C language for DV

Intro to C language

Generic coding in C for SoC DV

ARM V8/64bit aspects

Torsten Jaekel, June 2014

Content

- Some C language features:
  - Recap types
  - Recap pointers
- "*Veeneers*" and alignment
- `struct` and message passing
- `volatile`
- Loops
- Code Image (startup)
- Registers
- Test Bench Ports

Introduction

- C is a *Higher Level Assembly Code* language
- C supports to be platform and processor independent
- C allows all fundamental operations which can be done with machine code (assembly instructions or small sequences of it)

- C does NOT support high level, very abstract operations, e.g. string operations, complex math operations, network access, graphics (widgets)
- Such support is based on *Libraries* (pre-compiled functions)
- Compilers support special instructions via "*Intrinsics*", e.g.

```
__wfi()
__sev()
```

Recap some C features

C supports non-complex data types:

```
(singed/unsigned) char, short, int
long, long long
float, double
enum
* Pointers  //to any type, incl. functions
```

- C supports structures and unions:

```
struct      //structured (a record) of members
union       //sharing same memory location
```

- C allows to define new types (like aliases):

```
typedef
```

- **`enum`** and **`typedef`** should be used more often.
- They support type-safe definition and usage, way better as macros.

Generic use of types - I

int, **long**, **long long** and **pointers** are platform dependent:

```
int          : 32bit on V7 and V8 (Atmel: 16bit)
long         : 32bit on V7 | 64bit on V8
* pointers   : 32bit on V7 | 64bit on V8
```

- **sizeof()** *is our friend for generic coding:*

```
sizeof(int)     = 4
sizeof(long)    = 4 or 8
sizeof(char *)  = 4 or 8     //figure out platform
```

```
int myIntArray[20];                        //just change size here
char myCharArray[sizeof(int) * 20];  //same size

for (i = 0; i < (sizeof(myIntArray) / sizeof(int)); i++) {
    myIntArray[i] = myCharArray[i];
}
```

Generic use of types - II

Use `typedef` for platform dependent types:

```
typedef unsigned int        UInt32;
typedef unsigned long long UInt64;
```

- Benefit:
- We can modify the real type just at one place (all code lines using it will follow automatically).
- A platform header file can act as single place to adapt.

```
long *myPtr;
long myVar;
myPtr = &myVar;
*myPtr = 0x12;          //4 bytes or 8 bytes written !
myPtr++;                //incremented by 4 or 8?
//better:
UInt32 *myPtr;
*myPtr++ = 0x12;        //guaranteed number of bytes written
```

Generic use of types - III

Use `enum` instead of macros:

```
typedef enum {
        CMD_0,
        CMD_1
} E_Commands;
```

- Benefit:
- We are forced to use the right type, e.g. on function calls.
- We avoid mistakes when using macros or hard-coded values.

```
int myEnumFunction(int i, E_Commands cmd)
{
    switch (cmd) {
            case CMD_0 : return i + 1;
            case CMD_1 : return i * 2;
            case 2     : return i << 1; //WARNING!
    }
    return 0;
}

myEnumFunction(10, CMD_1);              //compiler will check
myEnumFunction(10, (E_Commands)1);      //OK but not nice
```

Pointers

Pointers are powerful but risky ("side effects")

```
//call by reference:
void myPtrFunction1(int i, /*const*/ int *j, int *result)
{
    *result = i + *j;        //slow! side effect! 2 mem accesses!
}
//call by value:
int myPtrFunction2(int j, int j)
{
    return i + j;            //fast! (just registers used)
}

int j;
int result;
j = 20;                                  //additional instruction needed
myPtrFunction1(10, &j, &result);         //what if result is a
                                         //shared variable? (multi-core !)

result = myPtrFunction2(10, 20);
printf("%d", myPtrFunction2(10, j); //parameter is result a
                                         //from function call
```

Pointers - I

Be aware of *alignment*!

```
char myByteArray[20];

void myWordFillFunction(int *ptr, int val, int size)
{
    while (size--) {
            *ptr++ = val;
    }
}

//are we sure myByteArray is word aligned? No guarantee!
myWordFillFunction((int *)myByteArray, 0x11, 20);

//obviously wrong - alignment violation!
myWordFillFunction((int *)&myByteArray[1], 0x11,
                    sizeof(myByteArray) / sizeof(int) );
```

