Appendix C

Mixed Languages Interface: Source Code

This annex is intended to supplement the "Mixed Languages Interface" chapter (chapter 3). It gives source listings and some technical comments for the mixed languages interface, as we are currently using it.

The source is split into a number of different files. These files are:

`CFLOAD.ASM` program that loads the FORTH++ system and then the C overlay if it is required.

`MAKELOAD.BAT` batch file that makes the FORTH loader program.

`CFINIT.C` contains the C initialisation code.

`CFORTH.H` header file containing macro definitions (`push`, `pop`, etc.).

`CFORTH1.C` an example "user file". This is the only file that the user should edit. We have provided an example module that provides a floating point maths extension, using the C floating point code.

`CFASM.ASM` holds the code for initialising the C and FORTH context switching area, in addition to the code for performing the context switch.

`MAKEOVLS.BAT, MAKEOVLL.BAT` make the overlay library, incorporating the user supplied code. Using the *S*mall or *L*arge memory models.

C.1 Loader

This is the Microsoft assembler source code for a program that loads in the FORTH++ system. It first returns as much memory as possible to MS-DOS. It will then proceed to load in the FORTH++ segments. For a given segment, the name of the file to load is obtained by taking the name of the loader program and replacing the `.COM` by the required segment extensions:

`.CDE`	Code segment
`.FOR`	FORTH data segment
`.STR`	String segment
`.NAM`	Name segment

[0]**This is a chapter taken from the Ph.D. thesis "Practical and Theoretical Aspects of Forth Software Development". A full copy of the thesis is available from the British library. Both the this chapter and the thesis are Copyright © Peter Knaggs.**

The stack segment is grabbed from memory, but is not initialised by this loader program.

