03: Stack, Interrupts

Microcontrollers

Stefan Huber
www.sthu.org

Dept. for Information Technologies and Digitalisation
FH Salzburg

Winter 2024

Stack

Usage of the stack

For function calls:[1]

```
// Paramters sometimes passed via stack
void f(int param) {
      // Local vars reside on stack
      int local;
      // Where to continue control flow,
      // i.e., the computation? Return
      // address is stored on the stack
      return;
}
```

To swap out registers:

```
...    // We use some register, say r7
g();   // Say we use r7 within g() as well
...    // The effect on r7 by g() has to be
       // neutralized, by caller or callee,
       // who temporarily swaps out r7 on
       // the stack
```

The stack data structure consists of the stack pointer only:

- It contains the address of the tip of the stack.
- The stack grows from higher to lower addresses and lives in the SRAM.
  - A `PUSH` instruction decrements the stack pointer by one, a `POP` increments by one.
  - A `CALL` decrements by two, a `RET` or `RETI` increments by two, because they push and pop the function's return address.

[1] Details vary between so-called calling convention, e.g., see https://en.wikipedia.org/wiki/X86_calling_conventions#List_of_x86_calling_conventions.

Stack frame

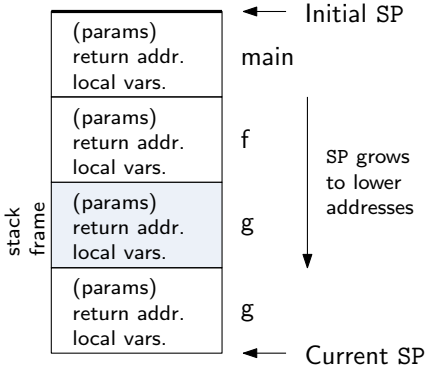
A stack frame is a portion of the stack "belonging" to a single call.

- It contains the return address and local variables.[2]
- Sometimes parameters are passed via the stack, sometimes via registers.

Warning

- Deep recursions cause invalid memory access by the stack.
- Buffer overflow security attacks alter the return address, e.g., by writing out-of-bounds in a local-variable array.

Function call sequence:
`main()` calls `f()` calls `g()` calls `g()` again



[1] It does not contain static variables of a function; they are stored globally.

Function call demo

Keywords:

- Stack pointer and stack dump
- Program counter
- Instructions `push`, `pop`, `call`, `ret`

Function call costs

Most expensive instructions: `CALL`, `RET`, `RETI` with 4 cycles.

- Access to stack requires data memory access, just like `PUSH` and `POP`.

CALL	k	Direct Subroutine Call	PC ← k	None	4
RET		Subroutine Return	PC ← Stack	None	4
RETI		Interrupt Return	PC ← Stack	I	4

Additional costs for function calls:

- Setup and clean up of stack frame
- Passing parameters and return value

Reducing function call costs in C

- Preprocessor macros
- Inline functions since C99

Inline functions

With C99 we can give the compiler a hint to inline a function [cppref-c-inline]:

- When calling a function, the function's body is placed inline instead of a function call.
- Inline functions are like macros, but with types.
- For small functions this is an optimization technique to save the function call costs.
- Inlining may lead to larger code size, i.e., the size grows linearly with the number of calls inlined.[3]

```
1 inline double sq(double x) {
2     return x * x;
3 }
4
5 inline double norm(double x, double y) {
6     return sqrt(sq(x) + sq(y));
7 }
```

- However, it is only a hint to the compiler, we cannot force it.

[3] Instruction cache effects may even revert the time savings gained from the savings of the function call costs. The compiler has cost models to decide whether inlining pays off.

Complete inline function demo

The previous code sample did not tell the whole story:

- We need to tell the compiler in which object file the machine code of the function shall land.
- We have a header file and an implementation file. The latter uses the `extern` keyword.

The header file `geom.h`:

```
inline double sq(double x) {
    return x * x;
}

inline double norm(double x, double y) {
    return sqrt(sq(x) + sq(y));
}
```

The implementation file `geom.c`:

```
#include "geom.h"

extern inline double sq(double x);
extern inline double norm(double x, double y);
```

Interrupts

Event handling and concurrency

Many use cases of microcontrollers are event-based:

- Act upon external triggers: drive made a half turn, object passed light barrier, button pressed, ...
- Performing periodic tasks when a timer event happened, e.g., closed-loop control algorithms
- Handling a byte received via a communication interface

Two approaches to act upon events:

Polling Cyclically testing in the main program whether an event happened and if so execute event handling routines.

