DESIGN OF SOFTWARE FOR EMBEDDED SYSTEMS (SWES)

Dr. Peter Tröger
Operating Systems Group, TU Chemnitz

C

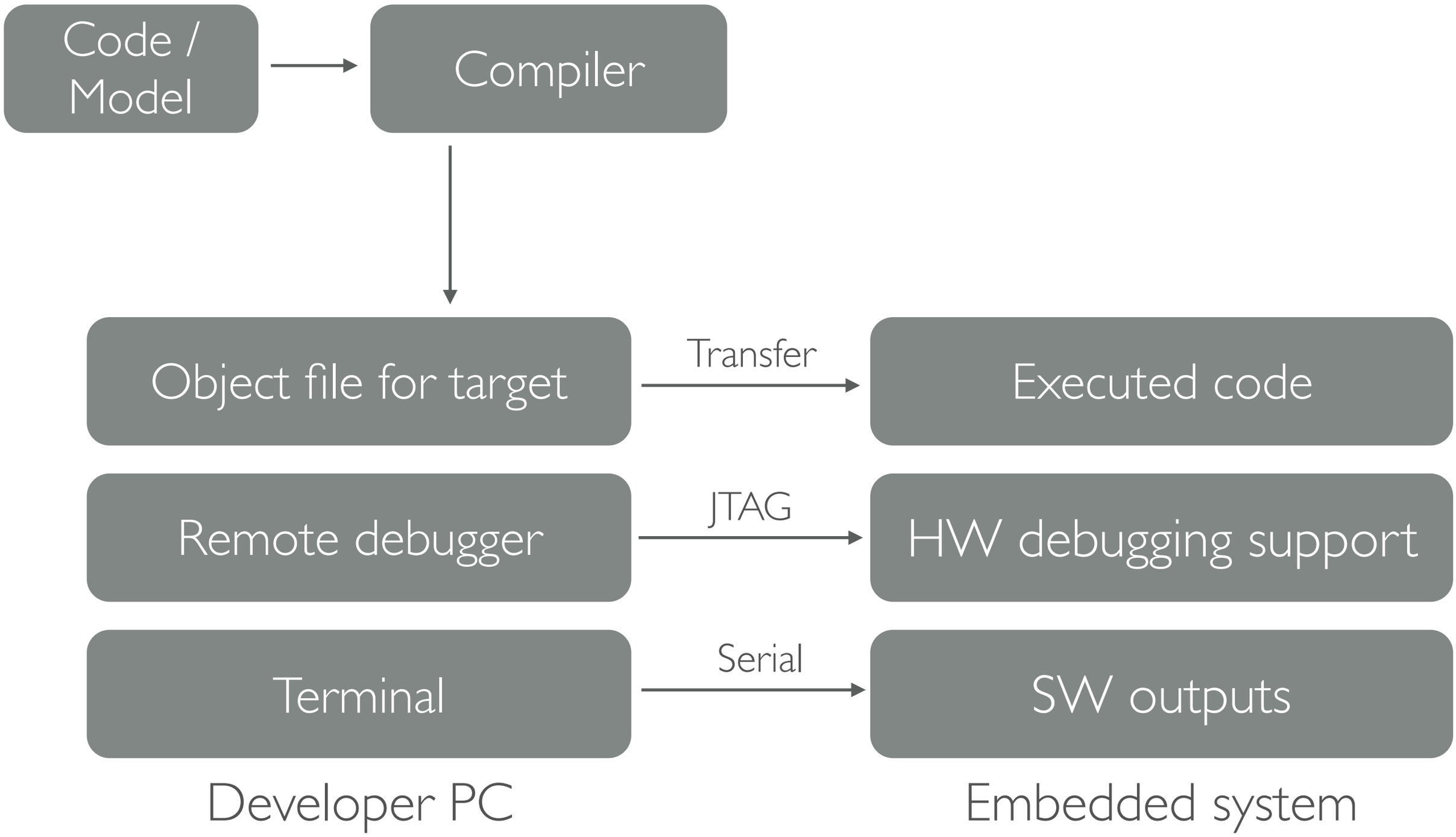
- C belongs to the most popular programming languages
  - Developed in the early 70s by Dennis Ritchie at Bell Labs
  - Imperative, structural, very small number of basic keywords
  - Portable and efficient => used for embedded systems
  - All major operating systems are written (mostly) in C
- Thin 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

```c
#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 for variables
  - Tells the compiler that the value may change outside the normal control flow of the program (e.g. by hardware)

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)

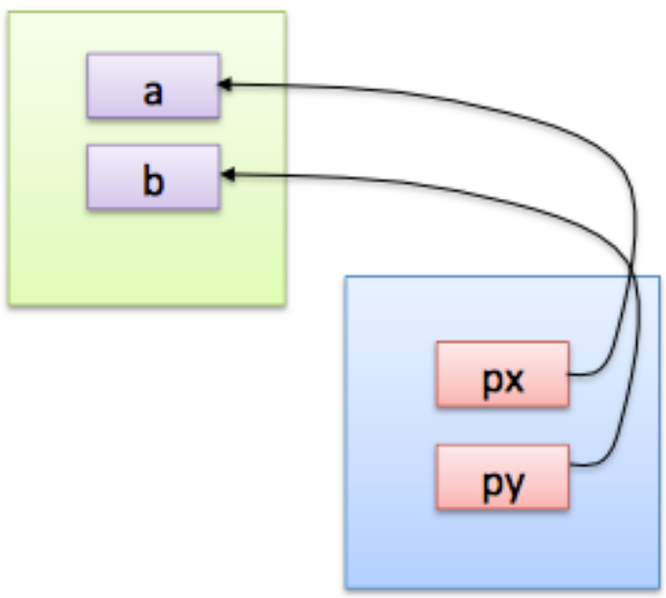
C POINTERS

- **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

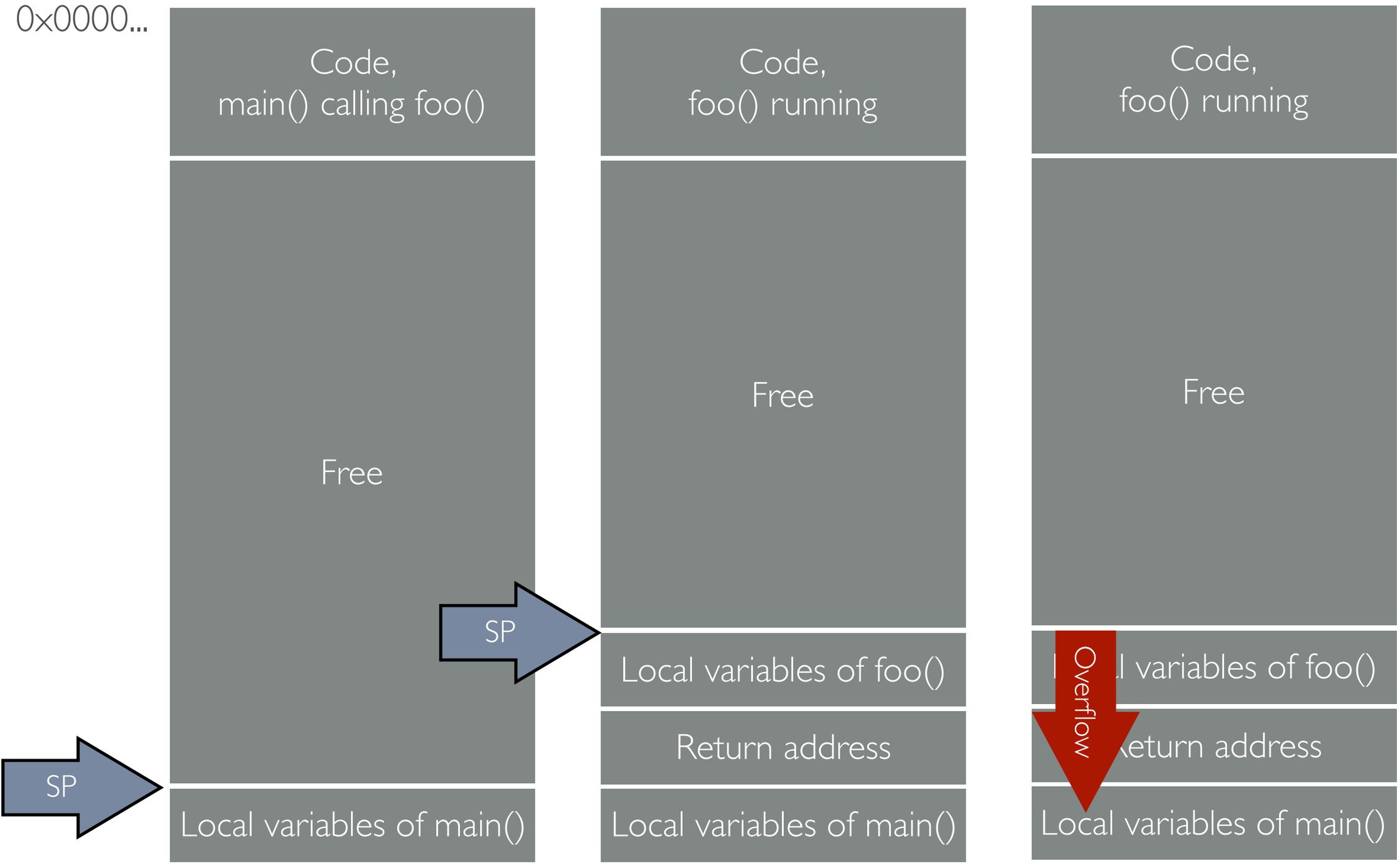
```
void strcpy1( char * s, char * t ) {
  int i = 0;
  while ((s[i] = t[i]) != ,\0')
    i++;
}