After loading in the FORTH system it will then load in a C module. Notice that this module is the last to be loaded. The loader will either take the C file from the same directory as the FORTH++ overlays or from a given location. It will then pass control to the C module to allow it to initialise before executing any of the FORTH code. If no C module is required the loader will simply execute the FORTH directly.

```
        PAGE    60,132  ; Set page size, lines x cols

        TITLE   'CForth++ Loader'

comment ;

This file is assembled to produce the Forth system loader.  The batch
file MAKELOAD.BAT contains the commands to assemble the file and convert
the object code to .COM format.  The file produced by MAKELOAD.BAT is
called FLOADER.TPT.  This file is a loader template which is patched and
renamed by the Forth SAVE-SYS routine to produce a customised loader for
a particular Forth system. The distribution files FPP.COM and CFORTH.COM
are examples of Forth loaders produced in this way.

When a loader such as CFORTH.COM is invoked as a DOS command, it detects
its own name (in this case "CFORTH") and loads in the Forth segments
(CFORTH.CDE, CFORTH.FOR, CFORTH.NAM and CFORTH.STR).  It also reserves
space for the Forth stack segment.  The segment address of each segment
is stored at reserved locations in the Forth code segment.

If a C overlay file was specified at SAVE-SYS time, its name will have
been patched into the loader file and it will be loaded and executed as
an overlay.  This overlay will be passed the PSP of the loader, within
which will be the address of the Forth code segment (in an unused FCB
address in the PSP).

If no C overlay file is specified control will be passed directly to the
Forth code segment.

;====================================================================

CFL_TEXT segment byte public 'CODE'
        assume  cs:CFL_TEXT,ds:CFL_TEXT,es:CFL_TEXT,ss:CFL_TEXT

        ORG     100h

entry   PROC    NEAR

; To enable loader locations to remain at a fixed position known to the
; Forth system, the code starts with a jump.

        jmp     start

; To allow a common location to place a debug breakpoint this far return
; instruction is placed at location 103 of the loader.  Hence when the
; Forth word TRAP is invoked, this far return is executed, thus allowing
; us to use a monitor program to assist in the debugging of our Forth
; code.

dbg     PROC    FAR
        ret
dbg     ENDP

; ====================================================================
;                            Data  Area
; ====================================================================
;
```

```
; Now we have the data area for the loader.  The default values held in
; this section will be altered to suit by Forth's SAVE-SYS command.

        ; Forth Segment sizes (in paragraphs)

csize:  dw     1000h    ; Code segment
fsize:  dw     1000h    ; Forth segment
nsize:  dw     1000h    ; Name segment
$size:  dw      800h    ; String segment
ssize:  dw     1000h    ; Stack segment

        ; Far address to execute Forth

eoff:   dw      3       ; Offset within Code segment to start execution
eseg:   dw      ?       ; Will be set when Forth code seg is loaded.


; C Base program name.
;
; This is a counted ASCIIZ sting, with the count including the
; terminating zero
;
; If the count is 0 then no C Base program is loaded and control is
; passed directly to the Forth++ Code segment
;
; If the count is -1 then the ASCIIZ string holds the Complete path name
; Otherwise the file name is added to the end of the default load path
;

cname   db      0
        db      7fh DUP(?)


; The following equates give addresses in the Forth code segment which
; are used to hold the addresses of other segments.

fsptr   equ     06h     ; Forth segment pointer
nsptr   equ     08h     ; Names
$sptr   equ     0ah     ; Strings
ssptr   equ     0ch     ; Stack
lsptr   equ     0eh     ; Loader segment pointer

; Address for Forth to call to re-enter C system.  This is initialised
; to point to an error handler by SAVE-SYS and is reset by the C overlay
; initialisation.

roptr   equ     10h     ; Offset
rsptr   equ     12h     ; Segment

; Forth++ Segment file name extensions

cfile   db      "CDE",0   ; Code
ffile   db      "FOR",0   ; Forth
nfile   db      "NAM",0   ; Name
$file   db      "STR",0   ; String


; ====================================================================
;                          Loader  Program
; ====================================================================
;

; Initialise all segments to point to the same (code) segment

start:  cld
```

```
        mov     ax,cs
        mov     ds,ax
        mov     es,ax
        mov     ss,ax
        mov     sp,OFFSET stack

; As we use features of DOS that only appeared (documented) in DOS 3.0
; we must make sure that the user is not running an earlier DOS system.

        mov     bp,OFFSET msg_1         ; Point to the error message
        mov     ah,30h
        int     21h
        cmp     al,3
        jae     resize
        call    abort

; Resize memory back down so that we can grab it

resize: mov     bx,64h                  ; keep enough memory to work in
        mov     ah,4ah
        int     21h                     ; resize allocated memory
        jnc     cont
        call    mem_err

; Find the current program name.  This is stored at the end of the
; environment table.  The environment table can be a maximum of 32K.
; Each entry is terminated with a 00 byte and the table is terminated
; with a second 00 byte (after an entry terminating 00 byte).  The name
; of the currently executing program is then stored as an ASCIIZ string
; 2 bytes on from the end of the environment table.

cont:   mov     bp,OFFSET msg_3

        mov     ax,cs:2ch       ; Get segment addr of Env. table
        mov     es,ax
        mov     cx,8000h        ; Max size of Env. table
        xor     ax,ax
        mov     di,ax           ; Set DI to start of Env. table

scan:   or      cx,cx
        jnz     scn1
        call    abort           ; If table too big, Env. error
scn1:   repnz   scasb           ; Find end of entry
        scasb                   ; Is it end of table
        jnz     scan

        add     di,2            ; Move DI to start of program name

; The currently executing program name is the full path name of this
; program (eg, C:\FORTH\CFORTH\CFORTH.COM).  We copy this name into an
; internal buffer so that we can change it as required.
; This name can be no longer than 7F bytes.

        push    ds

        push    es
        mov     ax,ds
        mov     es,ax
        pop     ds

        mov     si,di
        mov     di,OFFSET fname

        mov     cx,7fh
        rep     movsb
        pop     ds

; Scan through the currently executing program name (in the buffer) to
; find the . used in the '.COM' at the end of the name.  This is so that
; we can simply replace the 'COM' part of the filename with the relevant
; extension required for a given overlay.
;
; It is possible to have a '.' in a directory name, hence we must find
; the end of the filename (a maximum of 7fh characters) and scan back
; towards the start to find the correct '.'.  The location of the
; character after the '.' is stored in the variable  fdot  for later use
; by the  loadseg  subroutine.

        mov     cl,80h          ; Max for file name + 1
        mov     al,0            ; Terminating character
        mov     di,OFFSET fname
        repnz   scasb
        je      sdot
        call    abort           ; Filename to long (Env. error)

sdot:   mov     cl,6            ; Max characters back + 1
        mov     al,'.'
        std
        repnz   scasb
        cld
        je      gdot
        call    abort           ; Can't find the '.' (Env. error)

gdot:   add     di,2
        mov     WORD PTR fdot,di


; Load the Forth++ system.

; The Forth++ system is made up of a set of overlay files.  The
; following code simply loads in each of the overlays in turn.  The
; overlay name is made up by taking the currently executing path name
; and replacing the .COM with the relevant extension for the given
; overlay.  The subroutine  loadseg  preforms most of the work required
; for this function.

        ; Code segment

        ; The code overlay is the first to be loaded as this has to be
        ; patched with the segment addresses of the remaining overlays.
        ; The segment address of the code overlay is placed in ES and
        ; the offset to patch is placed in BP.  When we load the Code
        ; overlay we place the address of  eseg  in ES:BP so we can
        ; patch in the segment required for the inter-segment jump into
        ; the Forth system.

        mov     bx,WORD PTR csize       ; Size of Segment
        mov     bp,OFFSET eseg          ; Offset in ES patch seg addr.
        mov     dx,OFFSET cfile         ; Segment extension
        call    loadseg                 ; Load the Forth segment

        mov     ax,WORD PTR eseg        ; Recover overlay segment addr
        mov     es,ax                   ; Set as ES

        ; Set  rsptr  field of Forth code segment to contain the segment
        ; address of the code overlay.  If a C overlay file is loaded
        ; this setting will be overwritten by the C code to the segment
        ; address of the C code.  This is to assist Forth to set up an
        ; error trap for spurious C calls (ie, those made when no C
        ; overlay is present).

        mov     es:[rsptr],ax

        ; Forth segment

        mov     bx,WORD PTR fsize
        mov     bp,fsptr
        mov     dx,OFFSET ffile
        call    loadseg

        ; Name segment

        mov     bx,WORD PTR nsize
        mov     bp,nsptr
        mov     dx,OFFSET nfile
        call    loadseg

        ; String segment

        mov     bx,WORD PTR $size
        mov     bp,$sptr
        mov     dx,OFFSET $file
        call    loadseg

        ; Stack segment

        ; Because the stack segment does not require to be initialised
        ; we do not have an overlay for it.  Here we simply obtain the
        ; memory required for the stack and patch the code overlay
        ; directly.

        mov     bx,WORD PTR ssize
        mov     ah,48h
        int     21h
        jnc     ssseg
        call    mem_err
ssseg:  mov     es:[ssptr],ax

        ; Load segment

        ; The  lsptr  field of the code overlay is set to the segment
        ; address of the currently execution program.  The setup chain
        ; of the Forth system will then be able to inspect (and execute)
        ; any text given on the command line after the program name.

        mov     ax,cs
        mov     es:[lsptr],ax


; Do we need to load in the C base program?
;
; The byte at  cname  is the count for an ASCIIZ string holding the name
