

# Inhaltsübersicht

1. Einführung in Mikrocontroller
- 2. Der Cortex-M0-Mikrocontroller**
3. Programmierung des Cortex-M0
4. Nutzung von Peripherieeinheiten
5. Exceptions und Interrupts

# Kapitelübersicht

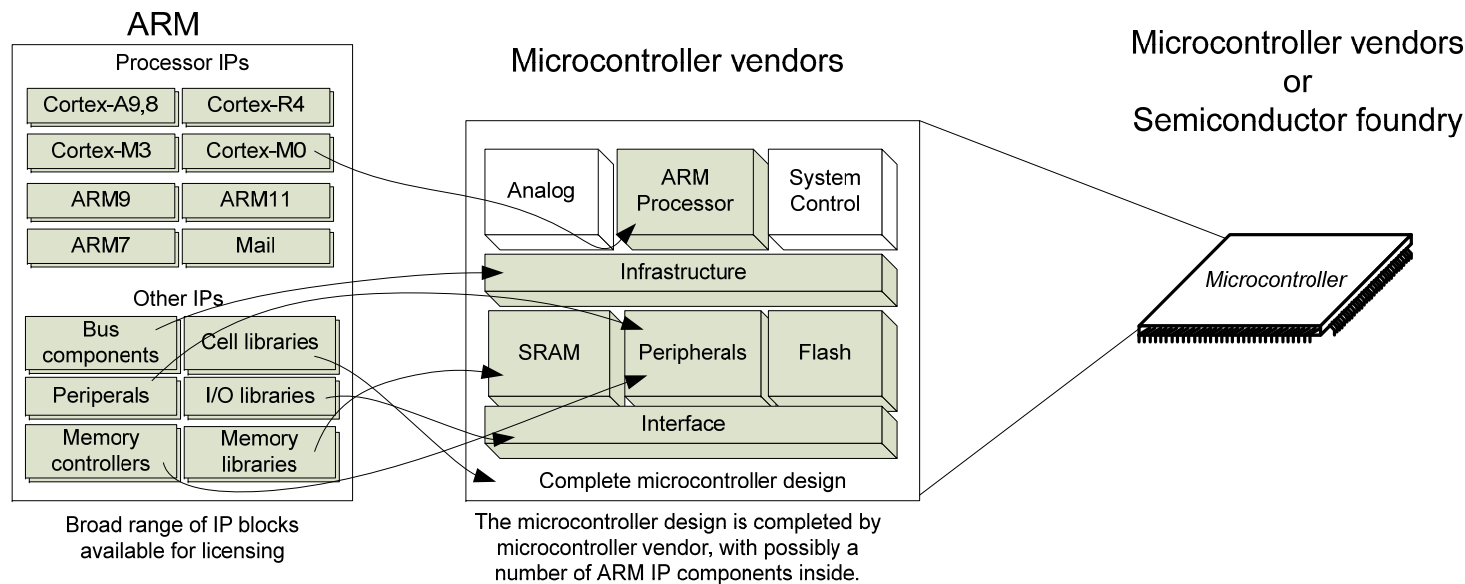
- I. Die ARM-Prozessoren
- II. Übersicht Cortex-M0
- III. Die Register des Cortex-M0
- IV. Die Speicherorganisation des Cortex-M0
- V. Die Startup-Sequenz des Cortex-M0
- VI. Stack und Heap im Cortex-M0

# Die Firma ARM

- Gegründet 1990 als Joint Venture von Acorn Computer, Apple Computer und VLSI Technology
- ARM steht für „Advanced RISC Machines“
- RISC steht für „Reduced Instruction Set Computer“ und kennzeichnet leistungsstarke Mikroprozessoren, in der Regel 32- oder 64-Bit-Prozessoren

# Das ARM Geschäftsmodell

- ARM stellt keine Chips her, sondern entwickelt Mikroprozessoren.
- ARM vergibt Lizenzen für die Mikroprozessoren an Chip-Hersteller (z.B. Texas Instruments, NXP, ST Microelectronics, NEC, Toshiba, Samsung)



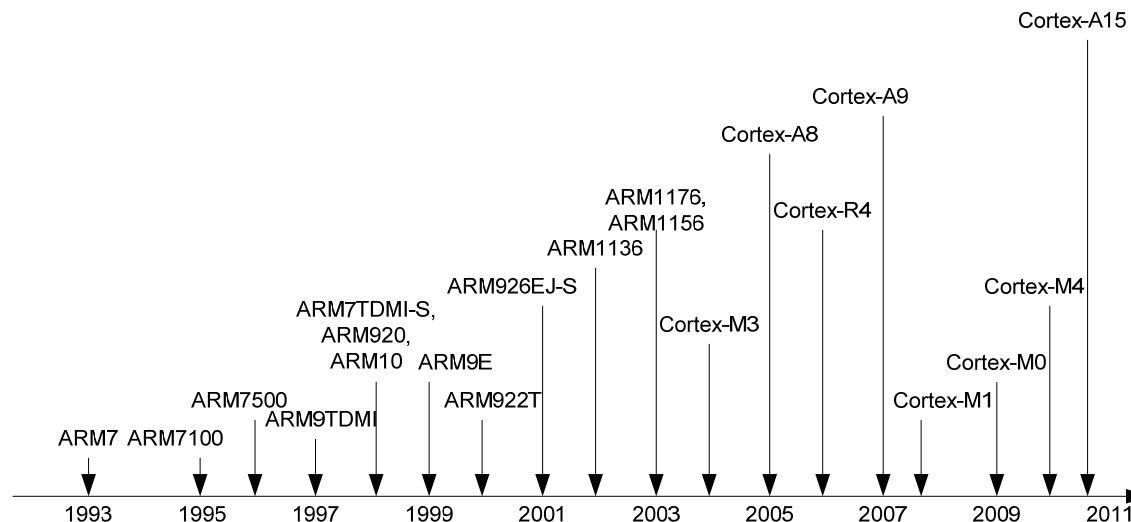
Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Die ARM Prozessoren

- Einer der erfolgreichsten Prozessoren ist der 1993 eingeführte ARM7. Er weist zum damaligen Zeitpunkt einige interessante Details auf:
  - Leistungsfähige 32-Bit RISC-Architektur
  - Bedingte Befehlsausführung
  - Befehlssatz besteht aus 32-Bit Instruktionen und 16-Bit Instruktionen („Thumb-Mode“)
- Der ARM7 wird aufgrund seiner geringen Leistungsaufnahme insbesondere in mobilen Geräten wie z.B. Mobiltelefonen eingebaut.

# Chronologie der ARM-Prozessoren

- Im Laufe der Zeit entwickelt ARM sowohl High-End-Prozessoren (z.B. für Smartphones) also auch Low-Cost-Prozessoren für preissensitive Mikrocontroller-Anwendungen.



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Beispiel: Samsung Galaxy SIII



Bildquelle:  
[www.chipworks.com](http://www.chipworks.com)



