FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

182.694 Microcontroller VU

FAKULTÄT FÜR !NFORMATIK

Martin Perner

SS 2017

Featuring Today:
Structured C Programming
Testing

Weekly Training Objective

- Already done
  - 3.1.1 C demo program †
  - 3.3.2 Interrupts †
  - 3.4.3 Generating periodic signals †
  - 3.8.1 Switches †
- This week
  - 3.1.3 Floating point operations ∗
  - 3.3.1 Interrupt & callback demo ∗
  - 3.6.1 UART receiver ∗
  - 3.6.2 UART sender ∗
- Next week
  - 3.4.4 PWM signals and glitches
  - 3.6.4 TWI (I^2C) ∗
  - 3.9.1 Keypad

Application 1 – Q&A

Any Questions regarding Application 1?

You should use a source control management system.

- For example: git
- `git bisect` may come in handy when something 'just' stopped working.

Structured C Programming

Important Paradigms

- background (BG) tasks vs. interrupt service routines (ISRs)
- callback functions

General approaches for common problems (Design Patterns).

Helpful extensions

Compile with "`-std=c99`" and take a look at

- `<util/atomic.h>` provides basic atomic operations.
- `<stdbool.h>` provides definition of `bool` as datatype.
- useful macros for
  interrupts, EEPROM, sleep-mode, CRC, **baudrate**, watchdog, ...
- detailed documentation: avr-libc manual

Backgrounding

Basic structure of a C-program

```
int main(void)
{
    init(); // calls initializtions of used modules
    for (;;) {
        background(); // calls background tasks
        sleep();      // put the MC into sleep mode
    }
}
```

An interrupt “terminates” the sleep. Then the background code is executed once, before the sleep mode is entered again.

Backgrounding

Basic structure of a background task

```
void background(void)
{
  if (INIT == cur_state) {
    bgtask_playIntroMusic();
    bgtask_showIntroAnimation();
  }
  else if (GOAL == cur_state) {
    ...
  }
}
```

Backgrounding

Problematic Behavior

- Animation will be played after `bgtask_playIntroMusic` returned.
- A background task can monopolize the execution!
  - This can cause a major increase in response-time of the system!
  - Break-up background tasks so that they can be called iteratively.

Backgrounding done “right”

Using “partial” functions for background task

```
void background(void)
{
  while (there_is_something_to_do()) {
    if (INIT == cur_state) {
      if (!done_music) {
        done_music = bgtask_playMoreIntroMusic();
      }
      if (!done_anim) {
        done_anim = bgtask_showMoreIntroAnimation();
      }
    }
    else if (GOAL == cur_state) {
        ...
    }
  }
}
```

Backgrounding done “right”

Pitfalls

- You need to ensure that every background function which may need to be processed, can be called.
  E.g., ISR enables a disco-light while there is still music to play.
- Pay attention to interrupts that change global/state variable while you process background tasks.
  E.g., ISR changes the center of a circle while it is drawn.

Callback Functions

When is polling not an option?

- long running computation
- library (interrupt) interacts with the program
- ...

Use callbacks!

- also called, resp. similar to, hooks
- provide function pointer(s) to a setup-function
- provided function(s) will be called by library upon certain event(s)

Callback Functions

Toy example of a callback function usage

```
void (*myCallbackFunction)(const uint8_t); // function pointer variable

ISR(MY_TIMER_INTERRUPT, ISR_NOBLOCK)
{
  /* execute the callback function */
  myCallbackFunction(1);
}

void set_callback(void (*callback)(const uint8_t))
{
  myCallbackFunction = callback;
}


void myfunc (const uint8_t i)
{
  ...
}

int main(void)
{
  set_callback(myfunc); //register myfunc as callback function
  for (;;) {}
}
```

While we are on the topic of interupts...

What does this progamm do?

```
bool timeout = false;

ISR(MY_TIMER_INTERRUPT, ISR_NOBLOCK)
{
  timeout = true;
}

int main (void)
{
  for (;;) {
    if (timeout == true) {
      turn_on_leds();
    }
  }
}
```

While we are on the topic of interupts...

What does this progamm do?

```
bool timeout = false;

ISR(MY_TIMER_INTERRUPT, ISR_NOBLOCK)
{
  timeout = true;
}

int main (void)
{
  for (;;) {
    if (timeout == true) {
      turn_on_leds();
    }
  }
}
```

On some versions of the toolchain the if-statement will be interpreted as false, always!

