System Programming

Prof. Dr. Antônio Augusto Fröhlich
guto@lisha.ufsc.br
http://www.lisha.ufsc.br/~guto

Nov 2006

Outline

- Interweaving system programming languages
  - Assembly, C, C++
- The stack
  - Function call conventions
- Run-time environments
- System calls
- I/O
- Interrupt handling

Programming: System X Application

- Application
  - User interface
  - High-level prog. lang.
    - High-level of abstraction
    - High productivity
    - Complex runtime
      - Automatic memory management
      - Active objects
      - Component repositories
  - World of JAVA, PHP, Python
- System
  - Hardware abstraction
  - Low-level prog. lang.
    - Resource-constrained environments
    - Little runtime overhead
    - Small runtime library
    - Direct access to hardware
  - World of C, C++

System Programming

- Paradigms
  - Structured
  - Object-oriented
- Languages
  - Assembly, C, C++
  - Ada? Eifel? JAVA?
- Tools
  - Preprocessors
  - Assemblers
  - Compilers
  - Linkers
  - Debuggers

Preprocessors

- Preprocessors do mostly simple textual substitution of program fragments
  - Unaware of programming language syntax and semantics
- CPP: the C Preprocessor
  - Directives are indicated by lines starting with `#`
  - Directives to
    - Include other files (`#include`)
    - Define macros and symbolic constant (`#define`)
    - Conditionally compile program fragments (`#ifdef`)

The C Programming Language

- Designed by Ritchie at Bell Labs in the 70's
  - As a system programming language for UNIX
  - Industry standard (ANSI C)
- The “portable assembly language”
  - Allows for low-level access to the hardware mostly like assembly does
  - Can be easily compiled for different architectures
- The “high-level programing language”
  - Compared to programming languages of its time
  - No longer suitable for most application development

Mixing C and Assembly (GCC)

- Why to embed assembly in a C program?
  - To gain low-level access to the machine in order to provide a hardware interface for high-level software constructs
- When the compiler encounters assembly fragment in the input program, it simply copies them directly to the output

```c
int main() {
  asm("nop");
  return 0;
}
```

compiling →

gcc -S

```
...
main:
    ...
    nop
    ...
```

Example of C with inline Assembly

- IA-32 context switch

```
void IA32::switch_context(Context * volatile *
  o, Context * volatile n)
{
    ASM("    pushfl                          \n"
        "    pushal                          \n"
        "    movl    40(%esp), %eax   # old \n"
        "    movl    %esp, (%eax)           \n"
        "    movl    44(%esp), %esp   # new \n"
        "    popal                           \n"
        "    popfl                           \n");
}
```

GCC Extended Assembly

- `asm` statements with operands that are C expressions
- Basic format

```
asm("assembler template"
    : output operands                /* optional */
    : input operands                 /* optional */
    : list of clobbered registers /* optional */
);
```

GCC Extended Assembly

- Assembler template
  - The set of assembly instructions that will be inserted in the C program
  - Operands corresponding to C expressions are represented by "`%n`" in the `asm` statement, with "n" being the order in which they appear in the statement
  - Example (IA-32)

```
int a = 10, b;
asm("movl %1, %0;"
    :"=r"(b) /* output operands */
    :"r"(a)  /* input operands */
    :        /* clobbered register */ );
```

GCC Extended Assembly

- Operands
  - Preceded by a constraint

    `r` operand must be in a general purpose register

    `m` operand must be in memory (any addressing mode)

    `o` operand must be in memory, address must be offsetable

    `i` operand must be an immediate (integer constant)

    `...` many others, including architecture-specific ones
  - Input operand constraints
    - Are met before issuing the instructions in the `asm` statement
  - Output operand constraints (begin with "`=`")
    - Are met after issuing the instructions in the `asm` statement

GCC Extended Assembly

- Clobber list
  - Some instructions can clobber (overwrite) registers and memory locations
  - By listing them, we inform the compiler that they will be modified and their original values should no longer be trusted
  - Example (IA-32)

```
int a = 10, b;
asm("movl %1, %%eax; movl %%eax, %0;"
    :"=r"(b) /* output operands */
    :"r"(a)     /* input operands */
    :"%eax"     /* clobbered register */ );
```

GCC Extended Assembly

- Volatile assembly
  - When the assembly statement must be inserted exactly where it was placed
  - When a memory region accessed by the assembly statement was not listed in the input or output operands
  - Example (IA-32)

```
int a=10;
asm __volatile__ ("movl %0, 0xfefa;"
        :              /* output operands */
        :"r"(a)        /* input operands */
        :              /* clobbered register */ );
```

The C++ Programming Language

- Designed by Stroustrup at Bell Labs in the 80's
  - As a multiparadigm programming language
  - Superset of C (a C program a valid C++ program)
  - Strongly typed
  - Supports object-oriented programming (classes , inheritance, polymorphism, etc)
  - Supports generative programming techniques (generic programming, static metaprogramming, etc)

The C++ Programming Language

- System software != applicative software
  - Rational use of late binding (polymorphism, dynamic casts, etc)
  - Extended use of static metaprogramming
  - Always take a look at the assembly produced

Mixing C++ and C

- C++ and C use different linkage and symbol generation conventions
  - C++ does name mangling
    - Symbols corresponding to member functions embed parameter types
- In order to call C functions from C++

`extern "C" { /* C function prototypes */ }`

- In order to call C++ functions from C
  - one has to know the mangled function names

Linking

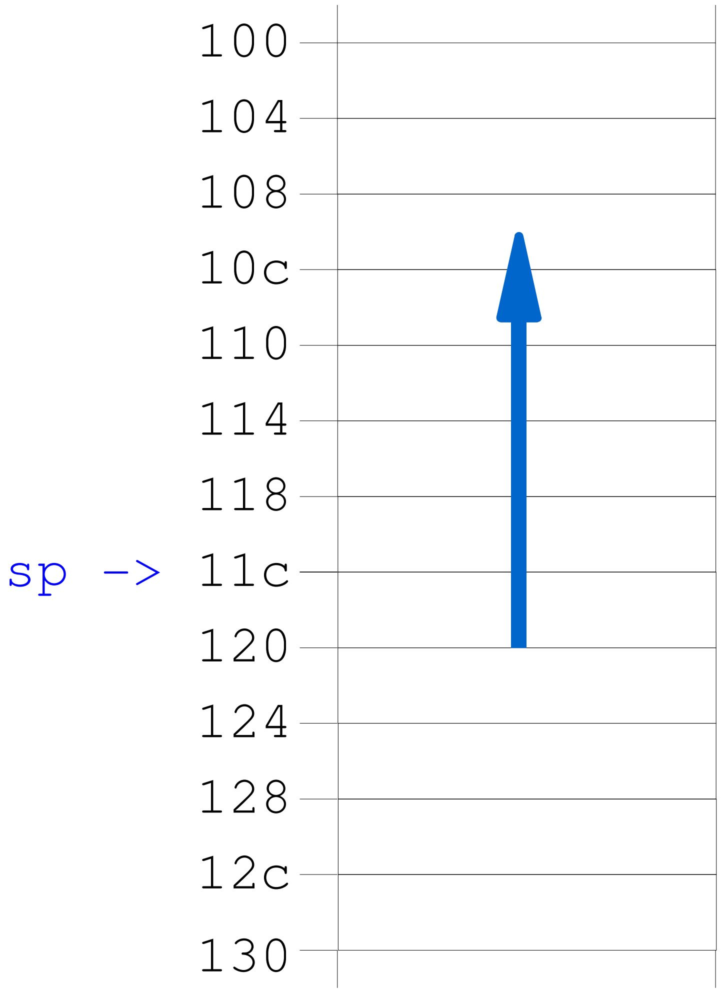
- Linkage
  - The process of collecting relocatable object files into a executable
- Styles of linking
  - Static linking, dynamic linking, runtime linking
- Linker scripts

```
SECTIONS {
.text 0x8000:    {
      *(.text)
      *(.rodata)
      *(.strings)
}   > ram
```

```
.data : {
      *(.data)
      *(.tiny)
}   > ram
}
```

