Mixed Language Programming using SockPuppet with Professional Versions of MPE’s VFX Forth

Connecting Forth and C libraries via SockPuppet – adding interactive capabilities to C

Introduction

With the ever increasing complexity of microcontrollers such as the Cortex cores and systems, manufacturers are providing development hardware together with software systems based on C libraries for easier design.
These supplied libraries also reduce the requirement for chip documentation.
The conventional approach to providing support for these development boards in Forth had been to manually port the C library sources to Forth. **SockPuppet** takes a different approach by providing an interface solution between Forth and C.
The Forth system calls the underlying C libraries.
In turn, this allows the details of the hardware to be abstracted away by the C libraries, whilst allowing the Forth system to provide a powerful, uniform and interactive user interface.

The MPE ARM/Cortex Forth Cross-Compiler supports calling functions in C or any other language that can provide functions using the AAPCS calling convention. This is an ARM convention documented in *IHI0042F_aapcs.pdf*. The option of calls with a variable number of parameters (**varargs**) are not supported in this version.

Example code in both Forth and C is available for the Professional versions of the ARM Cortex cross compiler. The example code provides a simple GUI for an STM32F429I Discovery board using the sample C code provided by ST and others.
The interface is defined for Cortex-M CPUs only. If an ARM32 interface is required, contact MPE - http://www.mpeforth.com/contact.htm

This work is directly inspired by Robert Sexton's Sockpuppet interface:

```
https://github.com/rbsexton/sockpuppet
```

His contribution and permission are gratefully acknowledged.

How the Forth to C SockPuppet Interface works

Each function that is exported from the C world to the Forth world appears as one of a number of types of calls. These words are called externs.
You can handcraft these words in assembler, but the compiler includes code generators for several techniques.
The call format and return values match the AAPCS standard used by ARM C compilers.

There are several ways to call externs. They all have their pros and cons and are discussed in the following sections.

- **SVC Calls**. You just need to know the SVC numbers.
  SVC calls provide the greatest isolation between sections of code written in other languages. The functions foreign to Forth are accessed by SVC calls and/or jump tables.
  The example solution uses SVC calls for most foreign functions. Regardless of the primary call technique used, all techniques rely on a small number of SVC calls.
- **Jump Table.** The base address of the table can be set at run time, e.g. by making a specific SVC call. The calling words fetch the run-time address from the table, given an index.
- **Double Indirect Call.** A primary jump table is at a fixed address and contains the addresses of secondary tables, which hold the actual routine addresses.
  The fixed address and both indices must be known at compile time. This technique is used by the TI Stellaris parts and some NXP parts to accesss driver code in ROM.
- **Direct Calls** to the address of the routine. You need to know the address at compile time.

There is a practical limit of four arguments, if you use SVC calls for the insulation between Forth and C. If you want this limit changed, call MPE! The other 3 interface methods do not suffer from this limit.

It is a matter of convention between the Forth and C code as to parameter passing order. It can be changed by either side. MPE’s convention is for the left-most Forth parameter to be passed in R0. This matches the AAPCS code used by the hosted Forth compilers such as VFX Forth for ARM Linux. We strongly suggest that you use the MPE convention in order to take advantage of the future Sockpuppet developments.

SVC Calls

The examples provided use the MPE calling convention and are illustrated in assembler as well as by using the code generator.

```
SVC( 67 ) void BSP_LCD_DrawCircle( int x, int y, int r );
\ SVC 67 draw a circle of radius r at position (x,y).
```

```
CODE BSP_LCD_DrawCircle \ x y r --
\ SVC 67 draw a circle of radius r at position (x,y).
  mov r2, tos                     \ r
  ldr r1, [ psp ], # 4            \ y
  ldr r0, [ psp ], # 4            \ x
  svc # __SAPI_BSP_LCD_DrawCircle
  ldr     tos, [ psp ], # 4       \ restore TOS
  next,
END-CODE
```

When the SVC call occurs, the Cortex CPU stacks registers R0-R3, R12, LR, PC, xPSR on the calling R13 stack with R0 at the lowest address. The SVC handler places the address of this frame in R0, extracts the SVC call number, and jumps to the appropriate C function.
The functions all have the prototype:

```
  void doSVCcallN( uint32_t *frame);
```

Note: the original `R0..R3` are obtained as `frame[0]..frame[3]`.
If you use the MPE convention, the Forth parameters appear left to right starting at `frame[0]`.