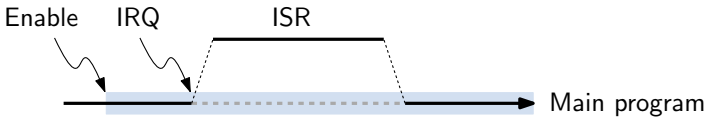
Interrupts Let the microcontroller detect when an event happened, *interrupt* the execution of the main program and execute a dedicated interrupt service routine (ISR).

Interrupts provide means for concurrency[4].

- Example: Motion control and serial communication at the same time.
- On a processor level, without a multi-tasking operating system.

[4] Dt. Nebenläufigkeit. Recall that *concurrency* and *parallel execution* have related but different meaning!

Execution overview



- When an Interrupt Request (IRQ) happens the corresponding Interrupt Service Routine[5] (ISR) is executed and interrupts the main program.
- Interrupts need to be enabled.

The interrupt controller orchestrates this mechanism.

[5] Also known as interrupt handler.

Interrupt controller

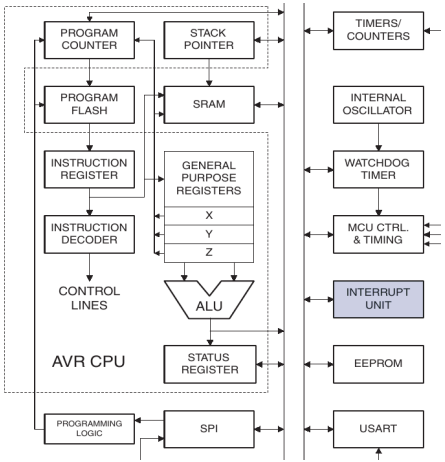


Figure: The interrupt controller of the ATmega32 takes care for the interrupt handling [ATmega32, p. 3, fig. 2].

ATmega32 interrupts

The ATmega32 provides a bunch of different interrupts:[6]

- Reset interrupt (power on reset, JTAG reset, watchdog reset, ...)
- External interrupts
- Timer interrupts
- Communication interface interrupts (SPI, USART, I^2C/TWI)
- Analog-Digital Converter, memory programming

6 ATmega32, p. 44.

Interrupt control

Enabling interrupts:

- Each interrupt has its dedicated interrupt enable (IE) bit.
- And there is Global Interrupt Enable (GIE) bit in the status register SREG.[7]
  - We sometimes disable all interrupts via GIE to implement *atomic sections*; after all, we deal with concurrency!

Both – GIE and the respective IE – must be set to enable the execution of the ISR.

[7] ATmega32, p. 10.

Interrupt control

Enabling interrupts:

- Each interrupt has its dedicated interrupt enable (IE) bit.
- And there is Global Interrupt Enable (GIE) bit in the status register SREG.[7]
  - We sometimes disable all interrupts via GIE to implement *atomic sections*; after all, we deal with concurrency!

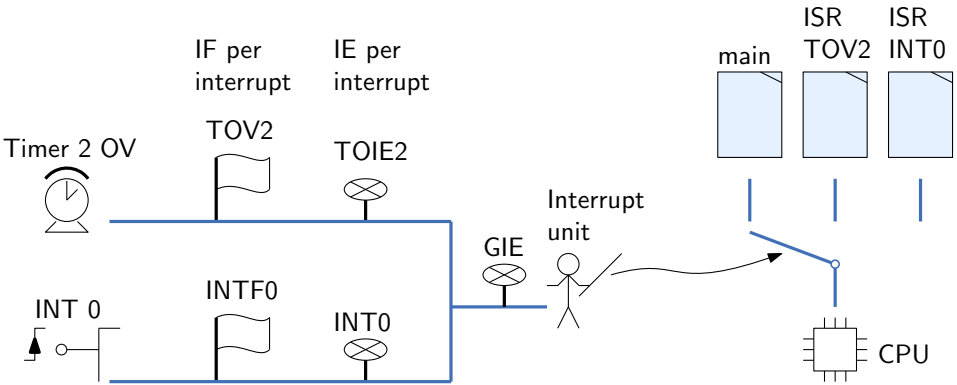
Both – GIE and the respective IE – must be set to enable the execution of the ISR.

