Digital embedded electronics

J.-M Friedt

FEMTO-ST/time & frequency department

jmfriedt@femto-st.fr

slides at `jmfriedt.free.fr`

December 15, 2024

List of presentations:

7 lessons introducing baremetal programming of the STM32:

1. Digital electronics basics
   L3/bachelor: analog aspects, power supply and consumption, datasheet analysis
   Overview of the various peripherals (RS232, SPI, timer, ADC)
   data format (size/encoding), masks, architecture
   Reminders on Atmega32U4 (Makefile, compilation, masks ...)
2. First steps with the STM32, peripheral addresses, internal architecture
3. gcc operation and optimizations:
   preprocess–compiler–assembler–linker, C to assembly language translation, pointers
4. libraries and software architecture (separating algorithm/hardware), simulators & stubs
   libopencm3, newlib & stubs, resources needed to execute libraries
5. Communication bus parallel/serial, synchronous/asynchrone
6. arithmetics on embedded systems
   integers v.s floating point number, converting an algorithm expressed with floats to integers, timers
7. interrupts and analog data acquisition, sampling rate
   interrupt vectors, clock management on STM32, ADC

All supporting material (bachelor and master) on `http://jmfriedt.free.fr`

Compile the compiler

(Cross)-compilation toolchain requires

1. the compiler (`gcc`)
2. tools for converting between the various binary formats (`binutils`)
3. **libraries** (`newlib`)

+ debugger (`gdb`) + tools for programming the microcontroller (`avrdude`, `stm32flash`, `dfu-programmer` ...)

```
configure --target=arm-none-eabi --prefix=${HOME}/sat
```

Automate this process using a script for generating a consistent set of tools [1].

[1] usage explanation at `http://jmfriedt.free.fr/summon_arm.pdf`

newlib

Resources needed by a library calculating with **floating point numbers**:

Always in thumb instruction set:	without stdio, with floating point division	12950
	without stdio, with floating point atan	17090
	with stdio, with floating point division	80397
	with stdio, with floating point atan	80397

Thumb is an ARM processor instruction subset encoded on 16-bits instead of 32-bits [2]

Various free, opensource implementations of the C standard library:

- glibc (`http://www.gnu.org/software/libc/`, used with the Linux kernel) ...
- ... and eglibc (*embedded* glibc, `http://www.eglibc.org/home`) now merged with `glibc`,
- uClibc (`http://www.uclibc.org/`, used by uClinux) for MMU-less systems, forked to uClibc-ng.

[2] J.-M Friedt, É. Carry, *Développement sur processeur à base de cœur ARM7 sous GNU/Linux*, GNU/Linux Magazine France **117** (Juin 2009), pp.40-59

Problem statement

In a C program, calling `malloc()` leads to an error during linking

```
ne-eabi/lib/thumb/cortex-m3/libc.a(lib_a-sbrkr.o): In function ‘_sbrk_r’:
/libc/reent/sbrkr.c:58: undefined reference to ‘_sbrk’
```

Same for `printf()`

```
ne-eabi/lib/thumb/cortex-m3/libc.a(lib_a-sbrkr.o): In function ‘_sbrk_r’:
/libc/reent/sbrkr.c:58: undefined reference to ‘_sbrk’
ne-eabi/lib/thumb/cortex-m3/libc.a(lib_a-writer.o): In function ‘_write_r’:
/libc/reent/writer.c:58: undefined reference to ‘_write’
ne-eabi/lib/thumb/cortex-m3/libc.a(lib_a-closer.o): In function ‘_close_r’:
/libc/reent/closer.c:53: undefined reference to ‘_close’
ne-eabi/lib/thumb/cortex-m3/libc.a(lib_a-fstatr.o): In function ‘_fstat_r’:
/libc/reent/fstatr.c:62: undefined reference to ‘_fstat’
ne-eabi/lib/thumb/cortex-m3/libc.a(lib_a-isattyr.o): In function ‘_isatty_r’:
/libc/reent/isattyr.c:58: undefined reference to ‘_isatty’
ne-eabi/lib/thumb/cortex-m3/libc.a(lib_a-lseekr.o): In function ‘_lseek_r’:
/libc/reent/lseekr.c:58: undefined reference to ‘_lseek’
ne-eabi/lib/thumb/cortex-m3/libc.a(lib_a-readr.o): In function ‘_read_r’:
/libc/reent/readr.c:58: undefined reference to ‘_read’
```

Using a library – newlib

https://sourceware.org/newlib/libc.html#Stubs: 17 stubs, "glue" between newlib and custom programs[3].

- **_exit**: exit a program execution without closing opened files
- environ:
- execve:
- fork:
- fstat:
- getpid:
- isatty:
- kill:
- link:
- lseek:
- open:
- **read**: read from a file
- sbrk: *used by* malloc for allocating memory
- stat:
- times:
- unlink:
- wait:
- **write**: write to a file descriptor, including stdout

[3] http://wiki.osdev.org/Porting_Newlib

Using a library – newlib

Usage example (sending stdio to a serial port)

```
_ssize_t _read_r(struct _reent *r, int file, void *ptr, size_t len)
{char c; int  i; unsigned char *p;
 p = (unsigned char*)ptr;
 for (i = 0; i < len; i++)
   {while ( global_index == 0) {} //  !uart0_kbhit() ) ;
    c = global_tab[0];global_index=0; // (char) uart0_getc();
    if (c == 0x0D) {*p='\0';break;}
    *p++ = c; jmf_putchar(c,NULL,0,0);
   }
 return len - i;
}

_ssize_t _write_r ( struct _reent *r, int file, const void *ptr, size_t len)
{int i; const unsigned char *p;
 p = (const unsigned char*) ptr;
 for (i = 0; i < len; i++)
    {if (*p == '\n' ) jmf_putchar('\r',NULL,0,0);
     jmf_putchar(*p++,NULL,0,0);
    }
 return len;
}

void jmf_putchar (int a,char *chaine, int* chaine_index,int USARTx)
{ if (chaine != NULL) {chaine[*chaine_index]=a;*chaine_index=*chaine_index+1;}
  else {if (usart_port==1)
         {USART_SendData (USART1, a);
          while(USART_GetFlagStatus(USART1, USART_FLAG_TC) == RESET) { ; }
         }
        else
         {USART_SendData (USART2, a);
          while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET) { ; }
         }
    }
}
```