The Stack

- Stores information about the active subroutines of a computer program
  - Return address
  - Local variables
  - Arguments (param. values)
  - Temporary storage
  - Object pointer (this)
- Stack frame
  - Every called subroutine
  - Frame pointer



Function Calling Convention

- Standardized method for a program to pass parameters to a function and receive a result value back from it
  - Parameters (registers, stack or a combination of both)
  - Function naming (mangling)
  - Setup/cleanup of stack frame (caller/callee)
- Standards
  - cdecl, stdcall, fastcall, iABI

Convention: cdecl

- Used by many C and C++ compilers for IA-32
- Parameters passed on the stack in right-to-left order
- Return value in EAX register
- Registers EAX, ECX, and EDX available for use in the function

Convention: cdecl

- C Program

```
int f1(int p1, int p2)
{
  return p1 + p2;
}

int main()
{
  int v1 = 1, v2 = 2,
 v3;
  v3 = f1(v1, v2);
  return v3;
}
```

- ASM [v3 = f1(v1, v2)]

; push arguments

```
pushl v2
pushl v1
```

; call function

```
call f1
```

; clear stack

```
addl $8, %esp
```

; move return value

```
movl %eax, v3
```

Convention: cdecl

- Function Prologue

```
; save old frame pointer
pushl %ebp

; set new frame pointer
movl %esp, %ebp

; allocate local variables
sub $n, %esp
```

- Function Epilogue

```
; deallocate local variables
movl %ebp, %esp

; restore old frame pointer
pop %ebp

; return to caller
ret
```

Convention: cdecl

```
f1:

; prologue
pushl %ebp
movl  %esp, %ebp

; EAX <- p2
movl  12(%ebp), %eax

; EAX <- EAX + p1
addl  8(%ebp), %eax ; p1

; epilogue
movl  %ebp, %esp
popl  %ebp
ret
```

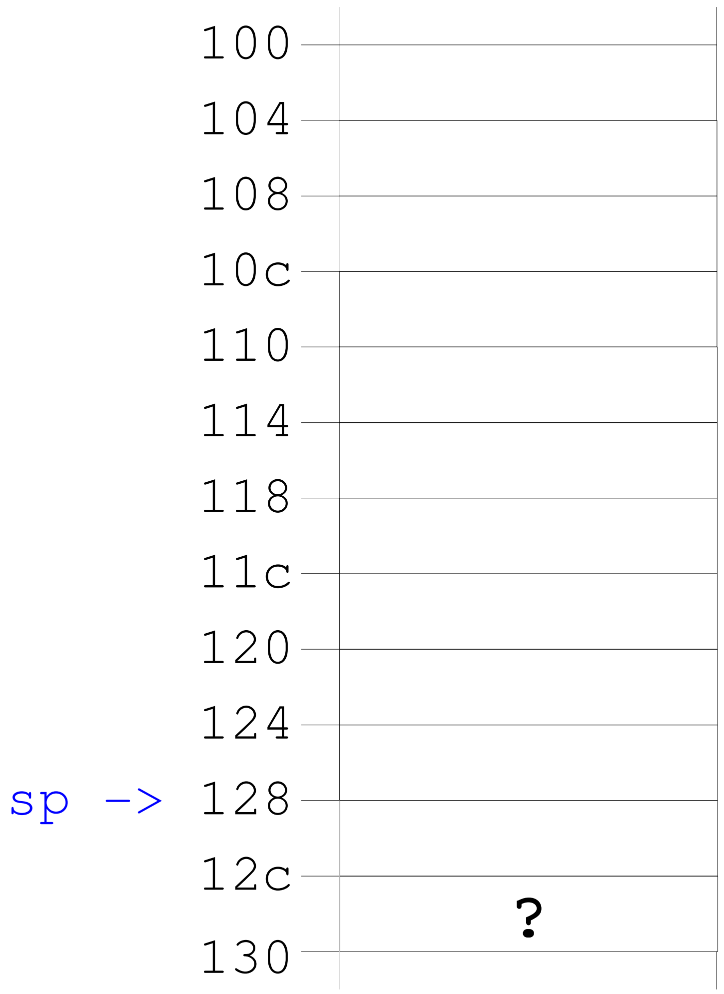
Convention: cdecl

```
main:

; prologue (3 x sizeof (int) )
pushl %ebp
movl  %esp, %ebp
subl  $12, %esp

; v1 <- 1
movl  $1, -12(%ebp)

; v2 <- 2
movl  $2, -8(%ebp)
```

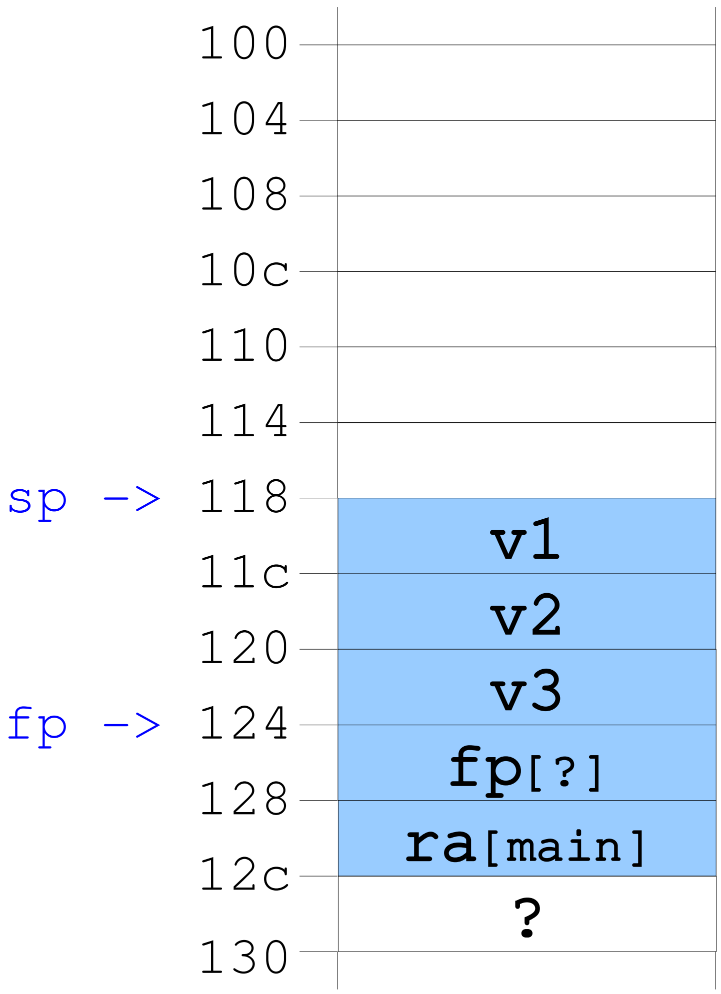
```
; call f1(v1, v2)
pushl -8(%ebp)        ; v2
pushl -12(%ebp)       ; v1
call  f1
addl  $8, %esp

; v3 <- EAX
movl  %eax, -4(%ebp)

; epilogue
movl  %ebp, %esp
popl  %ebp
ret
```