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)

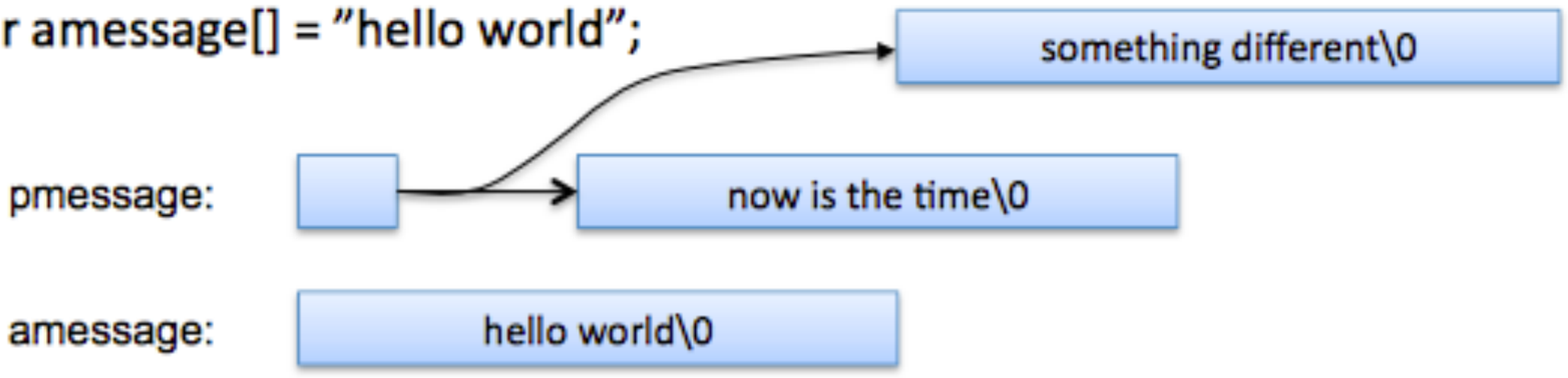
```
(* compare) ( "hello", "world" );
```