Demonstration

```
#ifdef use_stdio
  for (tempe=1;tempe<10;tempe++) {mf=mf*lf;}
  printf("\n+mf=%d\n",(int)mf);

  t=(char*)malloc(200); // malloc requires _sbrk
  printf("^%x\n^%x\n",(int)t[0],t[199]);
  memset(t,0x55,200);
  printf("^%x\n^%x\n",(int)t[0],t[199]);
#endif

  while (1)
  {
#ifndef use_stdio
   put_chars(welcome);
#else
   printf("%s",welcome);
#endif
   if (i<10) put_char(USART1,i+'0');
        else put_char(USART1,i+'A'-10);
   i++;if (i==16) i=0;
...
```

Demonstration

Tradeoff between development time, functionalities and resource usage

```
3240  main.bin
31824 main_stdio.bin
```

Here, displaying a message with `printf()` implies dynamic memory allocation using `malloc()`, floating point calculation

An emulator for testing software when hardware is not available:
`https://github.com/beckus/qemu_stm32`
`qemu-system-arm -M stm32-p103 -serial stdio -serial stdio -kernel main.bin`
assumes however that peripherals have been "correctly" (accurately) emulated.

`simavr` for the Atmega32U**2**

Stubs

- A main program includes all the “intelligence” of the software: algorithm
- This program exclusively calls hardware independent generic functions: *stubs*
- These functions accessing hardware are implemented in separate files
- Algorithm is linked to hardware calls during compilation (separate compilation)
- Ability to pass arguments to `Makefile`: variables and conditional tests
- Linking to generate an executable

algorithmique
```
led_on()
led_off()
communique()
```

matériel1
```
GP3DAT |= 0x00070000;
GP3DAT &= (~0x00070000);
COMTX = a;
```

matériel2, bibliothèque1
```
GPIO_SetBits(GPIOC,GPIO_Pin_2);
GPIO_ResetBits(GPIOC,GPIO_Pin_1);
USART_SendData (USART1, a);
```

matériel2, bibliothèque2
```
gpio_set  (GPIOA, GPIO4);
gpio_clear(GPIOA, GPIO4);
usart_send_blocking (USART1,a)
```

```
Makefile
  MODBUS = 0
  #include makefile.cortex_AD9958
  #include makefile.cortex_XE1203
  include makefile.aduc
  ifeq ($(MODBUS),1)
  ...
  endif
```

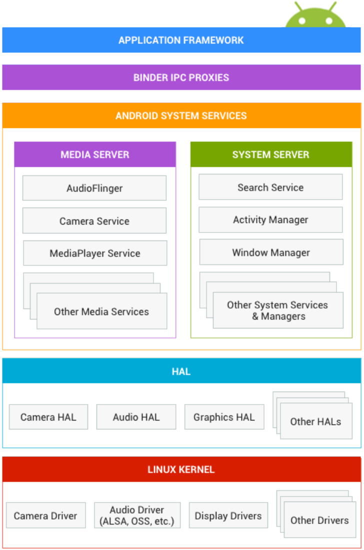
Edition de liens –> exécutable

→ “easy” replacement of the hardware platform without modifying the proven algorithm (HAL)

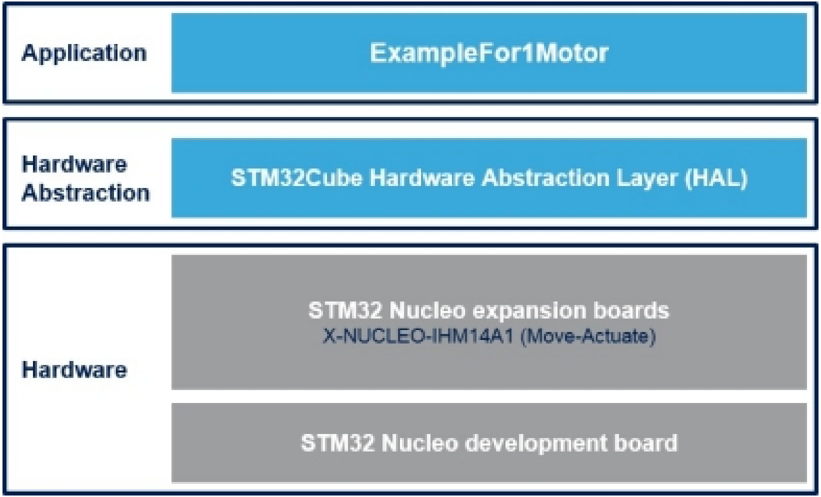
→ “dummy functions” on PC to emulate hardware access: unit tests

HAL: Hardware Abstraction Layer

Android



ST Microelectronics development framework



https://source.android.com/devices/architecture

https://www.st.com/en/embedded-software/x-cube-spn14.html

Software design

1. Separate software accessing hardware (peripheral initialization or hardware resource access) from algorithm
2. Separate compilation: select one of many hardware implementations or software emulation (gcc supports a wide range of platforms and processors)
3. Allows for testing/debugging software on PC before porting to the embedded microcontroller (reproducible simulation of various hardware environments), automated testing, especially pathological cases
4. Using a simulator to probe the internal state of the state machine executing the code
5. qemu for many platforms, now Renode, `simavr` for Atmega32U**2**

```
unsigned short interroge (unsigned short puissance, unsigned int freq,unsigned int offset,unsigned char cha
{ unsigned int v12;
  FTW0[1] = (freq & 0xFF000000) >> 24;
  FTW0[2] = (freq & 0xFF0000) >> 16;
  FTW0[3] = (freq & 0xFF00) >> 8;
  FTW0[4] = (freq & 0xFF);
[...]
  v12 = readADC12 ();
  readerF2_CLR;                         // coupe reception
  TIM_ITConfig (TIM3, TIM_IT_Update, ENABLE);
  return (v&0x03F);
}
```

