C

DESIGN OF SOFTWARE FOR EMBEDDED SYSTEMS (SWES)

Dr. Peter Tröger
Operating Systems Group, TU Chemnitz

C

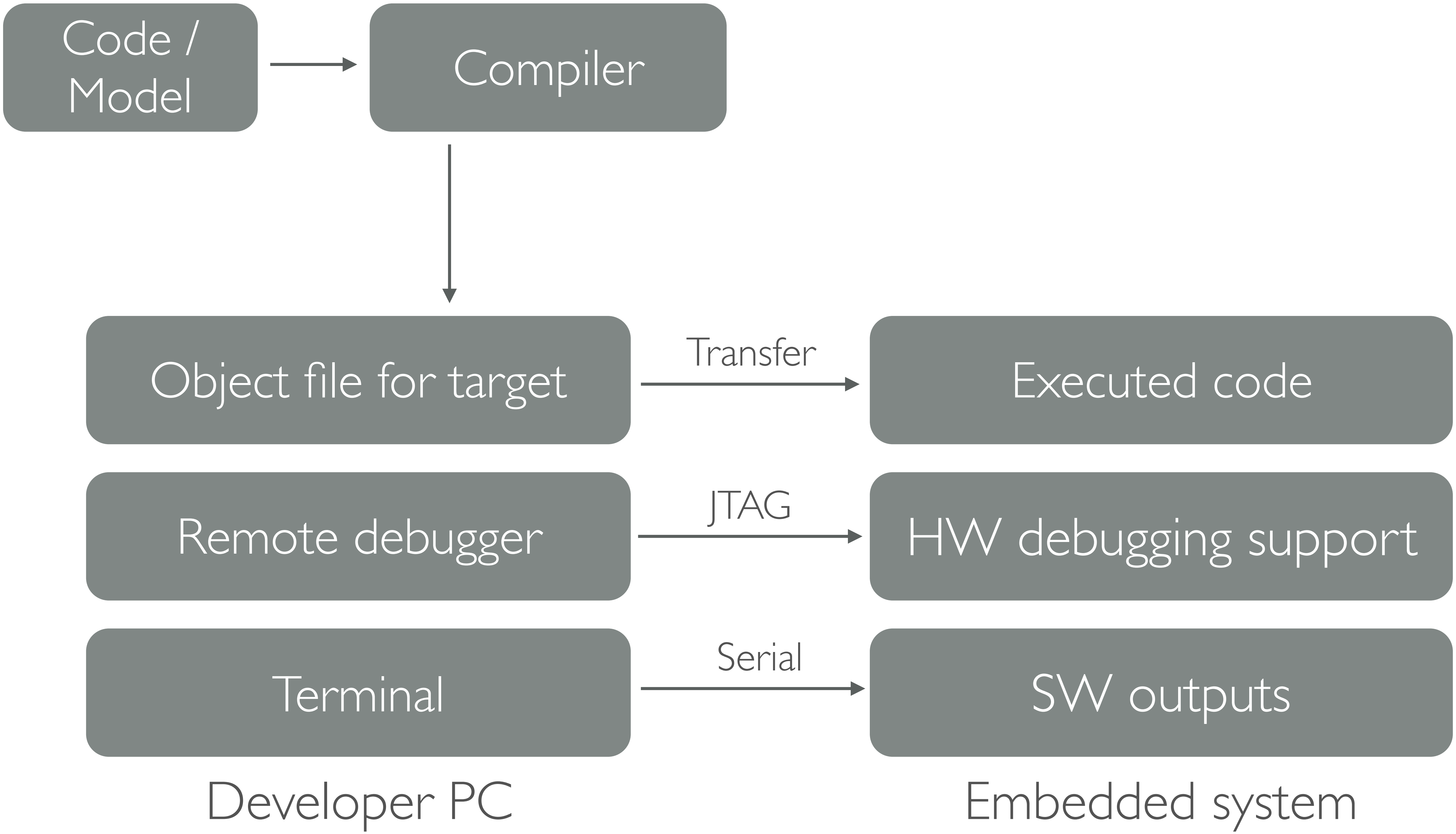
- C belongs to the most popular programming languages
  - Developed in the early 70s by Dennis Ritchie at Bell Labs
  - Imperative, structural, very small number of basic keywords
  - Portable and efficient => used for embedded systems
  - All major operating systems are written (mostly) in C
- Thin layer above assembler language
  - Data type semantics driven by hardware architecture
  - Direct memory manipulation, inline assembler supported
  - Few chances for compiler to check semantic correctness
- Standards: C89/C90, C95, C99, C11