Samsung Application Processor  
Exynos 4412, ARM Cortex-A9 QuadCore

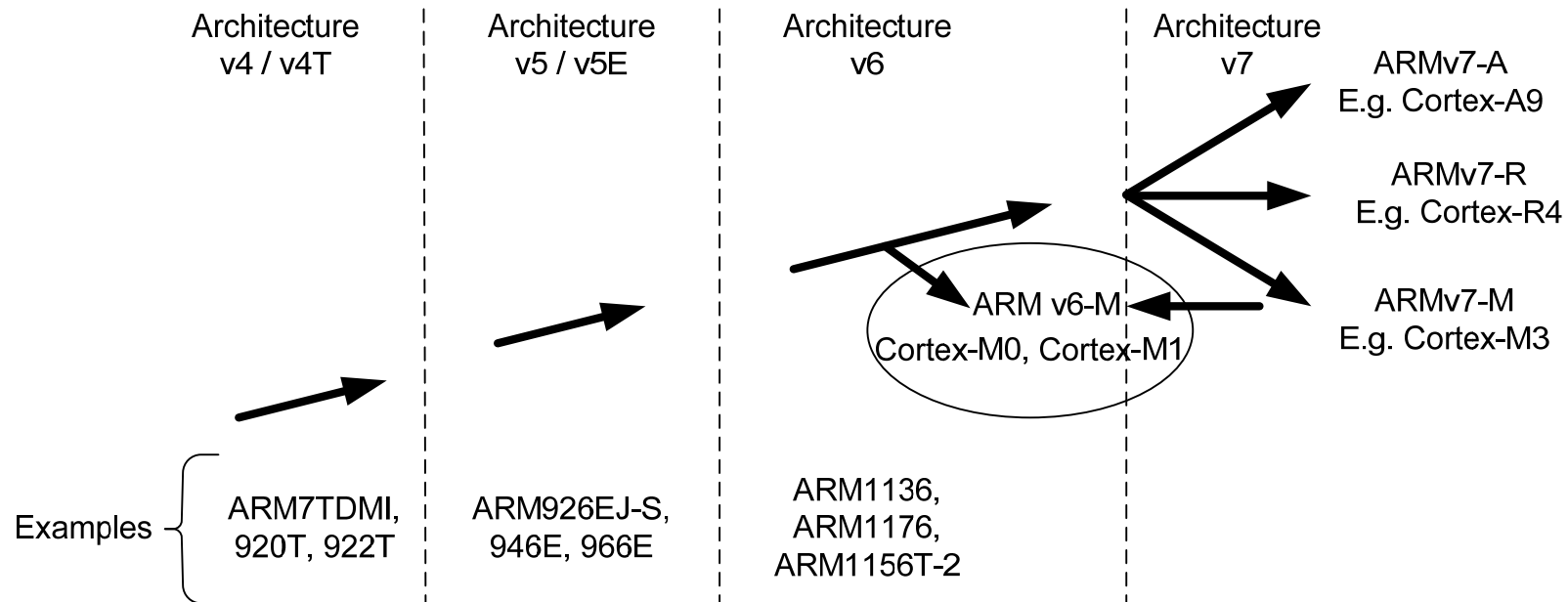
# Architektur – Prozessorkern - Chip

- Jeder Prozessorkern, z.B. ARM7, Cortex-A9 oder Cortex-M0, beruht auf einer von ARM definierten Architektur (z.B. ARMv4, ARMv6, ARMv7)
- Ein Chip eines Herstellers, z.B. Samsung Exynos 4412, beinhaltet einen oder mehrere ARM-Prozessorkerne
- Neben den Prozessorkerne sind weitere Komponenten, wie das Bussystem, Speicher und Peripherieeinheiten auf den Chips integriert.



# ARM Architekturen

- Im Laufe der Zeit wurden bei ARM eine Reihe von Architekturen entwickelt



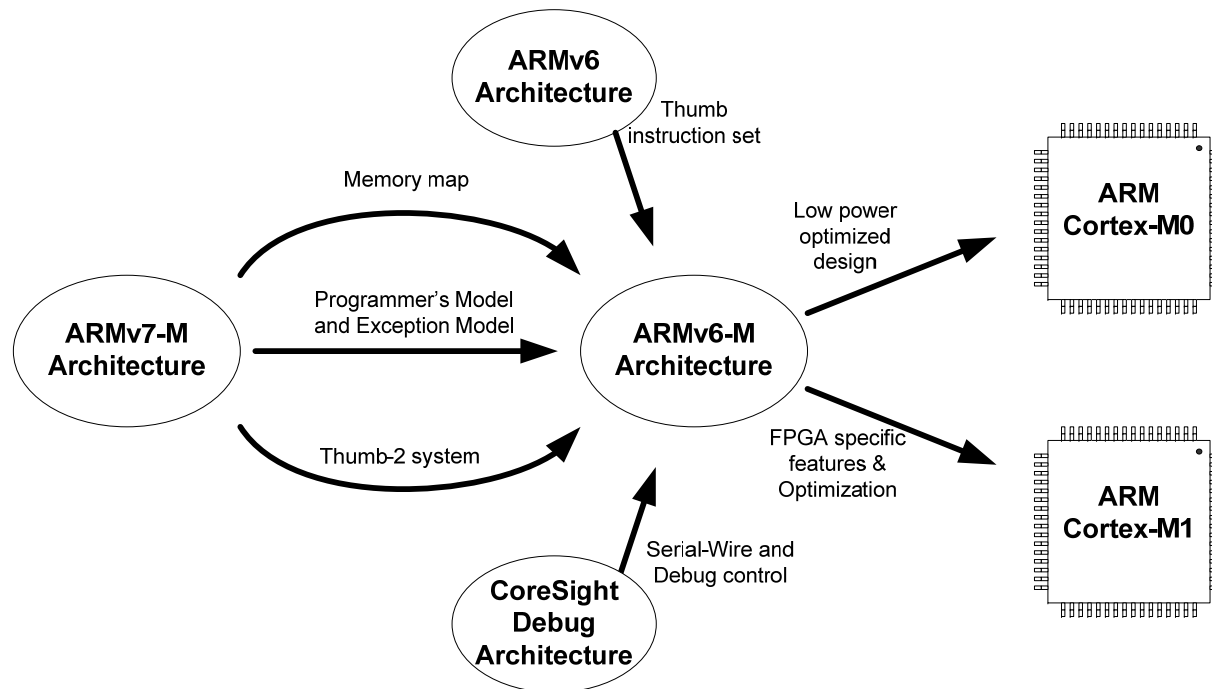
Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Die Cortex-Profile

- Um die steigende Diversifikation zu ordnen, definiert ARM mit der ARMv7-Architektur drei Anwendungs-„Profile“:
  - **A (Application Processor)**: Anwendungsprozessoren für leistungshungrige Anwendungen (z.B. Smartphones, Tablet-PCs)
  - **R (Realtime)**: Prozessoren für Echtzeit-Anwendungen wie Automobilelektronik
  - **M (Microcontroller)**: Prozessoren für Low-Cost Mikrocontroller-Anwendungen

# Der Cortex-M0

- Ist der kleinste und einfachste Cortex-Prozessor
- Beruht auf der ARMv6-M-Architektur



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

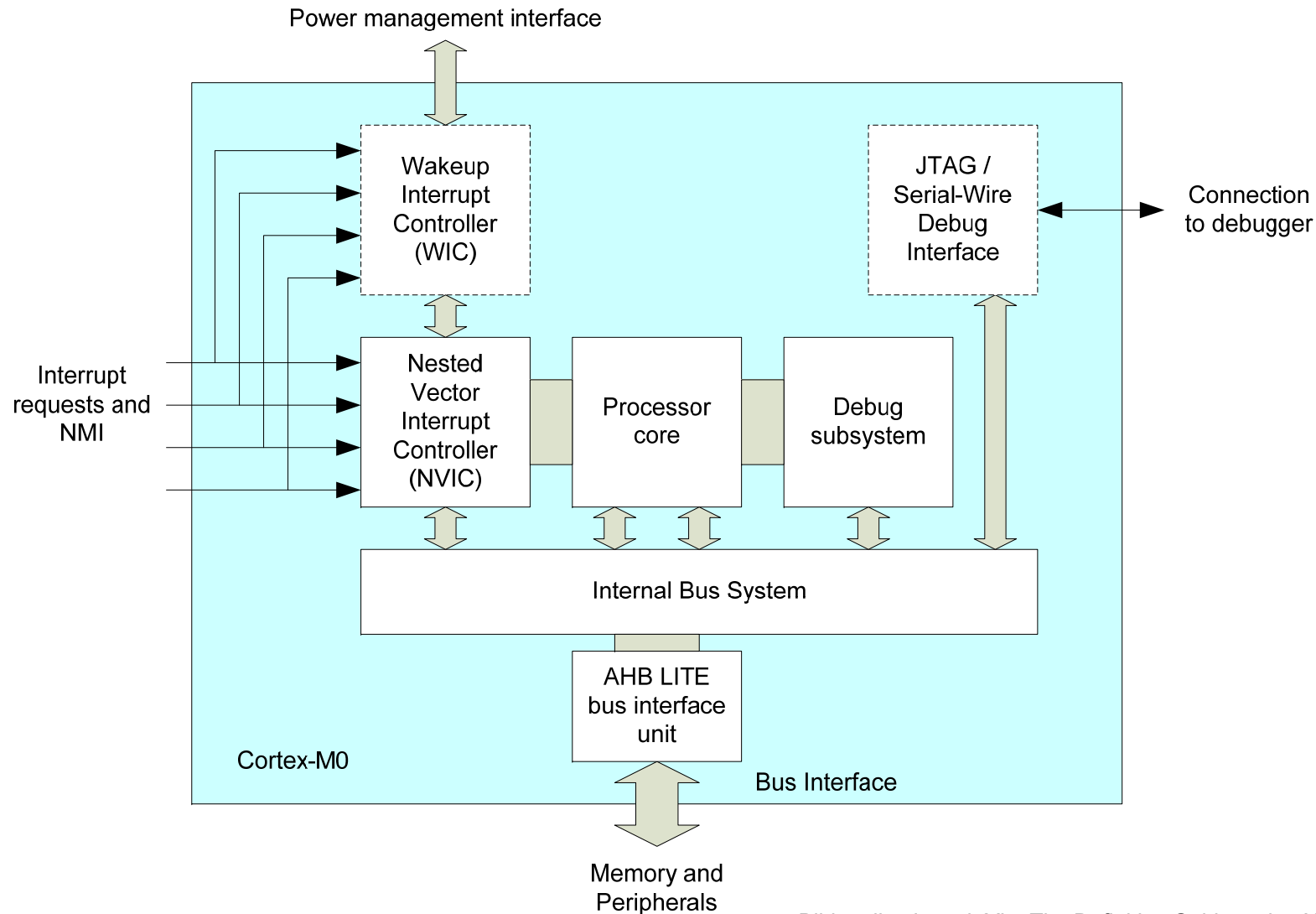
# Kapitelübersicht

- I. Die ARM-Prozessoren
- II. Übersicht Cortex-M0**
- III. Die Register des Cortex-M0
- IV. Die Speicherorganisation des Cortex-M0
- V. Die Startup-Sequenz des Cortex-M0
- VI. Stack und Heap im Cortex-M0

