDR.

`CODE C! \ char c-addr --`

Store the character CHAR at memory C-ADDR.

`CODE FILL \ c-addr u char --`

Fill LEN bytes of memory starting at ADDR with the byte information specified as CHAR.

`CODE TEST-BIT \ u c-addr -- flag`

AND the mask with the contents of addr and return the result. Byte operation.

`CODE SET-BIT \ u c-addr --`

Apply the mask ORred with the contents of c-addr. Byte operation.

`CODE RESET-BIT \ u c-addr --`

Apply the mask inverted and ANDed with the contents of c-addr. Byte operation.

`CODE TOGGLE-BIT \ u c-addr --`

Invert the bits at c-addr specified by the mask. Byte operation.

2.8 Miscellaneous words

`: SEARCH-WORDLIST ( c-addr u wid -- 0|xt 1|xt -1 )`

Search the given wordlist for a definition. If the definition is not found then 0 is returned, otherwise the XT of the definition is returned along with a non-zero code. A -ve code indicates a "normal" definition and a +ve code indicates an IMMEDIATE word.

`CODE NAME> \ nfa -- xt`

Move a pointer from an NFA to the CFA or "XT" in ANS parlance.

`CODE >NAME \ xt -- nfa`

Move a pointer from an XT back to the NFA or name-pointer. If the original pointer was not an XT or if the definition in question has no name header in the dictionary the returned pointer will be useless. Care should be taken when manipulating or scanning the Forth dictionary in this way.

`: >BODY \ xt -- a-addr`
Move a pointer from a CFA or "XT" to the definition BODY. This should only be used with children of CREATE. E.g. if FOOBAR is defined with CREATE foobar, then the phrase ' foobar >body would yield the same result as executing foobar.

`CODE DIGIT \ char n -- 0|n true`
If the ascii value CHAR can be treated as a digit for a number within the radix N then return the digit and a TRUE flag, otherwise return FALSE.

`CODE UPC \ char -- char'`
Convert a character to upper case.

`CODE S>D \ n -- d`
Convert a single number to a double one.

`CODE D>S \ d -- n`
Convert a double number to a single.

`CODE NOOP \ --`
A NOOP, null instruction.)

2.9 Portability helpers

Using these words will make code easier to port between 16, 32 and 64 bit targets.

`CODE ALIGNED \ addr -- a-addr`
Given an address pointer this word will return the next ALIGNED address subject to system wide alignment restrictions. Note that this definition is only necessary in the code file for CPUs that require alignment, but do not require alignment to a cell boundary.

`CODE CELL+ \ a-addr1 -- a-addr2`
Add the size of a CELL to the top-of stack.

`CODE CELLS \ n1 -- n2`
Return the size in address units of N1 cells in memory.

`CODE CELL- \ a-addr1 -- a-addr2`
Decrement an address by the size of a cell.

`CODE CELL \ -- n`
Return the size in address units of one CELL.

`CODE CHAR+ \ c-addr1 -- c-addr2`
Increment an address by the size of a character.

`CODE CHARS \ n1 -- n2`
Return size in address units of N1 characters.

2.10 Runtime for VALUE

`CODE VAL! \ n -- ; store value address in-line`

Store n at the inline address following this word.

`CODE VAL@ \ -- n ; read value data address in-line`

Read n from the inline address following this word.

2.11 Defining words and runtime support

`: COMPILE, \ xt -- ; compile call to xt - may be optimised`

Compile the word specified by xt into the current definition.

`: CONSTANT \ x "<spaces>name" -- ; Exec: -- x`

Create a new CONSTANT called "name" which has the value "x". When "NAME" is executed the value is returned on the top-of-stack.7

`: VARIABLE \ "<spaces>name" -- ; Exec: -- a-addr`

Create a new variable called "name". When "Name" is executed the address of the data-cell is returned for use with @ and ! operators.