HELLO WORLD

```
#include <stdio.h>
int main(void)
{
  printf("Hello, World\n");
  return 0;
}
```

- Implementation / source files (*.c)
- Declaration / header files (*.h)
- Object files (*.o on Unix, *.obj on Windows)
- Static library files (*.a on Unix, *.lib on Windows)
- Dynamic library files (*.so on Unix, *.dll on Windows)
- Entry point is always the *main()* function, result available in the OS

TECHNICAL ENVIRONMENT



C PREPROCESSOR

- C preprocessor
  - Simple text replacement for „#define“ and „#include“
- C Header Files
  - Separate declaration and implementation

```
# if SYSTEM == SYSV
  # define HDR ”sysv.h”
# elif SYSTEM == BSD
  # define HDR ”bsd.h”
# else
  # define HDR ”default.h”
# endif
# include HDR
```

  - „#include“ preprocessor directive includes one file in another file
  - Easiest way to include declaration into implementation file
  - Embedded world: Nice to separate hardware specifics
- Several predefined macros: __LINE__, __FILE__, __TIME__, ...

C HEADER FILES

```
#ifndef __LINUX_GPIO_H
#define __LINUX_GPIO_H

#define GPIOF_DIR_OUT  (0 << 0)
#define GPIOF_DIR_IN   (1 << 0)
#define GPIOF_INIT_LOW (0 << 1)
#define GPIOF_INIT_HIGH     (1 << 1)
#define GPIOF_IN            (GPIOF_DIR_IN)
#define GPIOF_OUT_INIT_LOW  (GPIOF_DIR_OUT | GPIOF_INIT_LOW)
#define GPIOF_OUT_INIT_HIGH (GPIOF_DIR_OUT | GPIOF_INIT_HIGH)

#ifdef CONFIG_GENERIC_GPIO
  #include <asm/gpio.h>
#else
  #include <linux/kernel.h>
  #include <linux/types.h>
  #include <linux/errno.h>
  struct device;
  struct gpio;
  struct gpio_chip;
  ...
#endif

#endif
```

C STATEMENTS

- Statement syntax has influenced C++, Java, C# and many others
  - `if(condition) statement else statement`
  - `for(init;condition;step) statement`
  - `while(condition) statement`
  - `do statement while(condition);`
  - `switch(condition) { case-block }`
  - Blocks
  - Expressions
  - `return, break, continue`

C DATA TYPES

- Only a few scalar basic types in C
  - **char** - **Smallest addressable unit of the machine**, at least 8 bit, contains character in local character set, may be signed or unsigned
  - **int** - Integer, supposed to be most efficient on the hardware
    - Qualifiers: **long** (at least 32 bit)**, short** (at least 16 bit)
    - sizeof(char) <=sizeof(short) <= sizeof(int) <= sizeof(long)
  - **float** - Floating point number with single precision
  - **double** - Floating point number with double precision
- Support for enumerations
- **signed, unsigned** - Type qualifiers

Common long integer sizes [edit]