# Cortex-M0: Merkmale

- 32-Bit RISC Prozessorkern
- Von-Neumann-Speicherarchitektur
- 56 Maschinenbefehle
  - 16-Bit Instruktionen
  - 32-Bit Instruktionen
- „Load-Store“-Architektur
- Interrupt-Controller (NVIC)
- Debug-Subsystem
  - Steuerung des Debugging, Einzelschrittausführung etc.
  - Realisierung von Haltepunkten (Breakpoints) etc.
  - Kommunikation mit dem Entwicklungsrechner über serielle Verbindung (JTAG)

# Cortex-M0 Blockdiagramm

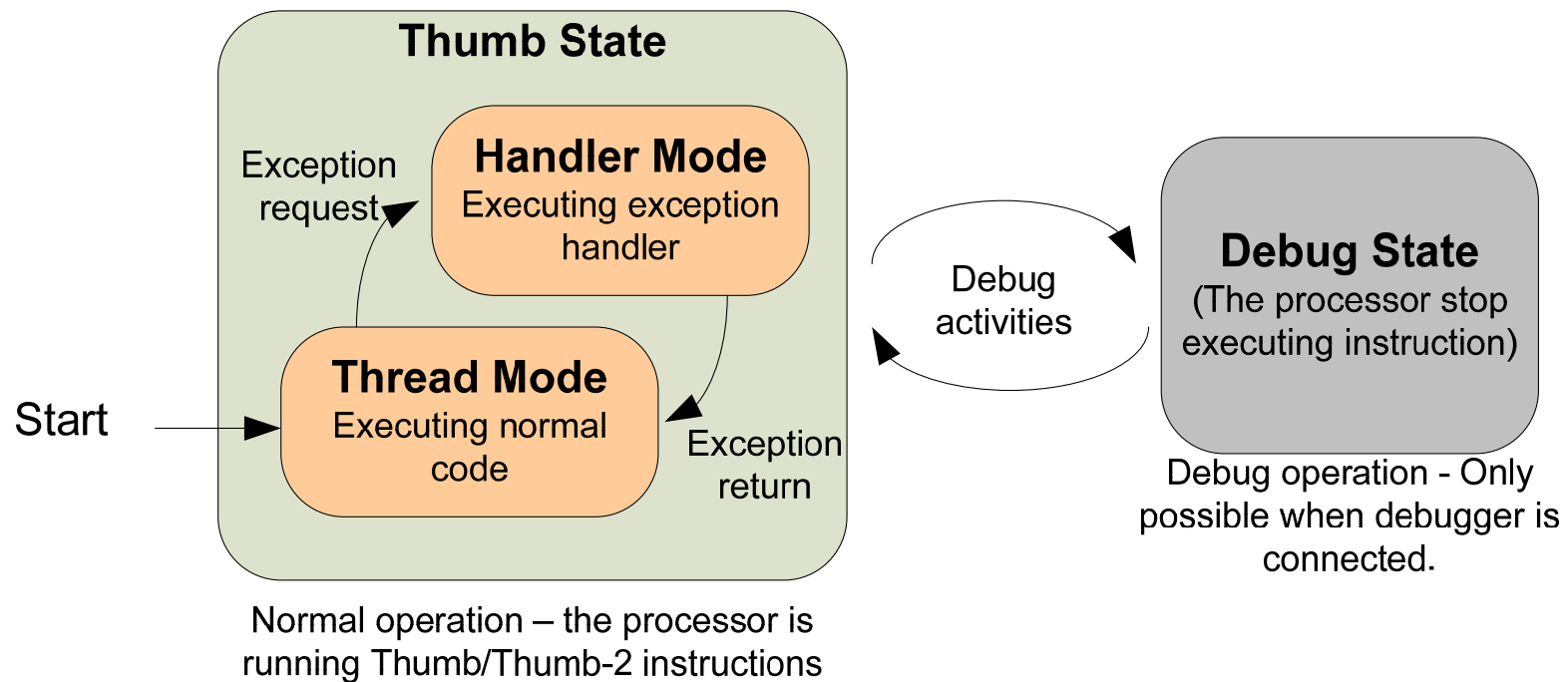


Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Betriebsmodi und Zustände

- Der M0 kann in zwei Zuständen sein:
  - „Thumb-State“: Der Prozessor führt ein Programm aus.
  - „Debug-State“: Der Prozessor wird angehalten und der Debugger kann auf Register des M0 zugreifen.
- Im „Thumb-State“ kann der Prozessor in einem von zwei Modi sein:
  - „Thread-Mode“: Wird für die normale Programmausführung benutzt.
  - „Handler-Mode“: Wird für die Ausführung von Ausnahmebehandlungsroutinen (Interrupt- oder Exception-Handler) benutzt.

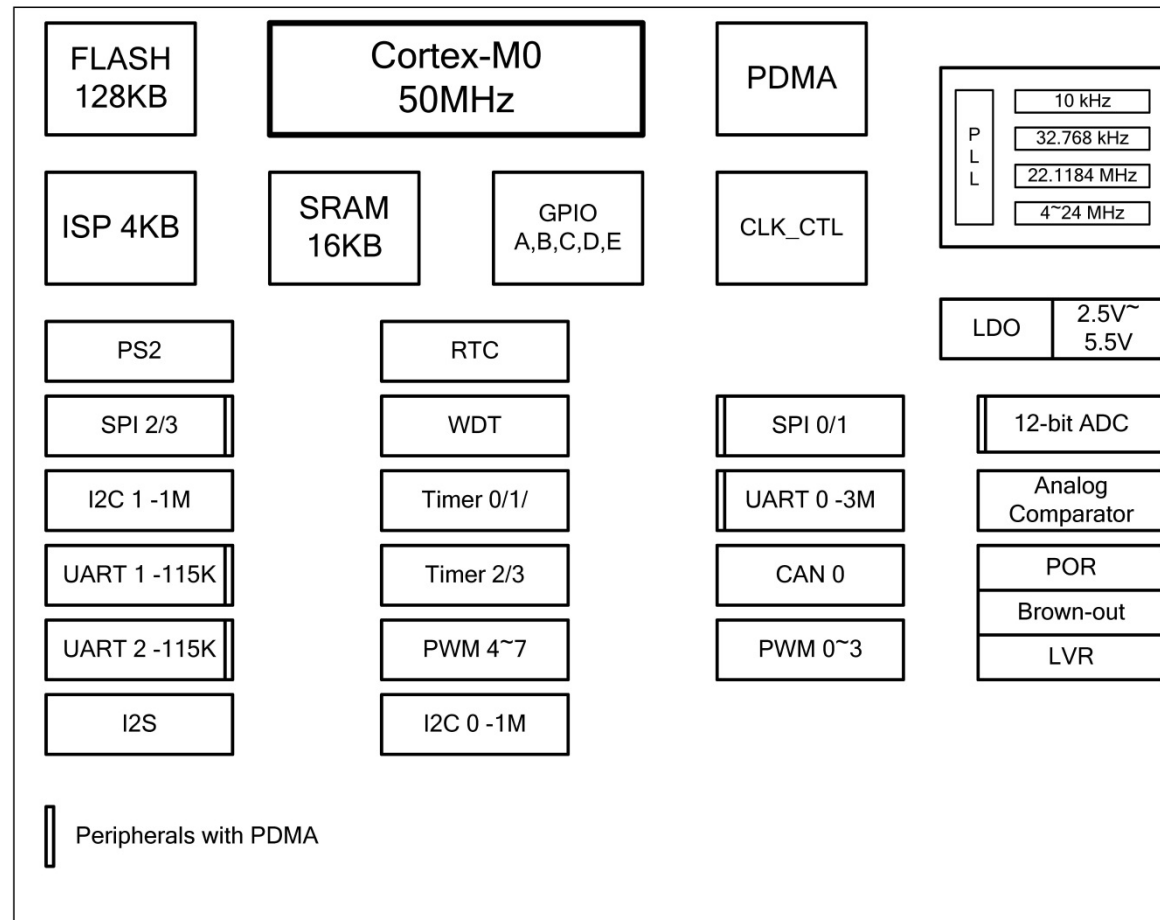
# Betriebsmodi und Zustände (2)