The volatile keyword

Volatile

```
bool volatile timeout = false;

ISR(MY_TIMER_INTERRUPT, ISR_NOBLOCK)
{
  timeout = true;
}

int main (void)
{
  for (;;) {
    if (timeout == true) {
      turn_on_leds();
    }
  }
}
```

volatile informs the compiler that `timeout` may change between subsequent accesses.

Side Effects

Avoid side effects!

- Never use something like `printf("DBG: %i", i++)` etc.
- Especially problematic in conditionally executed code!

Side Effects

Avoid side effects!

- Never use something like `printf("DBG: %i", i++)` etc.
- Especially problematic in conditionally executed code!

What will be?

`i` will be initialized to `1` in all cases and `id` is the identity function.

```
be = i++ + i
be = i + ++i
be = id(i++) + i
be = id(++i) + i
```

Side Effects

Avoid side effects!

- Never use something like `printf("DBG: %i", i++)` etc.
- Especially problematic in conditionally executed code!

What will be?

`i` will be initialized to `1` in all cases and `id` is the identity function.

```
be = i++ + i       // 2
be = i + ++i
be = id(i++) + i
be = id(++i) + i
```

Side Effects

Avoid side effects!

- Never use something like `printf("DBG: %i", i++)` etc.
- Especially problematic in conditionally executed code!

What will be?

`i` will be initialized to `1` in all cases and `id` is the identity function.

```
be = i++ + i      // 2
be = i + ++i      // 4
be = id(i++) + i
be = id(++i) + i
```

Side Effects

Avoid side effects!

- Never use something like `printf("DBG: %i", i++)` etc.
- Especially problematic in conditionally executed code!

What will be?

`i` will be initialized to `1` in all cases and `id` is the identity function.

```
be = i++ + i       // 2
be = i + ++i       // 4
be = id(i++) + i   // 3
be = id(++i) + i
```

Side Effects

Avoid side effects!

- Never use something like `printf("DBG: %i", i++)` etc.
- Especially problematic in conditionally executed code!

What will be?

`i` will be initialized to `1` in all cases and `id` is the identity function.

```
be = i++ + i       // 2
be = i + ++i       // 4
be = id(i++) + i   // 3
be = id(++i) + i   // 4
```

Short-Circuit Evaluations

Use the C-Standard, Luke!

```
int a = 0;
if (x() && a) {
   ...
}
else {
   ...
}
```

```
int a = 0;
if (a && x()) {
   ...
}
else {
   ...
}
```

Short-Circuit Evaluations

Use the C-Standard, Luke!

```
int a = 0;
if (x() && a) {
   ...
}
else {
   ...
}
```

```
int a = 0;
if (a && x()) {
   ...
}
else {
   ...
}
```

Depending on value of a, x() may or may not be called!
This can be used, e.g., to avoid division by zero!
Analog behavior for ||: evaluation is aborted after the first true evaluation.

The const keyword

Why const(ant) variables?

- **const** is used to declare variables that are written only once.
- You should declare whether this is your intention or not:
  `myfunc(uint8_t *str)`
  `myfunc(const uint8_t *str)`.
- C allows using pointers as input and output parameters.
  Also known as pass-by-references, compared to pass-by-value.
- Some kind of insurances: the compiler will not let you modify the variable.

ATMega1280's memories

- 128 KB program memory (flash)
  - Biggest memory in ATMega1280
  - Great for storing constants
- 8 KB internal RAM
  - RAM size is quite restricted
  - Beware of stack overflow
- 4 KB EEPROM
  - Useful for storing values that rarely change
  - Persistent storage
  - Limited number of read/write cycles

Using ATMega1280's memories in C

- Memory sections available:
  - `.text`: program memory, flash
  - `.data`: RAM and flash
  - `.bss`: RAM
  - `.eeprom`
- The `.data` memory section contains constants and variable initializations; it occupies both RAM and flash space.
  - RAM is volatile and therefore not programmable (before run-time).
- Therefore: put constant values into flash alone!
  - No need to occupy twice the required memory!
  - How? See Section 5 of the avr-libc manual.

The .initN sections (Section 4.6 avr-libc)

.init0 Weakly bound to __init(). If user defines __init(), it will be jumped into immediately after a reset.

.init1 Unused. User definable.

.init2 In C programs, weakly bound to initialize the stack, and to clear __zero_reg__ (r1).

.init3 Unused. User definable.

.init4 For devices with $>$ 64 KB of ROM, .init4 defines the code which takes care of copying the contents of .data from the flash to SRAM. For all other devices, this code as well as the code to zero out the .bss section is loaded from libgcc.a.

.init5 Unused. User definable.