If an interrupt request (IRQ) is raised the interrupt flag (IF) is set.

- If IF is set but the interrupt is disabled, the ISR is executed as soon as the interrupt is re-enabled.
- Hence, we do not miss interrupts if disabled, but only delay the ISR execution. Unless
  - the IF flag is cleared manually, or
  - more then one interrupt happens in that time. Keep the atomic sections short!

[7] ATmega32, p. 10.

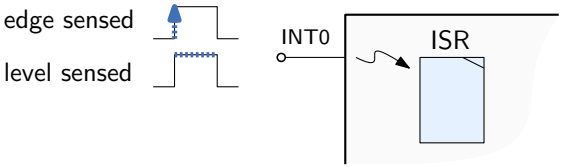
Interrupt control



External interrupts

When the external event happens (edge or level sensed[8]) then an IRQ is fired and the ISR is called.

- Example: React on pressed button, product crossing light barrier, fan making a half turn, ...



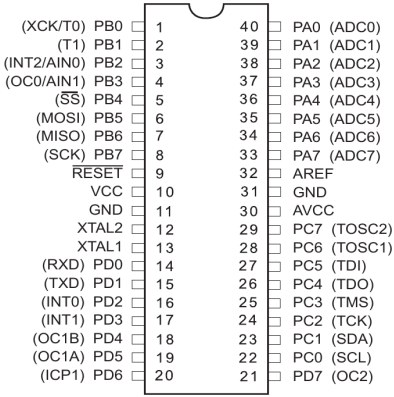


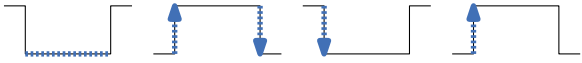
Figure: The ATmega32 has three external interrupts triggered by the pins INT0 to INT2.

8 Dt. Abtastung

External interrupt control

Trigger condition configured by Interrupt Sense Control bits in the MCUCR and MCUCSR registers[9][10]

7	6	5	4	3	2	1	0	
SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	



INT0 to INT2 is enabled by bits in the General Interrupt Control Register (GICR).The interrupt flags are found in the General Interrupt Flag Register (GIFR).

Further features:

- Can also be used for wake up from sleep modes.[11]
- Also works if pins are configured as output, which gives kind of software interrupts[12].

9 ATmega32, p. 66.

10 Note that "high level" is not supported. Hint: It would constantly trigger for high impedance at an input pin with pull-up enabled.

11 ATmega32, p. 34.

12 Software interrupts are used for system calls to an operating system, but for the ATmega32 there is hardly any use.

ISR with avr-gcc

```c
#include <avr/interrupt.h>

ISR (INT0_vect) {
    /* The ISR of external interrupt INT0. */
}

void init() {
    /* Make pin PD2 an input pin. */
    DDRD &= ~(1 << PD2);
    /* Enable external interrupt INT0. */
    GICR |= (1 << INT0);
    /* Set sense control for INT0 to any-edge. */
    MCUCR = (MCUCR & 0xfc) | 0x01;

    /* Set interrupts, i.e., set global interrupt enable bit. cli() is its
     * counterpart 'clear interrupts'. */
    sei();
}
```

Have a look at its objdump.

Spurious interrupts and noise cancellation

Interrupt is guaranteed if pulse holds longer than a clock period.

- Otherwise, it *may* raise an interrupt!

Noise on pins may trigger spurious external interrupts.

- Bouncing of mechanical switches or buttons cause them.
- Floating potentials are prone to noise.

Spurious interrupts are notorious error sources that are *very* hard to debug. Avoid them!

13 [ATmega32, p. 94]. Is only supported for the Input Capture pin ICP1.

Spurious interrupts and noise cancellation

Interrupt is guaranteed if pulse holds longer than a clock period.

- Otherwise, it *may* raise an interrupt!

Noise on pins may trigger spurious external interrupts.

- Bouncing of mechanical switches or buttons cause them.
- Floating potentials are prone to noise.

Spurious interrupts are notorious error sources that are *very* hard to debug. Avoid them!