The MPE convention leads to

```
void __SAPI_BSP_LCD_DrawCircle( uint32_t *frame )
{
  BSP_LCD_DrawCircle( frame[0], frame[1], frame[2] );
}
```

If there were return values, 32-bit values are returned in `frame[0]` and 64-bit values are returned in `frame[0]` and `frame[1]`, which correspond to R0 and R1 on return.

SVC calls provide the highest insulation between Forth and C, but suffer from several issues:

- The frame[] mechanism above is necessary, if the SVC despatch mechanism is implemented as a C table of function pointers. To permit arbitrary parameters for each call, the table must be in assembler.
- The frame[] mechanism is inefficient compared to a direct AAPCS handler. However, a more efficient mechanism will be offset by more register saves and restores.
- Because SVC calls are part of the CPU interrupt mechanism, you have to take care about how long a call takes. Changing the Cortex interrupt mechanism can fix this, but is complex.

Jump Table

In order to avoid the penalties of the SVC call mechnism, you can define an array of function pointers in C or assembler and call functions using an index into this table.

```
jumptable:
  dd func0          ; address of function 0
  dd func1          ; address of function 1
  ..
```

We still need to know the address of the jump table. This is found using an SVC call (15) and will be stored in a variable. The jump table address could be hard-coded, but given the complexity of twisting link map files and the like, the overhead of a single SVC call is preferable.

```
SVC( 15 ) void * GetDirFnTable( void );
\ Define SVC call that returns the address of the
\ jump table.

variable JT     \ -- addr
\ Holds the address of the jump table.
JT holdsJumpTable
\ Tell the cross compiler where the jump table address
\ is held.

: initJTI       \ -- ; initialise jump table calls
  GetDirFnTable JT !  ;

JTI( n ) int open(
  const char * pathname, int flags, mode_t mode
);
```

If constructed in assembler, the SVC despatch table and the main jump table can be the same table; it is just a question of what is put into the table.

Double Indirect Call Tables

Before use, you will have to declare the base address of the primary ROM table used for calling ROM functions. For Luminary/TI CPUs, this will probably be:

```
$0100:0010 setPriTable
```

Now you can define a set of ROM calls, for example, again for a TI CPU.

```
DIC( 4, 0 ) void ROM_GPIOPinWrite(
   uint32 ui32Port, uint8 ui8Pins, uint8 ui8Val
);
```

where

ROM_APITABLE	is an array of pointers located at 0x0100.0010.
ROM_GPIOTABLE	is an array of pointers located at ROM_APITABLE[4].
ROM_GPIOPinWrite	is a function pointer located at ROM_GPIOTABLE[0].

Parameters:

ui32Port	is the base address of the GPIO port.
ui8Pins	is the bit-packed representation of the pin(s).
ui8Val	is the value to write to the pin(s).

Description:

Writes the corresponding bit values to the output pin(s) specified by ui8Pins. Writing to a pin configured as an input pin has no effect. The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:
None.

To call this function, use the Forth form:

```
port pins val ROM_GPIOPinWrite
```

Direct calls

Where the address of the routine is known at Forth compile time, you can use a direct call.

```
DIR( addr ) int foo( int a, char *b, char c );
```

The Forth word marshalls the parameters and calls the subroutine at target address *addr*.

SVC call number list

The demonstration code provided here mostly uses SVC calls. To call the SVC, a small Forth word puts the arguments into registers, calls the SVC, and then returns a value if required.
SVCs are identified by a number, which corresponds here to an index into a table.

The list must match the equivalents in the C code `SVC_syscall_table[]`. To ease sharing of data between the Forth and C systems, you can use the **`enum`** parser described in the **"Interpreter directives"** chapter of the generic cross compiler manual.

The first 16 entries (0..15) are defined by the Sockpuppet system. Each application is free to use entries 16 onwards as suits the application.