Software design

1. Separate software accessing hardware (peripheral initialization or hardware resource access) from algorithm
2. Separate compilation: select one of many hardware implementations or software emulation (gcc supports a wide range of platforms and processors)
3. Allows for testing/debugging software on PC before porting to the embedded microcontroller (reproducible simulation of various hardware environments), automated testing, especially pathological cases
4. Using a simulator to probe the internal state of the state machine executing the code
5. qemu for many platforms, now Renode, `simavr` for Atmega32U**2**

```
unsigned short interroge(unsigned short puissance,unsigned int freq, \
  __attribute__((unused))unsigned int offset,__attribute__((unused))unsigned char chan)
{float reponse;
 reponse =exp(-((((float)freq-f01)/df)*(((float)freq-f01)/df))*3100.;
 reponse+=exp(-((((float)freq-f02)/df)*(((float)freq-f02)/df))*3100.;
 reponse+=(float)((rand()-RAND_MAX/2)/bruit); // ajout du bruit;
 if (reponse<0.) reponse=0.;
 if (reponse>4095.) reponse=4095.;
 usleep(60);
 return((unsigned short)reponse);
}
```

Interrupt emulation

PC version

```c
void handle_alarm(int xxx)
{ //  printf("RTC : %d %d %d\n",seconde,minute,heure);
  tim0+=10;
  seconde+=10;
  if (seconde>600){minute++;seconde=0;}
  if (minute>60) {heure++;minute=0;}
  alarm(1);
}

void handle_io(int xxx)
{ // io_happened=1;
  global_tab[global_index] = getchar ();
  if (global_index < (NB_CHARS - 1))
     global_index++;
}

int main() {
  signal(SIGALRM, handle_alarm);

  io_happened=0;
  struct termios buf;
  tcgetattr(0, &buf);
  buf.c_lflag &= ~(ECHO | ICANON);
  buf.c_cc[VMIN]=1;
  buf.c_cc[VTIME]=0;
  tcsetattr(0, TCSAFLUSH, &buf);
  signal(SIGIO, handle_io);
    int savedflags=fcntl(0, F_GETFL, 0);
  fcntl(0, F_SETFL, savedflags | O_ASYNC | O_NONBLOCK );
  fcntl(0, F_SETOWN, getpid());
  alarm(1);  // si on veut simuler le timeout
}
```

Microcontroller version

```c
void tim3_isr(void)  // VERSION LIBOPENCM3
{
 if (TIM_GetITStatus(TIM3,TIM_IT_Update)!=RESET)
  {TIM_ClearITPendingBit(TIM3,TIM_IT_Update);
   tim0++;
   seconde++;           // seconde en 1/10 seconde
   if (seconde > 600)
     {seconde = 0;minute++;}
  if (minute > 60)
     {minute = 0;heure++;}
  if (heure > 24)
     {heure = 0;}
  }
}

void usart1_isr(void)
{
  if( USART_GetITStatus(USART1, USART_IT_RXNE))
    {USART_ClearITPendingBit(USART1,USART_IT_RXNE);
     USART_ClearFlag(USART1,USART_IT_RXNE);
     {
      global_tab[global_index]=USART_ReceiveData(USART1);
      if (global_index<(NB_CHARS-1)) global_index++;
     }
    }
}
```

Benefit from analysis tools

valgrind to investigate memory leaks

```
int main()
{char *c;
 c=(char*)malloc(100);
}
```

is analyzed with

`valgrind --leak-check=full ./programme`
to indicate that

```
==20235== HEAP SUMMARY:
==20235==     in use at exit: 100 bytes in 1 blocks
==20235==   total heap usage: 1 allocs, 0 frees, 100 bytes allocated
==20235==
==20235== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==20235==    at 0x483577F: malloc (vg_replace_malloc.c:309)
==20235==    by 0x109146: main (in ...)
==20235==
==20235== LEAK SUMMARY:
==20235==    definitely lost: 100 bytes in 1 blocks
```

valgrind has detected that the heap was not empty upon execution completion

Benefit from analysis tools

valgrind to investigate illegal memory access

```
int main()
{char *c;
 c=(char*)malloc(100);
 c[100]=10;
 free(c);
}
```

is analyzed with

`valgrind -q ./programme`
to indicate that

```
==21801== Invalid write of size 1
==21801==    at 0x109163: main (in ...)
==21801==  Address 0x4a490a4 is 0 bytes after a block of size 100 alloc'd
==21801==    at 0x483577F: malloc (vg_replace_malloc.c:309)
==21801==    by 0x109156: main (in ...)
```

Strangely, while

`char *c; c=(char*)malloc(100); c[100]=10; free(c);`

and

`static char c[100]; c[100]=10;`

both store c on the heap, only the former allocation error is detected by `valgrind`.

Benefit from analysis tools

valgrind to investigate access to non-allocated memory areas
(less obvious to analyze on the stack: program counter corruption when attempting to return from the printf() function)

```c
int main()
{int i[3],k;
 for (k=0;k<10;k++) {i[k]=k;printf("%d ",i[k]);}
}
```

is executed to yield a Segmentation fault ...
... analyzed by valgrind -q ./my_program et tells us

```
==22408== Jump to the invalid address stated on the next line
==22408==    at 0x700000006: ???
==22408==    by 0x900000007: ???
==22408==    by 0x48ACBBA: (below main) (libc-start.c:308)
==22408==  Address 0x700000006 is not stack'd, malloc'd or (recently) free'd
==22408==
==22408==
==22408== Process terminating with default action of signal 11 (SIGSEGV)
==22408==  Access not within mapped region at address 0x700000006
==22408==    at 0x700000006: ???
==22408==    by 0x900000007: ???
==22408==    by 0x48ACBBA: (below main) (libc-start.c:308)
```