Convention: cdecl

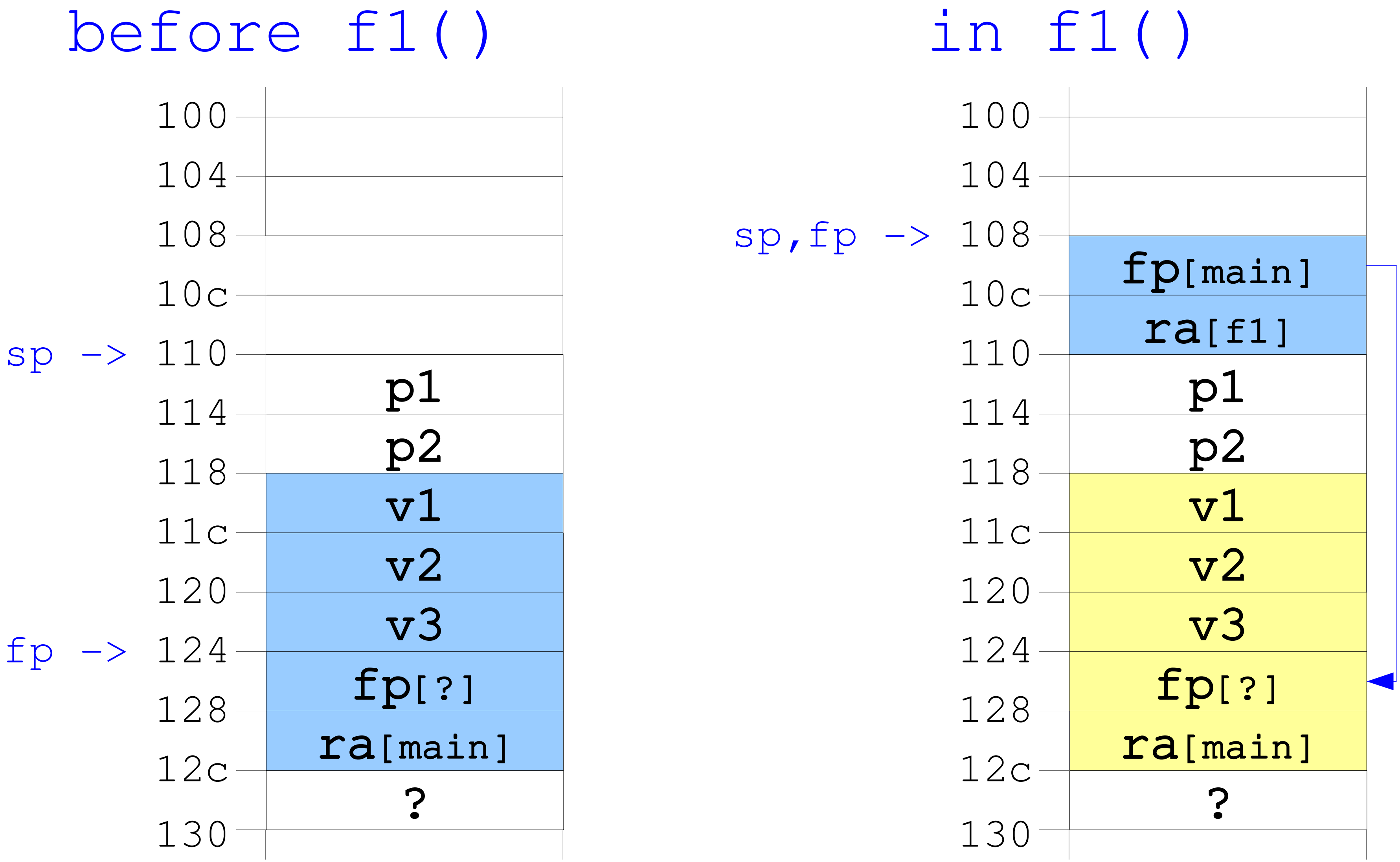
before main()



in main()



Convention: cdecl



Run-Time Environment

- Supports the execution of programs
  - Process initialization
  - Interaction with operating systems
    - Memory, I/O, etc
  - Exception handling
- Usually implemented by a combination of operating system + virtual machine + run-time libraries
  - C -> Unix + libc
  - JAVA -> ??? + JVM + ClassPath

C Standard Library: libc

- Typical functionality
  - Error handling
  - String manipulation
  - I/O
  - Time/date
  - Streams manipulations (file, buffers, etc)
- Implementation
  - Header files + library (object archive)
  - Influenced many other systems
- ANSI standard (C89/C99)

C++ Standard Library: libstdc++

- Organization
  - libc + STL
- Standard Template Library (STL)
  - Generic programming (compile-time)
  - Typical functionality
    - Containers
    - Iterators
    - Algorithms

JAVA Run-Time Environment

- Organization
  - JAVA Virtual Machine + class libraries (API, classpath)
  - Often implemented as a middleware
- Typical functionality
  - libstdc++ (I/O, streams, containers, etc)
  - Advances memory management (garbage collector)
  - Dynamic loading
  - Reification
  - Etc

Statically Linked Libraries

- Set of pre-compiled routines
  - Archive of object files
  - Symbol tables used for function location
- Copied into the target executable by the linker
  - Undefined symbols in the executable are searched in the library, causing the containing objects to be linked
- Resulting executable is standalone (though usually large)

Dynamically Linked Libraries

- Linker collects information about needed libraries and resolves symbols to a call table
- Loader loads needed libraries into the address space of process
  - Load-time: all objects are linked to the program as it is loaded
  - Run-time: objects are loaded on demand
- and updates the call tables
- Indirect calls through call tables

Shared Libraries

- Save memory avoiding duplicates
  - Dynamically linked libraries can be shared among several processes (programs)
- Implementation
  - Position independent code (only relative addresses)
  - Procedure linkage table stubs
  - Global offset tables
  - MS Windows DLLs
    - Pre-mapping instead of position independent
    - Seldom shareable (due to mapping conflicts)

Input/Output

- "The world outside the CPU"
  - I/O ports (GIOP, SPI, ...)
  - Buses (I2C, CAN, ...)
  - Peripheral devices
    - UART
    - Timers
    - EEPROM
    - Sensors
    - Actuators
    - ...

I/O Ports

- Connect external devices
  - Sensors, actuators, UART, ...
- Parallel (e.g. GPIO) or serial (e.g. SPI)
- Input, output or bidirectional
- Synchronous (clock) or asynchronous (strobe)
- Some can directly drive leds, buttons and TTL-style circuitry

I/O Mapping

- Register mapped
  - CPU registers directly map I/O ports
- Memory mapped
  - I/O ports and device registers are mapped in the processor's memory address space
  - chip select <- address decoder
- I/O mapped
  - I/O ports and device registers are mapped in a separate address space
  - chip select <- address decoder + I/O line

I/O Operation

- Polling x Interrupt
  - Polling
    - Processor continuously probes an I/O device's status register
  - Interrupt
    - I/O device notifies the processor when its status changes
- PIO x DMA
  - Programmed I/O
    - I/O device <=> CPU <=> memory
  - Direct Memory Access
    - I/O device <=> memory

Polling

- Sequential interrogation of devices for various purposes
  - Operational status, readiness to send or receive data, ...
- Processor continuously probes an I/O device's status register
  - Implies in busy waiting

Interrupts

- I/O device notifies the processor when its status changes by means of asynchronous signals named Interrupt Requests (IRQ)
- An interrupt request causes the processor to interrupt program execution and switch to an Interrupt Service Routine (ISR)
- Interrupts can usually be remapped and masked