BUFFER OVERFLOW



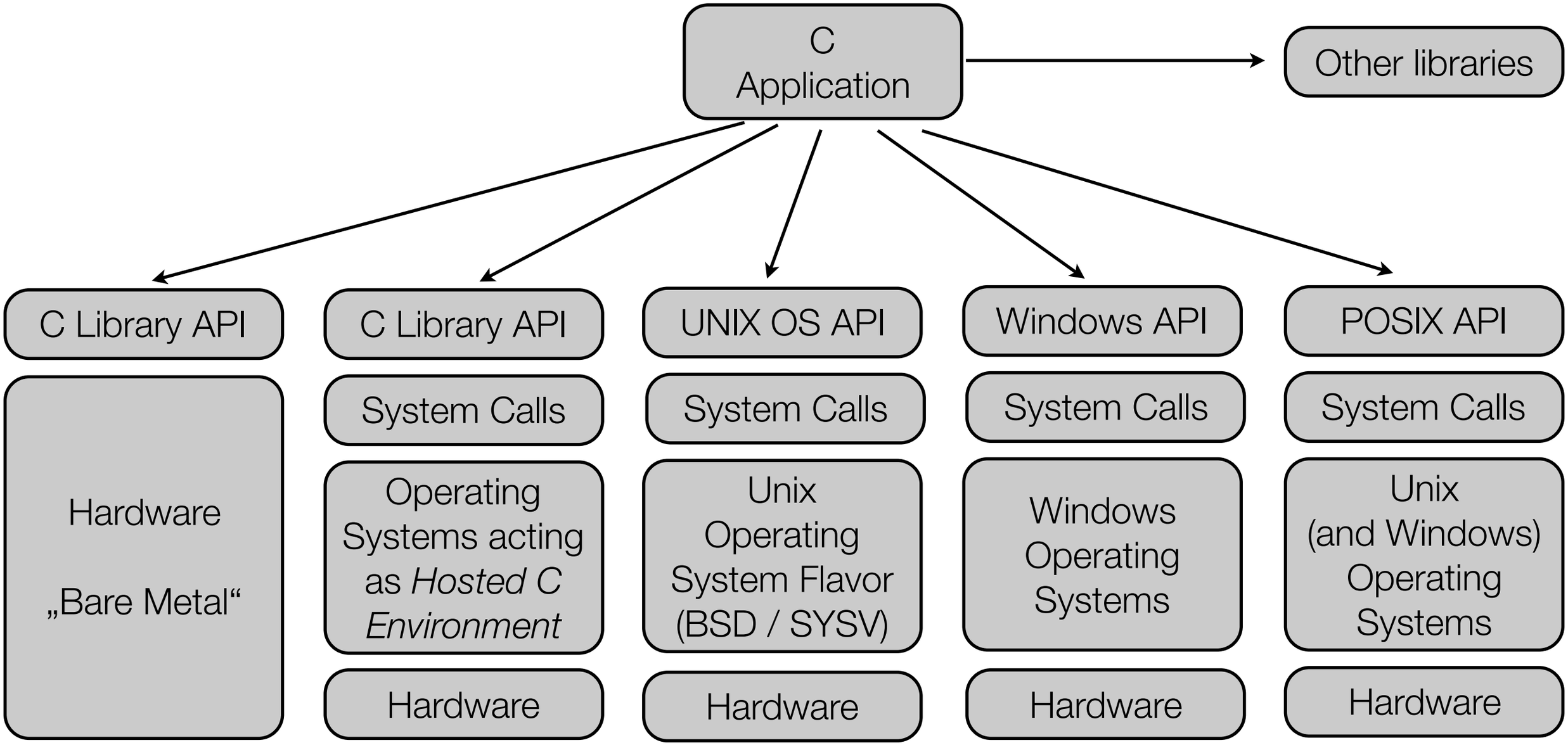
DEALING WITH STRINGS

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



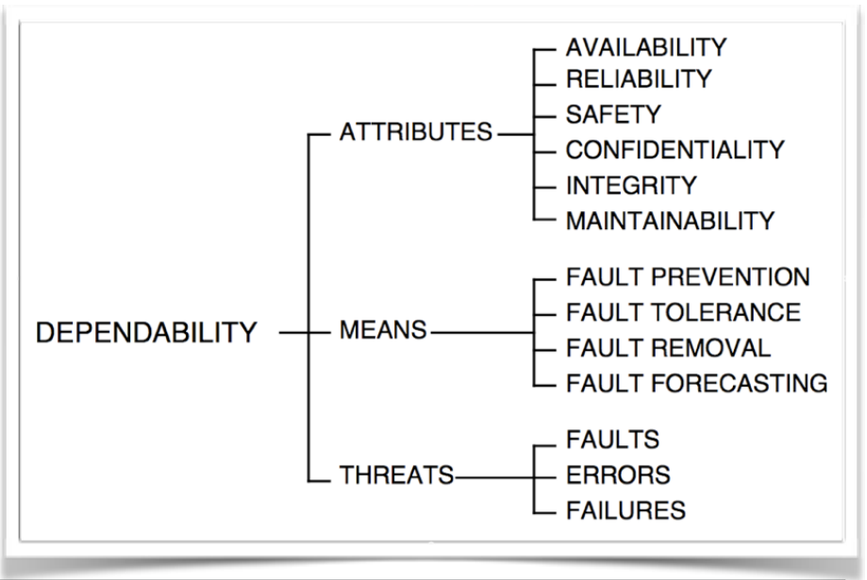
- *pmessage*: Pointer can be changed, but not the text
- *amessage*: Text can be changed
- [] and * can both be used on arrays
  - Pointer arithmetic may save a fixed-size index variable

APPLICATION PROGRAMMING INTERFACE



DEPENDABLE C CODE

- Low-level approach makes C code fast and memory-efficient
  - But also only minimal protection from programmer mistakes
- Different ways to achieve dependable C code
  - Fault avoidance - Prevent bugs from being introduced
    - Coding conventions or C code generation
    - Depends on understanding of typical fault causes
  - Fault removal - Find bugs before going into production
    - Testing, whole program analysis
  - Fault tolerance and fault prediction



DEPENDABLE C CODE

- Problematic properties of the C language
  - Intentionally implementation-defined behavior
    - Examples:
      Expression evaluation order, numerical types, register type
    - Chance for compiler optimizations
  - Intentionally non-portable semantics
    - Example: LOCALE in character / string handling
  - Intentionally undefined behavior
    - Example: Reaction on run-time problems,
      such as non-initialized variables being used