Digital filter The ATmega32 can enable noise cancellation: it takes over an input level only if it stayed constant for four clocks periods.[13] If bouncing pulses are longer than four clock periods then debouncing can be done in software for longer periods.

Analog filter Add a capacitor to stabilize the potential (which gives a low-pass filter).

[13] [ATmega32, p. 94]. Is only supported for the Input Capture pin ICP1.

Spurious interrupts and noise cancellation

Interrupt is guaranteed if pulse holds longer than a clock period.

- Otherwise, it *may* raise an interrupt!

Noise on pins may trigger spurious external interrupts.

- Bouncing of mechanical switches or buttons cause them.
- Floating potentials are prone to noise.

Spurious interrupts are notorious error sources that are *very* hard to debug. Avoid them!

Digital filter The ATmega32 can enable noise cancellation: it takes over an input level only if it stayed constant for four clocks periods.[13] If bouncing pulses are longer than four clock periods then debouncing can be done in software for longer periods.

Analog filter Add a capacitor to stabilize the potential (which gives a low-pass filter). Enable the pull-up resistor against floating potentials.

[13] [ATmega32, p. 94]. Is only supported for the Input Capture pin ICP1.

Interrupt vector table

When an interrupt is raised the ISR is called.

- More precisely, the CPU jumps to the corresponding entry in the interrupt vector table.
- A table entry has four bytes, which accommodates a `JMP` instruction to the actual ISR.
- Other microconrollers often contain the address of the ISR instead of a `JMP` instruction.

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match

Figure: Interrupt vector table at [ATmega32, p. p44].
Program address is word address rather than byte address.

```
00000000 <__vectors>:
   0:   0c 94 2a 00    jmp 0x54
   4:   0c 94 36 00    jmp 0x6c
   8:   0c 94 34 00    jmp 0x68
   c:   0c 94 34 00    jmp 0x68
  10:   0c 94 34 00    jmp 0x68
  14:   0c 94 34 00    jmp 0x68
  18:   0c 94 34 00    jmp 0x68
  1c:   0c 94 34 00    jmp 0x68
  20:   0c 94 34 00    jmp 0x68
  24:   0c 94 34 00    jmp 0x68
[...]
```

Interrupt priorities

If more than one interrupt is raised at the same CPU cycle, we require a deterministic strategy to decide which one to handle first.

- Interrupt priorities define a precedence order.[14]
- For the ATmega32 the position in the interrupt vector table is the priority, from high to low. The RESET has the highest, INT0 the second highest, et cetera.

[14] ATmega32, p. 13.

Life cycle of an interrupt

1. One or more interrupt flags are set. The current machine instruction is finished.
2. The raised interrupt of highest priority is determined.
   Its interrupt flag is cleared, the `PC` is pushed onto the stack and the `JMP` instruction in the interrupt vector table is executed.
   The interrupt response time is 4 cycles (like a `call`).
3. The ISR is executed. Its last instruction is `RETI` (return from interrupt, 4 cycles).

15 ATmega32, p. 15.

Life cycle of an interrupt

1. One or more interrupt flags are set. The current machine instruction is finished.
2. The raised interrupt of highest priority is determined.
   Its interrupt flag is cleared, the `PC` is pushed onto the stack and the `JMP` instruction in the interrupt vector table is executed.
   The interrupt response time is 4 cycles (like a `call`).
3. The ISR is executed. Its last instruction is `RETI` (return from interrupt, 4 cycles).

The interrupt latency is the time from the interrupt event occurring until the ISR being executed.

- It is at least 4 cycles. However, we have to add the current machine instruction (up to 4 cycles) and maybe also the wake up time from a sleep mode.

Before the next interrupt is served, at least one cycle of the main program is executed.[15]

- The main program does not starve, it is "only" slowed done.

[15] ATmega32, p. 15.

Interrupting interrupts

When an interrupt occurs at the ATmega32 then GIE bit is cleared.[16]

- Hence, (by default) an ISR cannot be interrupted by other interrupts. It forms an atomic section. As an general advice, keep ISRs short.
- The `RETI` instruction sets the GIE bit again.

However, one can set the GIE bit manually within the ISR and therefore enable nested interrupts.