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0



# System mit Cortex-M0: Nuvoton NUC130

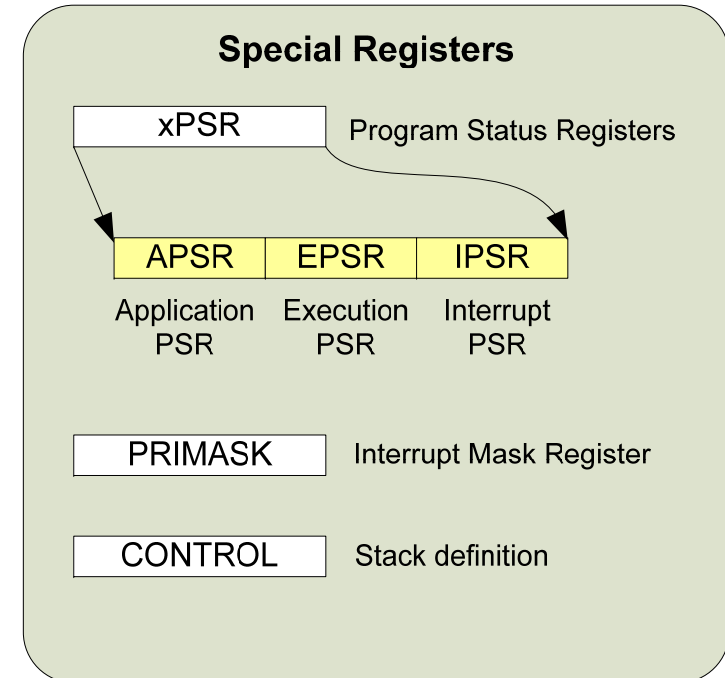
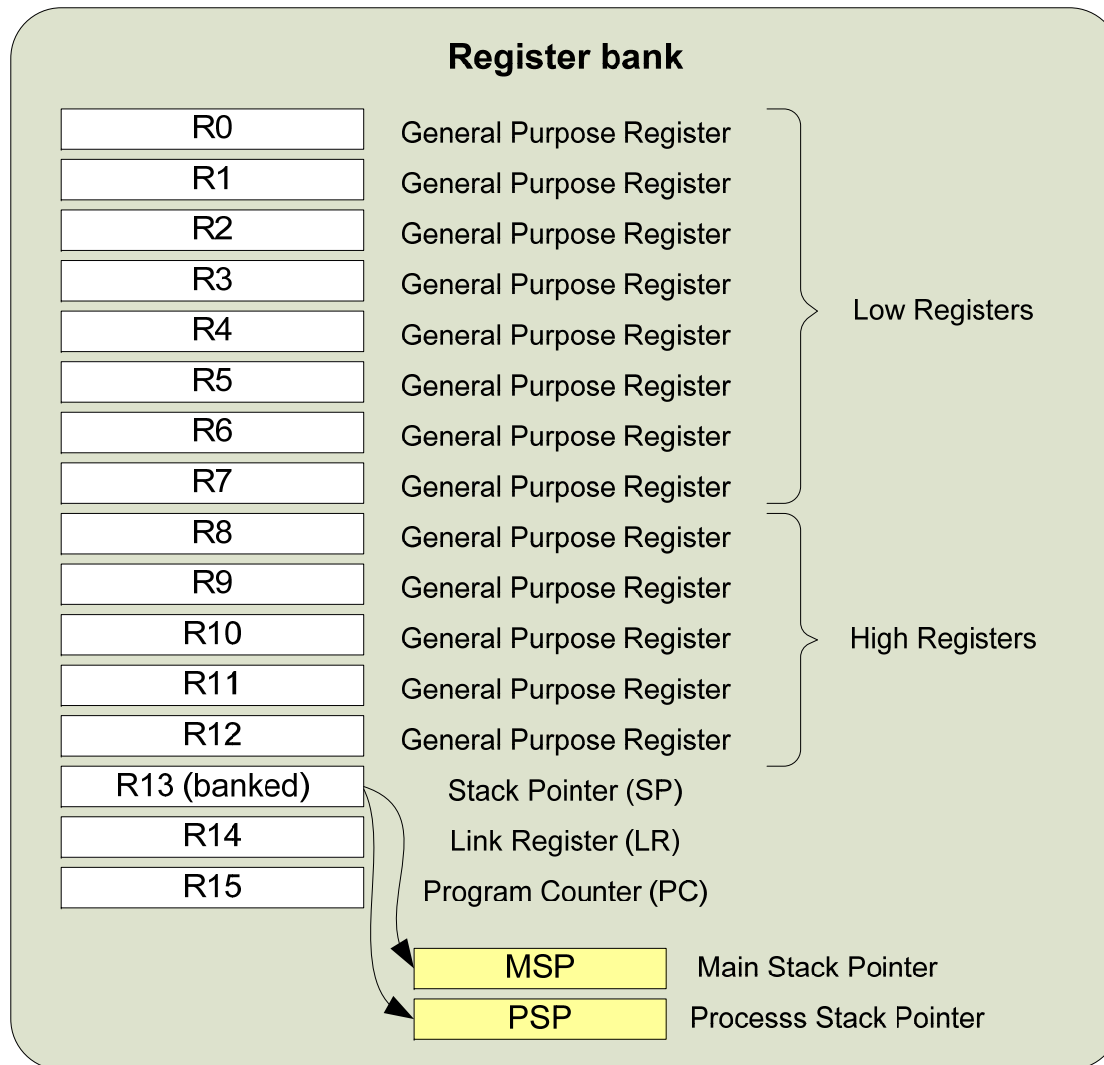


Bildquelle: Nuvoton Datenblatt NUC130

# Kapitelübersicht

- I. Die ARM-Prozessoren
- II. Übersicht Cortex-M0
- III. Die Register des Cortex-M0**
- IV. Die Speicherorganisation des Cortex-M0
- V. Die Startup-Sequenz des Cortex-M0
- VI. Stack und Heap im Cortex-M0

# Registerübersicht M0



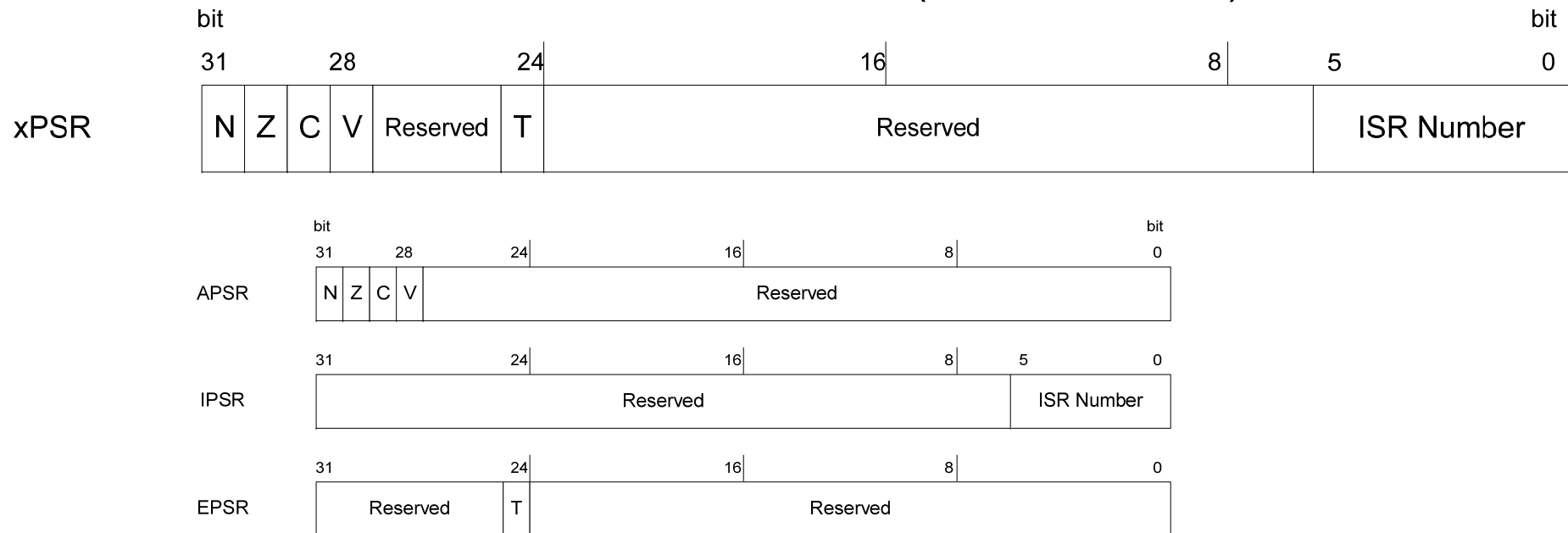
Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Die Register-Bank

- Der M0 verfügt über 16 Register (Load-Store-Maschine!):
  - R0 – R12: Arbeitsregister („general purpose“)
  - R13 – R15: Register mit speziellen Funktionen
- R13: „Stack Pointer“
  - Beinhaltet die Adresse des Stacks
  - Sind tatsächlich zwei unterschiedliche Register
    - MSP: Main Stack Pointer
    - PSP: Process Stack Pointer, für Betriebssysteme notwendig
- R14: „Link Register“
  - Speichert die Rückkehradresse bei Funktionsaufrufen
- R15: „Program Counter“
  - Zeigt auf die nächste auszuführende Instruktion