STYLE ISSUES

Code Structuring

```
if (a==b)   c=0;

if (a==b); c=0;
```

Expressions

```
a & b

a && b
```

Type conversion

```
int i;

float f, g;

g = f + i;

i = f + g;
```

Names & Scopes

```
int i=1;
{
  int i=2;
}
```

Operators

```
a = b

a == b
```

Readability

```
for (i=0; i<sizeof(s)-1&&(c=getchar())!='\n' && c!=EOF; i++)
  s[i] = c;
  s[i+1] = 0;
```

STYLE ISSUES

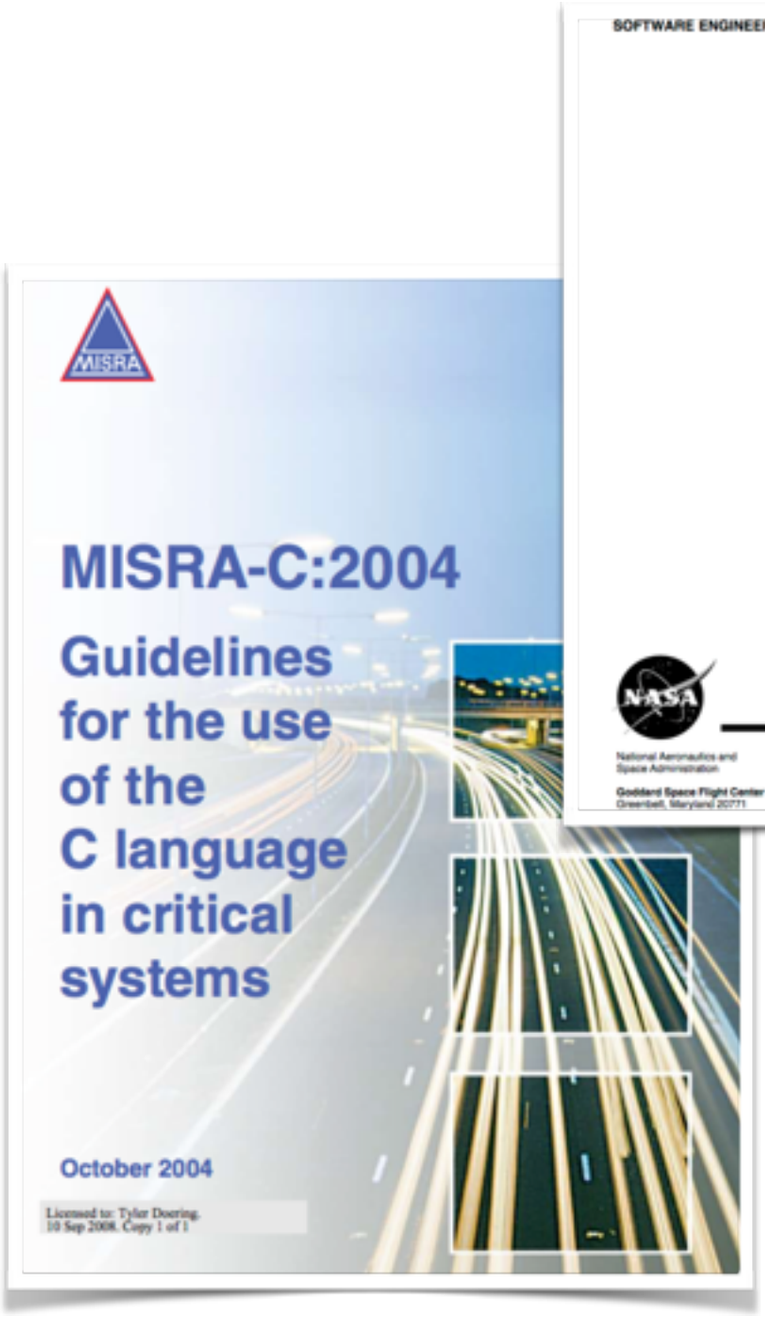
```
char **argv          // Pointer to char array
int (*daytab)[13] // Pointer to int array
int  *daytab [13] // Array of int pointers
void  *comp ()     // Function returning void pointer
void (*comp)()     // Pointer to function returning void

// Function returning pointer to an array, which contains
pointers to functions returning char
char (*(*x() ) [] )()

// Array of pointers to functions,
// which return a pointer to an array
char (* (*x[3])() )[5]
```