; of the C overlay file.  If this count is 0 (zero) then the C overlay
; is not required, so execution is passed directly to the Forth Code
; overlay.

        mov     ax,ds
        mov     es,ax

        mov     si,OFFSET cname         ; Count byte of C overlay name
        mov     al,[si]
        inc     si
        cmp     al,0                    ; Is the count zero ?
        jne     covl                    ; No => Load the C overlay
        jmp     DWORD PTR eoff          ; Yes => jump to Forth code seg

; The byte at  cname  did not contain a 0 (zero), hence we must load
; (and execute) the C overlay file.  If the count byte is -1 the
; following 7fh bytes give the full pathname of the C overlay file.
; However, if the count is not -1, it should be the length of the
; filename (including the terminating zero) for the C overlay.  This
; filename will be appended to the current load path, used to load in
; the other overlays.

covl:   cmp     al,-1
        je      covl_2

        ; The count byte indicates the number of characters to add to
        ; the end of the current load pathname.  In order to do this we
        ; must first find the first character of the file name at the
        ; end of the most recently loaded overlay (string).  This can be
        ; done by scanning backwards from the . used to indicate the
        ; overlay type, looking for the \ used to indicate a directory
        ; name.  If the \ is not found within the maximum number of
        ; characters allowed for a file name (8 + 3) then an Environment
        ; error is indicated.

        mov     bp,OFFSET msg_3
        push    ax

        ; Find the '\'
        mov     cx,11
        mov     di,WORD PTR fdot
        mov     al,'\'
        std
        repne   scasb
        cld
        je      covl_1
        call    abort

covl_1: add     di,2

        ; Copy the given filename onto the end of the path
        pop     cx
        mov     ch,0
        rep     movsb

        ; Load and execute the overlay
        jmp     cload

        ; The count byte was -1, so copy the full pathname into the
        ; internal buffer.

covl_2: mov     di,OFFSET fname
        mov     cx,7fh
        rep     movsb

cload:  ; Load and execute the C overlay file.  The name of the file to
        ; load is in the internal name buffer  fname  .

        ; Initialise all seven loadblock fields to 0000h

        mov     di,OFFSET loadblock
        push    di
        mov     cl,7
        xor     ax,ax
        rep     stosw
        pop     di
        ; Initialise the new loadblock so that the C overlay inherits
        ; the eseg:eoff values as the first four bytes of its default
        ; FCB.  This is to pass the segment:offset address of the Forth
        ; code overlays entry address to the C system.

        mov     ax,cs
        mov     [di+8],ax       ; Default FCB segment
        mov     [di+4],ax       ; Command line
        mov     ax,OFFSET eoff
        mov     [di+6],ax       ; Default FCB holding eseg:eoff
        mov     ax,80h
        mov     [di+2],ax       ; Command line tail

        ; Invoke the program (C Overlay file).  Note the execution is
        ; passed to the C overlay file to allow the C initialisation
        ; code to be executed before the Forth system is invoked.  The
        ; overlay file will invoke the Forth system via the execution
        ; address passed to it in the default FCB.

        mov     dx,OFFSET fname
        mov     bx,di
        mov     ax,4b00h        ; Load and execute an overlay
        int     21h
        mov     bx,cs
        mov     ss,bx
        mov     sp,OFFSET stack
        jnc     execok
        call    load_err

execok: mov     al,0
        jmp     exit

;===========
; Subroutine:  Loadseg -> Load a given Forth++ Segment overlay
;===========
;
; On entry:
;          bx => No of paragraphs required for overlay
;       es:bp => addr to place segment addr of new overlay
;          dx => overlay name
;        fdot => addr of first char after the '.' in load file name

loadseg LABEL   NEAR
        push    es

        ; Grab the memory
        mov     ah,48h
        int     21h
        jnc     rdseg
        call    mem_err

        ; We got it and its seg addr is in ax - patch it into the
        ; code overlay.
rdseg:  mov     es:[bp],ax

        ; set es:bx to the parameter block "loadblock"
        push    cs
        pop     es
        mov     bx,OFFSET loadblock

        ; Initialise the load parameter block.
        mov     [bx],ax            ; destination segment
        mov     WORD PTR 2[bx],0   ; load Offset is set to zero

        ; Copy the overlay extension name over the .COM
        mov     si,dx
        mov     di,WORD PTR fdot
        mov     cl,3
        rep     movsb

        ; set ds:dx to file name

        mov     dx,OFFSET fname

        ; load file as overlay
        mov     ax,4b03h
        int     21h
        jnc     rdok
        call    load_err

rdok:   pop     es
        ret


;===========
; Subroutine: Mem_err -> Display a memory error condition and abort
;===========
;

mem_err:mov     bp,OFFSET msg_2

;===========
; Subroutine: Abort -> Display an error condition and abort
;===========
;
; BP => Error message to display
; AX => Error code

abort:  push    bp
        push    ax

        mov     bp,OFFSET msg_A  ; <CR><LF>Forth++ Load error <
        call    display

        pop     dx
        pop     bp
        pop     ax
        sub     ax,3
        call    hexw             ; Address of error
        mov     al,'/'
        call    char             ; Separator
        mov     ax,dx
        call    hexw             ; Error code (AX at time of error)

        call    display          ; Error text (for user) '> xxx'
        mov     bp,OFFSET msg_B  ;  !<CR><LF><BELL>
        call    display

        mov     al,1

;===========
; Subroutine: Exit - Exit back to Dos
;===========

exit:   mov     ah,4ch           ; Exit back to Dos
        int     21h
        jmp     exit

;===========
; Subroutine: Load_err -> Display a load error condition and abort
```

```
;===========
;
; fseg:foff => Address of ASCIIZ string holding file name

load_err:
        mov     dx,ax

        mov     bp,OFFSET msg_A  ; <CR><LF>Forth++ Load error <
        call    display

        pop     ax
        sub     ax,3
        call    hexw             ; Execution address
        mov     al,':'           ; Separator char
        call    char
        mov     ax,dx            ; Error code
        call    hexw

        mov     bp,OFFSET msg_5  ; > Can't find:
        cmp     dx,2
        je      le_1
        cmp     dx,3
        je      le_1

        mov     bp,OFFSET msg_6  ; > access denied when loading <CR><LF>
        cmp     dx,5
        je      le_1

        mov     bp,OFFSET msg_2  ; > Out of memory
        cmp     dx,8
        je      le_2

        mov     bp,OFFSET msg_3  ; > Environment error
        cmp     dx,0ah
        je      le_2

        mov     bp,OFFSET msg_4  ; > in

le_1:   call    display          ; Display error message

        mov     bp,OFFSET fname  ; Display File name (or err msg)
le_2:   call    display

        mov     bp,OFFSET msg_B  ;  !<CR><LF><BELL>
        call    display
        mov     al,1
        jmp     exit

;===========
; Subroutine: Display -> Display a ASCIIZ string
;===========
;
; BP => addr of ASCIIZ string to be displayed
;

display:cld
        push    si
        mov     si,bp
d1:     lodsb
        cmp     al,0
        je      dr
        call    char
        jmp     d1
dr:     pop     si
        ret
```

```
;===========
; Subroutine: HexW -> Display a hex word
;===========
;
; AX => Word to be displayed as four hex digits
;

hexw:   push    ax
        mov     al,ah
        call    hexb
        pop     ax

;===========
; Subroutine: HexB -> Display a hex byte
;===========
;
; AL => Byte to be displayed as two hex digits
;

hexb:   push    ax
        mov     cl,4            ;***  This is required as MASM (4.0) is
        shr     al,cl           ;***  not able to accept: shr al,4
        call    hexc
        pop     ax

;===========
; Subroutine: HexC -> Display a hex character
;===========
;
; AL => Bits 0..3 to be displayed as a single hex digit
;

hexc:   and     al,0fh
        add     al,'0'
        cmp     al,'9'+1
        jc      char
        add     al,'A'-('9'+1)

;===========
; Subroutine: Char -> Display a character on the video
;===========
;
; On entry:
;       AL => Character to be displayed
;

char:   push    dx
        mov     dl,al
        mov     ah,2
        int     21h
        pop     dx
        ret

;======================================================================
;                               DATA AREA
;======================================================================

; Error messages

msg_A:  db      13,10,'Forth++ Load error <',0
msg_B:  db      ' !',13,10,7,0

; The DOS must be version 3.0 or above
```

```
msg_1:  db      '> Dos 3.0 (or above) required',0

; An out of memory error occurs when a request for memory is denied.

msg_2:  db      '> Out of Memory',0

; An Environment error occurs when:
; 1. The environment table is longer than its maximum 32K.
; 2. The currently executing file name is longer than its max (7F bytes)
; 3. The . can not be found at the end of the file name
; 4. The \ can not be found at the start of the file name
; 5. Returned as an error from the load (4B) function

msg_3:  db      '> Environment Error',0

; When the load (or load and execute) overlay function (4B) is invoked,
; it may return an error code.  The following error messages will be
; displayed dependent on the error code returned.
;
;     Out of memory - Insufficient memory to load the overlay
; Environment error - Bad Environment
;        Can't find - File does not exist
;     Access denied - File access is denied
;                in - Any other error condition

msg_4:  db      '> in ',0

msg_5:  db      '> Can''t find: ',0

msg_6:  db      '> Access denied when loading',13,10,0


;=====================================================================
;                          Variable Space
;=====================================================================

; The offset address of the name that is attempting to be loaded (in
; case of an Can't find error) is stored in an internal variable.

foff    dw        ?


; The loadblock used to load the overlays (and C base program)

loadblock dw      7 DUP(?)


; A buffer to store the currently executing program name.  The name in
; this buffer will be manipulated to form the correct path name for the
; overlay files that make up the CForth++ system.

fdot    dw        ?
fname   db        80h DUP(?)

; Some stack area.

stack   dw        160 DUP(?)

entry   ENDP
CFL_TEXT ends
        end     entry
```