Interrupts

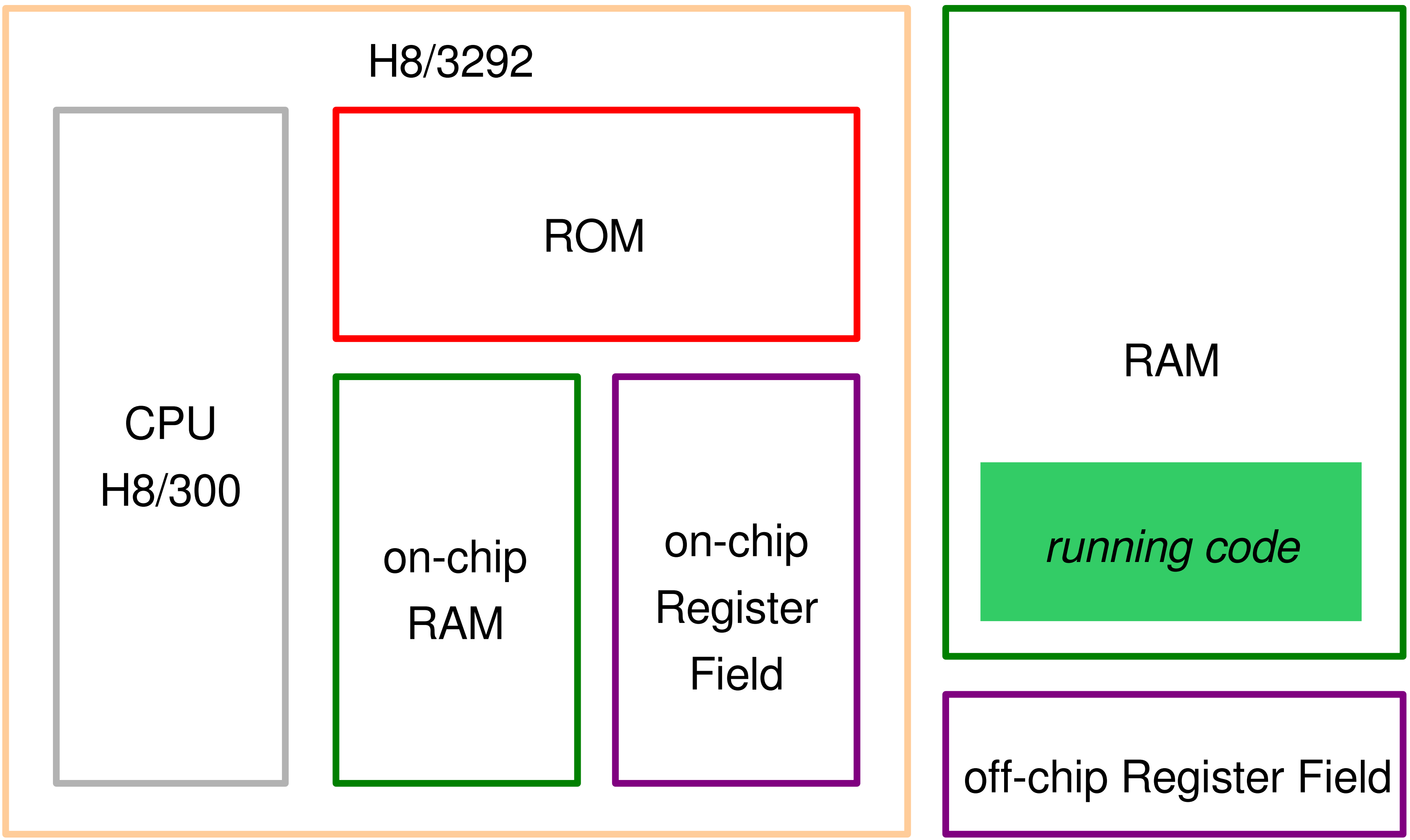
- Interrupt Vector
  - An array of pointers, indirect pointers or single instructions that leads to the ISRs
  - May reside in ROM (predefined) or in RAM (programmable)
- Interrupts triggered by external devices such as timers and sensors are known simply as interrupts
- Interrupts triggered by internal events such as arithmetic exceptions are know as exceptions

Case Study: Lego RCX (H8/3292)

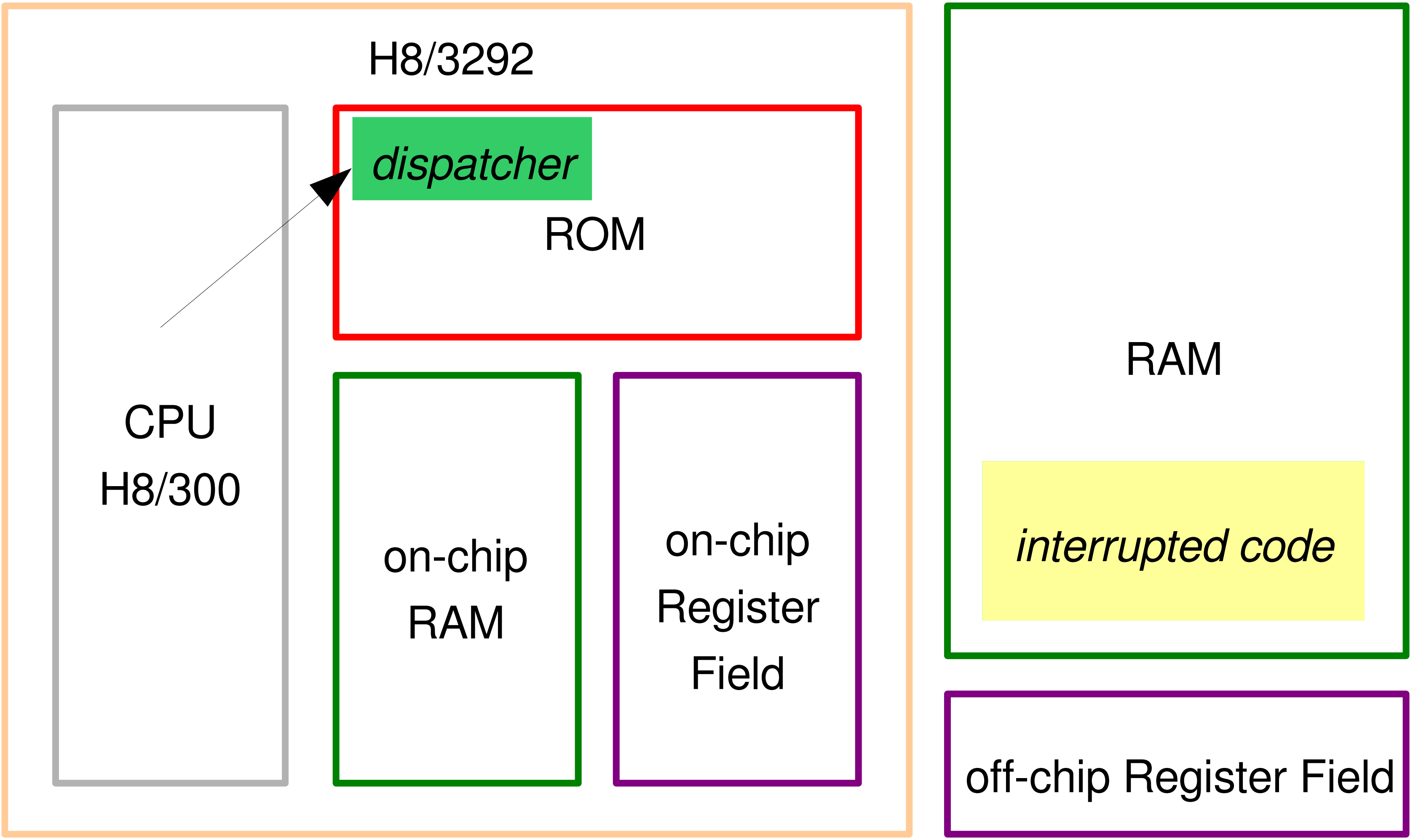
- H8/3292 interrupt table
  - Stored at `0x0000-0x0049` (ROM)
  - Redirected to a RAM interrupt table
  - Decreasing priority
- On-chip RAM interrupt table
  - Stored at `0xfd80-0xfdbf`
  - Pointers to user-defined handlers
- Masking
  - Globally (except NMI) CCR bit 7
  - Individually through the off-chip register field