Programming language	Approval Type	Platforms	Data type name	Storage in bytes	Signed range	Unsigned range
C ISO/ANSI C99	International Standard	Unix,16/32-bit systems[6] Windows,16/32/64-bit systems[6]	`long` †	4 (minimum requirement 4)	−2,147,483,648 to +2,147,483,647	0 to 4,294,967,295 (minimum requirement)
C ISO/ANSI C99	International Standard	Unix, 64-bit systems[6][8]	`long` †	8 (minimum requirement 4)	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615
C++ ISO/ANSI	International Standard	Unix, Windows, 16/32-bit system	`long` †	4 [9] (minimum requirement 4)	−2,147,483,648 to +2,147,483,647	0 to 4,294,967,295 (minimum requirement)
C++/CLI	International Standard ECMA-372	Unix, Windows, 16/32-bit systems	`long` †	4 [10] (minimum requirement 4)	−2,147,483,648 to +2,147,483,647	0 to 4,294,967,295 (minimum requirement)
VB	Company Standard	Windows	`Long`	4 [11]	−2,147,483,648 to +2,147,483,647	N/A
VBA	Company Standard	Windows, Mac OS	`Long`	4 [12]	−2,147,483,648 to +2,147,483,647	N/A
SQL Server	Company Standard	Windows	`BigInt`	8	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615
C#/ VB.NET	ECMA International Standard	Microsoft .NET	`long` or `Int64`	8	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615
Java	International/Company Standard	Java platform	`long`	8	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	N/A

[Wikipedia]

64 BIT DATA MODELS

Data model	short (integer)	int	long (integer)	long long	pointers/ size_t	Sample operating systems
LLP64/ IL32P64	16	32	32	64	64	Microsoft Windows (x86-64 and IA-64)
LP64/ I32LP64	16	32	64	64	64	Most Unix and Unix-like systems, e.g. Solaris, Linux, BSD, and OS X; z/OS
ILP64	16	64	64	64	64	HAL Computer Systems port of Solaris to SPARC64
SILP64	64	64	64	64	64	"Classic" UNICOS[34] (as opposed to UNICOS/mp, etc.)

[Wikipedia]

Data Type	LP32	ILP32	ILP64	LLP64	LP64
char	8	8	8	8	8
short	16	16	16	16	16
int32			32		
int	16	32	64	32	32
long	32	32	64	32	64
long long (int64)				64	
pointer	32	32	64	64	64

C DATA TYPES

- Several data representations depend on the underlying hardware
  - Ideal for hardware-oriented performance tuning
  - Use data type sizes close to register width / processor capabilities
  - Especially relevant with very small hardware (e.g. micro controllers)
  - Well-known issues with code correctness
    - Tradeoff: Potential performance vs. bug probability
- Floating points in accordance to IEEE 754
- **char** variables are technically just 8-bit integers
  - Value is position in the character set, e.g. ASCII, EBCDIC, UTF-8
- No native string type, but support for character arrays

MEMORY IN C

- Operating system provides virtual memory address space for process
  - Static variables, stored in separate region (**bss**)
  - Local variables, allocated on the **stack**
    - Each function call stores information on the stack
    - Return address, return values, parameters, local variables
  - Dynamically allocated memory regions in the **heap** (e.g. malloc)
  - Shared memory regions
- **volatile** keyword important for registers and shared global variables
  - Makes sure that the variable always lives in memory

MEMORY IN C

```
void IOWaitForRegChange(unsigned int* reg, unsigned int bitmask){
  unsigned int orig = *reg & bitmask;
  while (orig == (*reg & bitmask)) {;}
}
```

- Function to wait for register change
- Some interrupt routine will concurrently modify the value of ‚reg‘
- Code compiled with activated optimizations
- Visible effect
  - Function never returns, although register changes
- What is wrong ? (typical problem in embedded C coding)