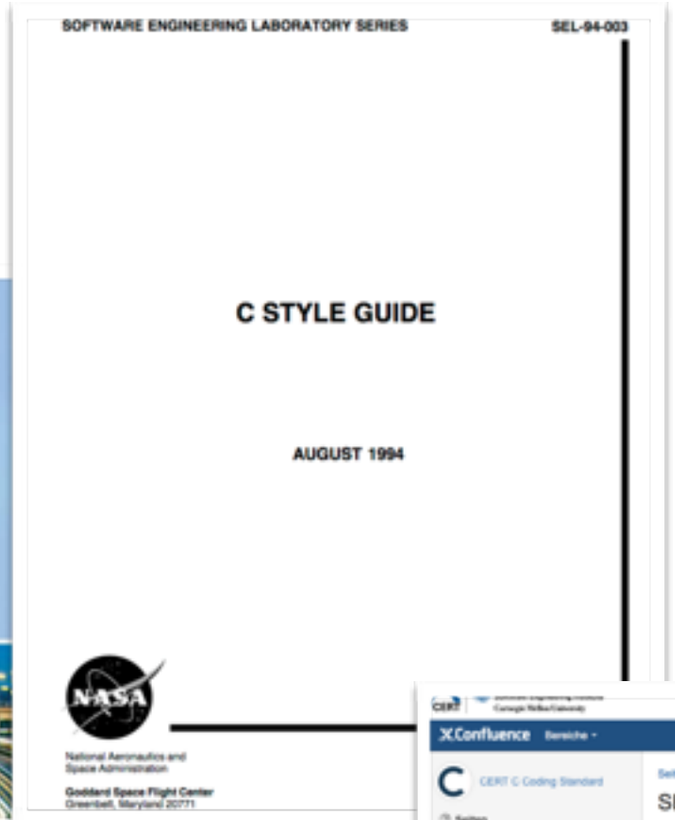
IOCCC.ORG

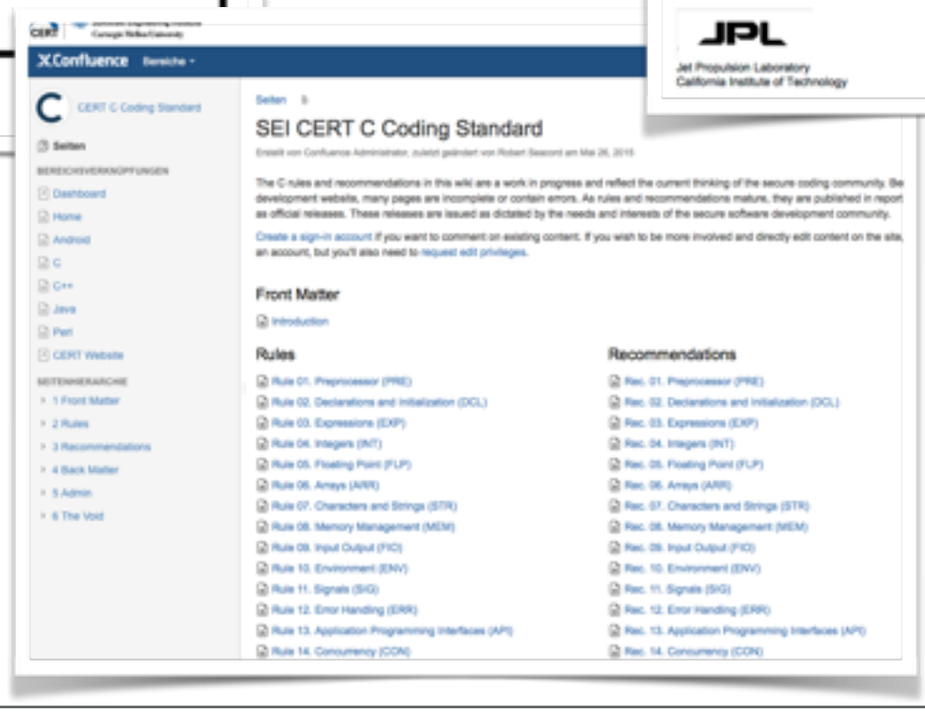
```
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#define _ float

                  _ CRUSH(_*LEG,int ARM,_*FINGER) {
          _ GORE; for(GORE=0,--ARM; ARM>=0; --ARM)GORE+=FINGER[ARM]*ARM
     [LEG]; return GORE; } _*BITE(){ _*BRAIN=calloc(sizeof(_),1<<17);int TOE
  =getc(stdin),EYE,SKULL=0; while((EYE=getc(stdin))!=EOF){ ++BRAIN[(TOE<<8) +
 EYE]; ++SKULL; TOE=EYE; } if(SKULL)for(TOE=0; TOE<8<<13; ++ TOE)BRAIN [TOE]/=
 SKULL; return BRAIN; } _ CHEW(_*GUT,_ BONE[][1<<16]){ int GRR; for(GRR=0; GRR<
 6; ++GRR){ GUT[GRR+256*256]=powf(1+expf(-CRUSH(BONE[GRR],1<<16,GUT)),-1); BONE
 [6][50+GRR]=GUT[256*256+GRR]*(1.-GUT[(8<<13)+GRR]); } BONE[6][81]=1/(1 +expf(-
 CRUSH(GUT+(1<<16),6,BONE[6]))); return BONE[6][82]=BONE[6][81]*(1.-BONE[6][81]
 ),BONE[6][81]; } _ GNAW(_ FLESH,_ LEG[][2<<15],_*EYE){ int UG,MMM; LEG[6][13]=
 FLESH-CHEW(EYE,LEG); LEG[6][14]=LEG[6][ 82]*LEG[6][13]; for(UG=0; UG<6; ++UG){
 LEG[6][34]=LEG[6][UG+(1<<7)-14]*LEG[6][7<<1]*LEG[6][UG]; for(MMM=0; MMM<4<<14;
 ++MMM)LEG          [UG][MMM]+=LEG[6][34]*.3*EYE[MMM]; LEG[6][UG]+=.3*LEG[6][14
  ]*EYE[              256*256+UG]; } return powf(LEG[6]          [13],2); } _
  **EAT(              char*TOMB){ DIR*BONE = opendir(             TOMB); int
  BRAIN=              0; struct dirent*TOOTH; _**BODY             =0; while
 (BONE?(TOOTH=       readdir(BONE)):0){ if(  TOOTH->             d_name[0]
 !=46){ char*MOAN=malloc(strlen(TOMB)+strlen(TOOTH->d_name       )+1); sprintf(
 MOAN, "%s%s",TOMB,TOOTH ->d_name); if(freopen(MOAN, "r",stdin)){ BODY=realloc(
 BODY,sizeof(_*)*(BRAIN+1)); BRAIN ++ [BODY]=BITE(); } } } return BODY=realloc(
 BODY,(1+BRAIN)*sizeof(_*)),BODY[BRAIN]=0,BODY; } int main(int GRR,char **UGH){
 _ BRAINS[7][1<<16],***CORPSES; int PUS,OOZE,UG; for(srand(time(0)),PUS=0; PUS<
  7; ++PUS)for(OOZE=0; OOZE<4<<14; ++OOZE)BRAINS[PUS][OOZE]=rand()/(_)RAND_MAX
   -.5;fread(BRAINS,sizeof(BRAINS),1,stdin); if(*UGH[1]==45){ GRR-=2; CORPSES
    =malloc(                                                  sizeof(_
    **)*GRR                                                    ); for(
     PUS=0;                                                      PUS<
    GRR; ++             PUS)CORPSES[PUS]=EAT(UGH[2+PUS]);       for(UG=
     0; UG<atoi       (&(UGH[1][1])); ++UG){ BRAINS[6][97]=      0; for(PUS
      =0; PUS<GRR; ++PUS)for(OOZE=0; CORPSES[PUS][OOZE]; ++OOZE)BRAINS [6][
        97]+=GNAW(1.-(_) PUS/(GRR-1),BRAINS,CORPSES[PUS][OOZE]); fprintf(
         stderr,"%d: %f\n",UG,BRAINS[6][97]); } fwrite( BRAINS, sizeof(
          BRAINS),1,stdout); } else for(UG=1; UG<GRR; ++UG)if(freopen(
              UGH[UG],"r",stdin))fprintf(stderr,"%s %f\n",UGH[UG],
                    CHEW(BITE(),BRAINS)); return 0; }
```

http://ioccc.org/years-spoiler.html

C STYLE GUIDES







RULES

- No dependence should be placed on C's operator precedence

```
x=(a*b)+c;
```

```
x=a*b+c;
```

- Only compound statements after
  `if, else, while, for, switch, case, do`

```
while (i > 0)
  *t++ = *s++;
```

```
while (i > 0){
  *t++ = *s++;
}
```

- Naming conventions for variables
  - Constants in UPPER CASE
  - Type names start with upper case letter, e.g. `struct Ports`
  - . . .