Vector	Source
0	reset
1 - 2	reserved
3	NMI
4 - 6	IRQs
7 - 11	reserved
12 - 18	16-bit timer
19 - 21	8-bit timer 0
22 - 24	8-bit timer 1
25 - 26	reserved
27 - 30	serial
31 - 34	reserved
35	ADI
36	WOVF

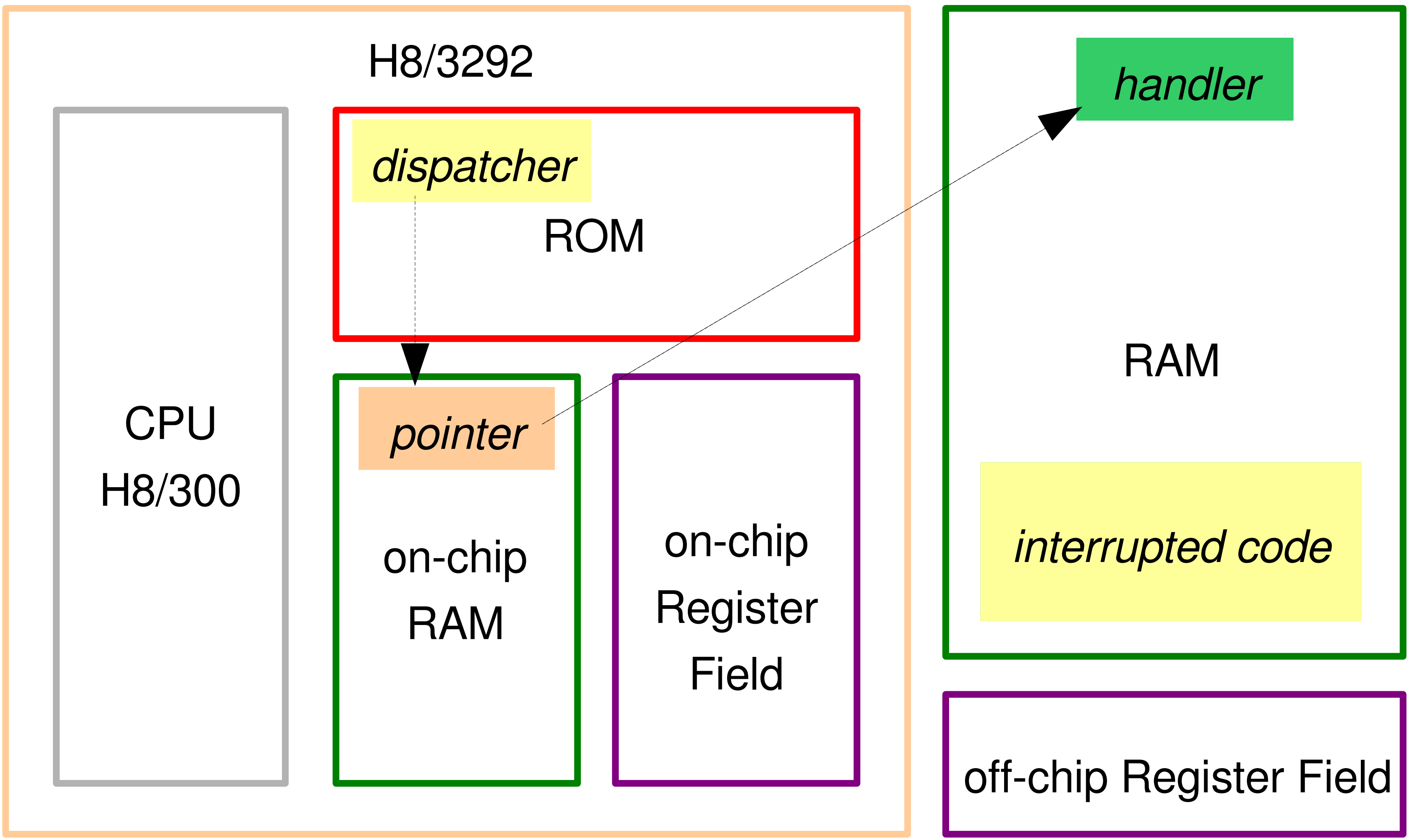
Interrupt Dispatching



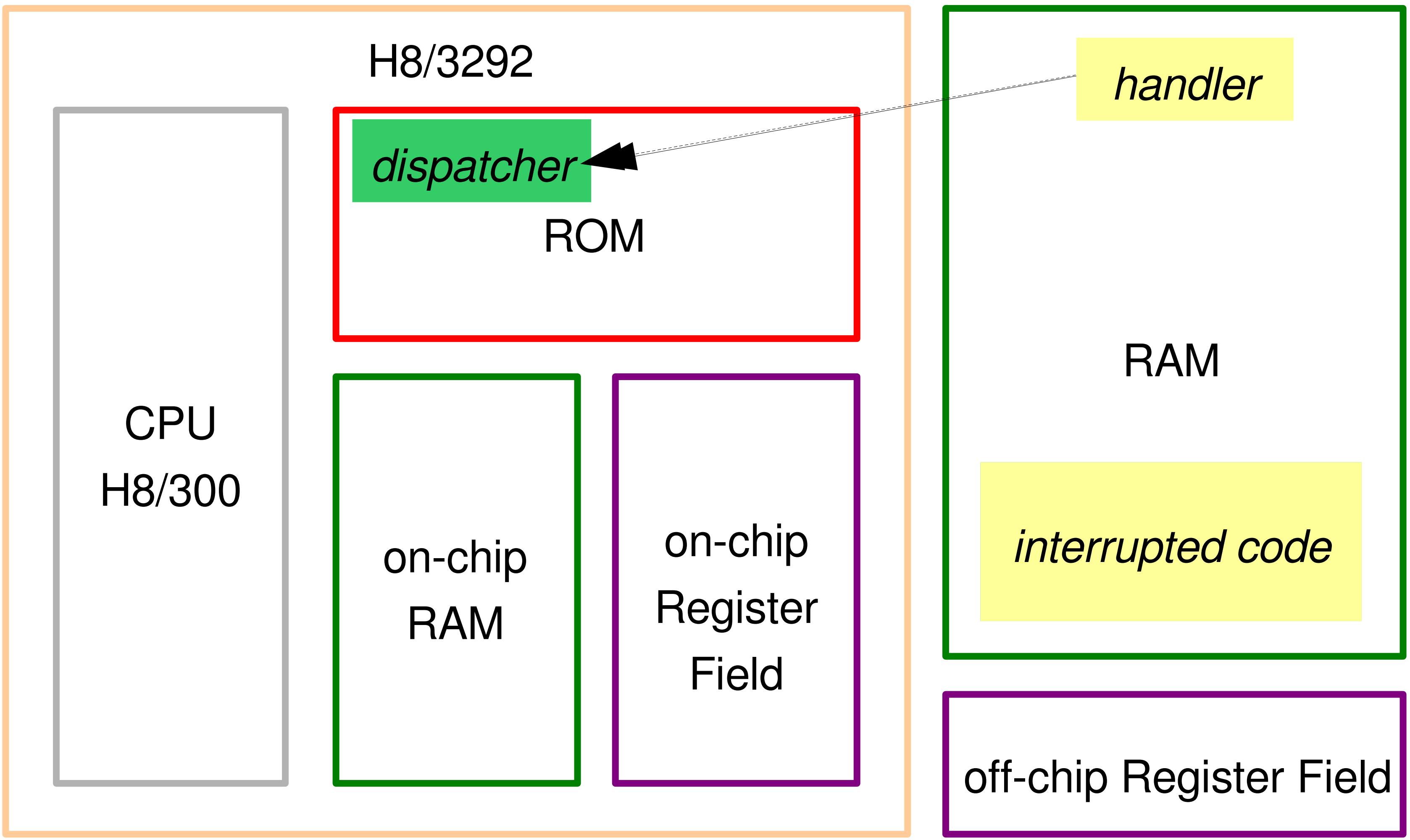
Interrupt Dispatching



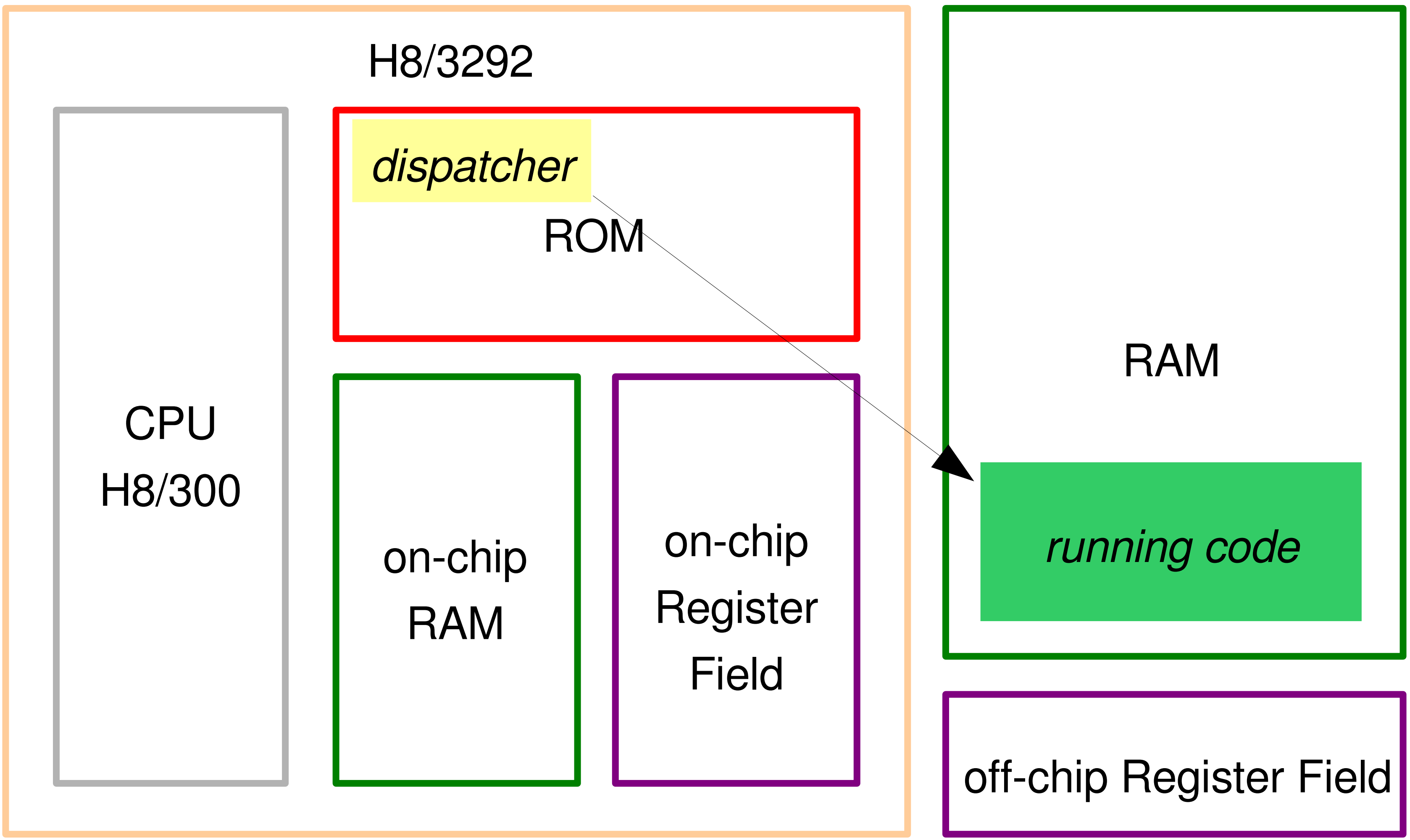
Interrupt Dispatching



Interrupt Dispatching



Interrupt Dispatching



LEGO RCX Interrupt Handling

- H8 dispatching

```
push pc
push ccr