Benefit from analysis tools

gprof to analyze execution time of each function

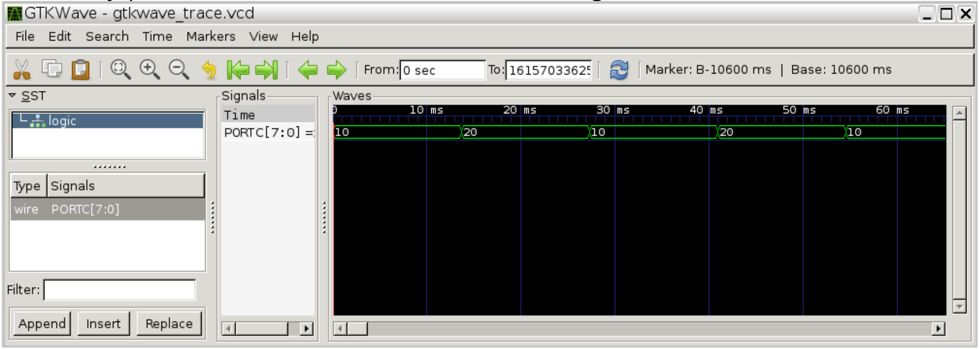
```
void function1s()
{volatile int k;for (k=0;k<0x1000000;k++) {};}

void function2s()
{volatile int k;for (k=0;k<0x2000000;k++) {};}

void function3s()
{volatile int k;for (k=0;k<0x3000000;k++) {};}

int main()
{function1s();
 function2s();
 function3s();
}
```

is compiled with the -pg option to generate upon execution a `gmon.out` log file ...
... analyzed with `gprof -bp ./my_program` and providing

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 55.56      0.10     0.10        1   100.00   100.00  function3s
 33.33      0.16     0.06        1    60.00    60.00  function2s
 11.11      0.18     0.02        1    20.00    20.00  function1s
```

Simulator

1. Use a simulator to probe the internal state of the machine executing the code sequence
2. qemu for many platforms, now Renode, simavr for Atmega32U2



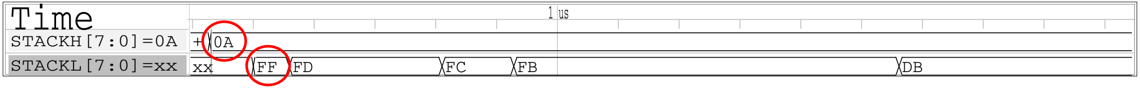
Screenshot of gtkwave used to display the evolution of port C.

Prefix program with simulation instructions:

```
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32");

const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
        { AVR_MCU_VCD_SYMBOL("PORTB"), .what = (void*)&PORTB, },
    };

int main {DDRB |=1<<PORTB5; while (1){PORTB^=1<<PORTB5;_delay_ms(1000);}}
```

Simulator

1. Use a simulator to probe the internal state of the machine executing the code sequence
2. qemu for many platforms, now Renode, `simavr` for Atmega32U**2**



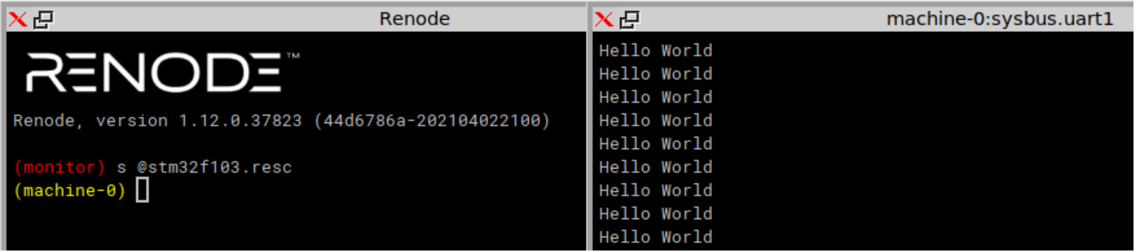
Prefix program with declaration of simulation conditions:

```
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32");

const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
 {AVR_MCU_VCD_SYMBOL("PORTC"), .what=(void*)(0x28), }, // &PORTC
 {AVR_MCU_VCD_SYMBOL("STACKL"),.what=(void*)(0x5D), }, // SP=0x3{DE}+0x20
 {AVR_MCU_VCD_SYMBOL("STACKH"),.what=(void*)(0x5E), }, // SP=0x3{DE}+0x20
};
```

Memory		Mnemonic	ATmega32U4	ATmega16U4
Flash	Size	Flash size	32KB	16KB
	Start Address	- 0x0000		
	End Address	Flash end	0x7FFF[(1)] 0x3FFF[(2)]	0x3FFF[(1)] 0x1FFF[(2)]
32 Registers	Size	-	32 bytes	32 bytes
	Start Address	-	0x0000	0x0000
	End Address	-	0x001F	0x001F
I/O Registers	Size	-	64 bytes	64 bytes
	Start Address	-	0x0020	0x0020
	End Address	-	0x005F	0x005F
Ext I/O Registers	Size	-	160 bytes	160 bytes
	Start Address	-	0x0060	0x0060
	End Address	-	0x00FF	0x00FF
Internal SRAM	Size	ISRAM size	2,5KB	1.25KB
	Start Address	ISRAM start	0x100	0x100
	End Address	ISRAM end	0x0AFF	0x05FF

Simulator

1. Use a simulator to probe the internal state of the machine executing the code sequence
2. qemu for many platforms, now Renode, `simavr` for Atmega32U**2**



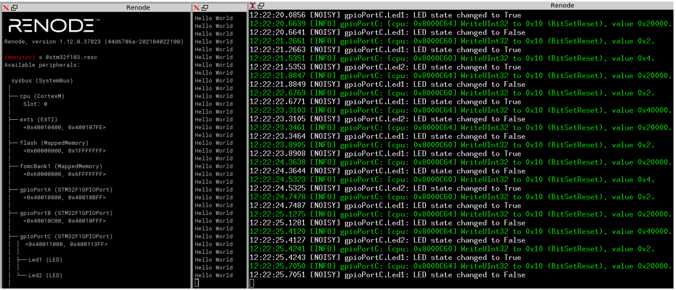
`stm32f103.resc`:

```
using sysbus
mach create
machine LoadPlatformDescription @platforms/cpus/stm32f103.repl
showAnalyzer uart2
showAnalyzer uart1
peripherals
logLevel -1 sysbus.gpioPortC.Led2
logLevel -1 sysbus.gpioPortC.Led1
logLevel -1 sysbus.gpioPortC
logLevel 3
sysbus LogPeripheralAccess sysbus.gpioPortC
macro reset
"""
    sysbus LoadELF @output/main.elf
"""
runMacro $reset
```

Simulator

1. Use a simulator to probe the internal state of the machine executing the code sequence
2. `qemu` for many platforms, now Renode, `simavr` for Atmega32U**2**

Adding custom LED support:



`stm32f103.resc`:

```
using sysbus
mach create
machine LoadPlatformDescription @platforms/cpus/stm32f103.repl
machine LoadPlatformDescription @stm32f1_led.repl
showAnalyzer uart2
showAnalyzer uart1
peripherals
logLevel -1 sysbus.gpioPortC.Led2
logLevel -1 sysbus.gpioPortC.Led1
logLevel -1 sysbus.gpioPortC
logLevel 3
sysbus LogPeripheralAccess sysbus.gpioPortC
macro reset
"""
    sysbus LoadELF @output/main.elf
"""
runMacro $reset
```

with `stm32f1_led.repl`

```
Led1: Miscellaneous.LED @ gpioPortC
gpioPortC:
    1 -> Led1@0
Led2: Miscellaneous.LED @ gpioPortC
gpioPortC:
    2 -> Led2@0
```

Emulating peripherals

An emulator simulates the bahaviour of the processor core, but how to emulate the behaviour of peripherals [4]?

Example of the analog to digital converter of the STM32:

```
static void stm32_adc_start_conv(Stm32Adc *s)
{uint64_t curr_time = qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL);
 int channel_number=stm32_ADC_get_channel_number(s,1);
 if (channel_number==16)
    {s->Vdda=rand()%(1200+1) + 2400;         // Vdda belongs to the interval [2400 3600] mv
     s->Vref=rand()%(s->Vdda-2400+1) + 2400; // Vref belongs to the interval [2400 Vdda] mv
     s->ADC_DR= s->Vdda - s->Vref;
    }
 else if (channel_number==17)i
    {s->ADC_DR= (s->Vref=rand()%(s->Vdda-2400+1) + 2400); //Vref [2400 Vdda] mv
    }
 else
    {s->ADC_DR=((int)(1024.*(sin(2*M_PI*qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL)/1e9)+1.))&0xfff);
    }
 s->ADC_SR&=~ADC_SR_EOC;  // indicates ongoing conversion
 s->ADC_CR2&=~ADC_CR2_SWSTART;
 // calls conv_complete when expires
 timer_mod(s->conv_timer,  curr_time + stm32_ADC_get_nbr_cycle_per_sample(s,channel_number));
}
```

[4] https://github.com/beckus/qemu_stm32/blob/stm32/hw/arm/stm32_adc.c#L700

Tackling a new architecture: RISC V

```c
#include <stdio.h>
int main()
{long x=0x12345678;
 char *c=(char*)&x;
 printf("\n%d\n",sizeof(long));
 printf("%hhx %hhx %hhx %hhx\n",c[0],c[1],c[2],c[3]);
}
```

Remember: `newlib` needs the *stub* `write()` to display a message on a known peripheral

```
FEMTO=[...]/riscv-probe/
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -mcmodel=medany -c h.c
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -mcmodel=medany \
  -nostartfiles -nostdlib -nostdinc -static -lgcc \
  -T $(FEMTO)/env/qemu-sifive_e/default.lds \
  $(FEMTO)/build/obj/rv32imac/env/qemu-sifive_e/crt.o \
  $(FEMTO)/build/obj/rv32imac/env/qemu-sifive_e/setup.o h.o \
  $(FEMTO)/build/lib/rv32imac/libfemto.a -o hello32
```

is executed in qemu with

```
$ [...]/riscv-qemu/riscv32-softmmu/qemu-system-riscv32 -M sifive_e \
  -nographic -kernel hello32
4
00000078 00000056 00000034 00000012
```

→ `long` is encoded as a 4-byte (32 bit) integer on this architecture

→ 0x78 appears first, the least significant bit on the lowest address location: *little endian*

Tackling a new architecture: RISC V

```
#include <stdio.h>
int main()
{long x=0x12345678;
 char *c=(char*)&x;
 printf("\n%d\n",sizeof(long));
 printf("%hhx %hhx %hhx %hhx\n",c[0],c[1],c[2],c[3]);
}
```

Remember: `newlib` needs the *stub* `write()` to display a message on a known peripheral
Compiling to a 64 bit architecture (options `-march=` and `-mabi`):

```
riscv64-unknown-elf-gcc -march=rv64imac -mabi=lp64 -mcmodel=medany -c h.c
riscv64-unknown-elf-gcc -march=rv64imac -mabi=lp64 -mcmodel=medany \
  -nostartfiles -nostdlib -nostdinc -static -lgcc \
  -T $(FEMTO)/env/qemu-sifive_e/default.lds \
  $(FEMTO)/build/obj/rv64imac/env/qemu-sifive_e/crt.o \
  $(FEMTO)/build/obj/rv64imac/env/qemu-sifive_e/setup.o h.o
  $(FEMTO)/build/lib/rv64imac/libfemto.a -o hello64
```

we obtain on the 64 bit qemu version the result

```
$ [...]/riscv-qemu/riscv64-softmmu/qemu-system-riscv64 -M sifive_e \
  -nographic -kernel hello64
8
00000078 00000056 00000034 00000012
```

→ `long` uses 8 bytes (64 bits)
→ still *little endian*

gdb basics

- Compile with symbols for debugging: `gcc -g ...`
- Connect to qemu with gdb [5]:

  1. `qemu-system-arm -M stm32-p103 -gdb tcp::3333 -S -kernel main.bin`
  2. `qemu --help`: "-s shorthand for -gdb tcp::1234"
  3. `gdb-multiarch main.elf`
     - `target remote :  3333`
     - `continue`
     - `bt`

```
(gdb) bt
#0  usart_wait_send_ready (usart=usart@entry=1073821696) at ../common/usart_common_f124.c:81
#1  0x0800089a in usart_send_blocking (usart=1073821696, data=<optimized out>) at ../common/usart_common_all.c:223
#2  0x080001ee in uart_putc ()
#3  0x08000450 in affchar ()
#4  0x0800047a in affshort ()
#5  0x0800026e in main ()
```

     - `list`