`: USER \ u "<spaces>name" -- ; Exec: -- addr`

Create a new USER variable called "name". The 'u' parameter specifies the index into the user-area table at which to place the data. USER variables are located in a separate area of memory for each task or interrupt. Use in the form: "$400 USER TaskData".

`: : \ C: "<spaces>name" -- colon-sys ; Exec: i*x -- j*x ; R: -- nest-sys`

Begin a new definition called "name".

`: :NONAME \ C: -- colon-sys ; Exec: i*x -- i*x ; R: -- nest-sys`

Begin a new code definition which does not have a name. After the definition is complete the semi-colon operator returns the XT of newly compiled code on the stack.

`: (;CODE) \ -- ; R: a-addr --`

Performed at compile time by ;CODE and DOES>. Patch the last word defined (by CREATE) to have the run time actions that follows immediately after (;CODE).

`: DOES> \ C: colon-sys1 -- colon-sys2 ; Run: -- ; R: nest-sys --`

Begin definition of the runtime-action of a child of a defining word. You should not use RECURSE after DOES>.

: DOCREATE, \ --
Compile the run time action of CREATE.

: CRASH \ -- ; used as action of DEFER
The default action of a DEFERed word. This will simply THROW a code back to the system.

: DEFER \ Comp: "<spaces>name" -- ; Run: i*x -- j*x
Creates a new DEFERed word. A default action, CRASH, is automatically assigned.

: 2CONSTANT \ Comp: x1 x2 "<spaces>name" -- ; Run: -- x1 x2
A two-cell equivalent to CONSTANT.

: 2VARIABLE \ Comp: "<spaces>name" -- ; Run: -- a-addr
A two-cell equivalent to VARIABLE.

: FIELD \ size x "<spaces>name" -- size+x ; Exec: addr -- addr+x

Create a new field within a structure definition of size n bytes.

2.12 Structure compilation

These words define high level branches. They are used by the structure words such as IF and AGAIN.

: >mark \ -- addr ; mark start of forward branch
Mark the start of a forward branch. HIGH LEVEL CONSTRUCTS ONLY.

: >resolve \ addr -- ; resolve absolute target of forward branch
Resolve absolute target of forward branch. HIGH LEVEL CONSTRUCTS ONLY.

: <mark \ -- addr ; mark start (destination) of backward branch
Mark the start (destination) of a backward branch. HIGH LEVEL CONSTRUCTS ONLY.

: <resolve \ addr ; resolve (at branch point) backward branch
Mark the start (destination) of a backward branch. HIGH LEVEL CONSTRUCTS ONLY.

synonym >c_res_branch >resolve \ addr -- ; fix up forward referenced branch
See >RESOLVE.

synonym c_mrk_branch< >mark \ -- addr ; mark destination of backward branch
See >MARK

2.13 Branch constructors

Used when compiling code on the target.

: `c_branch< \ addr -- ; lay BRANCH instructions`
Lay the code for BRANCH.

: `c_?branch< \ addr -- ; lay ?BRANCH instructions`
Lay the code for ?BRANCH.

: `c_branch> \ -- addr ; forward referenced branch`
Lay the code for a forward referenced unconditional branch.

: `c_?branch> \ -- addr ; forward referenced branch`
Lay the code for a forward referenced conditional branch.

2.14 Main structure compilers

: `DOCOLON, \ --`
Compile the runtime entry code required by colon definitions.

: `c_lit \ lit --`
Compile the code for a LITERAL.

: `c_drop \ --`
Compile the code for DROP.

: `c_exit \ --`
Compile the code for EXIT.

: `c_do \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- loop-sys`
Compile the code for DO.

: `c_?DO \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- | loop-sys`
Compile the code for ?DO.

: `c_LOOP \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2`
Compile the code for LOOP.

: `c_+LOOP \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2`
Compile the code for +LOOP.

: `c_case \ -- addr`
Compile the code for CASE.