ccr[7]=1 /* int disable */
RCX_Int_Table[n] =
   &rcx_handler;
H8_Int_Table[n]();
pop ccr
pop pc
```

- H8/300 Handler (ROM)

```
void h8_handler(void) {
  push r6
  mov  RCX_Int_Table[n], r6
  jsr  @r6
  pop  r6
  rte
};
```

- RCX Handler

```
void rcx_handler(void) {
  /* push registers */
  /* handle interrupt */
  /* pop registers */
};
```

- RCX Interrupt table

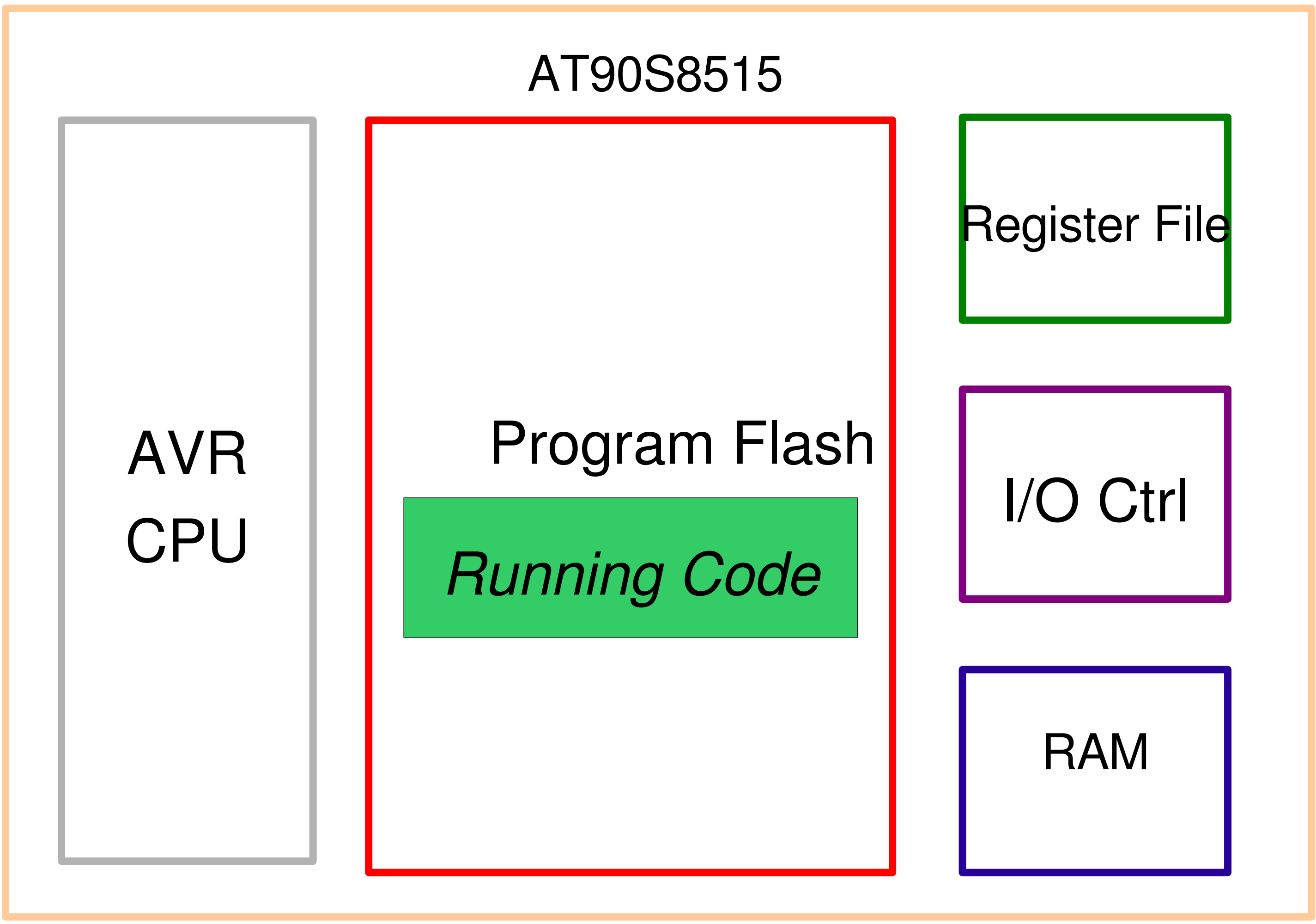
```
typedef void (RCX_Handler)(void);
RCX_Handler ** RCX_Int_Table = (RCX_Handler **)0xfd80;
```