In order to generate the loader program, you should invoke the **MAKELOAD** batch file provided. This will produce the (**.COM** format) loader template **FLOADER.TPT**. This is a simple data file, loaded into memory (from 0100h) of the current load segment.

As this program is less than 2 Kbytes in length, it means that we can store it as a couple of FORTH blocks directly. When we need to alter the variables we can simply copy the FORTH blocks to the required file name to produce the new program loader.

C.2 Making the loader

The **MAKELOAD** batch file will process the **CFLOAD.ASM** file into the system loader (**FLOADER.TPT**). The file is assembled, linked and then converted into the '**.COM**' format before being renamed **FLOADER.TPT** as required by FORTH++'s **SAVE-SYS** command.

```
rem *** Assemble the loader ***
masm cfload , , , ,

rem *** Link it into a .EXE file ***
link cfload ;

rem *** Now convert it into a .BIN ***
exe2bin cfload.exe

rem *** Copy it to the FLOADER.TPT template ***
copy cfload.bin floader.tpt

rem *** Delete unwanted files ***
del CFLOAD.LST
del CFLOAD.OBJ
del CFLOAD.CRF
del CFLOAD.EXE
del CFLOAD.BIN
```

C.3 Overlay initialisation

This is the initial C code that is executed immediately after the loader program has loaded in the C module. It simply calls the assembler code routine **FINIT** to initialise the FORTH interface and then enters into a tight loop passing control to the FORTH system. When control is returned to the C system, it pops an integer value off from the FORTH stack and uses it as an index into a jump table of C routines. The C routine is then executed and control is passed back to the FORTH system.

```
#include "cforth.h"

/*********************************************************************/
/***                                                              ***/
/***                    CFORTH1.C Definitions                     ***/
/***                                                              ***/
/***   The following initialised structures are defined by the   ***/
/***   customer code, in the file CFORTH1.C                       ***/
/***                                                              ***/
/***                                                              ***/

/*
 * jmptbl[] is a table that contains functions that can be invoked from
 * the Forth system.  A function number is used as an index into the
 * table.
 */
```

```
extern TBL jmptbl[];

/*
 * The function startup() is invoked after initialisation of the C
 * system to allow the customer code to perform any initialisation that
 * it may require.
 */

extern void startup(void);

/***                                                            ***/
/***                 End of CFORTH.C definitions                ***/
/***                                                            ***/
/******************************************************************/

/******************************************************************/
/***                                                            ***/
/***                       Forth Interface                      ***/
/***                                                            ***/
/***  The following code is for the C Main function.  This code is  ***/
/***  the interface between C system and the Forth system.  It      ***/
/***  also provides the calling mechanism to allow the Forth system ***/
/***  to invoke the C functions given in the jump table.  The       ***/
/***  external functions (FINIT and FORTH) can be found in the MASM ***/
/***  assembly source code file CFASM.ASM.                          ***/
/***                                                            ***/
/***  This code should NOT be altered unless you are sure about it! ***/
/***                                                            ***/
/******************************************************************/

/*
 * Declare the external values.  The function FINIT() is called to
 * initialise the Forth++ system.  It will initialise the context
 * switching area.  The function FORTH() will save the C environment,
 * build the Forth environment and then continue execution of the Forth
 * system.  When the Forth system wants to invoke a C function it will
 * return to FORTH().  FORTH() will then swap execution environments and
 * return execution to the C system.  The Forth Stack Pointer is updated
 * on entry/exit of the Forth system.
 */

/*
 * Under Zortech C++ we must declare these functions as having C type
 * linking.  Hence the next two lines would be:
 *
 *             extern "C" { void FINIT(void); }
 *             extern "C" { void FORTH(void); }
 *
 * Note: This is only required for C++, the Zortech C compiler will
 *       error when given this code.
 */

extern void far FINIT(void);
extern void far FORTH(void);

main(void)
{
 /* Invoke the Forth initialisation code */

 FINIT();

 /* Invoke customer C start up code */

 startup();
 /* Execute Forth and interpret any C calls */

 {
  unsigned int i;
  while(1)
    {
     FORTH();
     i = POP(int);
     jmptbl[i].function();
    }
 }
}
```

C.4 Context Switching

This is the code that actually does the hard work of transferring control between the FORTH and C systems. This file is designed as a C module and is to be linked in with the users C code.