.init6 Unused for C programs, but used for constructors in C++ programs.

.init7 Unused. User definable.

.init8 Unused. User definable.

.init9 Jumps into main().

Using .initN sections

Applying attributes to functions

```
void my_init_portb (void) __attribute__ ((naked, used, section (".init3")));

void my_init_portb (void)
{
   PORTB = 0xff;
   DDRB = 0xff;
}
```

This results in `my_init_portb` to be called during the init3-phase.

Linker scripts

- (in the lab) located at `/usr/avr/lib/ldscripts/`
- Mapping of virtual memory adresses for the ELF-file
- Very powerful!
- Define the order of the .initN calls.

Benchmarking memory usage

- After compilation, the **static** memory usage of your program can be analyzed via `avr-size`.
- Analyzing the **dynamic** memory usage can be done by
  (1) Code analysis (requires bounding recursion depth and heap usage).
  (2) Run-time analysis directly on the uC or via simulation (needs a software that monitors memory access).

Dynamic memory benchmark example

```
void memstatInit(void) __attribute__ ((naked, used, section (".init3")));
extern uint8_t _end;  // end of bss


// never call this function!
void memstatInit()
{
   register uint16_t i;

   for (i=(uint16_t)&_end; i<RAMEND; i++) {
      *(uint8_t*)i = (uint8_t)i; // or some magic string, e.g., 0xC0FFEE for 32-bit
   }
}
```

Dynamic memory benchmark example

```
mem_stat_t *memstatGetStat(void)
{
    static mem_stat_t stat;
    uint16_t i;
    uint8_t uninit;

    stat.maxStackSize = 0xffff;
    stat.maxHeapSize = 0xffff;
    // search end of stack, start at SP and go down
      ...
    // search end of heap, start at bss_end and go up
    uninit = 0;
    for (i=(uint16_t)&_end; i<RAMEND; i++) {
        if (*(uint8_t*)i == (uint8_t)i) {
            uninit++;
            if (uninit == 10) {
             stat.maxHeapSize = i - (uint16_t)&_end - 9;
             break;
            }
        } else { uninit = 0; }
    }
    return &stat;
}
```

Bitfields

- Memory constraints demand efficient programming; in particular, using a byte to store a single bit (flag) is infeasible.
- Thus: pack eight bits in a byte using techniques that efficiently pack and unpack bits (union, struct).
- Other bitwise techniques also come in handy, e.g., swap contents of two registers without using a temp register.

Struct/union example

RTC Config Register:

```
typedef struct {
  uint8_t rs :2;
  uint8_t :2;
  uint8_t sqwe :1;
  uint8_t :2;
  uint8_t out :1;
} ds1307_sqw_config_t;

typedef union {
  ds1307_sqw_config_t regs;
  uint8_t raw;
} ds1307_sqw_mem_t;
```

```
ds1307_sqw_mem_t sqwconf;
// clear previous config
sqwconf.raw = 0;

// set squarewave generator to 32kHz
sqwconf.regs.sqwe = 1;
sqwconf.regs.rs   = 3;

// write RTC register with magic function
DS1307_registerWrtie(DS1307_CONTROL,
        sqwconf.raw);
```

Testing

What is Testing?

Testing is the process of analyzing software to determine if the generated behavior adheres to the specification.

What is Debugging?

Debugging is the process of recognizing, locating, and correcting an error.

Disclaimer

The following is a really short overview and by far not complete.

The Big Do Not

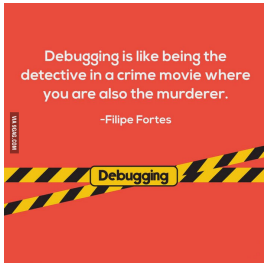

Figure: Debugging is like being the detective. http://9gag.com/gag/aojn3b0

A programmer should avoid attempting to test his or her own program.
— Glenford Myers – The Art of Software Testing

How can we test?

Two major classifications

Black-Box Testing

The specification is known, the source code is unknown.

White-Box Testing

The specification and the source code are known.

Black-Box Testing Techniques

- Equivalence class partitioning
- Boundary value analysis
- State based test
- Error guessing
- ...

Equivalence Class Partitioning

- Divide inputs into equivalence classes
- Ensure that an input of each class is tested

Example

The program must return `true` if the input is larger than 5. Otherwise, `false` must be returned.

- input > 5
- input ≤ 5

Boundary value analysis

Why working with boundary values?

Equivalence partitioning may not cover/test boundary conditions sufficiently.
E.g., the input 5 in the previous example.

