C function

```
char m;
fred(){
char * p = & m;
*p=4;
}
```

C Compiler

To compile

gcc -c test.c

Produces test.o which is an object file for the host machine e.g. intel architecture.

We can disassemble the test.o file with the tool objdump:

objdump -S test.o

X86 (dis)assembly of test.o

```
0: 55                          push   %rbp
1: 48 89 e5                    mov    %rsp,%rbp
4: 48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
b: 00
c: 48 8b 45 f8                mov    -0x8(%rbp),%rax
10:    c6 00 04                movb   $0x4,(%rax)
13:    90                      nop
14:    5d                      pop    %rbp
15:    c3                      retq
```

Cross Compiler for ARM

To compile

arm-none-eabi-gcc -c test.c

Produces test.o which is an object file for the ARM architecture.

We can disassemble the test.o file with the tool objdump:

arm-none-eabi-objdump -S test.o

ARM assembly of test.o

```
0: e52db004  push {fp}      ; (str fp, [sp, #-4]!)
4: e28db000  add   fp, sp, #0
8: e24dd00c  sub   sp, sp, #12
c: e59f301c     ldr         r3,   [pc,   #28]   ;   30
<fred+0x30>
10:   e50b3008  str    r3, [fp, #-8]
14:   e51b3008  ldr    r3, [fp, #-8]
18:   e3a02004  mov  r2, #4
1c:   e5c32000  strb  r2, [r3]
20:   e1a00003  mov  r0, r3
24:   e24bd000  sub   sp, fp, #0
28:   e49db004  pop   {fp}      ; (ldr fp, [sp], #4)
2c:   e12fff1e    bx lr
```

Run program on host architecture

```
//t.c
Main(){
printf(“hello”);
}
```

Compile with:

gcc t.c

gcc produces an executable called a.out which can be executed using the command line on the host machine:

a.out

Run Arm executable

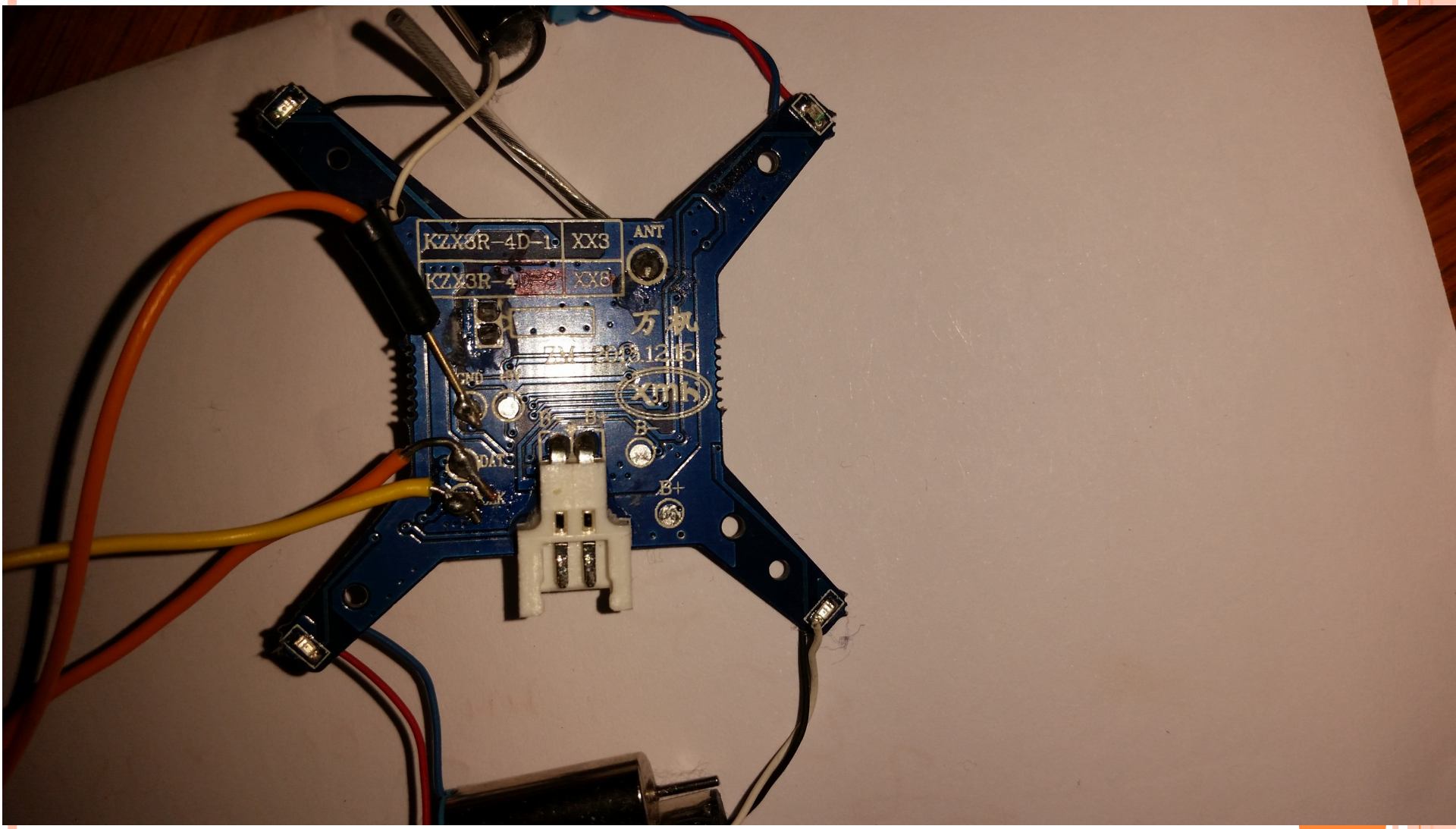
We need an Arm based board or simulate the arm architecture on the host.