C POINTERS

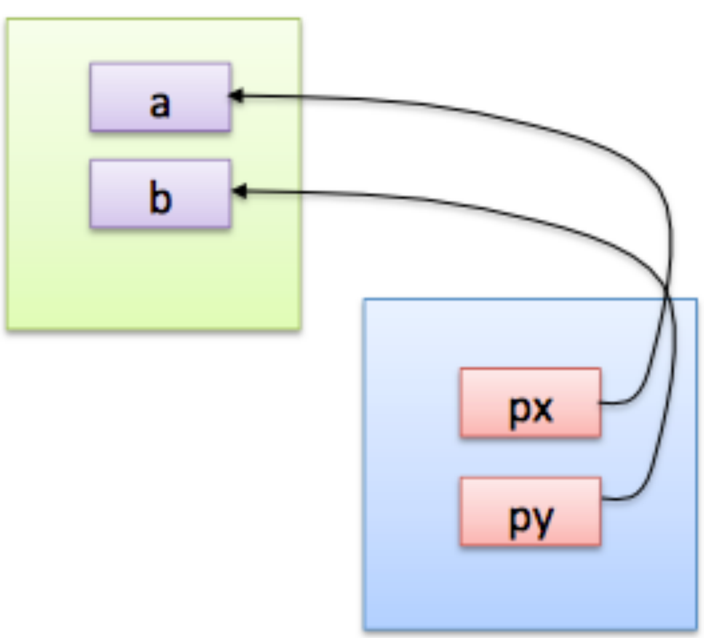
- **Pointer**: Variable that contains some memory address
    - Some location: Another variable, allocated heap memory, function implementation, operating system data structure, shared memory, ...
- Excessively used as concept in C
    - Maps directly to addressing in assembler language
    - Pointer variable is typed with respect to the data it points to
    - **& operator** for adress determination
    - *** operator** for de-referencing

```
int x = 1, y = 2, z[10];

int *ip;

ip = &x;

y = *ip; *ip = 0;

ip = &z[0];
```

C POINTERS

```
int x = 1, y = 2, z[10];

int *ip;

ip = &x;

y = *ip; *ip = 0;

ip = &z[0];
```

- C only knows call-by-value
- Implement call-by-reference by providing a pointer
  - Pointer value is copied




```
swap( &a, &b );

----------------------------

void swap( int *px, int *py)
{
   int temp = *px;
   *px = *py;
   *py = temp;
}
```

ARRAYS AND POINTERS

- Value of an array variable is the address of the first element
- Every array indexing operation can be expressed as pointer operation
  - Sometimes faster
- Array and pointer are not the same
  - Arrays are not variables, not allowed on left side of an expression
  - Arrays as function argument result in address of the first element
    - Allows to hand over only parts of the array to some function

```
pa = &a[0];

equals

pa = a;
```

```
a[i]

equals

*(a+i);
```

```
*(array_var+3)
```

```
func(&a[2]);
```

