# The Sockpuppet Forth to C interface

Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England
t: +44 (0)23 8063 1441
e: sfp@mpeforth.com
w: www.mpeforth.com

## Abstract

*As processors become ever more complex and the software we are asked to write becomes more complex, it takes ever longer to write the basic drivers for an embedded system. A full digital audio chain is vastly more complex than pumping DAC output into an audio amplifier. Silicon vendors provide C libraries to make using their chips easier. Rather than convert these libraries to Forth, MPE now provides a mechanism to call the C library from Forth.*

## Introduction

For microcomputers such as the Cortex cores and systems, manufacturers are providing software systems based on C libraries to make using their chips easier. Such libraries reduce the requirement for chip documentation at the expense of software documentation. This tendency has increased to the level that C header files include registers undocumented in the chip user manual. The manufacturer's focus has changed from documentation to time-to-first-hello. The penalty is that chips take longer to learn and the documentation has more errors.

The conventional approach to providing support for development boards in Forth has been to manually port the C library sources to Forth. The SockPuppet system 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. This MPE code is directly inspired by Robert Sexton's Sockpuppet interface:

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

His contribution and permission are gratefully acknowledged.

Interfacing code from programming language to another is usually called "**mixed language programming**". The MPE ARM/Cortex Forth cross-compiler supports Forth calling functions in C or any language that can provide functions that use the AAPCS calling convention. This is an ARM convention documented in IHI0042F aapcs.pdf. Calls with a variable number of parameters (varargs) are not supported.

The example code in both Forth and C is available for the Professional versions of the ARM Cortex cross compiler with automatic code generation of five interface types. The example code provides a simple GUI for an STM32F429I Discovery board using sample C code provided by ST and others. A version for the BBC micro:bit is in preparation. The interface is currently defined for Cortex-M CPUs only. All versions of the compiler can be used with hand-written assembler code.

## How the Forth to C interface works

Both Flash and RAM memory are partitioned, one pair for C and the other for Forth. Because of the arcane and undocumented nature of start-up for C compiler target code, the initial boot of the system is performed by the C code in order to make sure that the initialisation is correct.

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

Each calling technique has its own pros and cons. They are discussed in 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 call technique used by the majority of your code, 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. This technique has good performance and few problems.
- 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 TI’s Stellaris and some NXP parts to access 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 because Cortex CPUs automatically stack four registers for an interrupt. The other 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 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.

### *SVC calls*

The examples use the MPE calling convention and are illustrated in assembler as well as by using the code generator. The code generator interface is much to be preferred and preserves far more of the information in the C prototype. The decision to use the C prototype is deliberate and follows long-established practice in MPE’s hosted systems.

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

The code generator parses the extern definition above and generates the extern as a function with three parameters implemented as SVC call 67. If you really want to demonstrate your assembler prowess, the code below performs the same operation.

```
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/R4, extracts the SVC call number, reloads the AAPCS parameters from the frame and jumps to the appropriate C function. In this case

```
void BSP_LCD_DrawCircle(
  uint16_t Xpos, uint16_t Ypos, uint16_t Radius
);
```

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

- The SVC call mechanism is part of the Cortex interrupt and exception system. The assembler and/or C side of this uses code written in assembler to allow the C routines called from a jump table to to be AAPCS compliant.
- The SVC mechanism is inefficient compared to a direct AAPCS handler.
- Because SVC calls are part of the CPU interrupt mechanism, you have to care how long a call takes. Playing games with the Cortex interrupt mechanism can fix this, but is complex.

### *Jump table*

In order to avoid the run-time penalties of the SVC call mechnism, you can make an array of function pointers in C or assembler and call functions using an index into the 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 stored in a variable. The jump table address could be hard-coded, but given the horrors of perverting link map files and the like, the overhead of a single SVC call is preferable.

```
SVC( 15 ) void * GetDirFnTable( void );
\ Returns the address of the jump table.
variable JT \ -- addr
\ Holds the address of the jump table.
JT holdsJumpTable
\ Tell cross compiler where 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's just a question of what you put in the table.

### *Double indirect call tables*

Some vendors, particularly TI, use a table of tables approach. The sub-tables provide the API for a particular peripheral, e.g. UARTs. Before use, you 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).

To call this function, use the Forth form:

```
  port pins val ROM_GPIOPinWrite
```

### *Direct calls*

Where the address of the routine is known at the 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.

## Extracting information from C

It is convenient to have a certain amount of information available from the C portion of the code. This is supported by a few SVC calls that exist in all versions of the Sockpuppet API.

```
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.
```

In order to support data sharing between C and Forth, the C can export named objects which can appear as Forth words.

### *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-Term 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
```

A set of support words allow us to run down the table and create Forth **VALUE**s and **CONSTANT**s.

## Demonstration code

In order to evaluate the Sockpuppet technique and to provide a demonstration environment we decided to port the MPE PowerView GUI code to an STM32F429I Discovery board, which includes a small QVGA colour panel.


We made a decision to standardise on the gcc compiler maintained by ARM:
https://launchpad.net/gcc-arm-embedded
This seems to be a clean compiler, but it has a few deficiencies:

- It does not include a make utility,
- Every silicon vendor ships a different version of Eclipse with different make tools,
- They are all incompatible.

A good alternative for supported hardware is the online mbed compiler:
https://developer.mbed.org/

The alternative is just to take the silicon vendor's "free" tools, accepting that we will need a huge amount of disc space (almost free these days) and a degree of pain in learning the tool-chain. The days when you could just download and go are long past. Whatever you do, there will be pain.

## Conclusions

Mixed language programming for embedded systems is entirely feasible and productive.

Do not assume that the C libraries provided by the silicon vendors are bug-free.

You can use the Forth to debug the C.

Once you have set it up, it all works surprisingly well, but compared to Forth cross-compilation, the C compilation chain is baroque.

Using the C libraries for hardware access saves a huge amount of time reading chip documentation. As the use of silicon vendor C libraries increases, silicon vendor are placing less importance on correct documentation. We have already found devices whose C libraries depend on undocumented registers.

## Acknowledgements

Robert Sexton is responsible for the ideas and implementation of Sockpuppet. His dedication has made it a production-grade environment.

Elizabeth Rather convinced me long ago of the necessity of good notations. This convinced us to port the extern interface from hosted systems to the cross-compiler.