For an Arm board we need specialised hardware such as a jtag, st-link etc which interfaces to the debug interface on the cortex-m microcontroller.

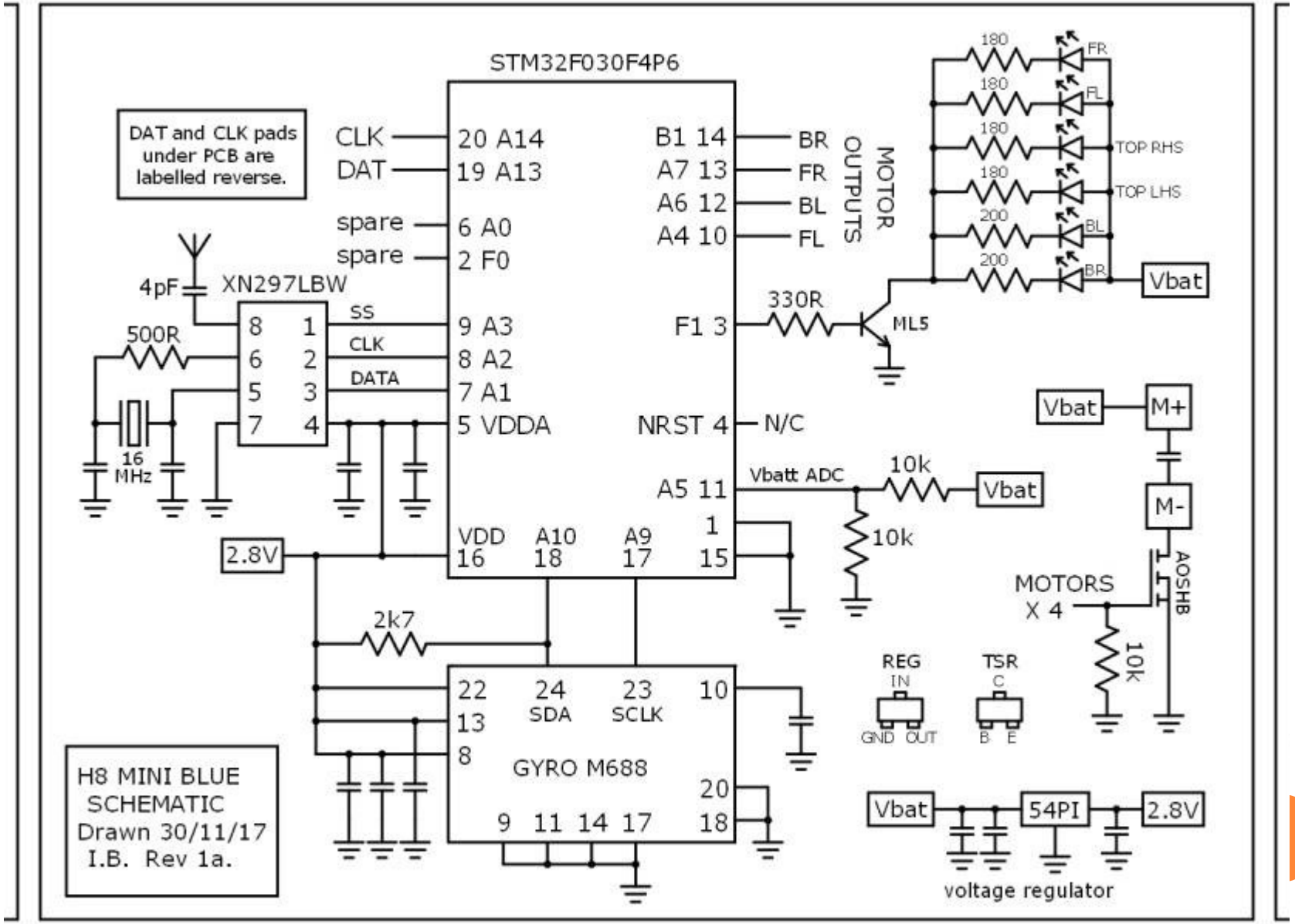
Cortex-M3 systems for example support two types of debug host interfaces and associated protocols. The JTAG interface, and the Serial-Wire interface.