Pointers - II

Don't mix immediates (*values*) with pointers

```
UInt32 val;
char *myPtr;
char *byteVar;

myPtr = &byteVar;                    //OK on all platforms

if ((UInt32)myPtr > 0x80000000) {    //what on V8, 64bit?

void myFunction(UInt32 addr, UInt32 val)
{
    UInt32 *ptr = (UInt32 *)addr;    //maybe OK
    *ptr = val;
}
myFunction((Uint32)&byteVar, 0x11);  //Warning on V8,
                                     //pointer (address) truncation!
```

Pointers - III

How to cast pointers properly?

Use **union** to make it platform independent (also automatic endian)

```
typedef union {
    UInt32          Addr32bit;    //immediate value
    void            *ptrAddr;     //address (any pointer)
} U_ADDR_CAST;

char anyVar;
U_ADDR_CAST addrCast;             //helper union variable

addrCast.ptrAddr = &anyVar;              //put pointer in
if (addrCast.Addr32bit & 0x3) {          //take value out
    //it was not 32bit word aligned
```

Veeners and Alignment

A *veener* is an (automatically) generated auxiliary code for "long jumps".

V8 (64bit) needs properly aligned *veeners*.

```
0x0000275c:    940014f5    ....    BL       $Ven$XX$L$$printf ; 0x7b30

$Ven$XX$L$$printf
        0x00007b30:    58000050    P..X    LDR      x16,{pc}+8 ; 0x7b38 ; [0x7b38] =
        0x00007b34:    d61f0200    ....    BR       x16
    $d
        0x00007b38:    34049810    ...4    DCD    872716304
        0x00007b3c:    00000000    ....    DCD    0
```

If the instruction code word cannot take the distance for a "*long jump*" the veener is needed.

V7 (32bit) assumes 4byte aligned, V8 (64bit) 8 byte alignment.

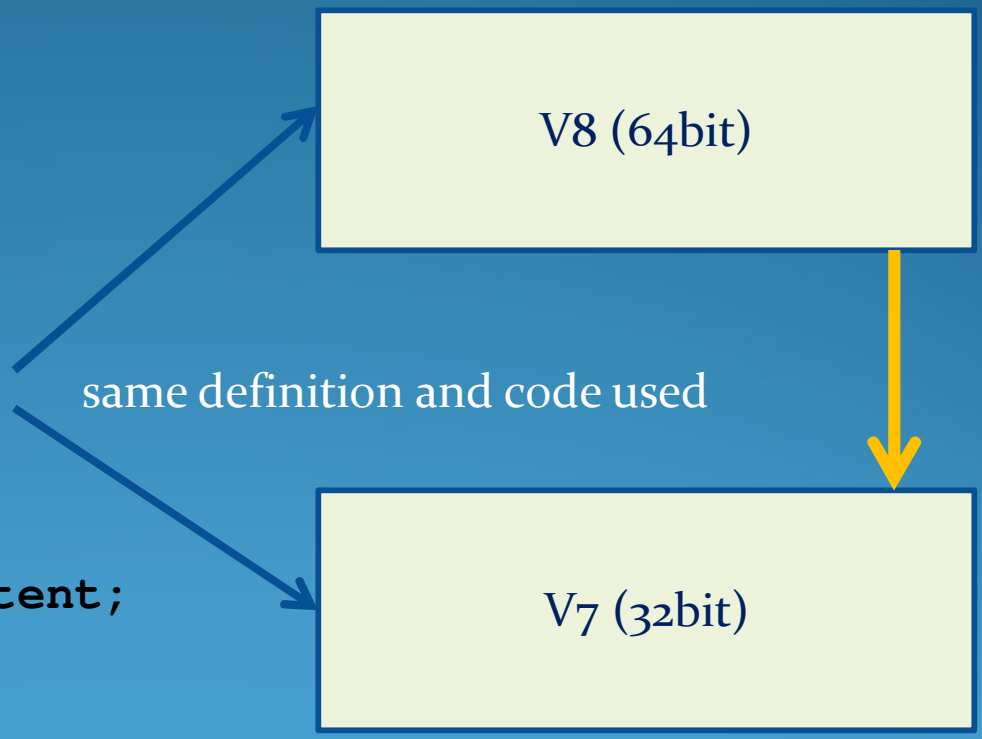
It might be necessary to force proper alignment for *Veeners*. Align the "*literal pool*" via scatter file.