This list will be different for every application.

```
const=equ       \ constants invisible on target
\ const=constant  \ constants visible on target
#00 const __SAPI_00_ABIVersion          \ SVC 00  - Required
#01 const __SAPI_01_GetLinkList         \ SVC 01  - Required
#02 const __SAPI_02_PutChar             \ SVC 02
#03 const __SAPI_03_GetChar             \ SVC 03
#04 const __SAPI_04_GetCharAvail        \ SVC 04
#05 const __SAPI_05_PutCharHasRoom      \ SVC 05
#06 const __SAPI_06_SetIOCallback       \ SVC 06
#07 const __SAPI_07_GetTimeMS           \ SVC 07 - Required
#08 const __SAPI_08_NVICReset           \ SVC 08

#10 const __SAPI_10_LauchUserApp        \ SVC 10 - Reserved
#11 const __SAPI_11_MPULoad             \ SVC 11 - Reserved
#12 const __SAPI_12_GetPrivs            \ SVC 12 - Reserved
#13 const __SAPI_13_GetUsageCPU         \ SVC 13 - Reserved
#14 const __SAPI_14_PetWatchdog         \ SVC 14 - Reserved
#15 const __SAPI_15_GetFnTable          \ SVC 15 - Required

#20 const __SAPI_20_GetIP               \ SVC 20
#21 const __SAPI_21_PBuf_Ram_Alloc      \ SVC 21
#22 const __SAPI_22_PBuf_Free           \ SVC 22
#23 const __SAPI_23_DNS_GetHostByName   \ SVC 23
#24 const __SAPI_24_UDP_GetPCB          \ SVC 24
#25 const __SAPI_25_UDP_Connect         \ SVC 25
#26 const __SAPI_26_UDP_Recv            \ SVC 26
#27 const __SAPI_27_UDP_Send            \ SVC 27
#28 const __SAPI_28_GetTimeMS           \ SVC 28
#29 const __SAPI_29_GetKeyButton        \ SVC 29
#30 const __SAPI_30_SetLeds              \ SVC 30
#31 const __SAPI_31_SetLedsHW           \ SVC 31
\ Graphics and Touchscreen
#32 const __SAPI_32_WaitForPressedState \ SVC 32
#33 const __SAPI_33_BSP_TS_GetState     \ SVC 33
#34 const __SAPI_34_Touchscreen_Cal     \ SVC 34
#35 const __SAPI_35_Touchscreen_demo    \ SVC 35

#37 const __SAPI_BSP_LCD_Init           \ SVC 37
#38 const __SAPI_BSP_LCD_GetXSize       \ SVC 38
#39 const __SAPI_BSP_LCD_GetYSize       \ SVC 39
#40 const __SAPI_BSP_LCD_LayerDefaultInit           \ SVC 40
#41 const __SAPI_BSP_LCD_LayerDefaultInitPixelForm  \ SVC 41
#42 const __SAPI_BSP_LCD_SetTransparency            \ SVC 42
#43 const __SAPI_BSP_LCD_SetLayerAddress            \ SVC 43
#44 const __SAPI_BSP_LCD_SetColorKeying             \ SVC 44
#45 const __SAPI_BSP_LCD_ResetColorKeying           \ SVC 45
#46 const __SAPI_BSP_LCD_SetLayerWindow             \ SVC 46
#47 const __SAPI_BSP_LCD_SelectLayer                \ SVC 47
#48 const __SAPI_BSP_LCD_SetLayerVisible            \ SVC 48
#49 const __SAPI_BSP_LCD_SetTextColor   \ SVC 49
#50 const __SAPI_BSP_LCD_SetBackColor   \ SVC 50
#51 const __SAPI_BSP_LCD_GetTextColor   \ SVC 51
#52 const __SAPI_BSP_LCD_GetBackColor   \ SVC 52
```

```
#53 const __SAPI_BSP_LCD_SetFont          \ SVC 53
#54 const __SAPI_BSP_LCD_GetFont          \ SVC 54
#55 const __SAPI_BSP_LCD_ReadPixel        \ SVC 55
#56 const __SAPI_BSP_LCD_DrawPixel        \ SVC 56
#57 const __SAPI_BSP_LCD_Clear            \ SVC 57
#58 const __SAPI_BSP_LCD_ClearStringLine        \ SVC 58
#59 const __SAPI_BSP_LCD_DisplayStringAtLine    \ SVC 59
#60 const __SAPI_BSP_LCD_StrAtLineMode          \ SVC 60
#61 const __SAPI_BSP_LCD_DisplayStringAt        \ SVC 61
#62 const __SAPI_BSP_LCD_DisplayChar      \ SVC 62
#63 const __SAPI_BSP_LCD_DrawHLine        \ SVC 63
#64 const __SAPI_BSP_LCD_DrawVLine        \ SVC 64
#65 const __SAPI_BSP_LCD_DrawLine         \ SVC 65
#66 const __SAPI_BSP_LCD_DrawRect         \ SVC 66
#67 const __SAPI_BSP_LCD_DrawCircle       \ SVC 67
#68 const __SAPI_BSP_LCD_DrawPolygon      \ SVC 68
#69 const __SAPI_BSP_LCD_DrawEllipse      \ SVC 69
#70 const __SAPI_BSP_LCD_DrawBitmap       \ SVC 70
#71 const __SAPI_BSP_LCD_FillRect         \ SVC 71
#72 const __SAPI_BSP_LCD_FillCircle       \ SVC 72
#73 const __SAPI_BSP_LCD_FillTriangle     \ SVC 73
#74 const __SAPI_BSP_LCD_FillPolygon      \ SVC 74
#75 const __SAPI_BSP_LCD_FillEllipse      \ SVC 75
#76 const __SAPI_BSP_LCD_DisplayOff       \ SVC 76
#77 const __SAPI_BSP_LCD_DisplayOn        \ SVC 77
#78 const __SAPI_BSP_LCD_GetFontTable     \ SVC 78

#81 const __SAPI_tsWaitUp                 \ SVC 81
#82 const __SAPI_tsWaitDown               \ SVC 82
```