FUNCTION MACROS

[https://www.securecoding.cert.org/]

```c
#include <string.h>

void func(const char *src) {
  /* Validate the source string; calculate size */
  char *dest;
  /* malloc() destination string */
  memcpy(dest, src,
    #ifdef PLATFORM1
      12
    #else
      24
    #endif
  );
  /* ... */
);
```

- Code considers platform specifics on memory copy operation
- `memcpy()` function may be implemented by a preprocessor macro
  - Compiler behavior undefined
- Either re-formulate code or forbid function macros

FUNCTION MACROS

```c
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  int m = ABS(++n);

  /* ... */
}
```

```c
m = (((++n) < 0) ? -(++n) : (++n));
```

```c
static inline int iabs(int x) {
  return (((x) < 0) ? -(x) : (x));
}
```

- Side effects in arguments to function macros may raise problems
- Different solutions
  - Perform `++n` before function call
  - Replace function macro with `inline` function

[https://www.securecoding.cert.org/]

TYPE CONVERSIONS

```c
void func(float f_a) {
  int i_a;

  /* Undefined if the integral part of f_a >= INT_MAX */
  i_a = f_a;
}
```

- Handle down-cast of values explicitly

[https://www.securecoding.cert.org/]

```c
#include <float.h>
#include <limits.h>

void func(float f_a) {
  int i_a;

  if (f_a >= ((float)INT_MAX -1.0) || f_a < ((float)INT_MIN +1.0)|| (f_a >= 0.0F && f_a < FLT_MIN)) {
    /* Handle error */
  } else {
    i_a = f_a;
  }
}
```


POINTERS

[https://www.securecoding.cert.org/]

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

void func(void) {
  char *tmpvar;
  char *tempvar;

  tmpvar = getenv("TMP");
  if (!tmpvar) {
    /* Handle error */
  }
  tempvar = getenv("TEMP");
  if (!tempvar) {
    /* Handle error */
  }
  if (strcmp(tmpvar, tempvar) == 0) {
    printf("TMP and TEMP are the same.\n");
  } else {
    printf("TMP and TEMP are NOT the same.\n");
  }
}
```

- `getenv` documentation declares that returned pointer should not be stored
- Code may lead to overwriting of first result
- Variables are considered the same even though they are not
- ‚Heisenbug‘

COMMENTING

- Code implements an algorithm
  - High-level comments for major software modules
  - Fine granularity comments explaining non-obvious details
- Header files should have comment block
  - Only readable part of library code
  - Version information, usage license, authors, support contact

```
/*****************************************************************
* Product: . .
* Version: . .
* Updated: November 26 2015
* Copyright (C) 2015-2015 Brillant student <brilliant@student.org>
* <licensing terms> (if any)
*****************************************************************/
```


COMMENTING

```
/*****************************************************************************************************\
* Project includes
\*****************************************************************************************************/
#include "mma845x.h"              // MMA845xQ macros
#include "iic.h"                  // IIC macros
#include "sci.h"                  // SCI macros
#include "spi.h"                  // SPI macros
#include "terminal.h"             // Terminal interface macros

/*****************************************************************************************************\
* Public macros
\*****************************************************************************************************/
/*****************************************************************************
**
**  General System Control
**
**  0x1802  SOPT1     System Options Register 1
**  0x1803  SOPT2     System Options Register 2
**  0x1808  SPMSC1    System Power Management Status and Control 1 Register
**  0x1809  SPMSC2    System Power Management Status and Control 2 Register
**  0x180B  SPMSC3    System Power Management Status and Control 3 Register
**  0x180E  SCGC1     System Clock Gating Control 1 Register
**  0x180F  SCGC2     System Clock Gating Control 2 Register
**  0x000F  IRQSC     Interrupt Pin Request Status and Control Register
*/

#define init_SOPT1    0b01000010
/*                      1100001U = reset
**                      |||xxx||
**                      |||   |+-- RSTPE      =0  : RESET pin function disabled
**                      |||   +--- BKGDPE     =1  : Background Debug pin enabled
**                      ||+------- STOPE      =0  : Stop Mode disabled
**                      |+-------- COPT        =1  : Long COP timeout period selected
**                      +--------- COPE        =0  : COP Watchdog timer disabled
*/

#define init_SOPT2    0b00000010
/*                      00000000 = reset
**                      |x||x|||
**                      | || ||+-- ACIC1      =0  : ACMP1 output not connected to TPM1CH0 input
**                      | || |+--- IICPS      =1  : SDA on PTB6; SCL on PTB7
**                      | || +---- ACIC2      =0  : ACMP2 output not connected to TPM2CH0 input
**                      | |+------ TPM1CH2PS  =0  : TPM1CH2 on PTA6
**                      | +------- TPM2CH2PS  =0  : TPM2CH2 on PTA7
**                      +--------- COPCLKS    =0  : COP clock source is internal 1kHz reference
*/
```

[freescale.com]

DATA TYPES

```
// Define struct overlay
typedef struct
{
     unsigned int  count;      // Offset 0x00
     unsigned int  max;        // Offset 0x02
     unsigned int  _reserved;  // Offset 0x04
     unsigned int  flags;      // Offset 0x06
} Counter;

// Create pointer to chip base address
Counter volatile * const pCounter = 0x10000000;

// Next line is equal to
// *((unsigned int *)0x10000002) = 5000;
pCounter->max = 5000;
pCounter->flags |= GO;

...