There are two assembler code routines in this file. `_FINIT` is called by the C initialisation code to initialise the C to FORTH data required by the `_FORTH` code. This will set up the initial FORTH entry address (including Code segment) as passed to it from the loader program (via the program segment prefix).

The `_FORTH` subroutine is called by the C when it wants to transfer control from the C system to the FORTH system. This subroutine simply stores the state of the C system (on the C stack) and then recovers the state of the FORTH system[1]. It will then pass control to the FORTH system.

When the FORTH system wants to pass control back to the C system, it will make an Inter-segment call to the label **freturn**. This code will save the current FORTH status, recover the C status (from the C stack) and return to the calling C code.

```
        PAGE    66,132
        TITLE   C to FORTH Interface  (P.J. Knaggs 08/08/90)

; This code is included as part of the C overlay.  It contains the MASM
; assembler code for the actual C to Forth interface.  Two routines are
; provided to be linked with the C system, they are _FINIT and _FORTH.

CFASM_TEXT segment byte public 'CODE'
        assume  cs:CFASM_TEXT,ds:_DATA

        SUBTTL  Initialise the Forth++ System

        PUBLIC  _FINIT
_FINIT  PROC    far

; void far _FINIT(void)
;
; Initialise the C data area for the context switching.  Also initialise
; the remaining part of the Forth system for the C overlay.

; The following equates give address in the Forth Code segment
; used to hold the address of the code to invoke the C

roptr   equ     10h     ; Return Offset pointer
rsptr   equ     12h     ; Return Segment pointer

ssptr   equ     0ch     ; Stack Segment pointer

; Save the C environment (on the C stack)
```

[1]From variables, as the C system needs access to the FORTH stack structure for argument handling

```
        pushf
        push    si
        push    di
        push    bp
        push    es
        push    ds

        mov     ax,_DATA
        mov     ds,ax

; Set Direction Flag to increment

        cld

; Extract the execution seg & offset of the Forth system from locations
; 5ch and 5eh of the PSP, where they will have been deposited by the
; Forth system loader.  Locations 5c and 5e are safe to use as they are
; in redundant PSP locations (actually a file control block).

        mov     ah,62h
        int     21h
        mov     es,bx

        ; Store the execution seg & offset in eseg & eoff

        mov     ax,WORD PTR es:5ch      ; Execution offset
        mov     eoff,ax

        mov     ax,WORD PTR es:5eh      ; Execution (Code) Segment
        mov     eseg,ax
        mov     es,ax

; We can now place the C execution vector into the Forth code segment.
; This is the address of the code that the Forth system is to execute
; when it transfers control back to the C system.  This is set to an
; error reporter by SAVE-SYS, but we now replace it to point to freturn.

        mov     ax,SEG freturn
        mov     es:[rsptr],ax          ; Return Seg Pointer
        mov     ax,OFFSET freturn
        mov     es:[roptr],ax          ; Return Offset Ptr

; We must set up the Forth stack (segment and offset) so that on the
; first execution of _FORTH the stack is at a sensible location.
; Thereafter it will be looked after by the Forth system.

        mov     ax,es:[ssptr]
        mov     WORD PTR sseg,ax
        mov     ax,4
        mov     WORD PTR soff,ax

; Recover the C environment from the stack

        pop     ds
        pop     es
        pop     bp
        pop     di
        pop     si
        popf
        ret
_FINIT  ENDP


        SUBTTL  Switch context between C and Forth and back again
        PUBLIC  _FORTH
_FORTH  PROC    far

; void FORTH(void) - Enter into the forth system
;
; When the C system has completed its task it will transfer control
; to the Forth system by executing this code.  When the Forth system
; wants to re-enter the C system it will invoke the code at freturn.
;
; The C environment is stored on the C stack before the SS:SP address
; is stored in cseg:coff.  All registers not saved can be discarded or
; recovered by the program.

; Save the C environment

        pushf
        push    si
        push    di
        push    bp
        push    es
        push    ds

; Save the C stack

        mov     ax,_DATA
        mov     ds,ax
        mov     cseg,ss
        mov     coff,sp

; Read Forth stack (from __FSP)

        mov     sp,soff
        mov     ss,sseg

; Set up for re-entry to Forth with a far return

        push    eseg
        push    eoff

; Restore Forth environment

        ; Forth++ uses the following registers:
        ;       SI, BP, BX, CS, DS, SS and SP
        ;
        ; We are changing the stack pointer so we must generate
        ; the new values of SS:SP (stored in the variable __FSP)
        ;
        ; The CS:IP value will be set when we return to the forth
        ; system.  The values are stored in eseg:eoff
        ;
        ; The remaining registers are stored in an environment buffer.
        ; They can not be stored on the Forth stack as the C code
        ; requires access to the Forth stack for argument passing, thus
        ; these values are stored in this environment buffer  fbuf .

        mov     si,fiip
        mov     bp,frsp
        mov     bx,fubp
        mov     ds,fds

; Clear direction flag (required in Forth inner interpreter)

        cld
```

```
; Execute the Forth system

        ret


; Forth to C
;
; When Forth is ready to invoke a C function, it will make an inter-
; segment call to the following code.  This will return to the C code
; which will, in turn, pop an integer value from the Forth stack and
; execute the corresponding jump table entry.

freturn label   near

; Set the data segment to access interface data

        push    ds
        mov     ax,_DATA
        mov     ds,ax

; Save Forth's environment

        ; As the C system requires access to the Forth stack (to get the
        ; function request number and any other argument passing) we
        ; can't store the systems state on the Forth stack.  Thus we
        ; must store it in the "environment buffer",  fbuf  .

        mov     fiip,si
        mov     frsp,bp
        mov     fubp,bx
        pop     fds

        ; Save Forth's re-entry point

        pop     eoff
        pop     eseg

        ; Save Forth's stack (in __FSP)

        mov     sseg,ss
        mov     soff,sp

; Recover the C environment

        ; Recover the C stack

        mov     ss,cseg
        mov     sp,coff

        ; Recover the C registers

        pop     ds
        pop     es
        pop     bp
        pop     di
        pop     si
        popf

; Return to the C system
        ret
_FORTH  ENDP

CFASM_TEXT ends

        SUBTTL  Data Area
```

```
_DATA   segment word public 'DATA'

; This is the Data Area required by the _FORTH function.

; The Following locations are the store for the Forth++ Environment

fbuf    label   word
fiip    dw      ?       ; Inner Interpreter ptr
frsp    dw      ?       ; Return Stack ptr
fubp    dw      ?       ; User Base ptr
fds     dw      ?       ; Data Segment

; Space to hold the C stack pointer

coff    dw      ?
cseg    dw      ?

; Space to hold the Forth Stack Pointer.

        PUBLIC  __FSP           ; Used by C to manipulate Forth Stack
__FSP   label   dword
soff    dw      ?
sseg    dw      ?

; Space to hold the Execution Address of the Forth system

eoff    dw      ?
eseg    dw      ?

_DATA   ends

        end
```