ARRAYS AND POINTERS

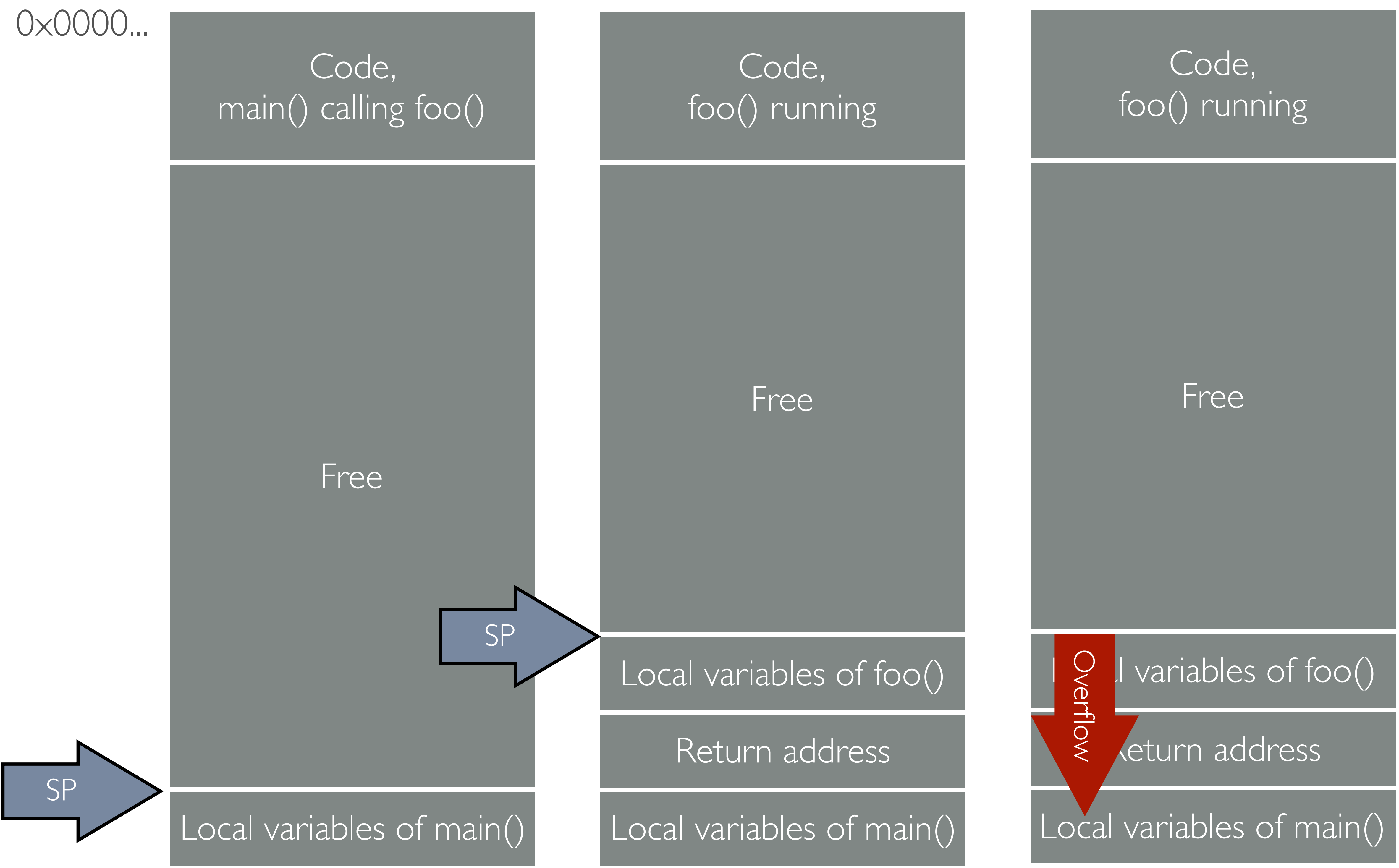
```
void strcpy1( char * s, char * t ) {
  int i = 0;
  while ((s[i] = t[i]) != ,\0')
    i++;
}

void strcpy2( char * s, char * t ) {
  while ((*s = *t) != '\0')
    { s++; t++; }
}
```

ARRAYS AND POINTERS

- Pointers can be added, subtracted and compared
  - Pointer arithmetics - very efficient and dangerous tool
  - Inc / dec steps in accordance to the data type being pointed to
  - All pointers can be converted to „void*“ and reverse
- No runtime checks for memory access through array index or pointer
  - Compiler converts it to native code, no underlying runtime
  - Illegal access may be defeated by operating system
  - Unintended access to process data possible
    (stack-based buffer overflow attack on return address)
- Pointer can reference functions (start address in code segment)

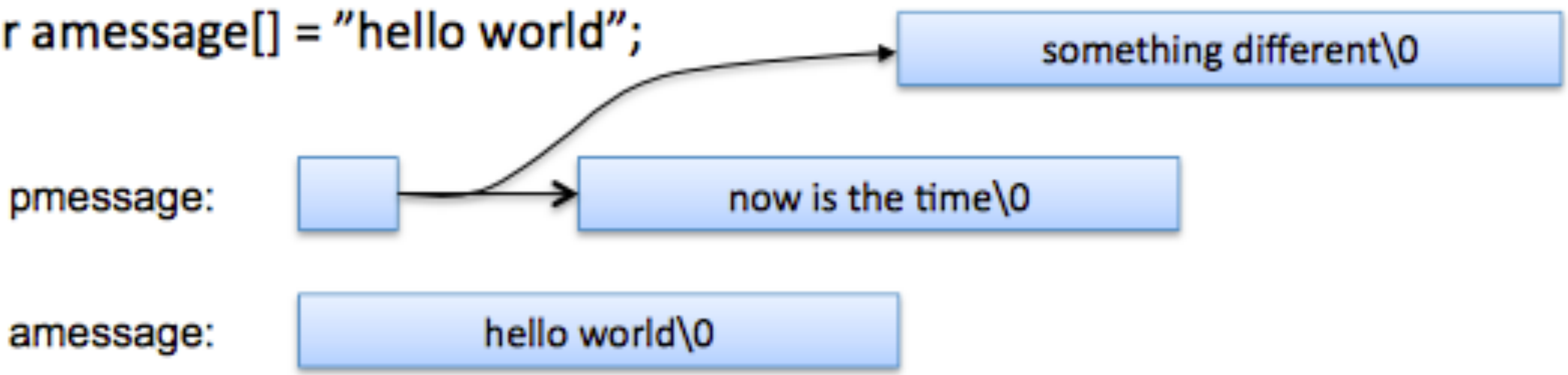
```
(* compare) ( ”hello”, ”world” );
```