Words containing an SVC call

`svc( 0 ) int SAPI-Version( void );`
SVC 00: Return the version of the API in use.

`svc( 1 ) int GetSharedVars( void );`
SVC 01: Get the address of the shared variable list.

`svc( 15 ) int GetSvcFnTable( void );`
SVC 15: Get the address of the SVC function table.

`: FN  \ n --`
Call a function in the SVC function table directly.
SVC calls that take a long time may/will block interrupts such as the system ticker, and thus will fail. To avoid this, such calls should be called directly so that they do not affect the interrupt priority level.
The functions **must** be declared as

```
  void function( void );
```

as the normal SVC parameter passing mechanism is completely bypassed. Note also that these functions inhibit the Forth multitasker for the duration of the call.
It is usually better to refactor the C library if you wish to use the Forth multitasker.

`svc( 28 ) int ticks( void );`
SVC 28: Get the ms counter. Returns a value in milliseconds that eventually wraps around.

`svc( 29 ) int SAPI_GetKeyButton( void );`
SVC 29: Get the Key Button value

`svc( 30 ) void SAPI_SetLeds( int mask );`
SVC 30: Set LED3 (bit0) and LED4 (bit1) via BSP.

`svc( 31 ) void SAPI_SetLedsHW( int mask );`
SVC 31: Set LED3 (bit0) and LED4 (bit1) via direct hardware access

`svc( 33 ) void SAPI_33_BSP_TS_GetState( void * TSstruct );`
SVC 33: Get the touchscreen coordinates into a structure.

Reserved calls

These calls are commented out by default and are not present in all systems. However, the SVC numbers are reserved. Note that some stack manipulation is required in some words - they may not be direct translations.

`CODE SetIOCallback  \ addr iovec read/write -- n`
Set the IO Callback address for a stream. *Iovec*, read/write (r=1), address to set as zero. Note the minor parameter swizzle here to keep the old value on TOS.

`CODE RestartForthApp ( addr ) \ --`
Do a stack switch and startup the user App. Its a one-way trip, so don't worry about stack cleanup.

`CODE MPULoad  \ --`
Ask for MPU entry updates.

`CODE privmode  \ --`
Request Privileged Mode. In some systems, this is a huge security hole.

`CODE PetWatchDog  \ --`
Refresh the watchdog

`CODE GetUsage  \ -- n`
The number of CPU cycles consumed in the last second.

GUI SVC calls

These calls are defined for the demonstration code only. Your application code may reuse the numbers for other functions.

`SVC( 49 ) void BSP_LCD_SetTextColor( int color );`
SVC 49: Set text and line drawing colour.

`SVC( 50 ) void BSP_LCD_SetBackColor( int color );`
SVC 50: Set background colour for text

`svc( 56 ) void BSP_LCD_DrawPixel( int x, int y, int colour );`
SVC 56: Draw a pixel at a at screen position (x,y)

`svc( 57 ) void SAPI_BSP_LCD_Clear( int colour );`
SVC 57: Set the LCD to the given colour.