`:  c_OF \ C: -- of-sys ; Run:  x1 x2 -- | x1`

Compile the code for OF.

`:  c_?OF \ C: -- of-sys ; Run:  flag --`

Compile the code for ?OF.

`:  c_ENDOF \ C: case-sys1 of-sys -- case-sys2 ; Run:  --`

Compile the code for ENDOF.

`:  FIX-EXITS \ n1..nn --`

Compile the code to resolve the forward branches at the end of a CASE structure.

`:  c_ENDCASE \ C: case-sys -- ; Run:  x --`

Compile the code for ENDCASE.

`:  c_END-CASE \ C: case-sys -- ; Run:  x --`

Compile the code for END-CASE.

Chapter 3

Interrupt handlers

The file 68K\INT68K.FTH contains generic ARM interrupt handlers in the v6.2 style.

The file 68K\TRAPS.FTH may be compiled after this file. TRAPS.FTH provides diagnostic information and other code such as divide by zero handling.

3.1 Interrupt management

`code SETI \ x -- ; set interrupt level x`

Global set interrupts to level n, where x=$0n00.

`: ei \ -- ; Enable interrupts`

Global enable interrupts to level 0.

`: di \ -- ; Disable interrupts`

Global disable interrupts to level 7.

`code SAVE-INT \ -- x ; save interrupt status`

Geturn interrupt status and then disable interrupts. Use [I and I] for new code.

`code RESTORE-INT \ x -- ; restore interrupt`

Restore state returned by SAVE-INT. Use [I and I] for new code.

`code [I \ R: -- x ; save SR on return stack and disable interrupts`

Preserve I/F status on return stack, disable interrupts. The state is restored by I].

`code I] \ R: -- x ; restore SR from return stack`

Restore interrupt status saved by [I from the return stack.

3.2 Interrupt service routines

Each predefined ISR has four components:

- A DEFERred word into which you ASSIGN the required action.
- A code fragment which references the DEFERred word and RETI.
- An assembler entry point.
- A vector assignment

`code RETI \ --`
: Return from high level interrupt.

`defer BUSERR-ISR \ --`
: The deferred word for a bus errot

`l: buserr-int`
: The code fragment which references BUSERR-ISR.

`proc buserr-int0`
: The assembly level entry point for the bus error handler.

The predefined DEFERred words are:

```
BUSERR-ISR ADDRERR-ISR ILLOP-ISR DIVZERO-ISR
CHKINST-ISR TRAPVINST-ISR PRIVVIOL-ISR TRACE-ISR
OPCODE1010-ISR OPCODE1111-ISR PHANTOM-ISR
AVECn-ISR where n=1..7
```

All these words have their associated other three components.

Chapter 4

68xxx multitasker

The 68xxx multitasker follows the model introduced with the v6.1 compilers. A few extensions are also provided.

4.1 Configuration - normally performed earlier

```
0 equ test-multi?  \ true to compile test code
```

If not previously defined, TEST-MULTI? is set to zero and test code is not compiled.

4.2 TCB data structure layout

```
cell       LINK    link to next task
cell       SSP     Saved Stack Pointer
cell       STAT    BIT 0   1 = running, 0 = halted
                   BIT 1   1 = message pending
                   BIT 2   1 = event triggered
                   BIT 3   1 = event handler run
                others  1 = set to run task, available to user
cell       TASK    Task # that sent message here
cell       MESG    Message address
cell       EVNTw   CFA of word run as event handler
```

This structure is allocated at the start of the USER area. Consequently the TCB of the current task is given by UP.

```
struct /TCB \ -- size
```

The structure used by the code that matches the description above.

4.3 Task handling primitives

`init-u0 constant main \ -- addr ; tcb of main task`

Returns the base address of the main task's USER area.

`0 value multi? \ -- flag ; true if tasker enabled`

Returns true if the tasker is enabled.

`: single \ -- ; disable scheduler`

Disable scheduler.