BUFFER OVERFLOW



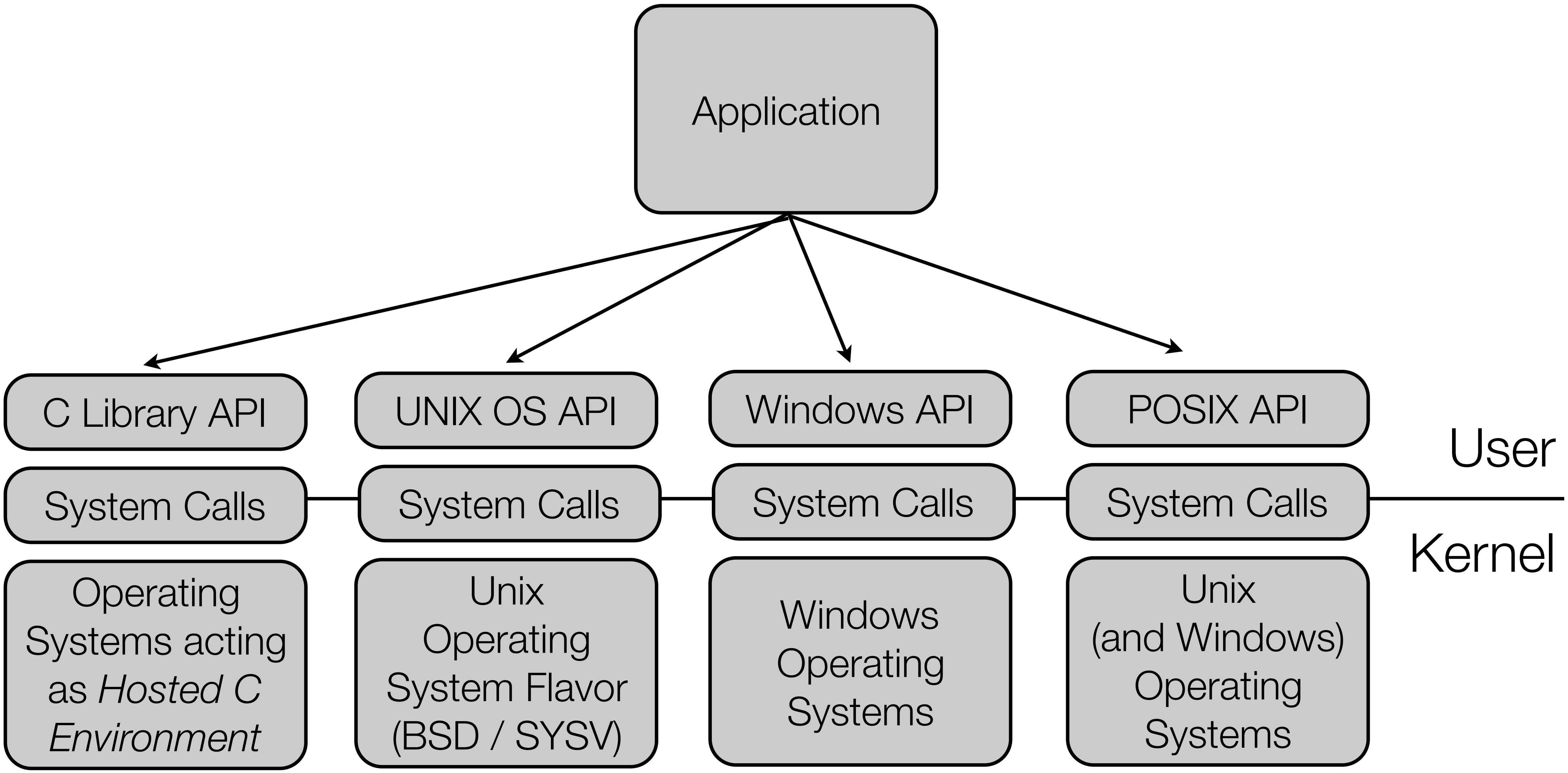
DEALING WITH STRINGS

```
char * pmessage = "now is the time";
char amessage[] = "hello world";
```