We use this hardware to “flash” the executable onto the board’s flash memory.

Solder to the SWD pins on back of the board



H8-mini quadcoptor schematic



The Serial Wire interface – Debug/Flashing

Connecting of the wires from the stlink dongle to the H8 board as follows:

ST LINK	H8 Board
SWCLK ---	DAT
GND ---	GND
SWDIO ---	CLK

provides the hardware component of SWD interface to the stm32F030F4 microcontroller using the st-link programmer.

The openocd tool provides the SWD protocol using the st-link hardware interface.

Technical Reference Manual – stm32F030F4 Section 26.3.1

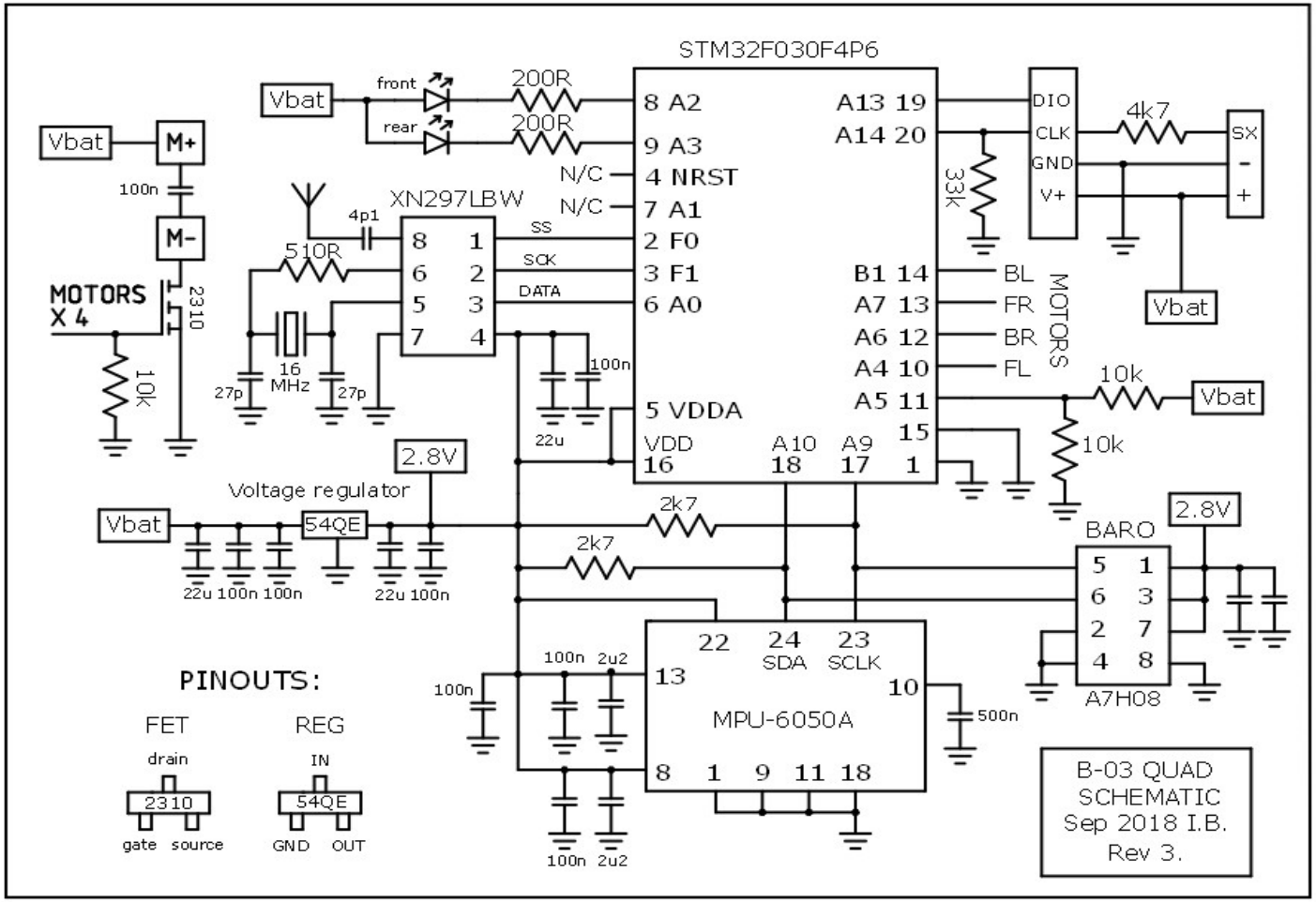
SWD port pins

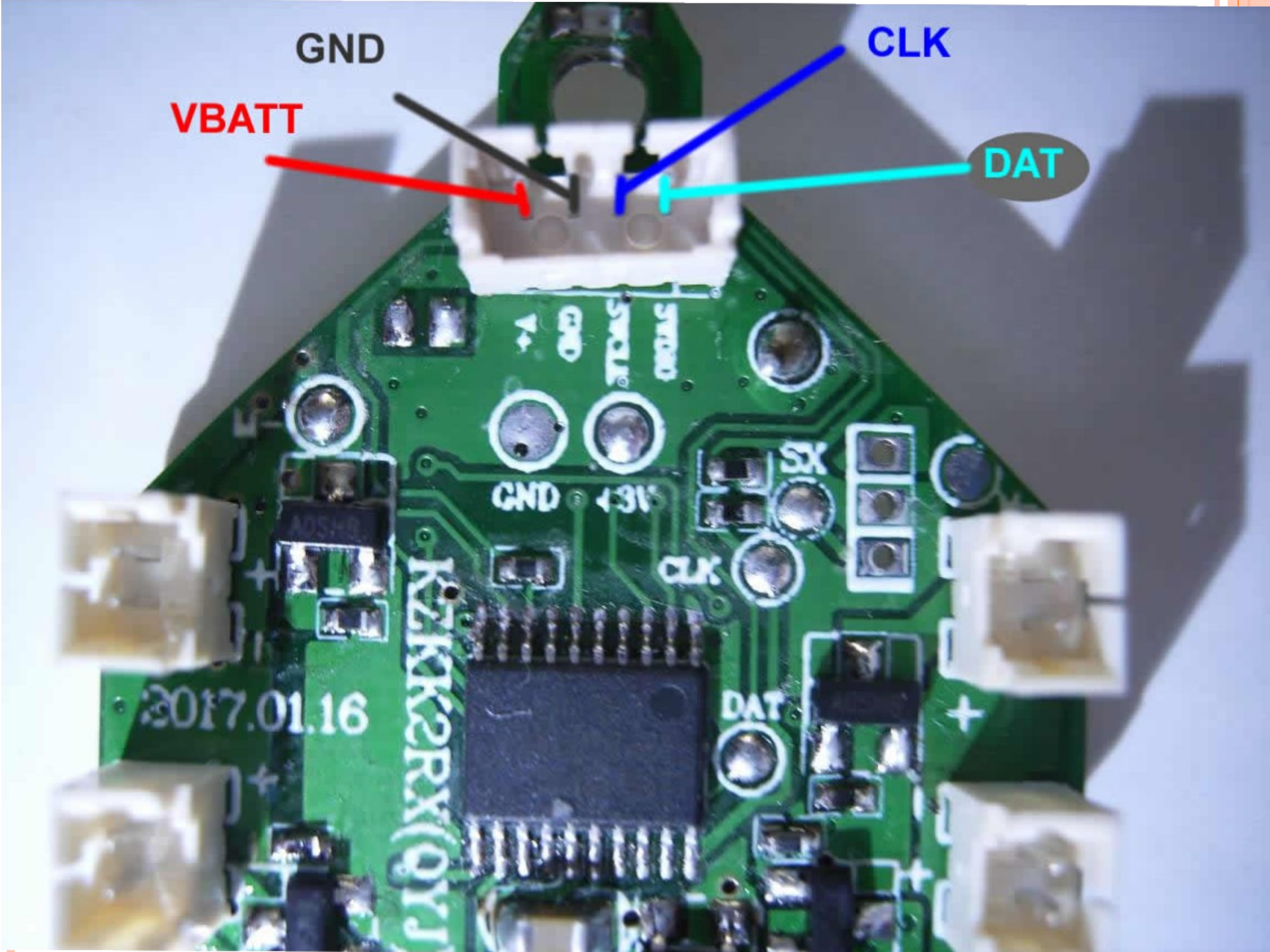
Two pins are used as outputs for the SW-DP as alternate functions of general purpose I/Os These pins are available on all packages.

