```
        .syntax         unified
        .cpu            cortex-m4
        .text

// int32_t Return32Bits(void) [COMPLETE];

        .global         Return32Bits
        .thumb_func
        .align
Return32Bits:
        LDR       R0,=10      // R0 <-- 10
            BX          LR                // return R0


// int64_t Return64Bits(void) [COMPLETE];

        .global         Return64Bits
        .thumb_func
        .align
Return64Bits:
        LDR       R0,=-10
            LDR         R1,=-1      // R0.R1 <-- -10
            BX          LR                // return R0.R1


// uint8_t Add8Bits(uint8_t x, uint8_t y) [COMPLETE];

        .global         Add8Bits
        .thumb_func
        .align
Add8Bits:
            ADD         R0,R0,R1    // R0 <-- R0 + R1 <-> x = x + y
            UXTB  R0,R0       // extend R0 from 8-bits to 32-bits
            BX          LR                // return R0


// uint32_t FactSum32(uint32_t x, uint32_t y) ;

        .global         FactSum32
        .thumb_func
        .align
FactSum32:
        PUSH{R4,LR}           // (push R4 to even out registers)
            ADD         R0,R0,R1    // R0 <-- R0 + R1 <-> x = x + y
            BL          Factorial   // R0 <-- Factorial(R0) <-> x = x!
            POP         {R4,PC}           // return R0
```

```
// uint32_t XPlusGCD(uint32_t x, uint32_t y, uint32_t z) ;

        .global         XPlusGCD
        .thumb_func
        .align
XPlusGCD:
        PUSH{R4,LR}
            MOV         R4,R0       // save first R0 ("original x" or x`) to R4
            MOV         R0,R1       // R1 --> R0 <-> x = y
            MOV         R1,R2       // R2 --> R0 <-> y = z
            BL          gcd               // R0 <-- gcd(R0,R1) <-> gcd(x,y)
            ADD         R0,R0,R4    // R0 <-- R0 + R4 <-> x = x + x`
            POP         {R4,PC}           // return R0

        .end
```

```
                    .syntax     unified
                    .cpu        cortex-m4
                    .text
                    // void OffBy0(void *dst, const void *src) ;
                    .global     OffBy0
                    .thumb_func
                    .align
OffBy0:             .rept 250               // 250 iterations = 1000 bytes to
compile / 4 bytes per iteration
                    LDR         R2,[R1],4   // Load dereferenced "src" (R1) into
a template register (R2)
                    STR         R2,[R0],4   // Store the template register value
into the "dst" pointer (R3)
                    .endr                   // End loop
                    BX          LR                // Return to program

                    // void OffBy1(void *dst, const void *src) ;
                    .global     OffBy1
                    .thumb_func
                    .align
OffBy1:             .rept 3                 // Copy the first three bytes (to
deal with the offset)
                    LDRB  R2,[R1],1
                    STRB  R2,[R0],1
                    .endr
                    .rept 249               //    OffBy0 minus one of the bytes
(Copy most of the data)
                    LDR         R2,[R1],4
                    STR         R2,[R0],4
                    .endr
```

```
                LDRB  R2,[R1],1   // Last byte
                STRB  R2,[R0],1
                BX    LR

                // void OffBy2(void *dst, const void *src) ;
                .global     OffBy2
                .thumb_func
                .align
OffBy2:         .rept 2                 // First two bytes
                LDRB  R2,[R1],1
                STRB  R2,[R0],1
                .endr
                .rept 249         //    Copy most of the data
                LDR         R2,[R1],4
                STR         R2,[R0],4
                .endr
                .rept 2                 // Last two bytes
                LDRB  R2,[R1],1
                STRB  R2,[R0],1
                .endr
                BX    LR

                // void OffBy3(void *dst, const void *src) ;
                .global     OffBy3
                .thumb_func
                .align
OffBy3:         LDRB  R2,[R1],1   // First Byte
                STRB  R2,[R0],1
                .rept 249
                LDR         R2,[R1],4   //    Copy most of the data
                STR         R2,[R0],4
                .endr
                .rept 3                 // Last three bytes
                LDRB  R2,[R1],1
                STRB  R2,[R0],1
                .endr
                BX    LR
                .end
```

```
        .syntax     unified
        .cpu        cortex-m4
        .text

// void WritePixel(int x, int y, uint8_t colorIndex, uint8_t
frameBuffer[256][240]) ;

        .global     WritePixel
```

```
        .thumb_func
        .align

WritePixel:
        PUSH            {R4,R5}                 // Load template registers
            ADD             R4,R0,R3            // R4 -> pointer that is x
bytes from the frame buffer
            LDR             R5,=240                 // R5 -> number of
columns
            MLA             R5,R1,R5,R4         // R5 -> number of columns * y
+ framebuffer pointer + x
            STRB        R2,[R5]                 // Store the color indix
            POP             {R4,R5}                 // ...