- *pmessage*: Pointer can be changed, but not the text
- *amessage*: Text can be changed
- [] and * can both be used on arrays
    - Pointer arithmetic may save a fixed-size index variable

APPLICATION PROGRAMMING INTERFACE



STANDARD C LIBRARY

- Standard C Library - 1989 first version from the American National Standards Institute (ANSI) as C89
- Last from 2011 (C11), glibc as most common implementation
- Focus on C source code portability for all operating systems
- Portable C programs are easy (e.g. Apache, MySQL)
- Portable non-C programs typically have a C-based runtime system
  - Hadoop (Java), Django (Python)
- Non-portable C programs are possible
  - Compiler-specific libraries, operating system - specific system calls

STANDARD C LIBRARY

- Set of default functions, available together with C compiler
- Function and data structures declared in multiple header files
- Typically implementation in one library file (libc.a, libc.so)
  - Automatically considered by linker
- Example: Input and output functions in `<stdio.h>`
  - `fopen, fclose, fread, fwrite, fprintf, ...`
  - Predefined file streams for `STDIN`, `STDOUT`, and `STDERR`
- Other functionality: Assertions (assert.h), mathematical functions (math.h), time handling (time.h), ...
- All embedded hardware vendors provide some C compiler + standard library for bare-metal development (guess why ?)

POSIX LIBRARIES

- **Portable Operating System Interface**
- Family of standards defined by IEEE since 1988
- Intention of offering a unified API across different UNIX-alike products
- Includes specification of shells and tools, since scripts are applications
- Focus on source code portability for Unix applications
- Operating systems can be certified for compliance
- Only a few operating systems are fully POSIX certified, including Windows NT 4.0
- Design goal for complying with governmental regulations
- Many design choices driven by the UNIX background (e.g. signals)

REAL-TIME IN C

- Real-time support not part of the language itself
  - POSIX interface in operating system typically used
- POSIX message queues
  - Communication between isolated processes
  - Multiple reader / multiple writer
  - Have internal structure
  - Status can be determined by communication partners
- POSIX signals
  - Asynchronous notification mechanism, common in UNIX apps
- Synchronization, priority scheduling and timer support

POSIX 1003.1B

- Additional demands on OS API for real-time systems
  - POSIX.1 - Fundamental system calls (e.g. *fork, read, write*)
  - POSIX 1003.1b - Real-time extensions (e.g. *pthread_setsched*)
    - Real-time signals
    - Message queues and other synchronization primitives
    - Priority scheduling (FIFO, round robin with quantum, …)
    - Memory locking
  - POSIX 1003.1c - Threading extensions (e.g. *pthread_create*)

POSIX 1003.1B

- Fixed priority scheduling
  - At least 32 priorities, support for round robin and FIFO
  - Policy and priority adjustable per thread / process
- Extended and safe real-time signal handling
  - At least 8 freely definable signals
  - Signal handling ordered by priorities
  - Signals can contain data
  - Signals are never lost
- Extended support for inter-process communication (IPC)
  - Prioritized messages, non-blocking send and receive