`: multi \ -- ; enable scheduler`

Enable scheduler.

`CODE pause \ -- ; the scheduler`

The software scheduler itself.

`code status \ -- task-status`

Returns the current task's status cell, but with the run bit masked out.

`CODE restart \ task -- ; mark task TCB as running`

Sets the RUN bit in the task's status cell.

`CODE halt \ task# -- ; reset running bit in TCB`

Clears the RUN bit in the task's status cell.

`: stop \ -- ; halt oneself`

HALT's the current task, and executes PAUSE.

4.4 Event handling

Event handling is only compiled if the equate EVENT-HANDLER? is set non-zero in the control file.

`: set-event \ task -- ; set event trigger in task TCB`

Set the event trigger in task TCB.

`: event? \ task -- flag ; true if task had event`

Returns true if true if task has received an event trigger which has not been cleared yet.

`: clr-event-run \ -- ; reset own EVENT_RUN flag`

Reset the current task's EVENT_RUN flag.

`: to-event \ xt task -- ; define action of a task`

Sets XT as the event handler for the task.

4.5 Message handling

Message handling is only compiled if the equate MESSAGE-HANDLER? is set non-zero in the control file.

`: msg? \ task -- flag ; true if task has message`

Returns true if task has received a message.

`: send-message \ addr task -- ; send message to task`

Send a message to a task.

`: get-message \ -- addr task ; wait for any message`

Wait for any message and return the message and the task it came from.

`: wait-event/msg \ -- ; wait for message or event trigger`

Wait for a message or an event trigger.

4.6 Task structure management

`code init-task \ xt task -- ; Initialise a task stack`

Initialise a task's stack before running it and set it to execute the word whose XT is given.

`: add-task \ task -- ; insert into list`

Add the task to the list of tasks after the current task.

`: sub-task \ task -- ; remove task from chain`

Remove the task from the task list.

`: initiate \ xt task -- ; start task from scratch and run it`

Start the given task executing the word whose XT is given, e.g.

```
['] <name> <task> INITIATE
```

`: sleeper \ xt task -- ; start task from scratch, but leave it HALTed`

Use in the form:

```
['] <action> <taskname> SLEEPER
```

to put a task on the active task list, but as if HALTed. SLEEPER allows you to make a task ready for waking up later, perhaps by another task. This avoids having to put STOP as the first word in a task. Note that SLEEPER does not call PAUSE. See also INITIATE.

```
:  terminate \ task -- ; stop task, and remove from list
```

Stop a task, and remove it from the list.

```
:  init-multi \ -- ; initialisation with multi-tasking
```

Initialise the multitasker and start it. If tasking is selected by setting the equate TASKING? in the control file, KERNEL62.FTH will automatically run this word. Make sure that your initialisation code includes INIT-MULTI or your code will crash.

```
:  his \ task uservar -- addr ; produce address of user var in another
task
```

Given a task id and a USER variable, returns the address of that variable in the given task. This word is used to set up USER variables in other tasks.

4.7 semaphores

The semaphore code is only compiled if the equate SEMAPHORES? is set non-zero in the control file.

A SEMAPHORE is an extended variable used for signalling between tasks, and for resource allocation. The counter field is used as a count of the number of times the resource may be used, and the arbiter field contains the TCB of the task that last gained access. This field can be used for priority arbitration and deadlock detection/arbitration.

```
:  semaphore \ -- ; -- addr [child]
```

Creates a semaphore which returns its address at runtime. Use in the form:

```
 Semaphore <name>
```

```
:  signal \ addr --
```

SIGNAL increments the counter field of a semaphore, indicating either that another item has been allocated to the resource, or that it is available for use again, 0 indicating in use by a task REQUEST waits until the counter field of a semaphore is non-zero, and then decrements the counter field by one. This allows the semaphore to be used as a COUNTED semaphore. For example a character buffer may be used where the semaphore counter shows the number of available characters. Alternatively the semaphore may be used purely to share resources. The semaphore is initialised to one. The first task to REQUEST it gains access, and all other tasks must wait until the accessing task SIGNALs that it has finished with the resource.