Structures - I

Be carful when structures (messages) are crossing boundaries and platforms

```
typedef struct {
    char    msgHdr;
    UInt32 *contentPtr;
} T_EXT_MSG;

UInt32 msgContent[20];
T_EXT_MSG msg;
msg.msgHdr = 0x11;
msg.contentPtr = msgContent;
IPC_Send_Msg(msg);
```

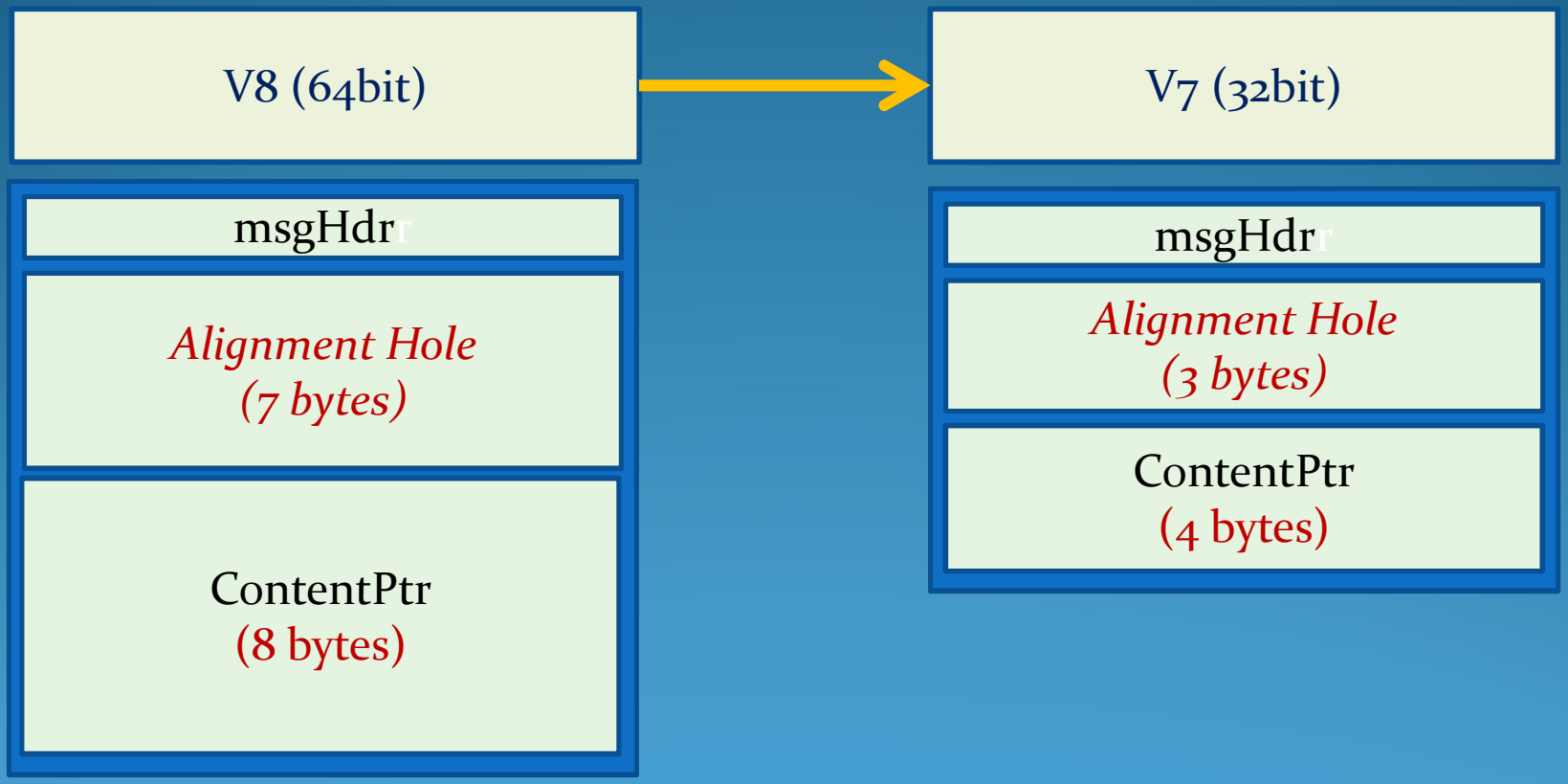


It compiles fine on both – but the message (as byte array) is NOT the same! – Alignments!