// Poll timer state
if (pCounter->flags &= DONE)
{ ... }
```

- Typical example for memory-mapped I/O
- Compiler determines offsets for individual registers
  - Allows struct-based access
- `struct` definition may not be portable
- Semantical, not syntactical issue

DATA TYPES

A++;

	8-bit S08	16-bit S12X	32-bit ColdFire
char near A;	inc A	inc A	move.b A(a5),d0 addq.l #1,d0 move.b d0,A(a5)
unsigned int A;	ldhx @A inc 1,x bne Lxx inc ,x Lxx:	incw A	addq.l #1,_A(a5)
unsigned long A;	ldhx @A jsr _LINC	ldd A:2 ldx A jsr _LINC std A:2 stx A	addq.l #1,_A(a5)

[freescale.com]

- Same single-line instruction on different processors
- Choice of data type impacts assembler code efficiency
- Trade-off between optimal portability and optimal performance

DATA TYPES

- Raw C data types allows compiler to choose most efficient storage

```
for (int i=0; i < N; i++) { ... }
```

- Most coding conventions recommend to not use them
  - Projects define their own data type abstractions
  - `<stdint.h>` for C99 provides portable data types
    - `int8_t`: signed 8-bit `uint8_t`: unsigned 8-bit
    - `int16_t`: signed 16-bit `uint16_t`: unsigned 16-bit
    - `int32_t`: signed 32-bit `uint32_t`: unsigned 32-bit
    - `int64_t`: signed 64-bit `uint64_t`: unsigned 64-bit

DATA TYPES

```
#include <limits.h>
#if (INT_MAX == 0x7fffffff)
typedef int SI_32;
typedef unsigned int UI_32;
#elif (LONG_MAX == 0x7fffffff)
typedef long SI_32;
typedef unsigned long UI_32;
#else
#warning "No 32 bit type."
#endif
```

```
// int j;
SI_32 j;
for (j = 0; j < 64; j++) {
  if (arr[j] > j*1024){
    arr[j]=0;
}}
```

CONST CORRECTNESS

- `const` keyword in C for read-only variable
  - Allows compiler to put value into ROM
  - Does not happen when `#define` is used instead
- In contrast to other languages, it is a property of the type
  - Part of compile-time type checking, which is good
- Most other languages allow to define constant objects instead

```
const int x=1;
```

```
int const *px;
```

```
int * const px;
```

```
void f(int& x);
// ...
const int i;
f(i);
```

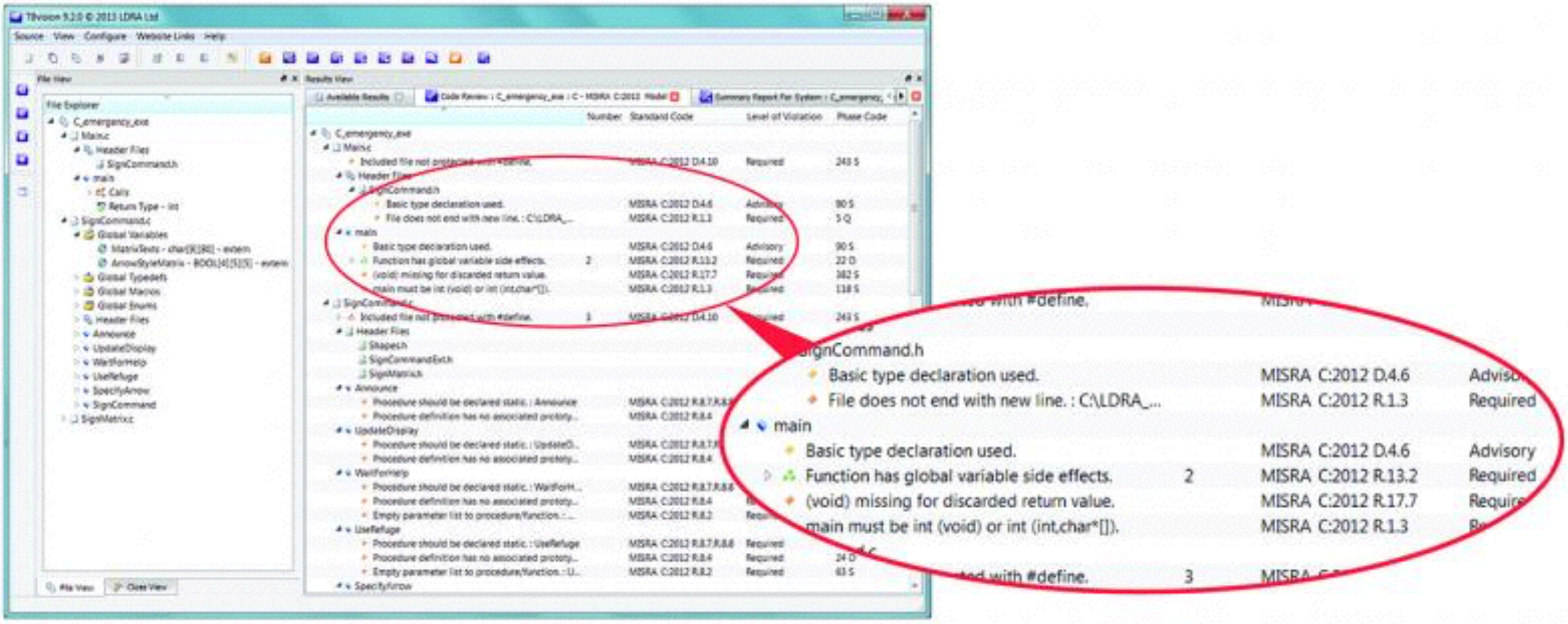

MISRA-C

- MISRA: Motor Industry Software Reliability Association
- MISRA-C: Programming standard from automotive industry
  - Restriction on C programming language („language sub-set“)
  - Goal is to make code as predictable as possible for disperse teams
  - Set of rules for C programs (**mandatory**, **required**, **advisory**)
    - Strict **process** for dealing with ignored non-mandatory rules
    - Rules either enforced with code generation or static checking
    - First version created in 1998, related to C89 standard
    - Third version MISRA-C:2012, related to C99 standard
- Since 2008 also MISRA-C++ available

MISRA-C RULES

- Categories of problems being tackled
  - Common programming errors in C
  - Underspecified language aspects, interpreted by compiler vendors
  - Common misconceptions of C language properties
  - Compiler errors and runtime errors
- Targeting human developers
- Some problems never occur in code generators
  - Rule may even impact performance (http://bit.ly/1pGpHpo)
- In MISRA-C 2012, 27 undecidable rules
  - No possible to evaluate rule in each and every case

MISRA-C:TOOL SUPPORT



[electronicdesign.com]

SOME MISRA-C RULES

- No nested comments
- Inline functions instead of function macros
- No direct comparison of floating points
- No pointer arithmetic, no octals
- No recursion
- Memory shall only be freed if allocated by the application itself
- Use of typedef's (with size and signedness) instead of basic types

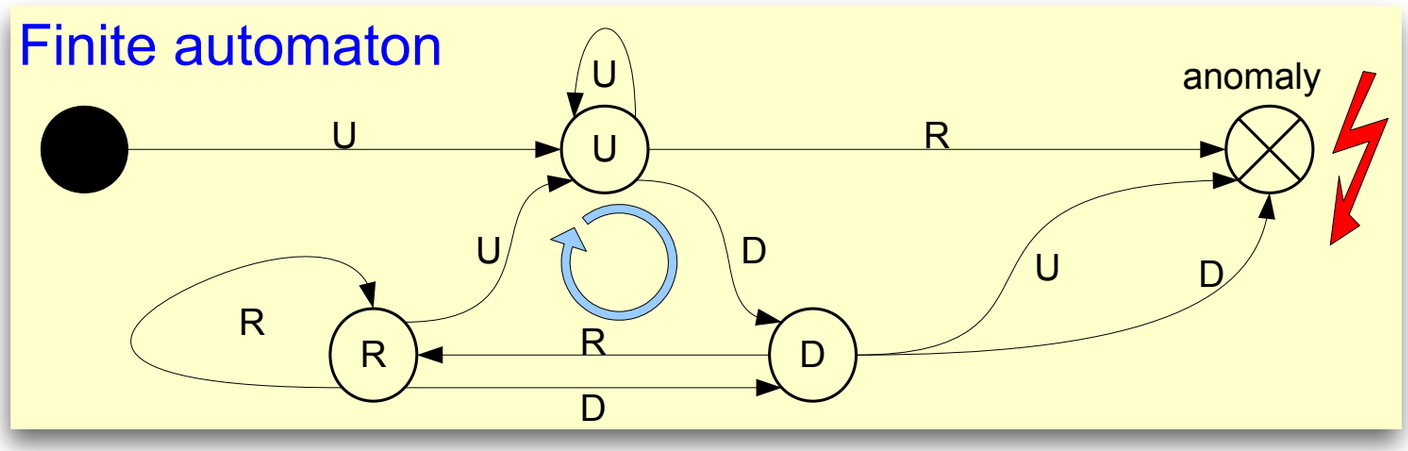
```
/*
... pages later...
foo();
/* ... */
... pages later...
*/
```