```
(gdb) list
78      void usart_wait_send_ready(uint32_t usart)
79      {
80              /* Wait until the data has been transferred into the shift register. */
81              while ((USART_SR(usart) & USART_SR_TXE) == 0);
82      }
```

- On PC when an operating system is supervising execution; `ulimit -c unlimited` to dump core for post-mortem debugging

[5] On Debian/GNU Linux: `gdb-multiarch` package

gdb basics

- Compile with symbols for debugging: `gcc -g ...`
- Connect to qemu with gdb [6]:
  1. `qemu-system-arm -M stm32-p103 -gdb tcp::3333 -S -kernel main.bin`
  2. `qemu --help`: "-s shorthand for -gdb tcp::1234"
  3. `gdb-multiarch main.elf`
     - `target remote : 3333`
     - `continue`
     - `break myfunction`

```
(gdb) break usart_wait_send_ready
Breakpoint 1 at 0x80008ae: file ../common/usart_common_f124.c, line 81.
(gdb) continue
Continuing.
Breakpoint 1, usart_wait_send_ready (usart=usart@entry=1073821696) at ../common/usart_common_f124.c:81
81              while ((USART_SR(usart) & USART_SR_TXE) == 0);
```

     - `info registers`

```
(gdb) info registers
r0             0x40011000          1073811456
r1             0x2                 2
r2             0xc34ff             799999
```

     - `print variable`

```
(gdb) print c
$1 = 6
```

- On PC when an operating system is supervising execution; `ulimit -c unlimited` to dump core for post-mortem debugging

[6] On Debian/GNU Linux: `gdb-multiarch` package

More gdb commands [7]

- `continue`: continue to next breakpoint or end
- `run`: run to next breakpoint or to end
- `step`: single-step, descending into functions
- `next`: single-step without descending into functions
- `finish`: finish current function, loop, etc..
- `info b`: list breakpoints
- `disable 1`: disable breakpoint 1
- `enable 1`: enable breakpoint 1
- `delete 1`: delete breakpoint 1

[7] https://ccrma.stanford.edu/~jos/stkintro/Useful_commands_gdb.html

Failing to get an output when executing on qemu?

Using the "official" qemu for ARM (`sudo apt install qemu-system-arm`) support STM32VLDiscovery:

1. Launch the simulation with gdb server enabled:

```
qemu-system-arm -M stm32vldiscovery -s -serial stdio -kernel test.bin
```

2. Connect gdb and probe the *ARM core* state:

```
$ gdb-multiarch test.elf
(gdb) target remote:1234
Remote debugging using :1234
0x080006fa in rcc_is_osc_ready (osc=osc@entry=RCC_HSE) at rcc.c:353
353                     return RCC_CR & RCC_CR_HSERDY;
(gdb) bt
Remote debugging using :1234
0x080006fa in rcc_is_osc_ready (osc=osc@entry=RCC_HSE) at rcc.c:353
353                     return RCC_CR & RCC_CR_HSERDY;
(gdb) bt
#0  0x080006fa in rcc_is_osc_ready (osc=osc@entry=RCC_HSE) at rcc.c:353
#1  0x0800072a in rcc_wait_for_osc_ready (osc=<optimized out>) at rcc.c:366
#2  0x08000dfe in rcc_clock_setup_pll (clock=0x80014b0 <rcc_hse_configs+84>)
    at rcc.c:1220
#3  0x080001f2 in core_clock_setup ()
#4  0x080002b4 in clock_setup ()
#5  0x0800035c in main ()
```

qemu does not emulate correctly HSE initialization and `libopencm3` waits for a hardware clock initialization flag...

- `libopencm3` is compiled with the -ggdb3 flag: `list` from gdb displays C source code (previous slide) [8]
- without -g flag, our program listing is displayed as assembly language

```
$ gdb-multiarch test.elf
(gdb) target remote localhost:1234
(gdb) continue
Continuing.
^C
Program received signal SIGINT, Interrupt.
0x08000266 in delay ()
(gdb) list .
Insufficient debug info for showing source lines at current PC (0x8000266).
```

- adding -g flag includes the C listing in the ELF file

[8] see `make V=1` output

- libopencm3 is compiled with the -ggdb3 flag: list from gdb displays C source code (previous slide) [9]
- without -g flag, our program listing is displayed as assembly language
- adding -g flag includes the C listing in the ELF file

```
$ gdb-multiarch test.elf
(gdb) target remote localhost:1234
(gdb) continu
Continuing.
^C
Program received signal SIGINT, Interrupt.
delay (delay=65535) at uart_f1.c:46
46                      __asm__("nop");
(gdb) list .
41
42        void delay(unsigned int delay)
43        {
44                volatile unsigned int i;
45                for(i=0;i<delay;i++)
46                        __asm__("nop");
47        }
48
49
50        void init_gpio(void)
```

[9] see make V=1 output

PC stubs v.s emulator

In both cases the software targeting the microcontroller is executed on the PC:

qemu emulator

- peripheral emulation matches embedded microcontroller register layout and timing
- opcodes are those of the targeted core and the ALU behaviour is emulated
- gdb allows accessing the microcontroller registers
- debug messages in emulator to access internal states not accessible in real hardware
- memory limitations included in emulator (stack on top of RAM...)
- instruction set limited to those of the emulated microcontroller core (no floating point, no MMU)

⇒ closer to real hardware but less flexible

stub on PC

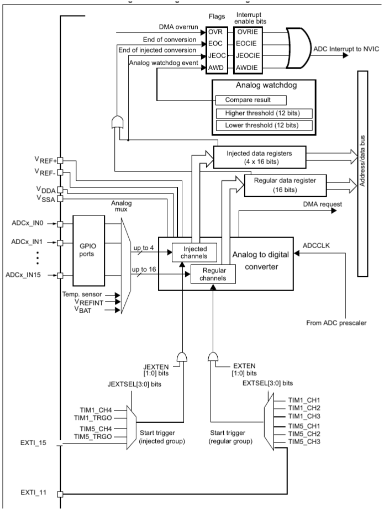
- behavioural description of peripheral access (`printf(...)`)
- opcodes are those of the host core and do not claim timing or behavioural accuracy on target
- analysis under operating system supervision (valgrind, gprof)
- inject synthetic signals to probe peripheral behaviour (could be done in emulator with external probes to communication interfaces – see `simavr` [a])
- no memory limitation since program executes as a task of the operating system

[a] J.-M. Friedt, *Émulation d'un circuit comportant un processeur Atmel avec simavr*, Hackable **34** (Jul.-Sept. 2020)

Conclusion

Separating algorithm and hardware access allows for

- using profiling and code analysis tools (`lint`, `valgrind` ...)
- unit testing (`cunit`)
- adress a new processor without changing the core processing algorithm.



Single analog-digital converter: 1×12-bit at 2.4 MSPs