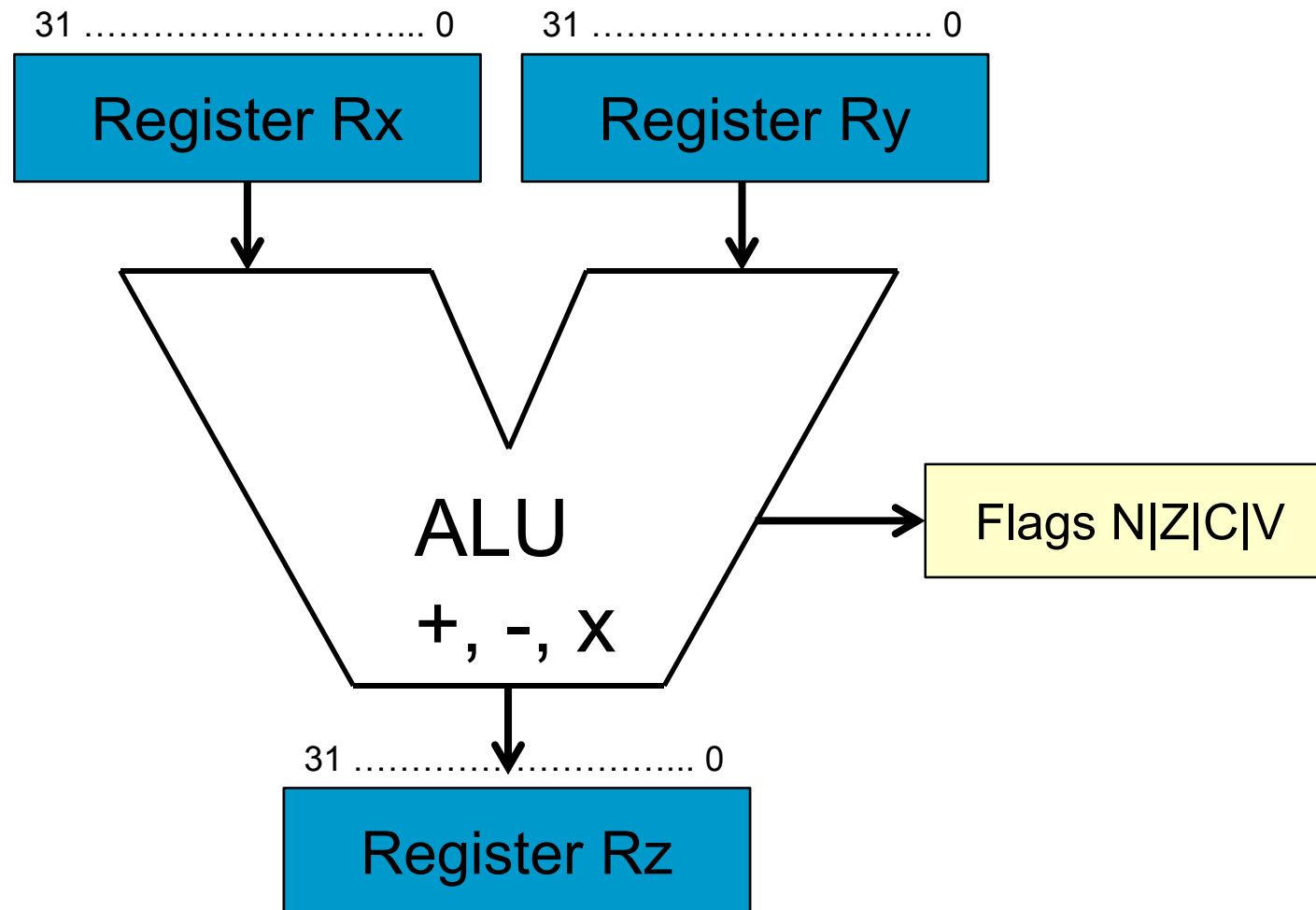
# Das Status Register

- Das „Program Status Register“ xPSR besteht aus drei Teilen:
  - APSR: „Flags“ der ALU (Negative, Zero, Carry, Overflow)
  - IPSR: Zeigt die Nummer des aktuell ausgeführten Interrupts an.
  - EPSR: T-Bit, ist beim M0 immer 1 (= Thumb-State)



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Die ALU



# Zahlenbereiche bei 32-Bit Daten

- Die ALU verarbeitet 32-Bit Daten:
  - Vorzeichenlos (C: unsigned long int)
  - Vorzeichenbehaftet (C: signed long int)
- Zahlenbereich „unsigned long int“:
  - Größte Zahl:  $2^{32}-1 = 4.294.967.295 = 0xFFFFFFFF$
  - Kleinste Zahl: 0
- Zahlenbereich „signed long int“:
  - Größte Zahl:  $2^{31}-1 = 2.147.483.647 = 0x7FFFFFFF$
  - Kleinste Zahl:  $-2^{31} = -2.147.483.648 = 0x80000000$

# C-Datentypen und Cortex-Prozessoren

- Neben 32-Bit Integer können auch die anderen C-Datentypen auf dem Cortex verwendet werden.
  - Die Verwendung von 8- oder 16-Bit Daten bietet allerdings keine Vorteile in der CPU, diese werden ebenfalls über die 32-Bit ALU und die Arbeitsregister verarbeitet.
  - Vorteil von kleineren Datentypen ist der geringere Speicherbedarf.
- Die Flags beziehen sich aber auf 32-Bit, bei den kleineren Integer-Datentypen (char, short int) werden die Flags i.d.R. nicht aktiv.
- Der M0 hat keine Floating-Point-Einheit und keinen Hardware-Dividierer. Entsprechende Operationen werden in Software realisiert, ebenso 64-Bit Operationen. Dies ist langsamer und benötigt mehr Programmcode.



# ALU Flags

- Die ALU verknüpft die Daten aus zwei Quell-Registern (Rx, Ry) und speichert das Ergebnis im Ziel-Register (Rz). Die Register sind 32 Bit groß.
- N(egative): Wird auf Bit 31 von Rz gesetzt. Das heißt, in Rz ist eine negative Zahl (2er-Komplement, K2), wenn  $N = 1$ .
- Z(ero): Wird auf 1 gesetzt, wenn  $Rz = 0$ .
- C(arry): Zahlenbereichsüberschreitung für vorzeichenlose Operationen. Addition:  $C = 1$  ist Überlauf; Subtraktion:  $C = 0$  ist Unterlauf („Borrow“, „low-aktiv“)
- V (Overflow): Zahlenbereichsüberschreitung für vorzeichenbehaftete Operationen (K2).

# Beispiel Flags

The screenshot shows a debugger interface with two main panes. The left pane, titled 'Registers', displays a list of registers and their values. The right pane, titled 'Disassembly', shows the assembly code for the current instruction. A yellow box is overlaid on the disassembly pane, containing a C code snippet with comments explaining the state of the flags.

Register	Value
R0	0x20000080
R1	0xFFFFFFFF
R2	0x00000001
R3	0x20000080
R4	0x00000444
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000444
R11	0x00000444
R12	0x00000000
R13 (SP)	0x20000470
R14 (LR)	0x00000157
R15 (PC)	0x000001A6
xPSR	0x61000000

Disassembly:

```
17: c = a + b; // 1. Überlauf
0x000001A4 188D ADDS r5,r1,r2
18:
0x000001A6
```

main.c

```
4 *
5 *
6 *
7 *
8 *****
9
10 int main (void) {
11
12     unsigned long int a, b, c;
13     signed long int x, y, z;
14
15     a = 0xFFFFFFFF;
16     b = 1;
17     c = a + b; // 1. Überlauf
18     a = 3;
19     b = 4;
20     c = a - b; // 2. Ergebnis negativ
21     b = 3;
22     c = a - b; // 3. Ergebnis null
23
24     x = 0x7FFFFFFF;
25     y = 1;
26     z = x + y; // 4. Überlauf (signed)
27 }
```

# Beispiel Flags

- 1. Überlauf (unsigned):
  - $a = 0xFFFFFFFF$ ,  $b = 1 \rightarrow c = 0$ ,
  - $Z = 1$  (weil  $c = 0$ ),  $C = 1$  (Überlauf)
- 2. Ergebnis negativ:
  - $a = 3$ ,  $b = 4 \rightarrow c = -1$
  - $N = 1$  (negativ),  $C = 0$  (Unterlauf, da unsigned)
- 3. Ergebnis null:
  - $a = 3$ ,  $b = 3 \rightarrow c = 0$
  - $Z = 1$  ( $c = 0$ ),  $C = 1$  (Subtraktion ohne Unterlauf)
- 4. Überlauf (signed):
  - $x = 0x7FFFFFFF$ ,  $y = 1 \rightarrow z = 0x80000000$
  - $V = 1$  (Überlauf),  $N = 1$  (negative Zahl)

# Überlauferkennung in C

- C bietet keine eingebauten Mechanismen für die Überlauferkennung oder den Zugriff auf Flags
- Lösungsmöglichkeiten:
  - Explizite Codierung der Überlauferkennung in C
  - Zugriff auf Flags in C über Compiler-spezifische Mechanismen oder spezielle Funktionen
  - Zugriff auf Flags über Assembler-Routinen
- Da dies einen gewissen Aufwand mit sich bringt, wird es i.d.R. nicht gemacht. Eine Bereichsüberschreitung sollte bei 32-Bit Zahlen nicht sehr wahrscheinlich sein.