4.8 TASK and START:

TASK <name> builds a named task user area. The action of a task is assigned and the task started by the word INITIATE

```
['] <action> <task> INITIATE
```

START: is used inside a colon definition. The code before START: is the task's initialisation, performed by the current task. The code after START: up to the closing ; is the action of the task. For example:

```
TASK FOO
: RUN-FOO
  ...
  FOO START:
  ...
  begin ... pause again
;
```

All tasks must run in an endless loop, except for initialisation code. When RUN-FOO is executed, the code after START: is set up as the action of task FOO and started. RUN-FOO then exits.

If you want to perform additional actions after starting the task, you should use INITIATE to start the task.

`variable task-chain \ -- addr`

Anchors list of all tasks created by TASK and friends.

`: task \ -- ; -- task ; TASK <name> builds a task`

Note that the cross-interpreter's version of TASK has been modified from v6.2 onwards to leave the current section as CDATA.

`: task \ -- ; -- task ; TASK <name> builds a task`

Creates a new task and data area, returning the address of the user area at run time. The task is also linked into the task chain anchored by TASK-CHAIN.

`: start: \ task -- ; exits from caller`

Used inside a colon definition. The code following START: up to the ending semi-colon forms the action of the task. The word containing START: finishes at START:.

4.9 Debugging tools

`: .task \ task --`

Display task's name if it has one.

`: .tasks \ task -- ; display all task names`
Display all the tasks anchored by TASK-CHAIN.

`: .running \ --`
Display running tasks.

Chapter 5

Minimal Umbilical code definitions

The file 68K\MIN68K.FTH contains the minimum code definitions required to support Umbilical Forth. If additional definitions are required, they may be copied to a new file from 68K\CODE68K.FTH or COMMON\KERNEL62.FTH.

5.1 Flow of control

`CODE (DO) \ limit index --`

The run time action of DO compiled on the target.

`CODE (?DO) \ limit index --`

The run time action of ?DO compiled on the target.

`CODE EXECUTE \ xt --`

Execute the code described by the XT. This is a Forth equivalent to an assembler JSR/CALL instruction.

5.2 Stack operations and maths

`Code Noop \ -- ; used by multi-tasker`

A NOOP, null instruction.)

`CODE MIN \ n1 n2 -- n'`

Given two data stack items preserve only the smaller.

`CODE MAX \ n1 n2 -- n1|n2`

Given two data stack items preserve only the smaller.

`CODE WITHIN? \ n1 n2 n3 -- flag`

Return TRUE if N1 is within the range N2..N3. This word uses signed arithmetic.

`CODE DNEGATE \ d1 -- d2`

Negate a double number.

`CODE ?NEGATE \ n1 flag -- n1|n2`

If flag is negative, then negate n1.

`CODE ?DNEGATE \ d1 flag -- d1|d2`

If flag is negative, then negate d1.

`CODE D+ \ d1 d2 -- d3`

Add two double precision integers.

`CODE D- \ d1 d2 -- d3`

Subtract two double precision integers. D3=D1-D2.

5.3 Multiply and divide

Different multiply and divide routines are compiled according to the setting of equate CPU32? which should be set non-zero for CPU32 CPUs such as the 68332 which have enhanced multiply and divide instructions.

`CODE UM* \ u1 u2 -- ud`

Perform unsigned-multiply between two numbers and return double result.

`CODE * \ n1 n2 -- n3`

Standard signed multiply. N3 = n1 * n2.

`CODE M* \ n1 n2 -- d3`

Signed multiply yielding double result.

`CODE UM/MOD \ ud u -- urem uquot`

Perform unsigned division of double number UD by single number U and return remainder and quotient.