- For the ATmega32, all interrupts can interrupt the current ISR.
- Other microcontoller use priorities or other means to control which interrupt can interrupt which.
- Beware that nested interrupts can be hard to debug!

```
1 ISR (INT0_vect, ISR_NOBLOCK) {
2     /* ISR_NOBLOCK says that the sei is called at the beginning to enable
3      * nested interrupts. */
```

16 ATmega32, p. 14.

Atomic sections

Concurrency requires locking.

- For microcontrollers, we typically simply turn off interrupts to form atomic sections.
- However, just calling `cli()`, `sei()` can be problematic:

```
void f() {
  /* GIE may or may not be set. */
  cli();
      /* Here comes the critical section... */
  sei();
  /* Now GIE is definetly set: perhaps a BAD SIDE EFFECT! */
}
```

A better method is to restore GIE, which is conveniently done by `ATOMIC_BLOCK`:[17]

```
void f() {
  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
      /* GIE is restored after leaving this block. */
  }
}
```

[17] See [AVR-libc-atomicblock] for details.

Design aspects of ISRs

Polling versus interrupts:

- Pro polling: Weak timing constraints, level sensing, long pulses, noisy signal
- Pro interrupts: Concurrent logic in main program, precise timing, infrequent events, edge sensing, no bouncing effects or spikes, main program could go to sleep mode

Logic in ISR vs. logic in main program:

- If the interrupt logic is small it can be done directly in the ISR.
- If the interrupt logic is heavy and not time critical then it shall be done in the main program. The ISR only registers the event, which is later handled in the main program. (Linux calls this upper and bottom half.)

Number of ISRs:

- Reducing the number of interrupts is generally a good thing. Sometimes an ISR can be spared and the work moved into the ISR of another interrupt.
- Example: We have a timer interrupt to cyclically display the value obtained from an ADC every few milliseconds. We do not need to setup a dedicated interrupt for the ADC completion but could simply trigger a new conversion in the timer ISR. (The ADC conversion is faster anyhow.)

References I

[ATmega32] *ATmega32: 8-bit AVR Microcontroller with 32KBytes In-System Programmable Flash*. Atmel Corporation. Feb. 2011.

[AVR-GCC-wiki] *AVR GCC*. URL: https://gcc.gnu.org/wiki/avr-gcc.

[AVR-libc-atomicblock] *AVR-libc: Atomic block*. URL: https://www.nongnu.org/avr-libc/user-manual/group__util__atomic.html.

[cppref-c-inline] *cppreference.com: inline function specifier*. URL: https://en.cppreference.com/w/c/language/inline.

[Dij] Edsger W. Dijkstra. *EWD1243a: The next fifty years*. URL: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1243a.html.

Historical note on interrupts

Dijkstra says in [Dij]:

> *In this connection the history of the real-time interrupt is illuminating. This was an invention from the second half of the 50s, [...]. Its advantage was that it enabled the implementation of rapid reaction to changed external circumstances without paying the price of a lot of processor time lost in unproductive waiting. The disadvantage was that the operating system had to ensure correct execution of the various computations despite the unpredictability of the moments at which the interrupts would take place[...]; the nondeterminism implied by this unpredictability has caused endless headaches for those operating system designers that did not know how to cope with it. We have seen two reactions to the challenge of this added complexity.*
> *[...]*
> *The difference was striking, showing once more that debugging is no alternative for intellectual control.*
> *[...]*
> *The moral is clear: prevention is better than cure, in particular if the illness is unmastered complexity, for which no cure exists.*

ISR: Saving registers

The ATmega32 does not save any registers.

- This includes the SREG register, which contains flags of arithmetic operations.
- However, the ISR should not leave "ghost-like" side effects to the main program!
- This is why the SREG register is manually saved and restored in the ISR. Likewise for the other registers used within the ISR. The C compiler does that for us.
- Some processors automatically save registers, in particular CISC machines with only few registers.

Analogous situation for ordinary function calls. Two approaches:

Caller saving The one calling a function is saving registers. The caller saves only the registers it uses and becomes immune to side effects. Does not work for ISRs.

Callee saving The one that is called is saving registers. The callee saves only those registers it uses and leaves no side effects.

The avr-gcc Application Binary Interface (ABI) defines register usage, calling conventions, and the like.[18]

[18] See [AVR-GCC-wiki]. For instance, avr-gcc does not support double-precision floating-point numbers.