# Weitere CPU Register

- PRIMASK: Besteht nur aus einem Bit (PRIMASK[0]). Wenn gesetzt (= 1), dann werden Interrupts blockiert (siehe Interrupts).
- CONTROL: Dient der Umschaltung zwischen Main Stack Pointer und Process Stack Pointer.

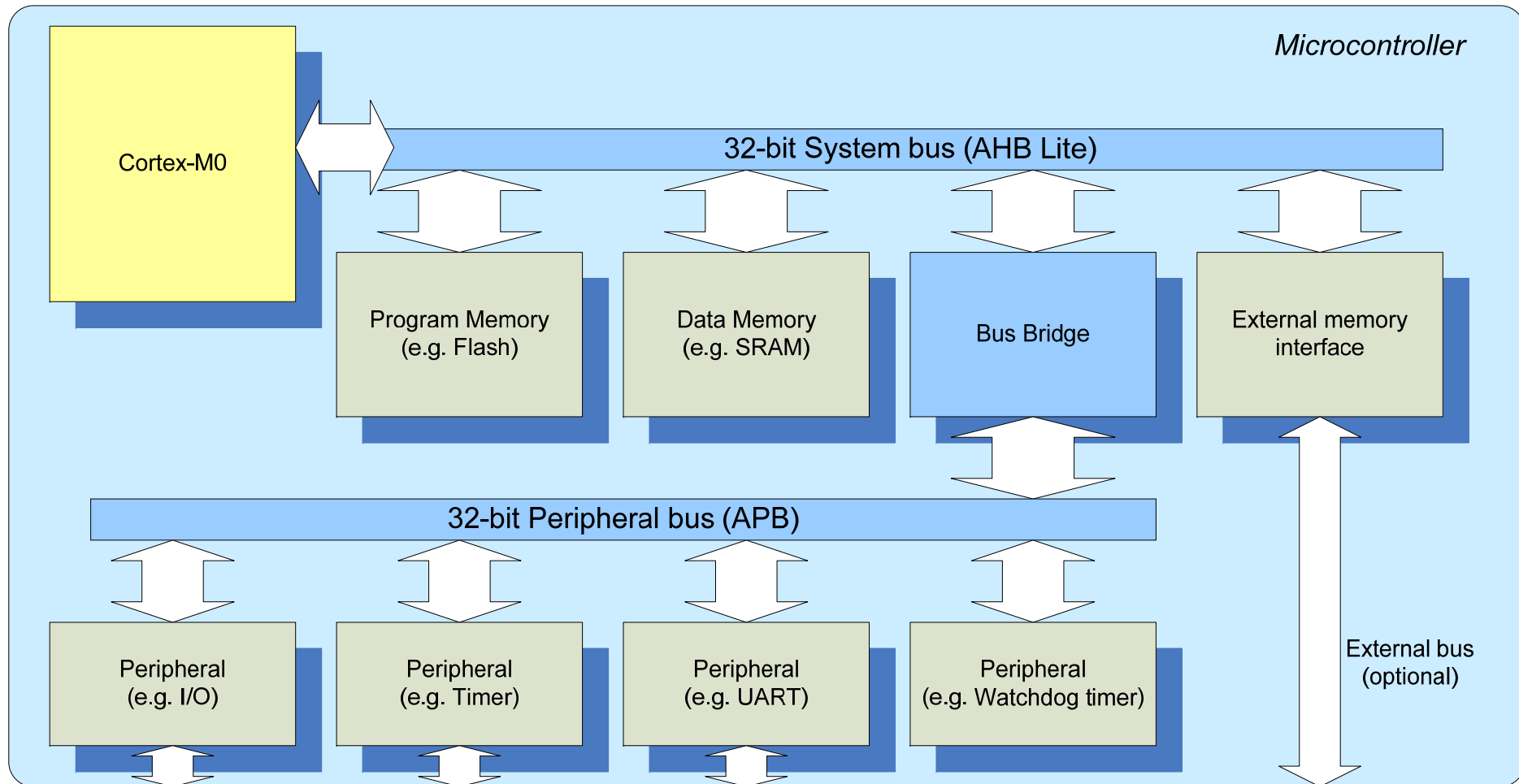
# Kapitelübersicht

- I. Die ARM-Prozessoren
- II. Übersicht Cortex-M0
- III. Die Register des Cortex-M0
- IV. Die Speicherorganisation des Cortex-M0**
- V. Die Startup-Sequenz des Cortex-M0
- VI. Stack und Heap im Cortex-M0

# Speicherübersicht

- Der Cortex-M0 hat einen linearen Adressraum von 4 GB ( $2^{32}$  Byte = 4.294967296 Byte)
- Der gesamte Adressraum ist in feste Bereiche eingeteilt, um die Portierung von Software zu erleichtern.
- Die M0-CPU verfügt über eingebaute Komponenten wie den Interrupt Controller (NVIC) oder die Debug-Einrichtungen.
  - Fester Adressbereich für alle M0-basierten Chips
  - Somit gleiches Programmiermodell für alle M0-Chips
- Die Chip-Hersteller nutzen die anderen Bereiche für die von Ihnen hinzugefügten Speicher- und Peripherieeinheiten (siehe z.B. Datenblatt Nuvoton NuMicro NUC130)

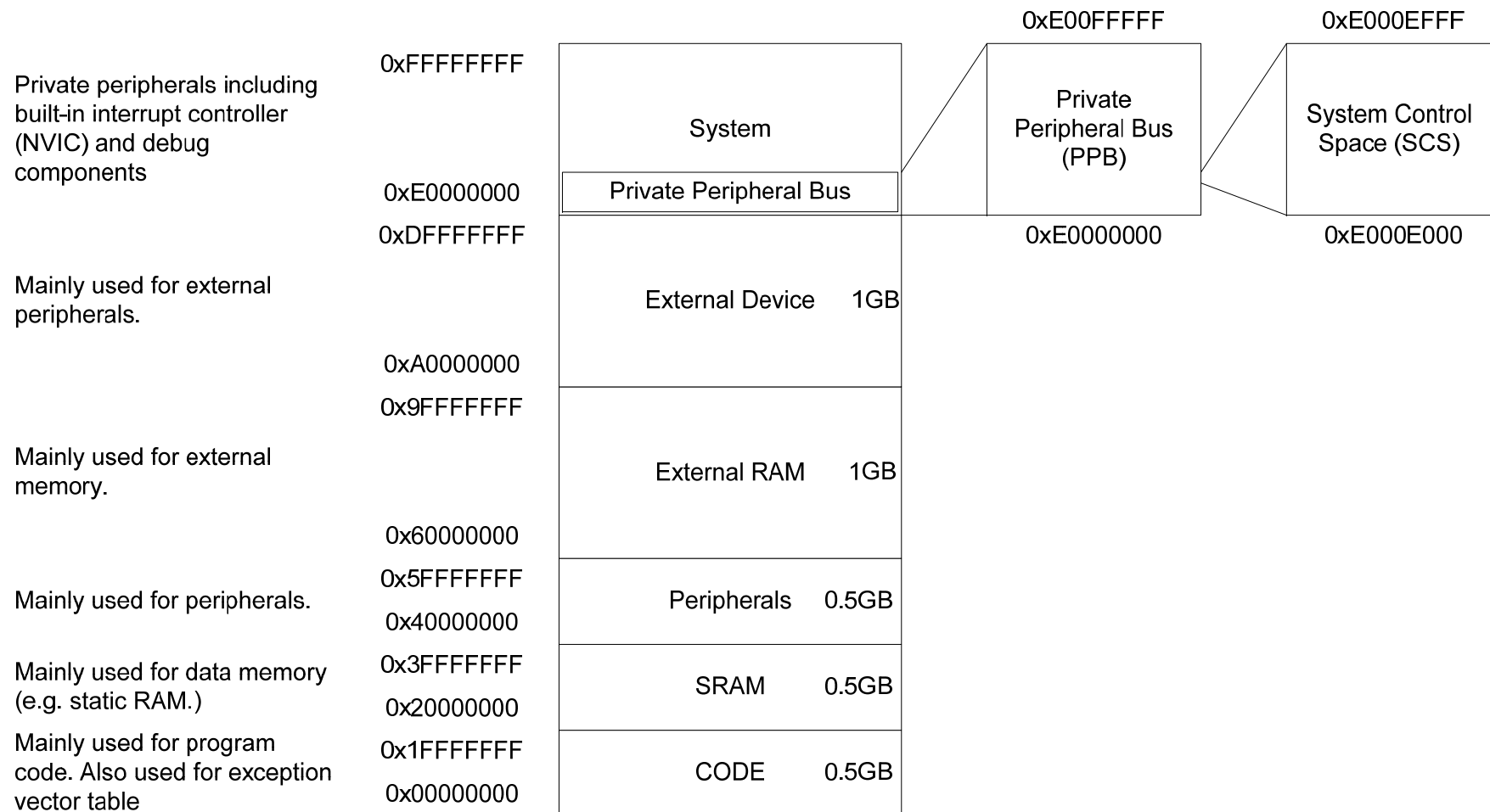
# Anschluss über Bussystem



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0



# Memory Map für Cortex-M0-Systeme



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Beispiel Memory Map des Nuvoton NUC130