```
void fn ( void )
{
  int32_t a;
  free(&a);
}
```

```
line_a |= 256; /* sets bit nr. 8 */
line_b |= 128; /* sets bit nr. 7 */
line_c |= 064; /* wrongly sets bits 2,4 and 5 */
```


MISRA-C VARIABLES

- All variables shall have a defined value before being used
- No unused variables
- No non-volatile variables with only one use
- Variables may be **undefined (U), defined (D),** or **referenced (R)**
  - UR anomaly: Initialization required
  - DU anomaly: Set and discard
  - DD anomaly: No read between two assignments



CODE GENERATION

- Some MISRA-C rules less reasonable for generated code
  - „No use of compiler-specific language extensions“
    - Tries to ensure portable C Code
    - No need when C code is just an intermediate language
  - „Wrap assembler code in C functions“
    - Avoids erroneous utilization of macros in manual coding
    - Destroys performance advantage from inline assembler
  - „No pointer arithmetic“
    - Code generator can be expected to work correctly here
- Some rules can be enforced on model level instead (e.g. no recursion)

MISRA-C CRITICISM

- Experienced embedded C developers not always agree to MISRA-C
  - Rules try to protect from inexperienced C programmers
    - Those people shouldn't write safety-critical code anyway !
  - Micro-controller programming needs to deal with scare resources
    - *goto* can save a lot of resources, but is not allowed
    - Recursion can save a lot of resources, but is not allowed
    - Pointer arithmetic is very efficient, and even used in OS kernels
    - Dynamic memory allocation is not allowed, waste of resources
  - No `stdio.h` allowed, but may be useful

http://www.knosof.co.uk/misracom.html

MISRA-C RELEVANCE

"Nonetheless, it should be recognized that there are other languages available which are in general better suited to safety-related systems, having (for example) fewer insecurities and better type checking. Examples of languages generally recognized to be more suitable than C are **Ada** and Modula 2. If such languages could be available for a proposed system then their use should be seriously considered in preference to C."

Source:
"MISRA-C:1998 - Guidelines for the use of the C language in vehicle based software"

ISO26262

- Standard for functional safety in automotive systems

Methods		ASIL			
		A	B	C	D
1a	One entry and one exit point in subprograms and functions[a]	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation[a,b]	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names[a]	+	++	++	++
1e	Avoid global variables or else justify their usage[a]	+	+	++	++
1f	Limited use of pointers[a]	o	+	+	++
1g	No implicit type conversions[a,b]	+	++	++	++
1h	No hidden data flow or control flow[c]	+	++	++	++
1i	No unconditional jumps[a,b,c]	++	++	++	++
1j	No recursions	+	+	++	++

[a] Methods 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

[b] Methods 1g and 1i are not applicable in assembler programming.

[c] Methods 1h and 1i reduce the potential for modelling data flow and control flow through jumps or global variables.

NOTE For the C language, MISRA C[3] covers many of the methods listed in Table 8.