This module is merged with the `CFINIT.C` module to form a library. This makes the linking process much simpler. In order to build this library one must first obtain the "object code" for the two modules. To assemble the `CFASM.ASM` file one would give the command:

```
MASM CFASM, CFASM, NUL, NUL /mx
```

While the `CFINIT.C` modules needs to be compiled:

```
TCC -c -ml -G CFINIT
```

Here we are compiling the C module with Borland's "Turbo C" compiler. The '`-c`' indicates that we want to compile the file. The '`-G`' instructs the compiler to optimise the code for speed rather than size. Finally the '`-ml`' option instructs the compiler to use the "*L*arge" memory model.

We are now in a position to make the library. This we do by giving the command:

```
TLIB CFORTH /C +CFASM +CFINIT
```

In this instruction we are asking the system to generate a library named `CFORTH.LIB` which is case sensitive with regard to the public labels. This library consists of merging the two object code files `CFASM.OBJ` and `CFINIT.OBJ` that we have just produced.

C.5 Stack access

C access to the Forth stack is provided via a set of type independent macros. The header file `CFORTH.H` defines these macros, it should be included into the users C code.

```
/*****************************************************************/
/***                                                           ***/
/***                        CFORTH.H                           ***/
/***                                                           ***/
/***   The following lines of code make up the file CFORTH.H.   ***/
/***                                                           ***/

/*
 * entry() is a macro definition designed to help in the setting up of
 * the jump table.  To use this macro you must be defining the jump
 * table, simply type:
 *
 *              entry(func)
 *
 * where func is the name of the code required for the given entry.
 */

#define entry(func)     { func }

/***                                                          ***
 *** The following definitions set up the types required by the  ***
 *** jump table.                                              ***
 ***                                                          ***/

/*
 * The type PFI is defined to be a Pointer to an Function returning an
 * Integer.
 */

typedef int (*PFI)();

/*
 * The table entry structure is defined to be of type PFI.
 */

#define TBL     struct tabentry
TBL { PFI function; };

/**************************************************************/
/*                                                            */
/*                    Stack Manipulations                     */
/*                                                            */
/* The following functions are defined to Manipulate the Forth  */
/* systems stack.  Items can be popped off the stack and pushed */
/* onto it.  All communication between the C system and the     */
/* Forth system should be conducted via these functions.        */
/*                                                            */
/**************************************************************/

/*
 * The Forth Stack Pointer is stored as a far pointer to a void.
 * This is stored in the CFASM module for access by the assembly
 * code.
 */

extern void far * _FSP;

/*
 * The DROP(type) macro is defined to drop an item of the given type
 * from the Forth stack.
 */

#define DROP(type)      (type far *)_FSP += 1

/*
```

```
 * The macro INDEX(type,n) is used to return the n-th stack item of the
 * given type.  Hence INDEX(int,0) will return the TOP int on the stack,
 * while INDEX(double,1) will return the second double on the stack.
 */

#define INDEX(type,n)   *( (type far *)_FSP+n )

/*
 * The macro IINDEX(type,offset) is used in the same manner as the
 * INDEX() macro except that the offset value is in stack cells and has
 * no regard to type size.
 */

#define IINDEX(type,n)  *( (type far *) ( (int far *)_FSP+n ) )

/*
 * The macro POP(type) is used to POP an item of the given type from the
 * Forth stack.
 */

#define POP(type)       *(type far *)_FSP; (type far *)_FSP += 1

/*
 * The PUSH(type,val) macro is used to PUSH a given value (val) of the
 * given type onto the Forth stack.
 */

#define PUSH(type,n)    (type far *)_FSP -= 1; *(type far *)_FSP = n

/*
 * Note:
 * The macros POP and PUSH should have been defined as follows:
 *
 *      #define POP(type)       *(type far *)(_FSP++);
 *      #define PUSH(type,n)    *(type far *)(--_FSP) = n;
 *
 * However Zortech C was the only system that could handle this complex
 * a definition.  For both Microsoft and Borland the definition has to
 * be split as above.
 */

/***                                                                 ***/
/***                    End of CFORTH.H                              ***/
/***                                                                 ***/
/***********************************************************************/
```

C.6 User code

The user places his code in a separate “user” module. The file CFORTH1.C is an example user module. In this example module, we provide a floating point maths system using the C floating point code rather than developing our own.

The user *must* provide the ‘jmptbl’ jump table. The routine ‘setup’ must also be provided to initialise any user code.

The user can write any C code they like in this file. The functions that they want to be accessible from FORTH must be given in the jump table.