Structures - II

Members are aligned (platform dependent)!

Pointers are platform dependent!



Don't use pointers in messages crossing boundaries!

Use well-controlled elementaty types only!

Volatile

Be aware of using volatile

- volatile is used to tell compiler:
- A) do not optimize the use of variable
- B) assume two consecutive reads will give different results
- Compilers use a real memory as model with assumption: what was written there will be kept: reading it back results always in the same if not another write done.

```
int i = 10;
while (i < 10) {
    //do something
}

//optimized into:
int i = 10;
//do nothing
```

```
volatile int i = 10;
while (i < 10) {
        //wait for other core
}

//should not be optimized
//and loop should be done
```

volatile is needed on HW registers and shared variables for multi-core scenarios.

Loops

Be aware of `while{}` and `do-{}-while`

```
volatile int i;
```

```
while (i) {
    i = READ_REG(APB);
}

//randomly failing!
```

```
do {
    i = READ_REG(APB);
} while (i);

//works OK
```

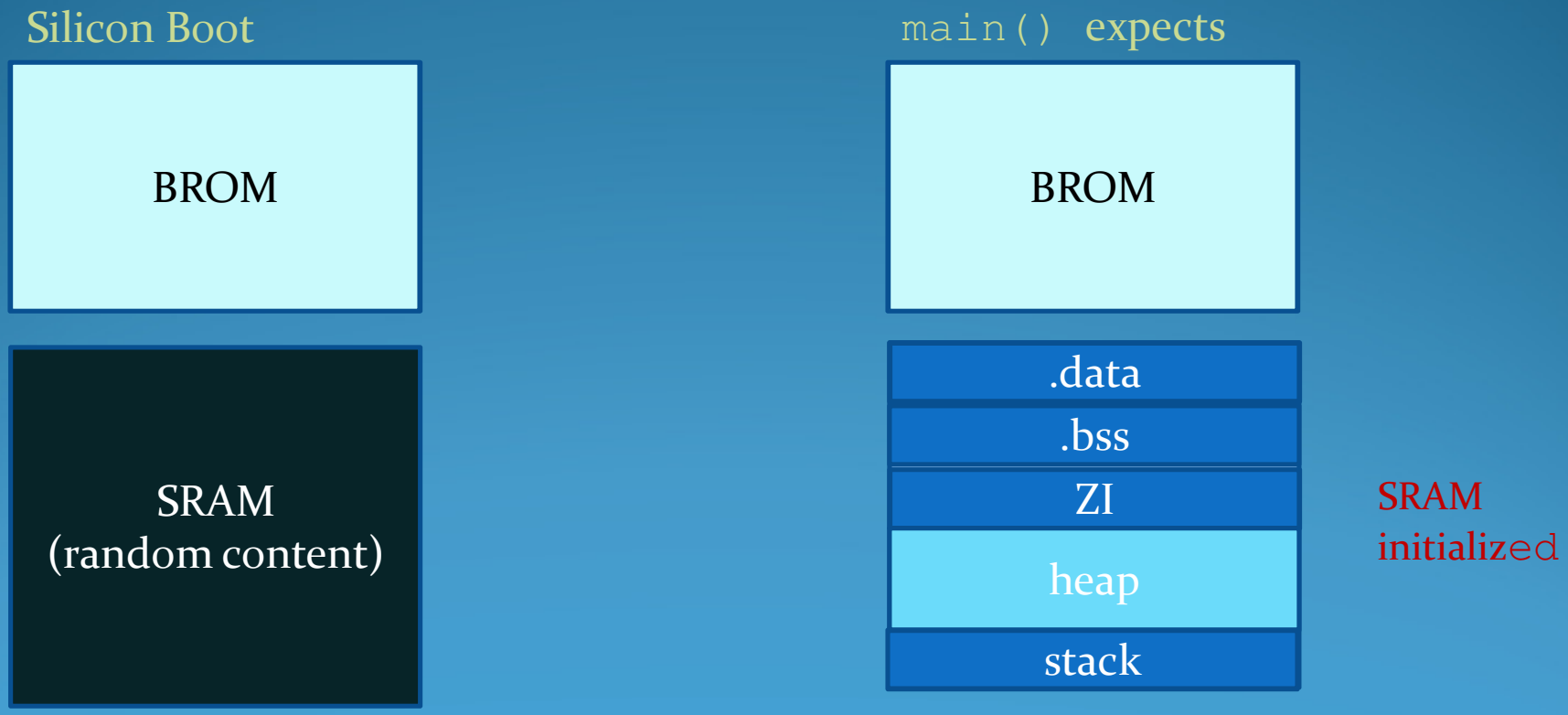
Local (type "auto") variables **are not initialized**!
Random values and possible to get i as 0 as well.