```
void init_adc(void)
{gpio_mode_setup (GPIOB, GPIO_MODE_ANALOG, GPIO_PUPD_NONE, GPIO0|GPIO1);

 adc_power_off (ADC1);
 adc_disable_scan_mode (ADC1);
// ADCCLOCK < 30 MHz, from APB2 => prescale tq APB2/DIV<30 MHz
 adc_set_clk_prescale (ADC_CCR_ADCPRE_BY4);
 adc_set_sample_time_on_all_channels (ADC1, ADC_SMPR_SMP_112CYC);
//si ADCCLOCK = 21 MHz, sampletime = 144 => Tconv = 7.4286us.
 adc_power_on (ADC1);
}

unsigned short read_adc (unsigned char channel)
{uint8_t channel_array[16];
 channel_array[0] = channel;
 adc_set_regular_sequence (ADC1, 1, channel_array);  // used channel
 adc_start_conversion_regular (ADC1);
 while (!adc_eoc (ADC1));
 return(adc_read_regular(ADC1));
}
```

Laboratory session

- Based on a program displaying a message using a custom function, replace your function with `printf()`, add the relevant stubs to link with newlib, and observe the binary size evolution. You will have to add `-lc -lnosys` [10] library linking flags to properly compile the executable binary: doing so, only `int _write(int file, char *ptr, int len)` needs to be overwritten, the other stubs being defined as empty by default.
- Observe the output when mixing printf display and custom display (`mon_putchar`). Interpret.
- Use the appropriate command to get rid of the issue you just observed. As a general rule, remember this behaviour of `printf()` when debugging C program displaying outputs on the console.
- Evaluate the time needed to display a message, *excluding* communication time, *i.e.* only computing the string output of `printf` but not sending to a communication port. How does it compare with your custom implementation?
- After compiling with the debugging option -g, analyze the content of the disassembled code using `arm-none-eabi-objdump -dSt` and find the location of your `_write` stub.
- Add some calls to `libm`, e.g. by computing the square root (`sqrt()`) or `atan()` of a floating point number (`float variable;`)

[10] see `https://github.com/eblot/newlib/blob/master/libgloss/libnosys/write.c` for the empty implementation of `_write` in `libnosys`

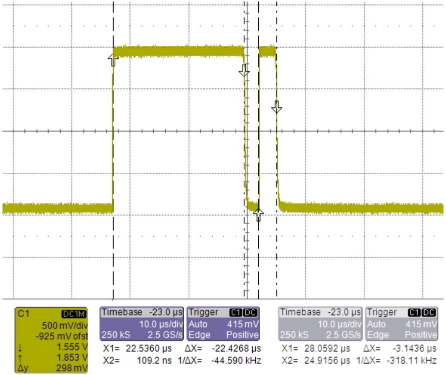
Solutions

- printf: 36496 byte .bin file, v.s 5080 bytes without

```
int _write(int file, char *ptr, int len)
{int i;
 for (i = 0; i < len; i++) {
      if (ptr[i] == '\n') mon_putchar('\r');
      mon_putchar(ptr[i]);
     }
  return i;
}
```

- The `printf()` function is buffered and only displays when a given amount of data has been stored

```
r
r
r
r
r
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10 Hello World 10
Hello World 10 Hello World 10 Hellr
r
r
r
r
r
```



- `fflush(0);` will force `printf()` to flush its buffer
- and display the content of the character array

Laboratory session (PC)

On the PC:

1. Write a program to compute recursively the factorial value of n defined as $n! = n \times (n-1) \times (n-2) \ldots \times 2$
2. Execute on the PC and check that the result of $4! = 24$ and $5! = 120$
3. If arguments passed as a command line argument, then `gdb --args ./program arg`
4. Execute under gdb[11] (arguments can be provided from with gdb with `set args`)
5. place a breakpoint at the `fact()` function with `b fact` and execute the program (`run`)
6. print the variable value: `print n`
7. continue execution until next breakpoint is reached: `continu`
8. step execution to next instruction: `step`
9. print the calling function history `bt`
10. print the stack pointer: `p $sp`
11. print the memory content at this address in decimal: `x/10 $sp`
12. print the memory content at this address in hexadecimal: `x/10x $sp`
13. print the memory content at this address as shorts in hexadecimal: `x/10hx $sp`
14. enable the text user interface: `tui enable`

Post-mortem analysis: `ulimit -c unlimited` to `core dump` and `gdb exec core`

[11] On Debian: `apt install gdb`

Laboratory session (ARM emulator)

ARM emulator (official qemu [12]):

1. compile with `arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 --static  -L$HOME/libopencm3/lib \`

   `-L$HOME/stm32/ -Tstm32f1.ld -nostartfiles program.c -o program -lopencm3_stm32f1`
2. convert to binary file with `arm-none-eabi-objcopy -Obinary program program.bin`
3. emulate with `qemu-system-arm -M stm32vldiscovery -serial stdio -kernel program.bin -S -s`
4. connect to gdb server [13]:

   ```
   gdb-multiarch program
   target remote localhost:1234
   ```
5. set breakpoint to fact()
6. continue execution
7. print stack pointer `print $sp` and stack content `x/10 $sp`
8. continue again and print stack: the next value of `n` is displayed
9. print heap content `x/10 0x20000000`
10. `print n` fails for lack of symbols: restart with `-g`