Adressbereich	Einheit
0x0000_0000 – 0x0001_FFFF	Flash Memory (Programm), 128 KB
0x2000_0000 – 0x2000_3FFF	SRAM Memory (Daten), 16 KB
0x6000_0000 – 0x6001_FFFF	Externer Speicher, 128 KB
0x4000_4000 – 0x5001_03FF	Peripherie (siehe Datenblatt)
0xE000_E010 – 0xE000_ED8F	System Control Space

# Datengröße und Endianess

- Das Speichersystem erlaubt Transfers von Bytes (8-Bit), Half Words (16-Bit) und Words (32-Bit)
- Der Cortex kann vom Chip-Hersteller konfiguriert werden, so dass die Multi-Byte-Daten im Speicher im „Big Endian“- oder im „Little Endian“-Format gespeichert werden.
  - Der Nuvoton NUC130 arbeitet im Little-Endian-Format
- Beim Zugriff auf Peripherieregister muss auf die korrekte Datengröße und den entsprechenden Datentyp geachtet werden (i.d.R. 32 Bit).

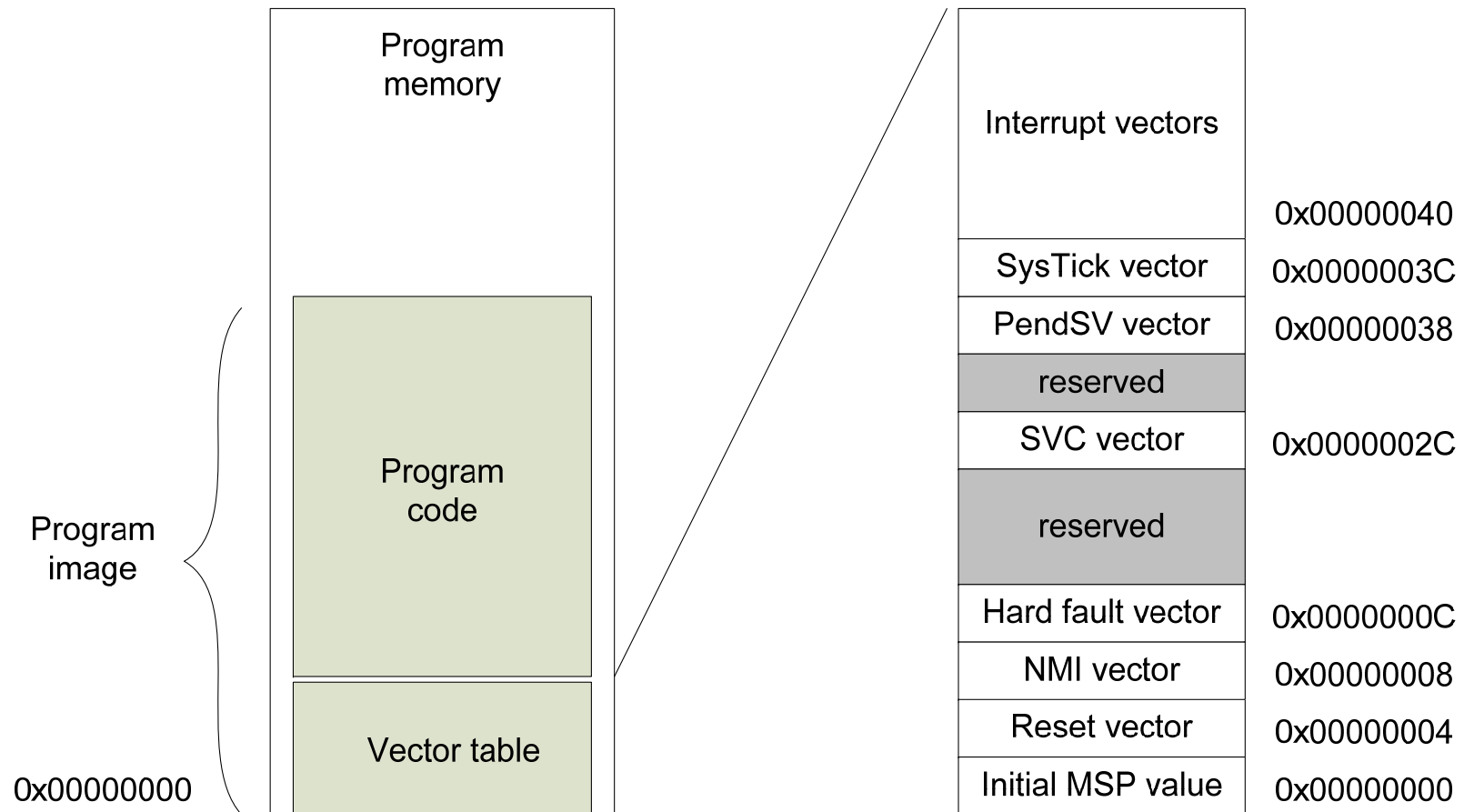
# Kapitelübersicht

- I. Die ARM-Prozessoren
- II. Übersicht Cortex-M0
- III. Die Register des Cortex-M0
- IV. Die Speicherorganisation des Cortex-M0
- V. Die Startup-Sequenz des Cortex-M0**
- VI. Stack und Heap im Cortex-M0

# Das „Programm-Image“

- Wir gehen davon aus, dass das Programm in den Flash-Speicher geladen wurde (wie, siehe später).
- Das zu ladende Programm-„Image“ besteht aus dem eigentlichen Programm und der „Vektortabelle“.
- Ein „Vektor“ beinhaltet die Anfangsadresse eines Interrupt Handlers (siehe später)
- Spezielle Vektoren:
  - 0x00000000: Anfangswert des Stack Pointers MSP
  - 0x00000004: „Reset Vector“, Anfangswert des PC

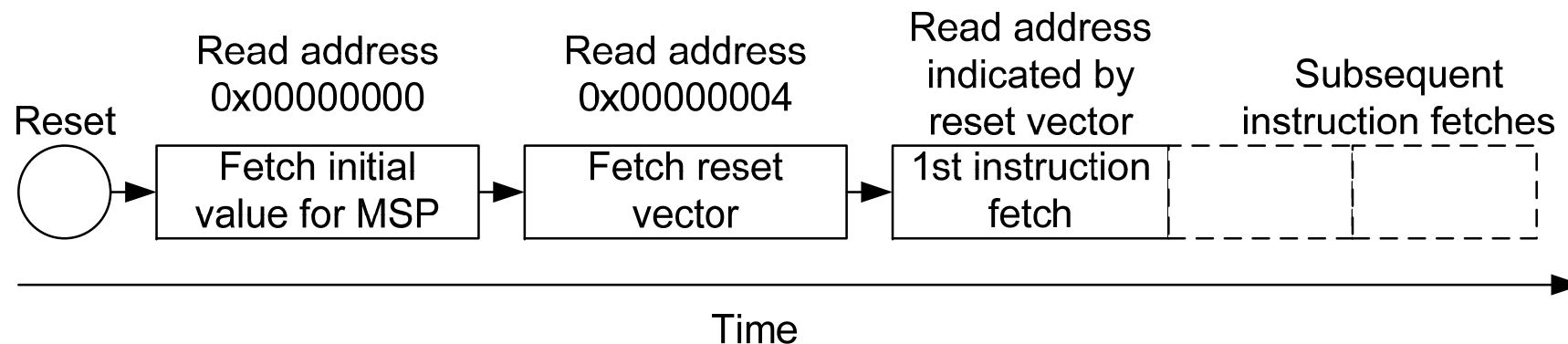
# Vektortabelle und Programm-Image



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Was passiert beim Reset des Prozessors?

- Wenn der Prozessor rückgesetzt wird (Reset, i.d.R. spezielle Taste am Board), dann führt er eine festgelegte Reset- oder Startup-Sequenz aus.
  - Der Stack Pointer wird mit seinem Anfangswert geladen.
  - Der PC wird mit dem Wert des Reset Vectors geladen und die Programmausführung beginnt an dieser Adresse.