Use `while-{}` if zero-times done (skip) is a valid case.
If at least one iteration is needed – use `do-{}-while`.
(it saves also an initialization instruction)

Code Image - I

Our code starts at `main()` – what is expected?

- SRAM is initialized with global variables (.data)
- Static variables are initialized (.bss)
- Zero-Initialized data (ZI) in RAM is set to zero

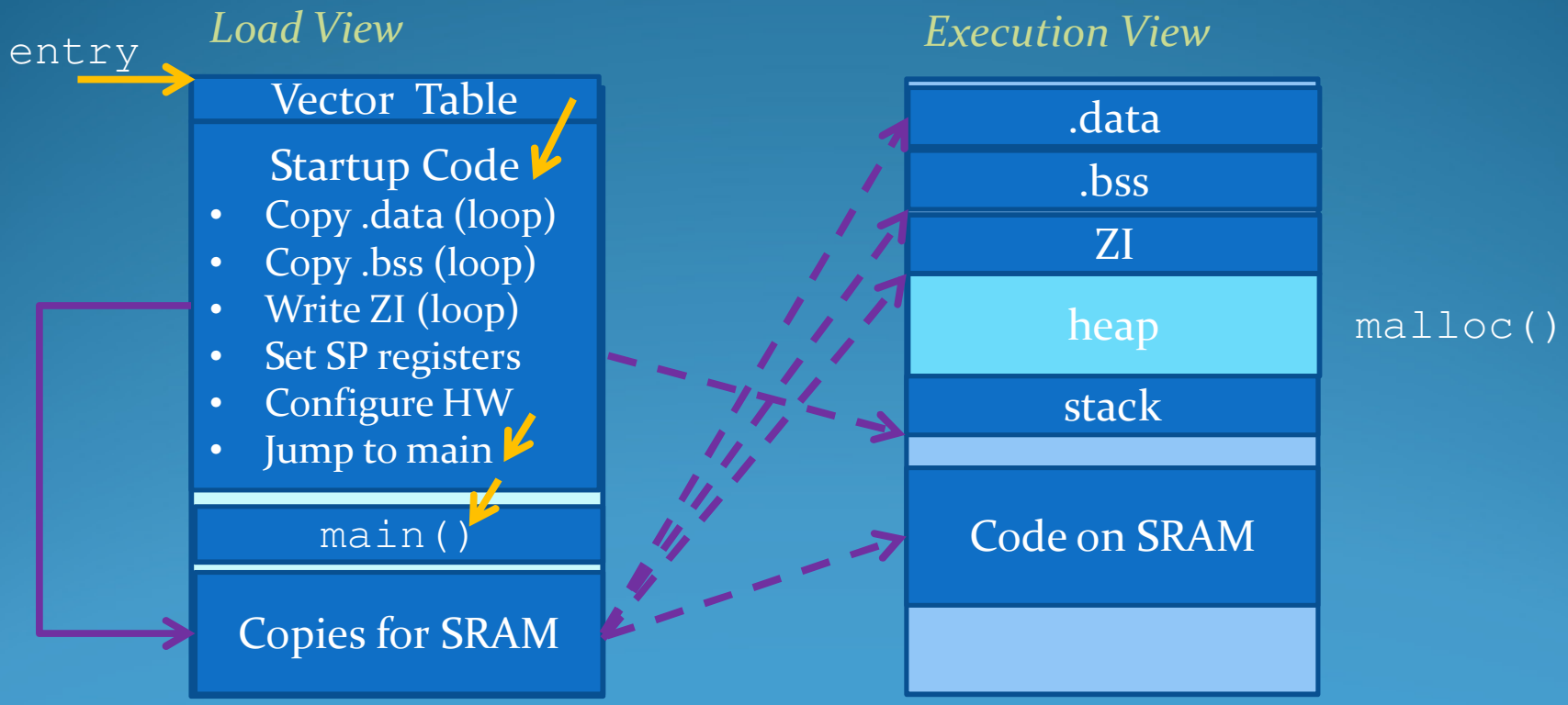


Startup code (before main) has to initialize memories

Code Image - II

Startup code will initialize system

- `scatterload`