POSIX 1003.1B

- High-resolution timers in nanosecond scale
  - Up to 32 timers per process
  - Operating system considers timer overflows
- Shared memory and locking
  - Code and data pages can be excluded from paging
- Synchronous I/O
  - Reading and writing of data without buffering
- Asynchronous priority-driven I/O
  - Non-blocking input and output
  - Ordered by priorities

EXAMPLE (RT LINUX)

```
#define MAX_SAFE_STACK (8*1024) // 8 KB
#define MAX_ALLOC_MEM (100*1024*1024) // 100 MB
void stack_prefault() {
    # Pre-fault pages up to the maximum planned stack size
    unsigned char dummy[MAX_SAFE_STACK];
    memset(&dummy, 0, MAX_SAFE_STACK);
}
int main(int argc, char* argv[]) {
    # Enable RT scheduling
    struct sched_param param;
    param.sched_priority = 50;
    pthread_setschedparam(pthread_self(), SCHED_FIFO, &param);
    # lock all current and future pages
    mlockall(MCL_CURRENT|MCL_FUTURE);
    stack_prefault();
    # Prevent malloc trimming on free, and mmap usage in malloc
    mallopt (M_TRIM_THRESHOLD, -1);
    mallopt (M_MMAP_MAX, 0);
    char* buffer = malloc(MAX_ALLOC_MEM);
    memset(&buffer, 0, MAX_ALLOC_MEM);
    free(buffer);
    // now it's safe to do RT stuff
}
```

https://www.kernel.org/doc/ols/2009/ols2009-pages-79-86.pdf

```
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#define MSG_SIZE       4096

void handler (int sig_num) { printf ("Received sig %d.\n", sig_num); }

void main () {
  struct mq_attr attr, old_attr;   // To store queue attributes
  struct sigevent sigevent;        // For notification
  mqd_t mqdes, mqdes2;             // Message queue descriptors
  char buf[MSG_SIZE];              // A good-sized buffer
  unsigned int prio;               // Priority

  attr.mq_maxmsg = 300;
  attr.mq_msgsize = MSG_SIZE;
  attr.mq_flags = 0;
  mqdes =  mq_open ("sideshow-bob", O_RDWR | O_CREAT, 0664, &attr);
  mqdes2 = mq_open ("troy-mcclure", O_RDWR | O_CREAT, 0664, 0);
  mq_unlink („troy-mcclure");      // declare as temporary queue, deleted when closed
  mq_getattr (mqdes, &attr);
  printf ("%d messages are currently on the queue.\n", attr.mq_curmsgs);
  if (attr.mq_curmsgs != 0) {
    attr.mq_flags = MQ_NONBLOCK;   // do not block on read / write attempts
    mq_setattr (mqdes, &attr, &old_attr);
    while (mq_receive (mqdes, &buf[0], MSG_SIZE, &prio) != -1)
      printf ("Received a message with priority %d.\n", prio);
    if (errno != EAGAIN) { perror („mq_receive()"); _exit (EXIT_FAILURE); }
    mq_setattr (mqdes, &old_attr, 0);
  }
  signal (SIGUSR1, handler);
  sigevent.sigev_signo = SIGUSR1;
  if (mq_notify (mqdes, &sigevent) == -1) {
    if (errno == EBUSY) printf ( "Another process has registered for notification.\n“ );
    _exit (EXIT_FAILURE);
  }

  for (prio = 0; prio <= MQ_PRIO_MAX; prio += 8) {
    printf ("Writing a message with priority %d.\n", prio);
    if (mq_send (mqdes, "I8-)", 4, prio) == -1) perror ("mq_send()");
  }
  mq_close (mqdes);
  mq_close (mqdes2);
}
```

[http://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/mq_overview.html]

MEMORY IN RT

- Memory management in real-time applications is critical
  - Standard operating systems optimize resource management
  - Dynamic allocation / deallocation logic to save memory resource
  - Leads to unpredictable timing behavior
- Adopted real-time operating system
  - Application developer must manually pin all relevant memory
- Purpose-built real-time operating system
  - Specific API for working with statically allocated memory regions