- Test corner cases, where the behavior of the program changes
- Test extreme values of the inputs
- Test values for the inputs which may cause overflows in internal calculations
- Invalid inputs!

State Based Test

If the program specifies/implements a state-graph

. . . test the state transitions and the reachability of the states.

Error Guessing

. . . the Tester guesses which inputs may lead to incorrect behavior.

Automated Testsystem

Automated testing?

Advantages:

- reproducible results
- fun (to setup)

Typically 10 times as much can be manually test while automated tests are set-up.

If an automated testsystem is used

ensure that there is some negative test executed from time to time.

White-Box Testing

Taking a look at the code

- Allows to apply metrics
- Be sure that everything that has been written has been tested

White-Box Testing

Taking a look at the code

- Allows to apply metrics
- Be sure that everything that has been written has been tested ?

White-Box Testing

Taking a look at the code

- Allows to apply metrics
- Be sure that everything that has been written has been tested ?

Code Reviews

Are considered important for many reasons. They take time and the participants need to be prepared.
Checkout `https://users.ece.cmu.edu/~koopman/essays/code_review_checklist.html` for a list of items which should be checked during a code review.

Line/Statement Coverage

Toy Example

```
if (a && b) { PORTA = 0xff; }
```

Line Coverage

Measures the lines executed.

- Problematic, as there is a difference between high-level and low-level lines!
- Testcase 1: $(a = 0, b = 0)$
- Has 100% line coverage (reached lines / all lines)
- yet PORTA is never set.

Decision/Branch Coverage

Toy Example

```
if (a && b) { PORTA = 0xff; }
```

Branch Coverage

Measures the percentage of branches taken.

- Testcase 1: $(a = 0, b = 1)$
- Testcase 2: $(a = 1, b = 1)$
- Not all path combination may be tested!

Condition Coverage

Toy Example

```
if (a && b) { PORTA = 0xff; }
```

Condition Coverage

Complete condition coverage means that every conditions (variable in boolean expression) in every decision has taken all possible values at least once.

- Testcase 1: $(a = 1, b = 0)$
- Testcase 2: $(a = 0, b = 1)$
- Useless without further extension/metrics!
  E.g., combine Condition and Decision Coverage; min(CC, DC)

Baseline Testing

Cyclomatic complexity

The cyclomatic complexity (cg) is the number of linear independent paths through a program's control flow graph (CFG).
Calculation: Either $cg(G) = e - v + 2$, or $cg(G) = p + 1$ where p is the number of binary decisions (e.g, `if` or loop-abort condition).
Attention: Boolean operators with short-circuit evaluations increase $cg(G)$ by one!

Structured Testing

Execute tests so that a set of linear independent paths is passed. This set of paths is a basis for the vector space and minimal. The size of the set is $cg(G)$.

Baseline Testing

Cyclomatic complexity

The cyclomatic complexity (cg) is the number of linear independent paths through a program's control flow graph (CFG).
Calculation: Either $cg(G) = e - v + 2$, or $cg(G) = p + 1$ where p is the number of binary decisions (e.g, `if` or loop-abort condition).
Attention: Boolean operators with short-circuit evaluations increase $cg(G)$ by one!

Structured Testing

Execute tests so that a set of linear independent paths is passed. This set of paths is a basis for the vector space and minimal. The size of the set is $cg(G)$.

Who do we get linear independent paths?

Baseline Testing

Construction by Baseline Method

- Define a baseline path.
  This path must run through the whole method, i.e., no early aborts. Ideally this is also a very typical execution path.
- Toggle first decision in baseline path. Decisions following on this new path can be toggled arbitrarily. Try to keep the maximum number of decisions as in the baseline.
- Start with baseline, toggle the second decision, and so on.
- After all decisions of the baseline have been toggled, toggle the first decision. Use this path as a "new" baseline and toggle its decision which where not in the original baseline path.
- Continue until every decision has been toggled, the generation of test cases is complete.

Baseline Testing

Problems/Shortcomings

- Covering all edges can, usually, be accomplished with fewer paths.
- The fact that all paths can be generated by linear combinations of the linear independent paths, does not guarantee that every path is tested!
- The determination of the linear independent paths may be complicated.
- Due to dependencies in the code, it may not be possible to achieve $cg(G)$-independent paths.

Baseline Testing

Toy Example

```
if ( a || b ) {
    c = 1;
}
else {
    c = 2;
}

if (1 == c) {
    d = c/b;
}
```

What is the cyclomatic complexity for this snippet?