`CODE FM/MOD \ d1 n2 -- rem quot ; symmetric division`

Perform a signed division of double number D1 by single number N2 and return remainder and quotient using floored division. See the ANS Forth specification for more details of floored division.

`CODE SM/REM \ d1 n2 -- rem quot ; symmetric division`

Perform a signed division of double number D1 by single number N2 and return remainder and quotient using symmetric (normal) division.

`code /mod \ n1 n2 -- rem quot ; 6.1.0240`

Signed division of N1 by N2 single-precision yielding remainder and quotient.

`code / \ n1 n2 -- quot ; 6.1.0230`

Standard signed division operator. n3 = n1/n2.

`code mod \ n1 n2 -- rem ; 6.1.1890`

Return remainder of division of N1 by N2. n3 = n1 mod n2.

`: */MOD \ n1 n2 n3 -- rem quot`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the remainder and quotient. The point of this operation is to avoid loss of precision.

`: */ \ n1 n2 n3 -- n4`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the quotient. The point of this operation is to avoid loss of precision.

`: M/ \ d n1 -- n2`

Signed divide of a double by a single integer.

`: MU/MOD \ d n -- rem d#quot`

Perform an unsigned divide of a double by a single, returning a single remainder and a double quotient.

5.4 Stack manipulation

`CODE SP@ \ -- x`

Get the current address value of the data-stack pointer.

`CODE SP! \ x --`

Set the current address value of the data-stack pointer.

`CODE RP@ \ -- x`

Get the current address value of the return-stack pointer.

`CODE RP! \ x --`

Set the current address value of the return-stack pointer.

`CODE ROLL \ xu xu-1 .. x0 u -- xu-1 .. x0 xu`
Rotate the order of the top N stack items by one place such that the current top of stack becomes the second item and the Nth item becomes TOS. See also ROT.

`CODE ON \ a-addr --`
Given the address of a CELL this will set its contents to TRUE (-1).

`CODE OFF \ a-addr --`
Given the address of a CELL this will set its contents to FALSE (0).

5.5 string and memory operators

`CODE 2@ \ a-addr -- x1 x2`
Fetch and return the two CELLS from memory ADDR and ADDR+sizeof(CELL). The cell at the lower address is on the top of the stack.

`CODE 2! \ x1 x2 a-addr --`
Store the two CELLS x1 and x2 at memory ADDR. X2 is stored at ADDR and X1 is stored at ADDR+CELL.

`CODE c+! \ b addr --`
Add N to the character (byte) at memory address ADDR.

`CODE COUNT \ c-addr1 -- c-addr2' u`
Given the address of a counted string in memory this word will return the address of the first character and the length in characters of the string.

`CODE CMOVE \ c-addr1 c-addr2 u --`
Copy U bytes of memory forwards from C-ADDR1 to C-ADDR2.

`CODE CMOVE> \ c-addr1 c-addr2 u --`
As CMOVE but working in the opposite direction, copying the last character in the string first.

`CODE FILL \ c-addr u char --`
Fill U bytes of memory starting at C-ADDR with the byte information specified as CHAR.

`CODE (") \ -- a-addr ; return address of string, skip over it`
Return the address of a counted string that is inline after the CALLING word, and adjust the CALLING word's return address to step over the inline string. See the definition of (.") for an example.

`: (C") \ -- c-addr`
The run time action compiled by C".

`: (S") \ -- c-addr u`
The run time action compiled by S".

5.6 Umbilical versions of defining words

`chere is-action-of constant`

The runtime code for a CONSTANT.

`chere is-action-of variable`

The runtime code for a VARIABLE.

`chere is-action-of user`

The runtime action of a USER variable.

`: CRASH \ -- ; used as action of DEFER`

The default action of a DEFERed word. A NOOP.

`chere is-action-of DEFER \ Comp: "<spaces>name" -- ; Run: i*x -- j*x`

The runtime action of a DEFERred word.