Case Study: AVR

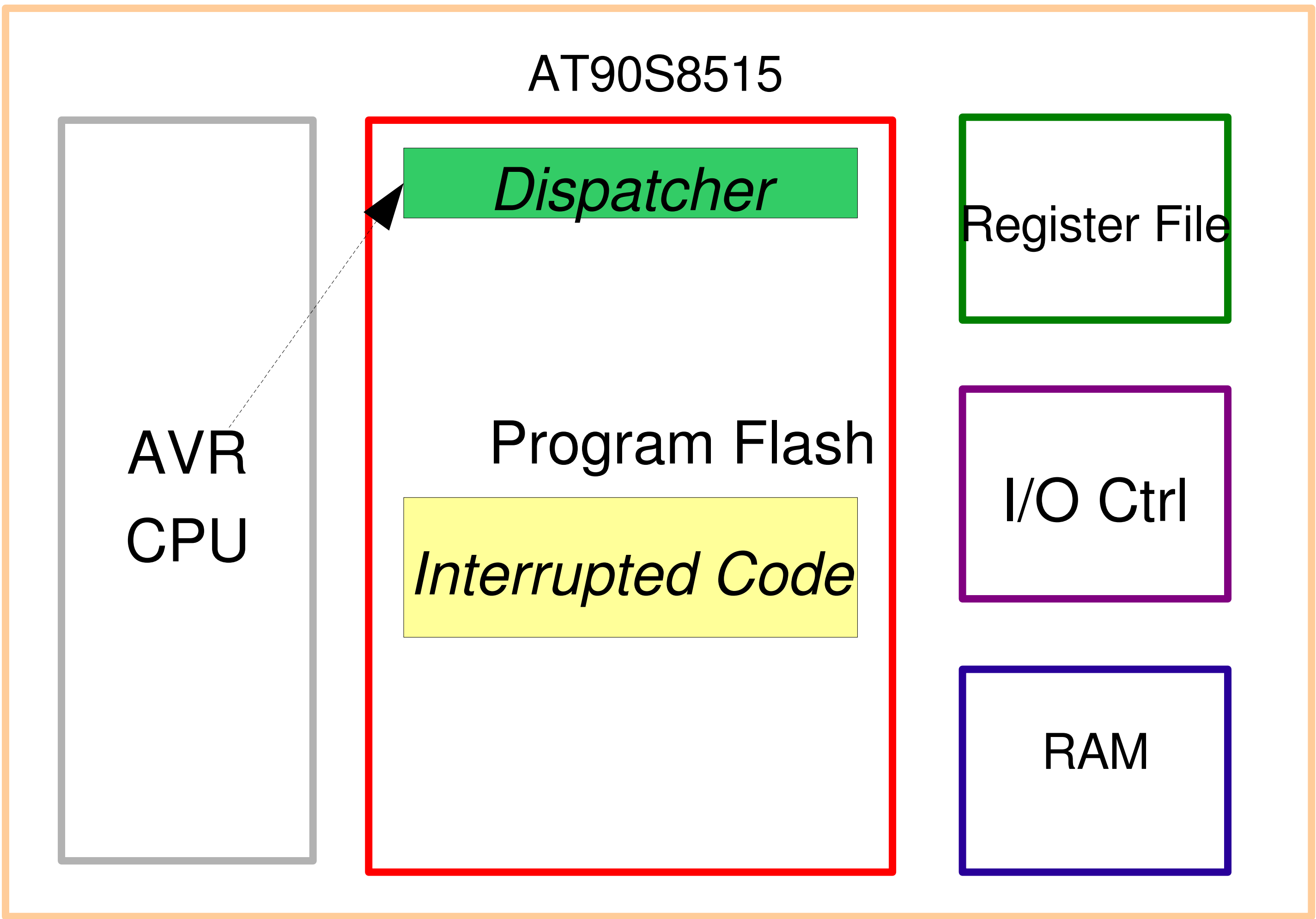
- AT90S interrupt vector
  - Stored in the first 14 words of program memory
  - Usually `rjumps` to ISRs
  - Decreasing priority
- Masking
  - I bit in SREG (Global Interrupt Enable)
  - Specific bits of device's control registers
    - GIMSK (IRQ0 and IRQ1)
    - TIMSK (Timers)
    - UCR (UART)
    - SPCR (SPI)

Vector	Source
0	Reset
1	IRQ0 (external)
2	IRQ1 (external)
3..7	timer1 events
8	timer0 overflow
9	SPI Tx complete
10..12	UART events
13	Analog comparator