            BX          LR                      // Return and restore
registers


// uint8_t *BitmapAddress(char ascii, uint8_t *fontTable, int charHeight, int
charWidth) ;

        .global     BitmapAddress
        .thumb_func
        .align

BitmapAddress:
        PUSH            {R4,R5}                 // Load template registers
            ADD             R4,R3,7                 // R4 -> width + 7
            LDR             R5,=8               // ...
            UDIV        R4,R4,R5            // R4 -> (width + 7)/8
            MUL             R4,R4,R2            // R4 -> columns * rows or
height
            SUB             R5,R0,32            // R5 -> ASCII - space
character or 32
            MUL             R4,R4,R5            // R4 -> ASCII * rows *
columns
            ADD             R0,R4,R1            // Store R4 + fontTable
            POP             {R4,R5}                 // ...
            BX          LR                      // Return and restore
registers



// uint32_t GetBitmapRow(uint8_t *rowAdrs) ;

        .global     GetBitmapRow
        .thumb_func
        .align

GetBitmapRow:
```

```
        LDR             R0,[R0]                 // Load the variable into a
32-bit format
            REV             R0,R0               // Reverse the byte
        BX          LR                      // Return R0.

            .end
```

```
            .syntax     unified
            .cpu        cortex-m4
            .text

/*
void FloodFill(int32_t x, int32_t y, uint32_t old_clr, uint32_t new_clr)
    {
    uint32_t *ppxl ;

    if (OutOfBounds(x, y)) return ;

    ppxl = PixelAdrs(x, y) ;
    if (*ppxl!= old_clr) return ;
    *ppxl = new_clr ;

    FloodFill(x - 1, y, old_clr, new_clr) ;
    FloodFill(x + 1, y, old_clr, new_clr) ;
    FloodFill(x, y - 1, old_clr, new_clr) ;
    FloodFill(x, y + 1, old_clr, new_clr) ;
    }
*/

            .global     FloodFill
            .thumb_func
            .align

// R0 = x, R1 = y, R2 = old_clr, R3 = new_clr

FloodFill:
            // Your code here...
                  PUSH  {R4,R5,R6,R7,R8,R9,R10,LR}

                  // Prerequisite
                  LDR         R4,=0       // R4: used as a "false" conditional
                  MOV         R5,R0       // R5 <- x
                  MOV         R6,R1       // R6 <- y
                  MOV         R7,R2       // R7 <- old_clr
                  MOV         R8,R3       // R8 <- new_clr

                  // First Part: Evaluation of "OutOfBounds"
```

```
                BL          OutOfBounds // Call OutOfBounds
                CMP         R0,R4
                BNE         FloodExit   // OutOfBounds = true -> EXIT
                MOV   R0,R5       // Load x back into the R0
                MOV         R1,R6       // Load y back into R1

                // Second Part: Assignment of "ppxl" through "PixelAdrs"
                BL          PixelAdrs   // Call PixelAdrs (R0 <- ppxl)
                LDR   R9,[R0]           // R9 <- *ppxl
                CMP         R9,R7
                BNE   FloodExit   // *ppxl != old_clr -> EXIT
                STR         R8,[R0]           // Store new_clr into R9
                MOV   R0,R5       // Load x back into the R0

                // Third Part: Expansive Recursion of "FloodFill"
                SUB         R0,R5,1         // x - 1
                MOV   R1,R6
                MOV   R2,R7
                MOV   R3,R8
                BL          FloodFill

                ADD         R0,R5,1     // x + 1
                MOV   R1,R6
                MOV   R2,R7
                MOV   R3,R8
                BL          FloodFill

                SUB         R1,R6,1           // y - 1
                MOV   R0,R5
                MOV   R2,R7
                MOV   R3,R8
                BL          FloodFill

                ADD         R1,R6,1     // y + 1
                MOV   R0,R5
                MOV   R2,R7
                MOV   R3,R8
                BL          FloodFill

FloodExit:
                POP         {R4,R5,R6,R7,R8,R9,R10,LR}
                BX          LR
            .end
```

```
        .syntax     unified
        .cpu        cortex-m4
        .text
```

```
        //.set        BitBanding,1        // Comment out if not using
bit-banding

// void PutBit(void *bits, uint32_t index, uint32_t bit) ;

        .global     PutBit
        .thumb_func
        .align

PutBit:
            .ifdefBitBanding               // >> First Part <<
            SUB         R0,R0,0x20000000   // : take the array and subtract the
base
            LSL         R0,R0,5                        // : multiply by 32
            ADD         R0,R0,R1,LSL 2           // : add the index * 4
            ADD         R0,R0,0x22000000   // : add the alias to get the
address
            STR         R2,[R0]                        // : update the address
with the new bit
            BX          LR                             // -----

            .else                                  // >> Second Part <<
            PUSH  {R4,R5,R6,R7,R8,LR}
            LSR         R4,R1,3                        // : byte index --> R4