`svc( 60 ) void BSP_LCD_DisplayStringAtLineMode( int line, char * string, int mode);`
SVC 60 Display the text at a on line a using the given mode

`svc( 61 ) void BSP_LCD_DisplayStringAt( int x, int y, char * string, int mode );`
Draw a string at position (x,y) in the given alignment mode.

```
$01 constant CENTER_MODE   \ center mode
$02 constant RIGHT_MODE    \ right mode
$03 constant LEFT_MODE     \ left mode
```

`svc( 63 ) void BSP_LCD_DrawHLine( int x, int y, int length );`
SVC 63: Draw a horizontal line at position (x,y) of length.

`svc( 64 ) void BSP_LCD_DrawVLine( int x, int y, int length );`
SVC 64: Draw a vertical line at position (x,y) of length.

`svc( 65 ) void BSP_LCD_DrawLine( int x1, int y1, int x2, int y2 );`
SVC 65: Draw a line between two points.

`svc( 66 ) void BSP_LCD_DrawRect( int x, int y, int w, int h );`
SVC 66: Draw a rectangle.

`svc( 67 ) void BSP_LCD_DrawCircle( int x, int y, int radius );`
SVC 67: Draw a circle of the given radius at screen position (x,y)

`SVC( 70 ) void BSP_LCD_DrawBitmap( int x, int y, void * image );`
SVC 70: draw the bitmap image at addr at the screen position (x,y).

`svc( 71 ) void BSP_LCD_FillRect( int x, int y, int w, int h );`
SVC 71: Draw a filled rectangle at (x,y) of size (w,h).

`svc( 72 ) void BSP_LCD_FillCircle( int x, int y, int radius );`
SVC 72 draw a filled circle of the given radius at screen position (x,y)

Debug tools

`: mainSP@        \ -- addr`
Return the main stack pointer.

`: procSP@        \ -- addr`
Return the process stack pointer.

Sharing data between Forth and C

In the MPE environment, **`VALUE`**s work very well for this. **`VALUE`**s are defined at compile time and can be updated at runtime. So, the basic logic is - walk the dynamic list, and if you find something that is already defined, assume it's a **`VALUE`** otherwise, generate a **`CONSTANT`**.

The code is based around 32 byte records. They are accessed from both Forth and C.

C Linkage structure

```
#define DYNLINKNAMEMLEN 22

typedef struct {
  // This union is a bit crazy, but it's the simplest way of
  // getting the compiler to shut up.
  union {
     void (*fp) (void);
     int*  ip;
     unsigned int    ui;
     unsigned int*  uip;
     unsigned long* ulp;
  } p;             ///< Pointer to the object of interest (4)
  int16_t size;    ///< Size in bytes (6)
  int16_t count;   ///< How many (8)
  int8_t kind;     ///< Is this a variable or a constant? (9)
  uint8_t strlen; ///< Length of the string (10)
  const char name[DYNLINKNAMEMLEN]; ///< Null-Terminated C string.
} runtimelink_t;
```

When the Forth system powers up, it runs the Forth word **`dy-populate`** which uses SVC call 01 to get the address of the `dynamiclinks[]` table, and walks through the table creating Forth named variables whose addresses match those in the C system.

A Forth word **`dy-show`** is provided to list the entries in the table.

Forth Linkage structure

```
interpreter
: hword  2 field  ;
: byte   1 field  ;
target

struct /runtimelink     \ -- len
\ Forth equivalent of the C structure above.
  int   fdy.val         \ usually a pointer     0, 4
  hword fdy.size        \ size in bytes         4, 2
  hword fdy.count       \ how many              6, 2
  byte  fdy.type        \ variable or constant  8, 1
  byte  fdy.nlen        \ name length           9, 1
  22 field fdy.zname    \ zero terminated name  10, 22
end-struct
```

The accessor words just read the fields defined above. They are defined as compiler macros. For interaction on the target, use the field names above.

```
compiler
: dy.val        fdy.val @ ;      \ addr -- n
: dy.size       fdy.size w@ ;    \ addr -- w
: dy.count      fdy.count w@ ;   \ addr -- w
: dy.type       fdy.type c@ ;    \ addr -- c
: dy.name       fdy.nlen  ;      \ addr -- addr'
target
```

Accessing shared data from Forth

`: dy-recordlen  \ -- n`
Return the recordlength. Its the first thing.