Table 108. SW debug port pins

SW-DP pin name	SW debug port		Pin assignment
	Type	Debug assignment	
SWDIO	IO	Serial Wire Data Input/Output	PA13
SWCLK	I	Serial Wire Clock	PA14

Schematic for bwhoop




GND
CLK
VBATT
DAT
GND
CLK
DAT
SX
2017.01.16

QEMU

QEMU (short for Quick Emulator) is a free and open-source hosted hypervisor that performs hardware virtualization.

QEMU is a hosted virtual machine monitor: it emulates CPUs through dynamic binary translation.

It provides a set of device models, enabling it to run a variety of unmodified guest operating systems.

QEMU

It also can be used together with KVM in order to run virtual machines at near-native speed (requiring hardware virtualization extensions on x86 machines).

QEMU can also do CPU emulation for user-level processes, allowing applications compiled for one architecture to run on another.

QEMU-System-arm

QEMU has generally good support for ARM guests. It has support for nearly fifty different machines.

ARM hardware is much more widely varying than x86 hardware.

ARM CPUs are generally built into "system-on-chip" (SoC) designs created by many different companies with different devices, and these SoCs are then built into machines which can vary still further even if they use the same SoC.

QEMU-system-arm

Even with fifty boards QEMU does not cover more than a small fraction of the ARM hardware ecosystem.

Because ARM systems differ so much and in fundamental ways, typically operating system or firmware images intended to run on one machine will not run at all on any other.

QEMU-system-arm

As well as the more common "A-profile" CPUs (which have MMUs and will run Linux) QEMU also supports the Cortex-M3 and Cortex-M4 "M-profile" CPUs.

The M-profile CPUs are microcontrollers used in very embedded boards.

There are only two boards which use the M-profile CPU at the moment: "lm3s811evb" and "lm3s6965evb" (which are both TI Stellaris evaluation boards).

ARM Versatilepb

ARM System Emulators supports many peripheral devices.

According to the QEMU user manual, the ARM Versatilepb baseboard emulates the following devices: •

ARM926E, ARM 1136 or Cortex-A8 CPU.

- PL190 Vectored Interrupt Controller.
- Four PL011 UARTs.
- pL110 LCD controller.
-

QEMU-System-arm

PL050 KMI controller for keyboard and mouse.

- PL181 MultiMedia Card Interface with SD card.
- SMC 91c111 Ethernet adapter.
- PCI host bridge (with limitations).
- PCI USB controller.
- LS153C895A PCI SCSI Host Bus Adapter with hard disk and CD-ROM devices.

ARM Versatilepb board emulated on QEMU

(2). The ARM Versatile board architecture is well documented by on-line articles in the ARM Information Center.

(3). QEMU can boot up the emulated ARM Versatilepb virtual machine directly. For example, we may generate a bin executable file, t.bin, and run it on the emulated ARM VersatilepbVM by:

qemu-system-arm —M versatilepb —m 128M
—kernel t.bin —serial mon:stdio

Add two numbers and put them into a memory location

```
 .text
   .global start
start:
    mov r0, #1          // r0 = 1
    MOV R1, #2           // r1 = 2
    ADD R1, R1, R0       // r1 = r1 + r0
    ldr r2, =result     // load address of result to r2
    str r1, [r2]         /* result = r1 */
stop: b stop
    .data
result: .word 0         /* result */
```

Compiler explorer - https://gcc.godbolt.org/

Add two numbers and put them into a memory location

```
 .text
   .global start
start:
    mov r0, #1          // r0 = 1
    MOV R1, #2           // r1 = 2
    ADD R1, R1, R0       // r1 = r1 + r0
    ldr r2, =result     // load address of result to r2
    str r1, [r2]         /* result = r1 */
stop: b stop
    .data
result: .word 0         /* result */
```