```c
#include <stdio.h>
#include <stdlib.h>

int fact(int n)
{volatile int tmp=n; // on stack
 if (n>2)
    return(fact(n-1)*n);
}

int main(int argc, char** argv)
{static int val=5;  // on heap
 if (argc>1) val=atoi(argv[1]);
 printf("resultat %d\n",fact(val));
 fact(val);
}
```

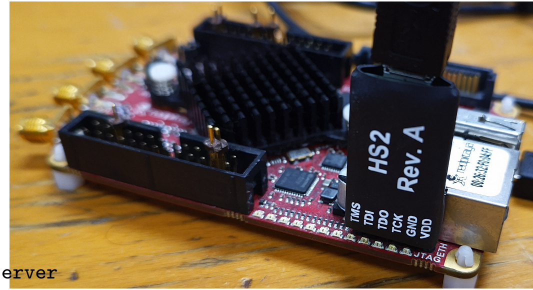
See FOSDEM session on debuggers at `archive.fosdem.org/2024/schedule/track/debuggers-and-analysis/`

[12] On Debian/GNU Linux: `qemu-system-arm`
[13] On Debian/GNU Linux: `gdb-multarch`

GDB for debugging embedded hardware

- JTAG[14] probe, e.g. Digilent JTAG HS2 supported by OpenOCD (JTAG $\leftrightarrow$ USB)
- JTAG bus: TDI/TDO (Test Data In/Out pins), TMS (Test Mode State pin), TCK (JTAG Return Test Clock), RESET
- many derivates for accessing microcontroller internals with fewer pins:
  - STM32 boards: STLink connector (notice that STM32 evaluation boards can be used as STLink interface to another STM32) [15]
  - Atmega boards: ICE (In-Circuit Emulation using *debugWire*) connector [16] on PF(4, 5, 6, 7), ICSP[17] on PB(1, 2, 3)
  - MSP430 [18]
  - MIPS [19]
  - Stellaris SWD (Serial Wire Debug)[20]

All these hardware interfaces are handled by a GDB server

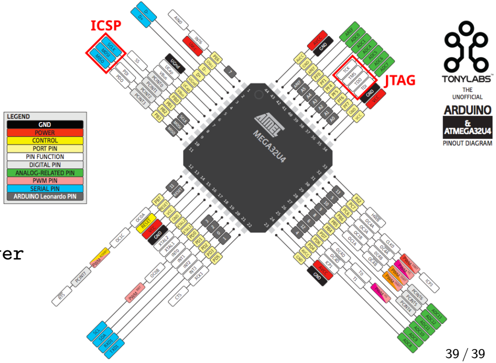


[14] Joint Test Action Group
[15] https://github.com/stlink-org/stlink
[16] https://avarice.sourceforge.io/
[17] https://github.com/tetofonta/gdb-debug-wire-integrated-server
[18] https://github.com/dlbeer/mspdebug
[19] https://github.com/sergev/ejtagproxy
[20] https://reversepcb.com/swd-vs-jtag/

GDB for debugging embedded hardware

- JTAG[14] probe, e.g. Digilent JTAG HS2 supported by OpenOCD (JTAG $\leftrightarrow$ USB)
- JTAG bus: TDI/TDO (Test Data In/Out pins), TMS (Test Mode State pin), TCK (JTAG Return Test Clock), RESET
- many derivates for accessing microcontroller internals with fewer pins:
  - STM32 boards: STLink connector (notice that STM32 evaluation boards can be used as STLink interface to another STM32) [15]
  - Atmega boards: ICE (In-Circuit Emulation using *debugWire*) connector [16] on PF(4, 5, 6, 7), ICSP[17] on PB(1, 2, 3)
  - MSP430 [18]
  - MIPS [19]
  - Stellaris SWD (Serial Wire Debug)[20]

All these hardware interfaces are handled by a GDB server



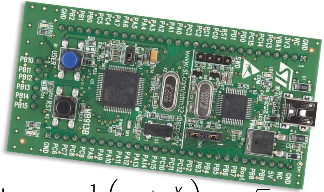
[14] Joint Test Action Group
[15] https://github.com/stlink-org/stlink
[16] https://avarice.sourceforge.io/
[17] https://github.com/tetofonta/gdb-debug-wire-integrated-server
[18] https://github.com/dlbeer/mspdebug
[19] https://github.com/sergev/ejtagproxy
[20] https://reversepcb.com/swd-vs-jtag/

Laboratory session (ARM hardware [21])

1. GDB server using the STLink protocol found on ST Microelectronics evaluation boards: `apt install stlink-tools` or compile `https://github.com/stlink-org/stlink`
2. run `st-util` to connect to the board: `lsusb` should indicate `ID 0483:3744 STMicroelectronics ST-LINK/V1` and upon connecting, st-util will answer

   ```
   INFO common.c: STM32F1xx_CL: 64 KiB SRAM, 128 KiB flash in at least 2 KiB pages.
   INFO gdb-server.c: Listening at *:4242...
   ```

3. Compile with `Makefile.stm32f100` to use the right clock settings (24 MHz). Check the linker script: the RAM size must be 8 KB.
4. launch `gdb-multiarch test.elf` to run the gdb client
5. `target remote localhost:4242` (or `target extended-remote localhost:4242` to avoid `st-util` from quitting every time we leave gdb) will bring the message from `st-util`: `gdb-server.c: GDB connected.`
6. `load test.elf` to load the program in the STM32F100 memory
7. `info breakpoints` to get a list of breakpoints
8. alternative to `st-util`: openOCD



Exercises:

- LD3 is PC9, LD4 is PC8: see how `usart_f1.c` handles the different settings between STM32F103 or STM32F100 of the Value Line evaluation board.
- Write a program for computing the square root of an integer following Newton's method $x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n}\right) \rightarrow \sqrt{y}$ and read the result from gdb. Compare with the execution time and binary size of `libm`'s `sqrt`

[21] `www.st.com/resource/en/user_manual/um0919-stm32vldiscovery-stm32-value-line-discovery-stmicroelectronic`