`: dy-first  \ -- addr`
return the first entry in the dynamic variable list

`: dy-next  \ addr -- addr`
return the next entry in the dynamic variable list

`: dy-stuff  \ n xt --`
Store n in the VALUE defined by xt

`: make-const  \ n addr len --`
Create a **`CONSTANT`** of name *addr/len* and value *n* by laying down a header and some machine code. A key trick is that we have to lay down a pointer to the first character of the definition after we finish This is what a constant looks like after emerging from the compiler.

```
( 0002:0270 4CF8047D Lx.} )  str r7, [ r12, # $-04 ]!
( 0002:0274 004F .O )        ldr r7, [ PC, # $00 ] ( @$20278=$7D04F84C )
( 0002:0276 7047 pG )        bx LR
```

`: dy-create  \ addr --`
Look for an entry and if its a value, update it.

`: dy-populate  \ --`
Walk the table and make the constants.

`: dy-print  \ addr --`
Dump out an entry as a set of constants

`: dy-show  \ --`
Walk the table and show the entries

`: dy-compare  \ addr n addr -- n`
Given a record address and a string, figure out if it is the one we're looking for.

`: dy-find  \ addr n -- addr|0`
Walk the dynamic variable list and find a string, return its address if found, else 0.

The Sockpuppet C code

The demonstration system runs on an STM32F429I Discovery board with a QVGA colour display. The Forth demonstration code uses a mixture of SVC calls and direct calls into the C jump table. This section documents the C code. The examples are taken from the file *SockPuppetInterface.c*.

SVC handler

The SVC handler runs a short piece of assembler that permits the use of both the main and process stacks. This part of the current code is not AAPCS compliant and forces the use of a slightly clumsy argument extraction technique. A future version of the code will be fully AAPCS compliant for up to four arguments.

```
void SVC_Handler(void)
{
    __asm( "tst    lr,#0x4" );           // Figure out which stack
    __asm( "ite    eq" );
    __asm( "mrseq  r0,msp" );            // @ Main stack
    __asm( "mrsne  r0,psp" );            // Process/Thread Stack

    __asm( "ldr    r1,[r0,#24]" );       // Get the stacked PC
    __asm( "ldrb   r1,[r1,#-2]" );       // Extract the svc call number
    __asm( "and    r1, #0x7F" );         // Range Checking 128 element table

    __asm( "ldr    r2,=SVC_syscall_table" );
    __asm( "ldr    r2, [r2, r1, LSL #2]" );
    __asm( "mov    pc,r2" );
    // "@ No more code...  We've already jumped."
}
```

On entry to the C procedure R0 contains the address of the stacked data frame, where **`frame[0]`** contains the calling R0. Data is returned by setting **`frame[0]`**.

```
svc( 56 ) void BSP_LCD_DrawPixel( int x, int y, int colour );
\ SVC 56: Draw a pixel at a at screen position (x,y)
```

jumps to the C function below:

```
void __SAPI_BSP_LCD_DrawPixel( uint32_t *frame )
{
   BSP_LCD_DrawPixel( frame[0], frame[1], (uint32_t)frame[2] );
}
```

The following function returns data.

```
void __SAPI_15_GetFnTable( uint32_t *frame )
{
    frame[0] = (uint32_t)SVC_syscall_table;
}
```

Building the demonstration

The demonstration code is for an STM32F429I Discovery board, which includes a QVGA colour display.

The Forth sources are provided in the Forth cross-compiler distribution in the *Lib/SockPuppet* directory. The Forth toolchain is the MPE Forth cross compiler.

The demonstration C source code is provided as a ZIP file with the cross compiler download.
The C toolchain is the version of GCC maintained by ARM at:

```
https://launchpad.net/gcc-arm-embedded
```

Flash programming is performed by using ST's STM32 ST-Link utility, usually found at:

```
http://www2.st.com/content/st_com/en/products
/embedded-software/development-tool-software/stsw-link004.html
```

The ST-Link software and drivers must be installed before the board can be programmed.

Building the C layer

Windows Batch files are supplied in the *C_files* folder:

- MakeAll.bat – make the C library system
- MakeAllFlash.bat – make the C library system and download it to the target Flash.