Assembler

-

arm-none-eabi-as -o ts.o ts.s

The assember creates ARM object code from the assembly language file

Assembly language is an abstraction/model for programmers to hide away the details of the machine

Building the executable

- Assembler

arm-none-eabi-as -o ts.o ts.s

- - Linker

arm-none-eabi-ld -T t.ld -o t.elf ts.o

Linker Script

```
ENTRY(start)
SECTIONS
{
  . = 0x10000;              // load exe to address 0x10000
  .text : { *(.text) }      // place all code (.text) sections here
  .data : { *(.data) }
  .bss  : { *(.bss)  }
  . = ALIGN(8);
  . = . + 0x1000;           /* 4kB of stack memory */
  stack_top = .;
}
```

Building the executable

- Dump symbols in executable and their addresses

```
arm-none-eabi-nm t.elf

0001001c t result
00011038 T stack_top
00010000 T start
00010014 t stop

arm-none-eabi-objcopy -O binary t.elf t.bin
```

t.bin file is only 52 bytes in size

od -x t.bin

Address	**hex dump**
0000000	0001 e3a0 1002 e3a0 1000 e081 2004 e59f
0000020	1000 e582 fffe eaff 001c 0001 0000 0000
0000040	1341 0000 6100 6165 6962 0100 0009 0000
0000060	0106 0108
0000064	

Arm-none-eabi-objdump t.elf

```
00010000 <start>:
   10000:   e3a00001  mov  r0, #1
   10004:   e3a01002  mov  r1, #2
   10008:   e0811000  add  r1, r1, r0
    ;  10018 <stop+0x4>
1000C:  e59f2004   ldr r2, [pc, #4]
   10010:   e5821000  strr1, [r2]

00010014 <stop>:
   10014:   eafffffe  b  10014 <stop>
   10018:   0001001c  .word0
```

Arm-none-eabi-objdump t.elf

```
00010000 <start>:
   10000:   e3a00001   mov  r0, #1
```

Address in memory is 10000 hex

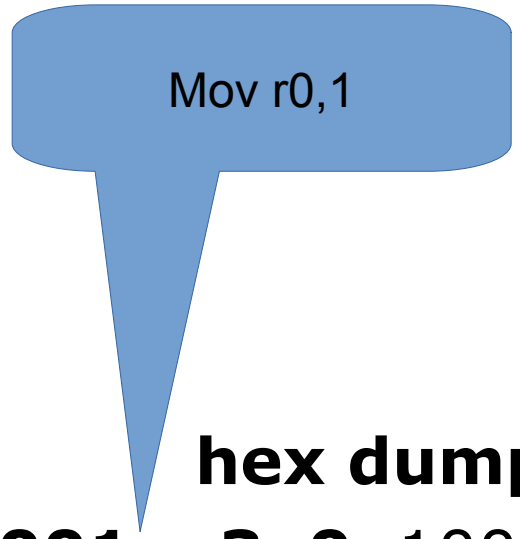
The data at that memory location is e3a00005

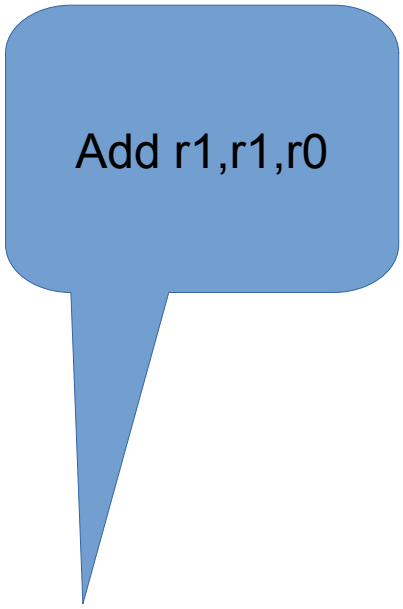
This data is an Arm instruction of 4 bytes which is

Mov r0,#5

t.bin file is only 52 bytes in size

od -x t.bin





Address	**hex dump**							
0000000	**0001**	**e3a0**	1002	e3a0	**1000**	**e081**	2004	e59f
0000020	1000	e582	fffe	eaff	001c	0001	0000	0000
0000040	1341	0000	6100	6165	6962	0100	0009	0000
0000060	0106	0108						
0000064								

GNU arm assembler has several pseudo

e.g. the manual has

ldr <register> , = <expression>

If expression evaluates to a numeric constant then a MOV or MVN instruction.