A function that is to be invoked from FORTH must not have any arguments and must not return a value. All argument passing should be performed by means of the macros provided in CFORTH.H. These macros actually manipulate the FORTH stack directly (as a C data structure).

```
/***********************************************************************/
```

```
/***                                                               ***/
/***                          CFORTH Header                        ***/
/***                                                               ***/
/***  You must include the CFORTH.H header file in order to define ***/
/***  the macros used to manipulate the Forth stack.  It also      ***/
/***  defines the structure of the jump table.                     ***/
/***                                                               ***/
/*********************************************************************/

#include "cforth.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*********************************************************************/
/***                                                               ***/
/***                     User Functions:                           ***/
/***                                                               ***/
/***  The following are the C functions which are to be called by  ***/
/***  Forth.  After they have been defined the functions will be   ***/
/***  placed in a jump table.  These functions do not pass         ***/
/***  parameters in normal C fashion and must have the prototype:  ***/
/***                                                               ***/
/***     int func(void)                                            ***/
/***                                                               ***/
/***  Ie, default functions taking no arguments.  Actually the     ***/
/***  functions pass their arguments via the Forth stack, which    ***/
/***  appears to the C program as a data structure.  Forth stack   ***/
/***  arguments are accessed with PUSH and POP macros.             ***/
/***                                                               ***/
/***  To pop a value use:      x = POP(type);  eg, x = POP(int);   ***/
/***  To push a value use:     PUSH(type,x);   eg, PUSH(int,-1);   ***/
/***                                                               ***/
/*********************************************************************/

/*
 * Declare variables to hold forth segment and offset.  A Forth call
 * will set fseg to the Forth memory space segment address. fseg and
 * ofs are used when a 16 bit Forth memory address is passed to a C
 * function.
 */

unsigned fseg, ofs;

/***                                                         ***
 ***            Provide floating point support.              ***
 ***                                                         ***/

/*
 * declare floating point stack and stack pointer.
 */

double fs[100], *fsp;
int base_pointer;

/***                                                               ***
 ***  Set up initial values for floating point stack pointer and   ***
 ***  floating point base pointer.  A margin of 5 elements is      ***
 ***  allowed to cater for stack underflow, which Forth will check ***
 ***  for after interpreting each command line.                    ***
 ***                                                               ***/

call_finit() { fsp=fs+5; base_pointer = 5; }

/***                                                               ***
```

```
 ***  The following code is invoked by the Forth system to set the  ***
 ***  fseg variable to the Forth data segment.  Thus allowing far   ***
 ***  addresses to be calculated (with MK_FP).                      ***
 ***                                                                ***/

setfseg() { fseg = POP(int); }

/*******************************************************************
 ***                                                             ***
 ***                  Floating point functions                   ***
 ***                                                             ***
 *******************************************************************/

call_fdiv() { *(fsp-1) = *(fsp-1) / *fsp; fsp-- ; }

call_float() { ++fsp; *fsp = (double) POP(int); }

call_fminus() { *(fsp-1) = *(fsp-1) - *fsp; fsp-- ; }

call_fmult() { *(fsp-1) = *(fsp-1) * *fsp; fsp-- ; }

call_fplus() { *(fsp-1) = *(fsp-1) + *fsp; fsp-- ; }

call_int() {
  int n;
  n = (int) *fsp; fsp--; PUSH(int,n); }

fppfrom() { PUSH(float,*fsp); fsp--; }

tofpp() { ++fsp; *fsp = POP(float); }

fdup() { ++fsp; *fsp = *(fsp-1); }

fdrop() { --fsp; }

fswap() {
  double temp;
  temp = *fsp; *fsp = *(fsp-1); *(fsp-1) = temp; }

fover() { ++fsp; *fsp = *(fsp-2); }

frot() {
  double temp;
  temp = *fsp; *fsp = *(fsp-2); *(fsp-2) = *(fsp-1); *(fsp-1) = temp; }

setstackpointer() { fsp = fs + POP(int); }

fetchstackpointer() { PUSH(int,fsp-fs); }

setbasepointer() { base_pointer = POP(int); }

fetchbasepointer() { PUSH(int,base_pointer); }

framefetch() {
  int index;
  index = POP(int); ++fsp; *fsp = fs[index+base_pointer+1]; }

framestore() {
  int index;
  index = POP(int); fs[index+base_pointer+1] = *fsp; --fsp; }

d_to_f() { ++fsp; *fsp = POP(long); }

f_to_d() { PUSH(long,*fsp); --fsp; }
```

```
fless() {
  if (*(fsp-1) < *fsp) { PUSH(int,-1); } else { PUSH(int,0); }
   --fsp; --fsp; }

fgreater() {
  if (*(fsp-1) > *fsp) { PUSH(int,-1); } else { PUSH(int,0); }
   --fsp; --fsp; }

fequal() {
  if (*(fsp-1) == *fsp) { PUSH(int,-1); } else { PUSH(int,0); }
   --fsp; --fsp; }

f0greater() {
  if (*fsp > 0) { PUSH(int,-1); } else { PUSH(int,0); }
  --fsp; }

f0less() {
  if (*fsp < 0) { PUSH(int,-1); } else { PUSH(int,0); }
  --fsp; }

f0equal() {
   if (*fsp == 0) { PUSH(int,-1); } else { PUSH(int,0); }
  --fsp; }

fdepth() { PUSH(int, fsp-fs-base_pointer); }

call_acos() { *fsp = acos(*fsp); }

call_asin() { *fsp = asin(*fsp); }

call_atan2() { *(fsp-1) = atan2(*fsp,*(fsp-1)); --fsp; }

call_cos() { *fsp = cos(*fsp); }

call_sin() { *fsp = sin(*fsp); }

call_cosh() { *fsp = cosh(*fsp); }

call_sinh() { *fsp = sinh(*fsp); }

sincos() {
  double temp;
  temp = *fsp; *fsp++ = sin(temp); *fsp = cos(temp); }

call_exp() { *fsp = exp(*fsp); }

call_fabs() { *fsp = fabs(*fsp); }

call_floor() { *fsp = floor(*fsp); }

call_frexp() {
  int n;
  *fsp = frexp(*fsp,&n); PUSH(int,n); }

call_ldexp() {
  int exp;
  exp = POP(int); *fsp = ldexp(*fsp,exp); }

call_log() { *fsp = log(*fsp); }

call_log10() { *fsp = log10(*fsp); }

call_modf() { ++fsp; *fsp = modf(*fsp,fsp-1); }

call_pow() { *(fsp-1) = pow(*(fsp-1),*fsp); --fsp; }
```

```
call_pow10() { *fsp = pow10(*fsp); }

call_sqrt() { *fsp = sqrt(*fsp); }

fround() {
  double ipart;
  if( modf(*fsp,&ipart) >= 0.5 ) ipart=ipart+1;
  *fsp = ipart; }

fmax() { if( *(fsp-1) < *fsp) *(fsp-1) = *fsp; --fsp; }

fmin() { if( *(fsp-1) > *fsp) *(fsp-1) = *fsp; --fsp; }

fnegate() { *fsp = -*fsp; }

dfpp_from() { PUSH(double,*fsp); --fsp; }

to_dfpp() { ++fsp; *fsp = POP(double); }

call_atan() { *fsp = atan(*fsp); }

call_tan() { *fsp = tan(*fsp); }

dfp_store() {
  double far *ptr;
  ofs = POP(int); ptr = MK_FP(fseg,ofs); *ptr = *fsp; --fsp; }

dfp_fetch() {
  double far *ptr;
  ofs = POP(int); ++fsp; ptr = MK_FP(fseg,ofs); *fsp = *ptr; }

/*******************************************************************/
/***                                                             ***/
/***                          Jump Table                         ***/
/***                                                             ***/
/***  The function table follows.  Functions placed in this table ***/
/***  are invoked by number when a C function call is received    ***/
/***  from Forth.  The code to process Forth C calls and invoke   ***/
/***  these functions is in CFINIT.C.  The entry macro, which in  ***/
/***  used to place entries into this table, is defined in the    ***/
/***  header file CFORTH.H                                        ***/
/***                                                             ***/
/*******************************************************************/


TBL jmptbl [] =
{
/* floating point initialisation */
entry(setfseg),

/* floating point support */
entry(call_fdiv),         entry(call_finit),    entry(call_float),
entry(call_fminus),       entry(call_fmult),    entry(call_fplus),
entry(call_int),          entry(fppfrom),       entry(tofpp),
entry(fdup),              entry(fdrop),         entry(fswap),
entry(fover),             entry(frot),          entry(setstackpointer),
entry(fetchstackpointer), entry(setbasepointer),entry(fetchbasepointer),
entry(framefetch),        entry(framestore),    entry(d_to_f),
entry(f_to_d),            entry(fless),         entry(fgreater),
entry(fequal),            entry(f0greater),     entry(f0less),
entry(f0equal),           entry(fdepth),        entry(call_acos),
entry(call_asin),         entry(call_atan2),    entry(call_cos),
entry(call_sin),          entry(call_cosh),     entry(call_sinh),
entry(sincos),            entry(call_exp),      entry(call_fabs),
entry(call_floor),       entry(call_frexp),    entry(call_ldexp),
entry(call_log),         entry(call_log10),    entry(call_modf),
entry(call_pow),         entry(call_pow10),    entry(call_sqrt),
entry(fround),           entry(fmax),          entry(fmin),
entry(fnegate),          entry(dfpp_from),     entry(to_dfpp),
entry(call_atan),        entry(call_tan),      entry(dfp_store),
entry(dfp_fetch)
};

/*******************************************************************/
/***                                                             ***/
/***                        Start Up                             ***/
/***                                                             ***/
/***  The following code is executed as part of the initialisation ***/
/***  sequence.  It should perform any initialisation required by   ***/
/***  the C code.                                                 ***/
/***                                                             ***/
/*******************************************************************/

void startup()
{
 call_finit();
}
```