- copy from ROM, iterate to write zero to SRAM

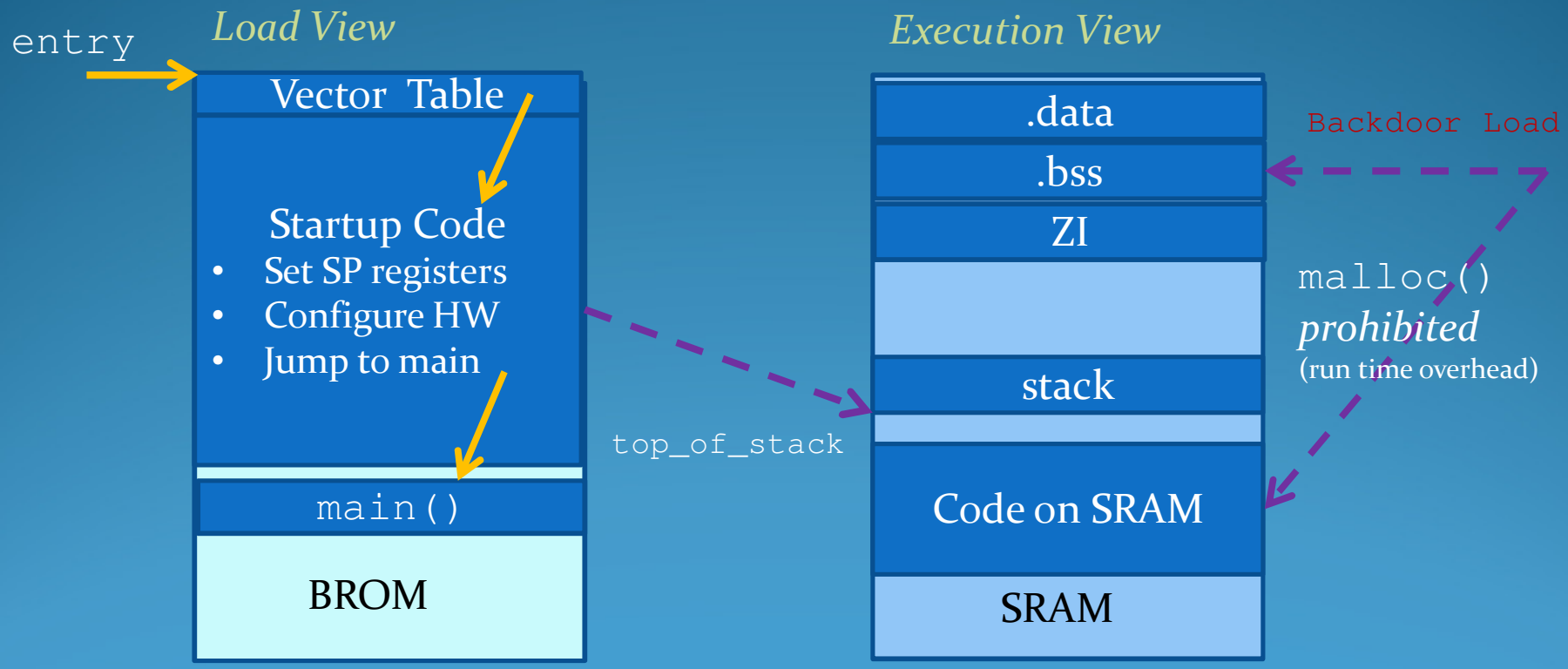


***Scatter File* (ARM) or *Linker Command File* (GNU) has a "*load*" and an "*execution view*", linker creates auxiliary code and "meta data".**

Code Image - III

Backdoor Load on DV shortens startup

- Initialize ALL memories for Execution View



Define *Scatter File* (ARM), *Linker Command File* (GNU) for "*Execution View*", parse AXF/ELF file and generate backdoor load images (*fromelf*)

HW Registers

DV or Linux like style?