Otherwise the constant will be placed into the

nearest literal pool (if it not already there) and a PC relative LDR instruction will be generated.

```
 ldr r2, =result     // load address of result to r2
1000C:  e59f2004    ldr r2, [pc, #4]

00010014 <stop>:
   10014:   eafffffe   b   10014 <stop>
   10018:   0001001c   .word 0
```

```
 ldr r2, =result     // load address of result to r2
1000C:  e59f2004    ldr  r2, [pc, #4]

00010014 <stop>:
   10014:    eafffffe   b   10014 <stop>
   10018:    0001001c   .word 0
```

at address 10018

Call QEMU to emulate the executable

qemu-system-arm -M versatilepb -kernel t.bin -nographic -serial /dev/null

QEMU 2.5.0 monitor - type 'help' for more information

(qemu) pulseaudio: set_sink_input_volume() failed

pulseaudio: Reason: Invalid argument

pulseaudio: set_sink_input_mute() failed

pulseaudio: Reason: Invalid argument

(qemu)

Running example on QEMU

info registers

(qemu) info registers

R00=00000001 R01=00000003 R02=0001001c R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=00010014
PSR=400001d3 -Z-- A svc32
s00=00000000 s01=00000000 d00=0000000000000000
s02=00000000 s03=00000000 d01=0000000000000000
s04=00000000 s05=00000000 d02=0000000000000000
s06=00000000 s07=00000000 d03=0000000000000000

Running example on QEMU

The r2 register holds 0x1001C the address of result.

xp /wd 0x1001C

Shows the value of result.

Disassemble using QEMU

```
(qemu) xp /i 0x10000
0x00010000:  e3a00005      mov   r0, #5   ; 0x5
```

Address in memory is 0x00010000

The data at that memory location is e3a00005

This data is an Arm instruction of 4 bytes which is Mov r0,#5

Disassemble 8 instructions

```
(qemu) xp /8i 0x10000
0x00010000:  e3a00005     mov   r0, #5  ; 0x5
0x00010004:  e3a01009     mov   r1, #9  ; 0x9
0x00010008:  e0811000     add   r1, r1, r0
0x0001000c: e59f2004    ldr   r2, [pc, #4]    ; 0x10018
0x00010010:  e5821000     str    r1, [r2]
0x00010014:  eafffffe     b0x10014
0x00010018:  0001001c     andeq r0, r1, ip, lsl r0
0x0001001c:  0000000e     andeq r0, r0, lr
```

The 0x1001c is the address of result which has no meaning disassembled.

sum Array example

```
 .text
     .global start
start:    ldr sp, =stack_top
//The BL instruction causes a branch to label, and copies the address
// of the next instruction into LR (R14, the link register).
     bl  sum
stop:     b   stop
sum:
     mov r0, #0
     ldr r1, =Array    // r1 has address of Array
                         // like int *p = Array.
                         // int Array[10];
     ldr r2, =N        // int * nptr = &N;
     ldr r2, [r2]      //  r2 = *nptr
```