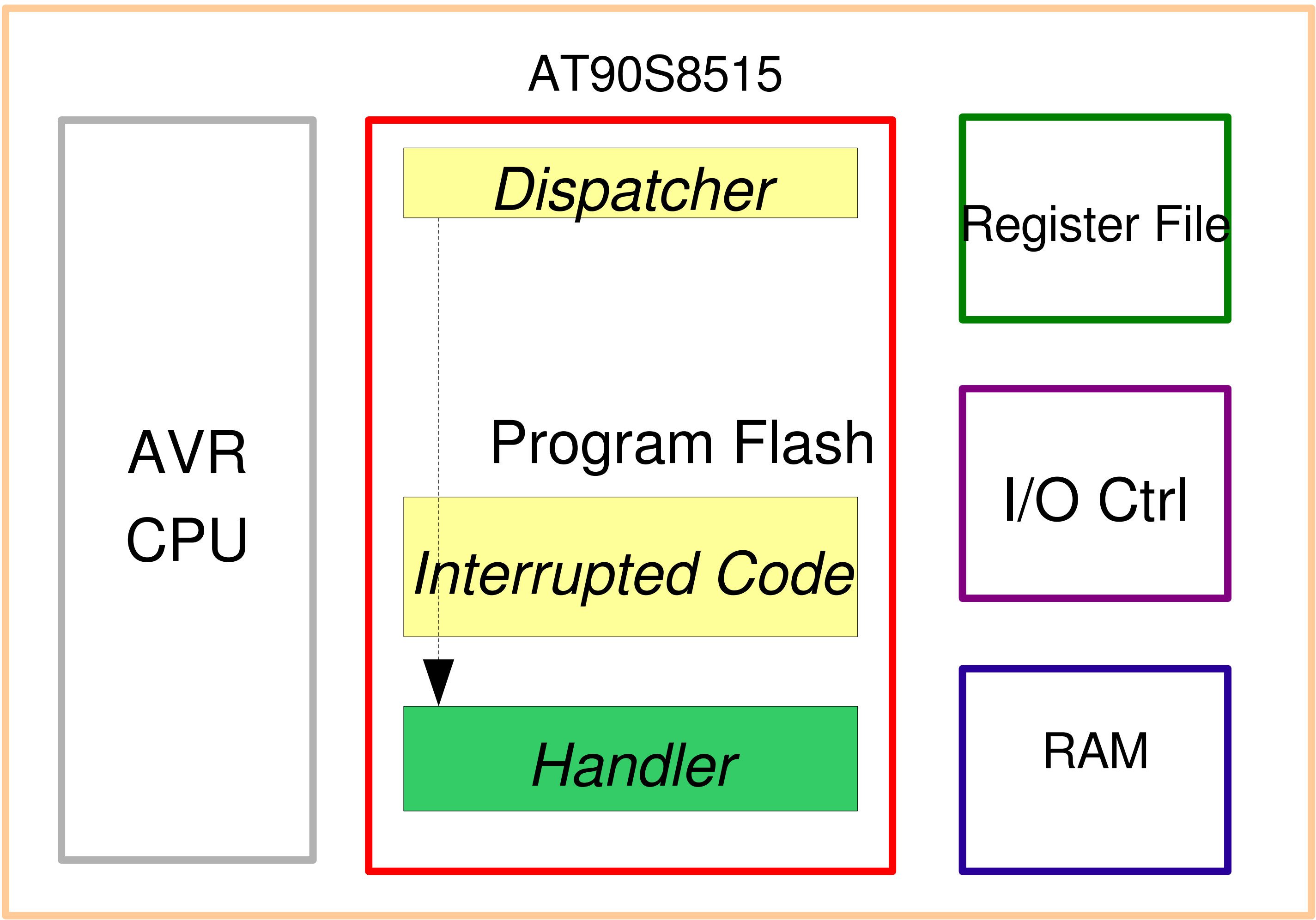
Interrupt Dispatching



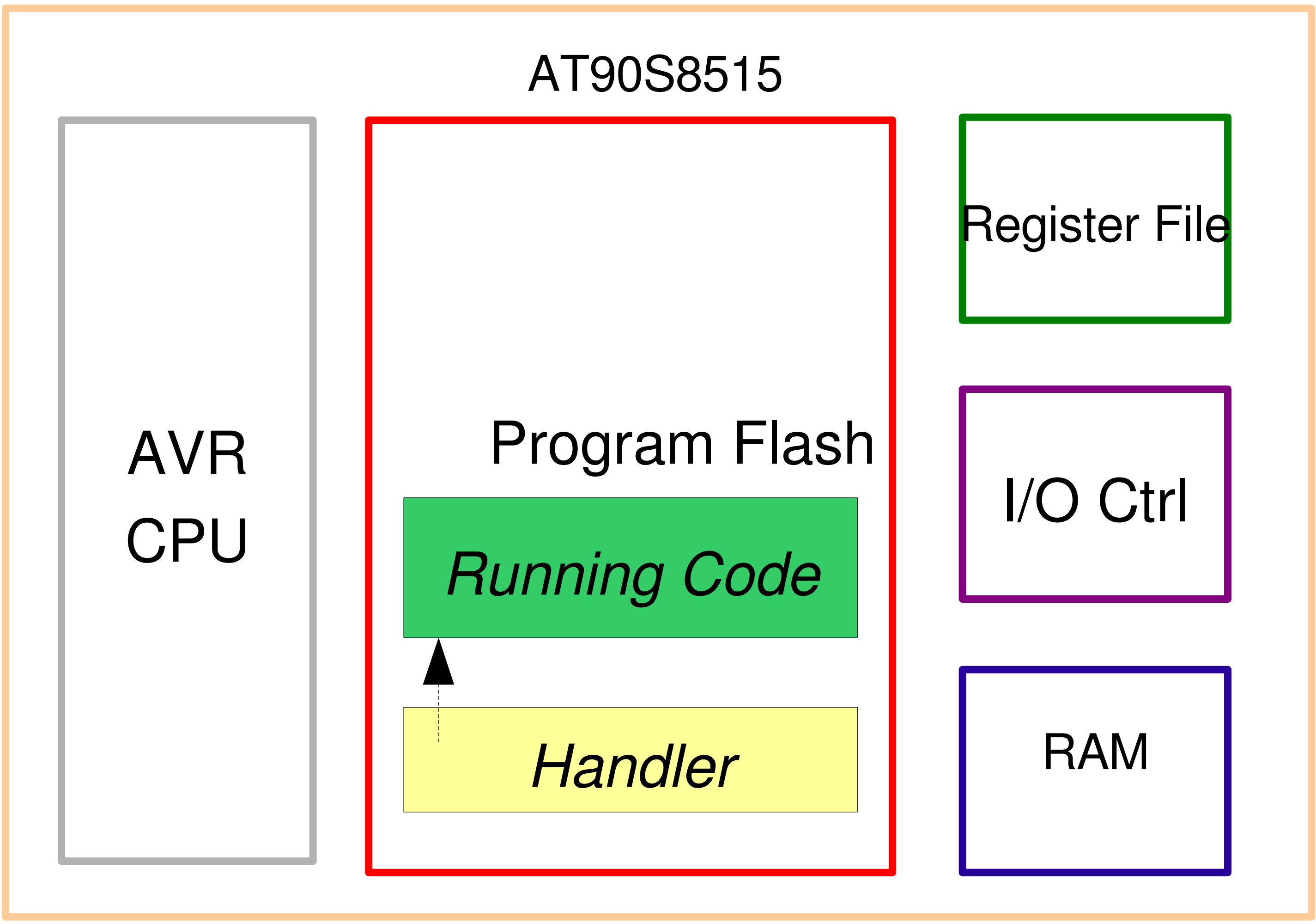
Interrupt Dispatching



Interrupt Dispatching



Interrupt Dispatching



Case Study: AVR

- After an interrupt is triggered, interrupts are globally disabled
  - Subsequent interrupt requests are flagged and executed in order of priority after the first ISR returns
  - The `reti` instruction reenables interrupts
  - Users may use cascading interrupts by reenabling interrupts in the ISR
  - External Interrupts (IRQs) are only flagged as long as the IRQ signal is active

Case Study: AVR (handler)

```
interrupts:
    rjmp reset      ; reset
    reti            ;
    reti            ;
    reti            ;
    reti            ;
    reti            ;
    reti            ;
    reti            ;
    rjmp timer      ; timer 0 overflow
    reti            ;
    reti            ;
    reti            ;
    reti            ;
    reti            ;
```

```
reset:
    ldi  r20, 0x02  ; reset handler
    out  0x3e, r20
    ldi  r20, 0x5f
    out  0x3d, r20
    sei
main:
    rjmp main       ; application
timer:
    inc r0          ; timer overflow
                      handler
    reti
```

Case Study: AVR (handler programming)

```
#define IRQ0     __vector_1
#define SIGNAL   __attribute__ ((signal))

int main (){
  while(1);
  return 0;
}

void IRQ0(void) SIGNAL;

void IRQ0(void)
{
  PORTB = ~PORTB;
}
```