Baseline Testing

Toy Example

```
if ( a || b ) {
    c = 1;
}
else {
    c = 2;
}

if (1 == c) {
    d = c/b;
}
```

What is the cyclomatic complexity for this snippet? 4

Baseline Testing

Toy Example

```
if ( a || b ) {
    c = 1;
}
else {
    c = 2;
}

if (1 == c) {
    d = c/b;
}
```

What is the cyclomatic complexity for this snippet? 4
Due to the data-dependency (on c), only 3 linear independent paths can be generated!

Cyclomatic Complexity

Relation to Code Complexity

It is argued that a high $cg(G)$ leads to bad testability and maintainability. The justification is that a small change in the specification may lead to a large number of test cases which need to be modified. Furthermore, changing a small part may have side-effects on the remaining code.

Coding Guideline App1

Recall that the coding guideline for application 1 limits the $cg(G)$ to 10, and number of lines to 100.

For determining both we use `pmccabe`.

Note that for $cg(G)$, the modified cyclomatic complexity is used, which treats `switch` differently.

Testing

Best Practice

- Keep test cases non-overlapping. Minimize the number of test cases which are affected by a change of the specification. Cf. equivalence classes!
- Document the execution of a test:
  - Version of the tested software
  - Version of the test specification
  - Name of the tester and date
  - Concrete results for every test case. Especially for expected results which are a range, e.g.: Result shall be within $[50, 60] \Rightarrow$ Passed, 52.

The Problem with Metrics

Why are Metrics used?

- Putting a number to things allows to "judge"
- Allows to define a point when tests are considered "done"

Problems

- Metrics can be cheated (coverage reached, but outputs not checked).
- Not necessarily correlated with code quality
- Depending on the metric, even with 100% coverage there may still be bugs.

The Problem with Metrics

Why are Metrics used?

- Putting a number to things allows to "judge"
- Allows to define a point when tests are considered "done"

Problems

- Metrics can be cheated (coverage reached, but outputs not checked).
- Not necessarily correlated with code quality
- Depending on the metric, even with 100% coverage there may still be bugs.

> *Testing shows the presence, not the absence of bugs*
> — *Edsger Dijkstra*

The Problem with Metrics

Toy example

```
int d = 0;

if ( a || b ) {
    c = 1;
}
else {
    c = 2;
}

if (1 == c) {
    d = c/b;
}
```

Line Coverage

Using the test cases:

- $(a = 0, b = 0) \Rightarrow (c = 2, d = 0)$
- $(a = 0, b = 1) \Rightarrow (c = 1, d = 1)$

Coverage: 100%

$(a = 1, b = 0) \Rightarrow (c = 1, d = \perp)$

Division by zero not caught!

The Problem with Metrics

Toy example

```
int d = 0;

if ( a || b ) {
    c = 1;
}
else {
    c = 2;
}

if (1 == c) {
    d = c/b;
}
```

Line Coverage

Using the test cases:

- $(a = 0, b = 0) \Rightarrow (c = 2, d = 0)$
- $(a = 0, b = 1) \Rightarrow (c = 1, d = 1)$

Coverage: 100%

$(a = 1, b = 0) \Rightarrow (c = 1, d = \perp)$

Division by zero not caught!

Similar examples can be found for other coverages.

Debugging

A Test failed; what now?

1. Find bug
2. Fix bug
3. rerun tests
4. goto 1

First Things First

Check your Setup (The Coding Environment)

- automate the build (make)
- Enable compiler warnings/errors and do not ignore them blindly
- Debug top to bottom (execution wise)
- Fail early
- RTFM (read the f*** manual)

Tracing the Program Execution

Time Insensitive Code

- Write to LCD
- UART transmission

Time Sensitive Code

- Use bit patterns on some output to show some specific state.
  Might require an oscilloscope to check the trace.
- Store data in a buffer and transmit later.
  - Buffer insertion can influence the timing
  - Issues for high-speed/high-data scenarios (more data generated than emitted)

Typical Bug Sources

Manifestation After Module Integration

- Shared global state/variables
- Memory access violations
- Timing issues

In these cases, the integrated module might only trigger the bug, but is not the source!

Single Module

- Incorrect module (Timer/ADC) settings
- Logic bugs
- ...

Innovative Debugging Techniques

Rubber Duck Debugging

Take a rubber duck and explain it your problem, in simple words! Take it through your code, line-by-line, and explain the code to the duck.

Summary

- Put constants into the flash (program) memory
- Use background tasks
- Analyze the static and dynamic memory usage of your programs
- Apply bitwise techniques
- Read avr-libc manual
- Testing is hard
- Keep your programs simple
- Do not just trust a metric thrown at you

Questions?