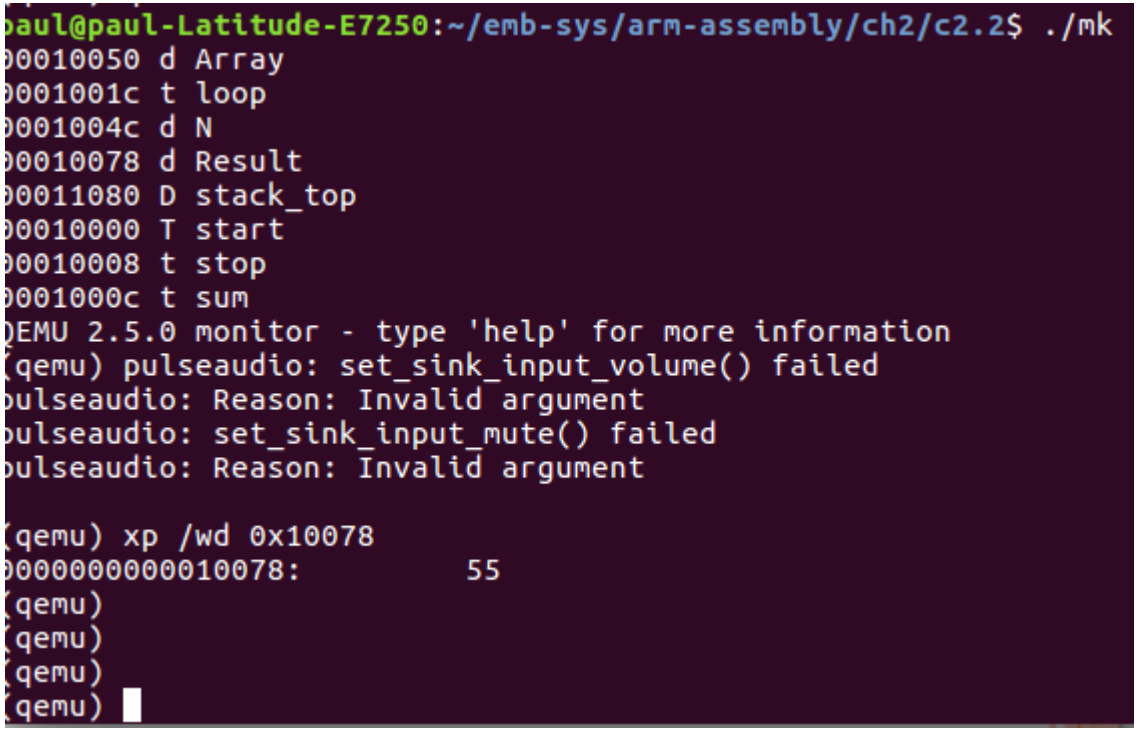
```
// int Array[10] = {1,2,3,4,5,6,7,8,9,10};
// int N = 10;
// int sum;      r0 used for sum
// for (int i=0;i<N;i++)
//      sum += *p++;
loop:
    ldr r3, [r1], #4      // r3 = *p++
                          // load value at address r1 into
                          // r3 and increment r1 by 4
                         // increment by 4 because words
                          // are 4 bytes
    add r0, r0, r3        // r0+= r3 i.e. sum += *p++
    sub r2, r2, #1        //  r2--
    cmp r2, #0            // if (r2 !=0)
    bne loop              // goto loop
```

```
// int Result;
//  int *ip;     r4 is used for ip
    ldr r4, =Result      // ip = &Result
    str r0, [r4]          // *ip = sum
    mov pc, lr       // move back to stop
.data
N: .word  10
Array:    .word  1,2,3,4,5,6,7,8,9,10
Result:   .word  0
```

Sum Array example

```
loop:
    ldr r3, [r1], #4
    add r0, r0, r3
    sub r2, r2, #1
    cmp r2, #0
    bne loop
    ldr r4, =Result
    str r0, [r4]
    mov pc, lr
.data
N: .word  10
Array:    .word  1,2,3,4,5,6,7,8,9,10
Result:   .word  0
```

Qemu show value of result



Show instructions at sum

```
(qemu) xp /8i 0x1000c
0x0001000c:  e3a00000      mov  r0, #0  ; 0x0
0x00010010:  e59f1028      ldr  r1, [pc, #40]   ; 0x10040
0x00010014:  e59f2028      ldr  r2, [pc, #40]   ; 0x10044
0x00010018:  e5922000      ldr  r2, [r2]
0x0001001c:  e4913004      ldr  r3, [r1], #4
0x00010020:  e0800003      add  r0, r0, r3
0x00010024:  e2422001      sub  r2, r2, #1      ; 0x1
0x00010028:  e3520000      cmp  r2, #0  ; 0x0
(qemu) xp /12i 0x1000c
0x0001000c:  e3a00000      mov  r0, #0  ; 0x0
0x00010010:  e59f1028      ldr  r1, [pc, #40]   ; 0x10040
0x00010014:  e59f2028      ldr  r2, [pc, #40]   ; 0x10044
0x00010018:  e5922000      ldr  r2, [r2]
0x0001001c:  e4913004      ldr  r3, [r1], #4
0x00010020:  e0800003      add  r0, r0, r3
0x00010024:  e2422001      sub  r2, r2, #1      ; 0x1
0x00010028:  e3520000      cmp  r2, #0  ; 0x0
0x0001002c:  1afffffa      bne  0x1001c
0x00010030:  e59f4010      ldr  r4, [pc, #16]   ; 0x10048
0x00010034:  e5840000      str  r0, [r4]
0x00010038:  e1a0f00e      mov  pc, lr
(qemu)
```