            AND   R5,R1,7                          // : bit index --> R5
            ADD         R0,R0,R4                   // : add the byte index to the
bits: address --> R0
            LSL         R6,R2,R5                   // : shift the bit by bit
index
            LDR         R7,=1                      // (dummy variable for BIC)
            LSL   R7,R7,R5
            LDRB  R8,[R0]                          // : R8 concerns the target
byte of the array
            BIC         R8,R8,R7                   // : zero out R8 up to bit
index
            ORR         R8,R8,R6                   // : add in the new data from
R6
            STRB  R8,[R0]                          // : update the bit array
            POP         {R4,R5,R6,R7,R8,LR}
            BX          LR                             // -----
            .endif



// uint32_t GetBit(void *bits, uint32_t index) ;
        .global     GetBit
        .thumb_func
        .align
```

```
GetBit:
            .ifdefBitBanding               // >> First Part <<
            SUB         R0,R0,0x20000000   // : See the first part of "PutBit"
            LSL         R0,R0,5
            ADD         R0,R0,R1,LSL 2
            ADD         R0,R0,0x22000000
            LDR         R0,[R0]                        // : get the bit
            BX          LR                             // -----

            .else                                  // >> Second Part <<
            PUSH  {R4,R5,R6,LR}
            LSR         R4,R1,3                        // : byte index --> R4
            AND   R5,R1,7                          // : bit index --> R5
            ADD         R0,R0,R4                   // : add the byte index to the
bits: address --> R0
            LDRB  R6,[R0]                          // : R6 concerns the target
byte of the array
            LSR         R6,R5                      // : Shift the target byte
right by the bit index
            AND         R0,R6,1                        // : Isolate the
rightmost bit
            POP         {R4,R5,R6,LR}
            BX          LR                             //  (Return R0) -----

            .endif

        .end
```

```
        .syntax     unified
        .cpu        cortex-m4
        .text

// void Bills(uint32_t dollars, BILLS *bills) ;

        .global     Bills
        .thumb_func
        .align

Bills:
            PUSH  {R4,R5,R6}         // R0 = dollars, R1 = bills pointer
            MOV         R4,R1              // R4 = struct pointer
            MOV         R5,R0              // R5 = amount left (R6 is a dummy
variable for calculation)

            LDR         R1,=3435973837     // on twenties
            UMULL R0,R1,R1,R5
```

```
            LSR         R0,R1,4
            STR         R0,[R4]                 // store quotient into the
pointer

            ADD         R6,R0,R0,LSL 2     // update amount (by multiplying the
quotient and subtracting from the real)
            SUB         R5,R5,R6,LSL 2     // Note: 20 = 5 * 4 = (1 + (1 << 2))
<< 2

            MOV         R0,R5              // restore the amount in R0
            MOV         R1,R4              // restore the pointer in R1

            POP         {R4,R5,R6}
            B           Common



// void Coins(uint32_t cents, COINS *coins) ;

        .global     Coins
        .thumb_func
        .align

Coins:
            PUSH  {R4,R5,R6}         // R0 = cents, R1 = coins pointer
            MOV         R4,R1              // R4 = struct pointer
            MOV         R5,R0              // R5 = amount left

            LDR         R1,=1374389535     // on twenty-fives
            UMULL R0,R1,R1,R5
            LSR         R0,R1,3
            STR         R0,[R4]

            ADD         R6,R0,R0,LSL 2     // update amount
            ADD         R6,R6,R6,LSL 2
            SUB         R5,R5,R6           // Note: 25 = 5 * 5 = ((1 + (1 <<
2)) + (1 + (1 << 2)) << 2)

            MOV         R0,R5              // restore the amount in R0
            MOV         R1,R4              // restore the pointer in R1

            POP         {R4,R5,R6}
            B           Common

Common:
            PUSH  {R4,R5,R6,LR}

            MOV         R4,R1              // store struct pointer
            MOV         R5,R0              // store amount left
```

```
    ADD         R4,R4,4                 // update pointer
    LDR         R1,=3435973837      // on tens
    UMULL R0,R1,R1,R5
    LSR         R0,R1,3
    STR         R0,[R4]

    ADD         R6,R0,R0,LSL 2      // update amount
    SUB         R5,R5,R6,LSL 1      // Note: (1 + (1 << 2)) << 1 = 10

    ADD         R4,R4,4                 // update pointer
    LDR         R1,=3435973837      // on fives
    UMULL R0,R1,R1,R5
    LSR         R0,R1,2
    STR         R0,[R4]

    ADD         R6,R0,R0,LSL 2      // update amount
    SUB         R5,R5,R6            // Note: 1 + (1 << 2) = 5

    ADD         R4,R4,4                 // update pointer
    STR         R5,[R4]                 // on ones (straight store)

    POP         {R4,R5,R6,LR}
    BX          LR                      // return
.end
```