C.7 Making the C Overlay

To make the C overlay, one must first compile the user code `CFORTH1.C` with the command:

```
TCC -c -ml -G -d CFORTH1
```

The '`-d`' option is instructing the compiler to merge any duplicate strings it may find.

Having compiled the user code successfully we are able to link the file with the `CFORTH.LIB` library that we produced earlier:

```
TLINK /c COL CFORTH1, CFORTH.OVL, CFORTH, CFORTH CL
```

In this command, we are linking the user code `CFORTH1.OBJ` with the standard prefix code `COL.OBJ`, the library `CFORTH.LIB` and the standard C library `CL.LIB`. We have instructed the system to produce the overlay file `CFORTH1.OVL`. In addition to this, the command will also produce a map file `CFORTH.MAP`. Finally the '`/c`' option is given to instruct the system that all label names are case sensitive.

As MS-DOS does not provide a "make" facility, we provide two batch files that will compile the C/FORTH interface library and the users code into the C overlay that would be loaded by the CFORTH++ system. The `MAKEOVLL.BAT` file uses the *L*arge memory model:

```
rem *** Make the library ***
masm cfasm cfasm nul nul /mx
tcc -c -ml -G -d -r- cfinit
tlib cforth /C -+cfasm -+cfinit

rem *** Compile the overlay CFORTH1.C ***
tcc -c -ml -G -d cforth1

rem *** Link it with the Large libraries ***
tlink /x \tc\lib\c0l cforth1, cforth1.ovl, cforth, cforth \tc\lib\mathl
         \tc\lib\emu \tc\lib\graphics \tc\lib\cl
```

It is also possible to use the *S*mall memory model. For this one should use the `MAKEOVLS.BAT` batch file. This should be used if at all possible, as the C system requires less overhead for a small model, rather than the large memory model.

```
rem *** Make the library ***
masm cfasm cfasm nul nul /mx
tcc -c -ms -G -d -r- cfinit
tlib cforth /C -+cfasm -+cfinit

rem *** Compile the overlay CFORTH1.C ***
tcc -c -ms -G -d cforth1

rem *** Link with the Small libraries ***
tlink /x \tc\lib\c0s cforth1, cforth1.ovl, cforth, cforth \tc\lib\maths
         \tc\lib\emu \tc\lib\graphics \tc\lib\cs
```

This batch file differs from the previous only in that it compiles both the `CFINIT` module and the `CFORTH1` user module using the small memory model. It also links the system together with the standard small libraries (`c0s`, `maths` and `cs` as opposed to `c0l`, `mathl` and `cl`).