These files trundle around the distribution to run the make systems, gcc and to program the Flash.

The C system is based on the STM32F4 LCD system by Pierpaolo Bagnasco, available from:

```
http://www.pierpaolobagnasco.com/category/stm32f4xx/
http://www.pierpaolobagnasco.com/2014/07/13/stm32f429-discovery-display/
```

Pierpaolo uses Eclipse to auto-generate makefiles so that the ST supplied libraries can be compiled using the GNU C ARM command line interface tools

```
http://thehackerworkshop.com/?p=1056
```

To avoid being tied to any particular IDE, the makefiles are now edited manually. If you want to know the gory details of editing makefiles produced by Eclipse, Google will help to find the solution.

The following files have been modified or added:

- C_files\src\main.c (modified)
- C_files\src\ SockPuppetInterface.c/h (added)
- C_files\src\stm32f4xx_it.c (modified)

Main.c initialises hardware components that we are using, then enters an infinite loop testing for the presence of the Forth system in the high 1 Mbyte of FLASH memory.
If found, the C system jumps to the StartCortex entry point in the Forth system.

SockPuppetInterface.c/h define the SockPuppet interface for the C toolchain.

stm32f4xx_it.c defines the SVC exception handler that jumps to the function in the SVC_syscall_table corresponding to the SVC number.

The C compiler used is the GNU Tools ARM Embedded\5.2 2015q4 gcc-arm-none-eabi-5_2-2015q4-20151219-win32.exe available from here:

```
  https://launchpad.net/gcc-arm-embedded/5.0/5-2015-q4-major
```

The makefile Make.exe is from GNU ARM Eclipse\Build Tools\2.6-201507152002 gnuarmeclipse-build-tools-win32-2.6-201507152002-setup.exe available from here:

```
http://sourceforge.net/projects/gnuarmeclipse/files/Build%20Tools/
```

A copy of make.exe is included in the C_files\Debug\ directory.

The output file is *C_files\Debug\Display.hex* and runs from location 0x0800000. 1 Mbyte of Flash and 128K bytes of RAM are allocated for the C system. This is grotesquely over the top but works. For a production environment the memory map should be edited.

Building the Forth system

There is a control file for the SockPuppet demo:

```
  Cortex/Hardware/STM32F4/SP429disco.ctl
```

Edit this file so that the line

```
 afterwards sh "C:\MyApps\ST-Linkv3.8.1\ST-LINK Utility\ST-LINK_CLI.exe"  -P
SP429DISCO.hex  -Rst
```

contains the correct path to the command line version of the ST-Link utility.

Set up a new AIDE project to compile this file or compile it directly. At the end the Forth image will have been programmed and the application run.

Required target files

The build processes should have programmed the required files. In case you just have the hex files, you can program them into the board using the ST-Link utility. Find the files

- Display.hex - the C image. Usually in *C_files\Debug\Display.hex*.
- SP429DISCO.hex - the Forth image. Usually in *<xArmCortex>\Cortex\Hardware\STM32F4\SP429DISCO.hex*.

Memory map

The STM32F429 chip has 2 Mbytes of Flash and 256 Kbytes of RAM on chip. The available memory is divided equally between the C and the Forth portions:

```

C System Flash      0x08000000 0x100000
Forth System Flash  0x08100000 0x100000
C System RAM        0x20000000 0x020000
Forth System RAM    0x20020000 0x020000
```

The STM32F429 fetches the stack and PC from address 0 at reset, and 0x400 bytes from 0x08000000 are initially mapped to address 0, allowing the chip to effectively boot from the start of the C system Flash at 0x08000000.

The C system checks the location 4 bytes offset from the start of the Forth system Flash, 0x08100004, which contains the address of the word **`StartCortex`** in the Forth system. Note that the value at this address always has bit 0 set to 1, indicating Thumb code, since the entire chip runs in Thumb mode. The value at the start of the Forth system Flash, 0x08100000, normally contains the address to be used for the stack, but the xArmCortex Forth start-up code has been modified not to use this value, but to continue to use the C system stack. This allows C library functions to be called in their own stack environment.

If the value at 0x08100004 is not 0xFFFFFFFF or 0x00000000 the C system hands control over to the Forth system, which can then access C library functions via the SockPuppet interface.

Southampton, 28 April 2016

Version 1.0