- DV uses RDBs : based on macros and `#define`
- Linux (and embedded systems) prefer `struct` and "*device pointers*"

DV

```
#define APB1_base       0x34000000
#define APB2_base       0x34010000
#define reg0_offset     0
#define reg1_offset     4
#define WRITE_REG(block, reg, val)
           *((volatile UInt32 *)(block + reg)) = val)

WRITE_REG(APB1_base, reg0_offset, 0x11);
WRITE_REG(APB2_base, reg0_offset, 0x22);
```

Linux like device

```
typedef struct {
          volatile UInt32       reg0;
          volatile UInt32       reg1;
} T_APB_BLOCK;

#define APB1_base     0x34000000
#define APB2_base     0x34010000

T_APB_BLOCK *devicePtr;

devicePtr = (T_APB_BLOCK *)APB1_base;
devicePtr->reg0 = 0x11;

devicePtr = (T_APB_BLOCK *)APB2_base;
devicePtr->reg0 = 0x22;
```

If we have several instances of same block – how to write code flexible to deal with it (*code reuse*)?

TB Ports

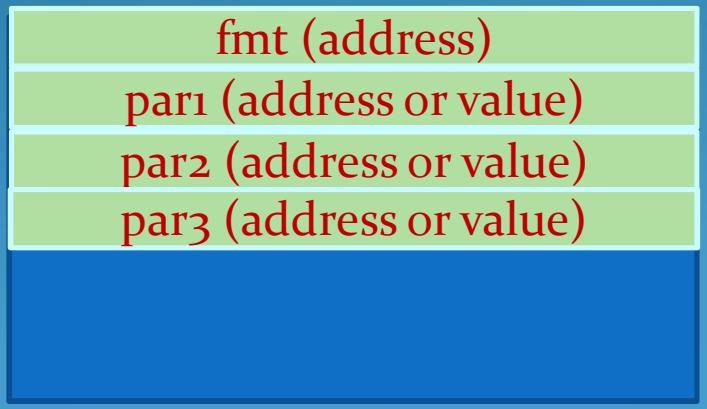
External TB ports are platform agnostic!

- **Example: `printf` implemented on TB**

V7 (32bit)

```
printf("%s : %d", "sub-string", 10);
```

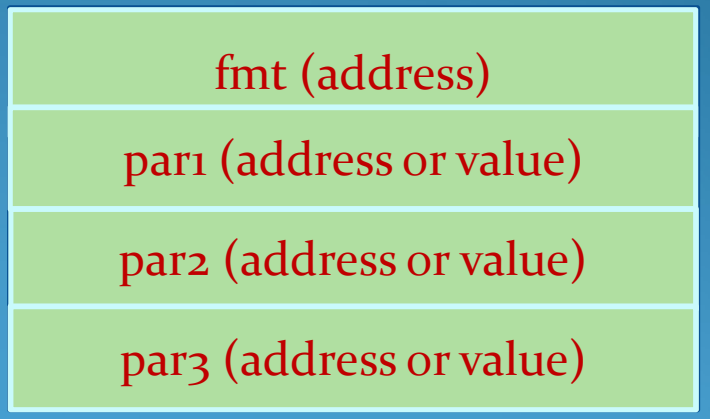
```
__asm printf(char *fmt, …) {
            //R0 : par0 : 32bit address to fmt
            //R1 : par1 : 32bit (an address here)
            //R2 : par2 : 32bit (value, here 10)
            //R3 : par3 : 32bit (value or address)
```



All as 32bit

V8 (64bit)

```
__asm printf(char *fmt, …) {
            //W0 : par0 : 64bit address to fmt
            //W1 : par1 : 64bit (an address here)
            //W2 : par2 : 64bit (value, here 10)
            //W3 : par3 : 64bit (value or address)
```



All as 64bit

TB has to know how to interpret parameters (32bit vs, 64bit)