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Kapitelübersicht

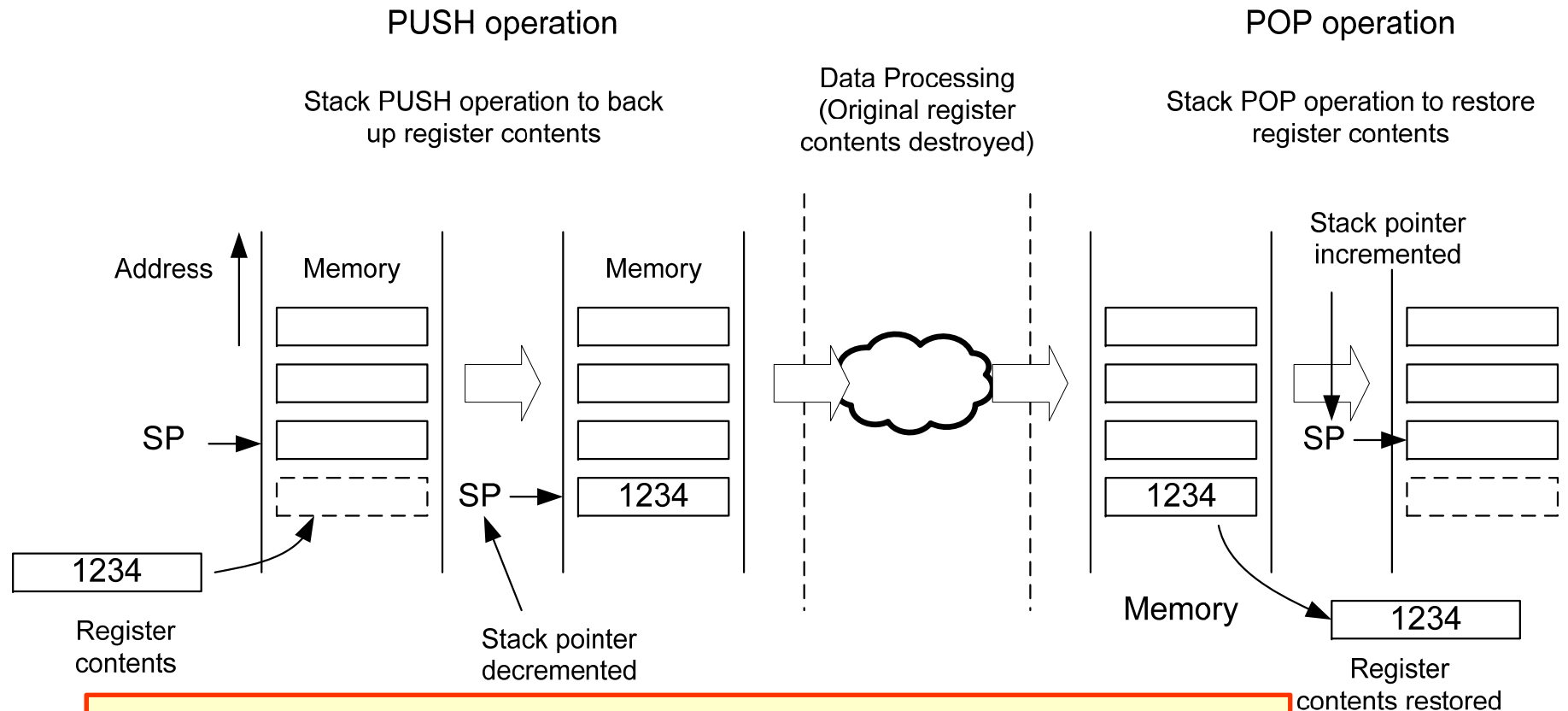
- I. Die ARM-Prozessoren
- II. Übersicht Cortex-M0
- III. Die Register des Cortex-M0
- IV. Die Speicherorganisation des Cortex-M0
- V. Die Startup-Sequenz des Cortex-M0
- VI. Stack und Heap im Cortex-M0**



# Wozu wird ein „Stack“ benötigt?

- Ein Stack (dt.: Stapel- oder Kellerspeicher) arbeitet nach dem LIFO-Prinzip (Last-In, First-Out).
- Er wird insbesondere bei Funktionsaufrufen (z.B. in C) benötigt:
  - Um Inhalte von Arbeitsregistern zu retten.
  - Für die lokalen Variablen (automatische Variablen).
  - Für Übergabe der Argumente und Rückgabewerte.
  - Er ist insbesondere für rekursive Funktionen notwendig
- Der Stack wird über einen „Stack Pointer“ verwaltet und wächst nach oben oder unten.
  - Beim M0 werden im Assembler die Befehle PUSH und POP für die Arbeit mit dem Stack verwendet. Der Stack im M0 wächst nach unten.

# Der Zugriff auf den Stack im M0



„Full Descending“: Stack wächst nach unten, SP zeigt auf das letzte Datum welches auf den Stack gebracht wurde („Full“). Das Datum ist immer 32 Bit groß.

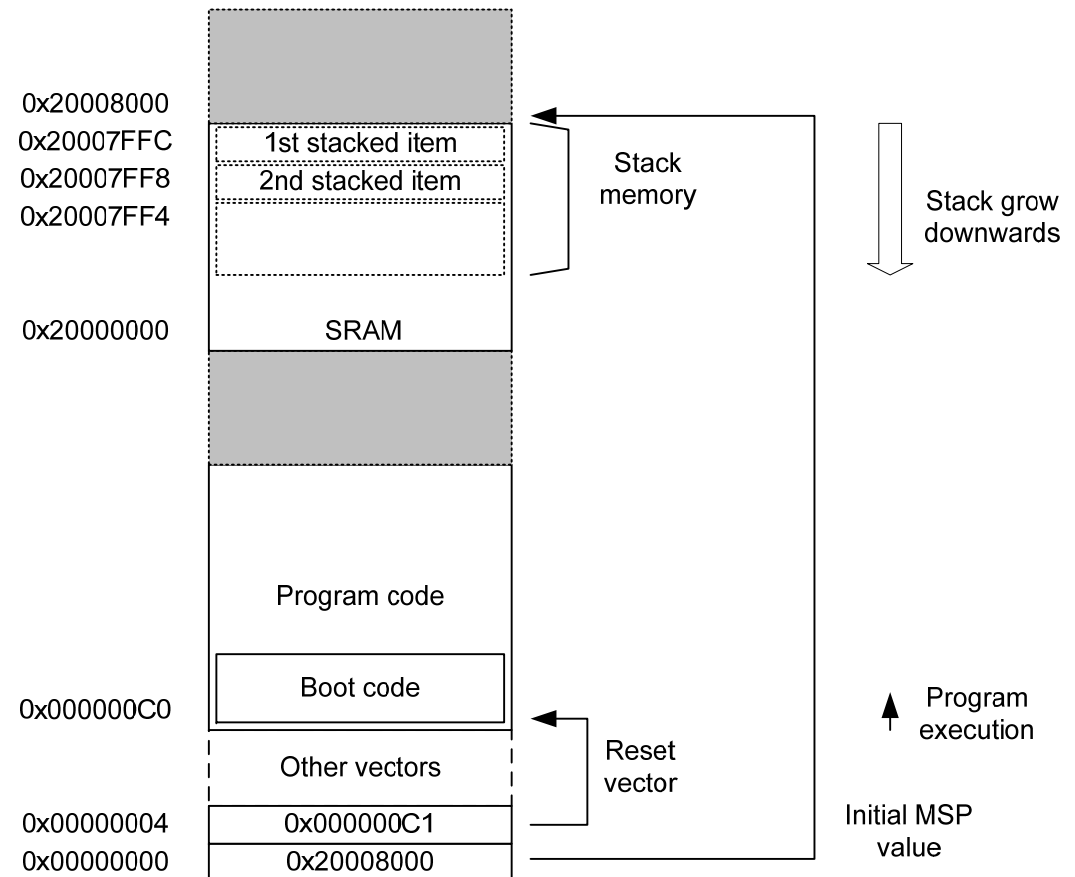
Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Was passiert bei Funktionsaufrufen?

- Die aufrufende Funktion:
  - Bringt die Übergabeargumente auf den Stack
  - Ggf. kann die Rücksprungadresse auf den Stack gebracht werden
  - Nach Rückkehr vom Unterprogramm wird das Ergebnis vom Stack geholt.
- Die aufgerufene Funktion:
  - Rettet die Registerinhalte auf den Stack
  - Implementiert die lokalen Variablen auf dem Stack
  - Holt am Ende die Registerinhalte vom Stack und gibt den Platz für die lokalen Variablen wieder frei
  - Bringt den Rückgabewert auf den Stack.

# Anordnung des Stack im M0-System

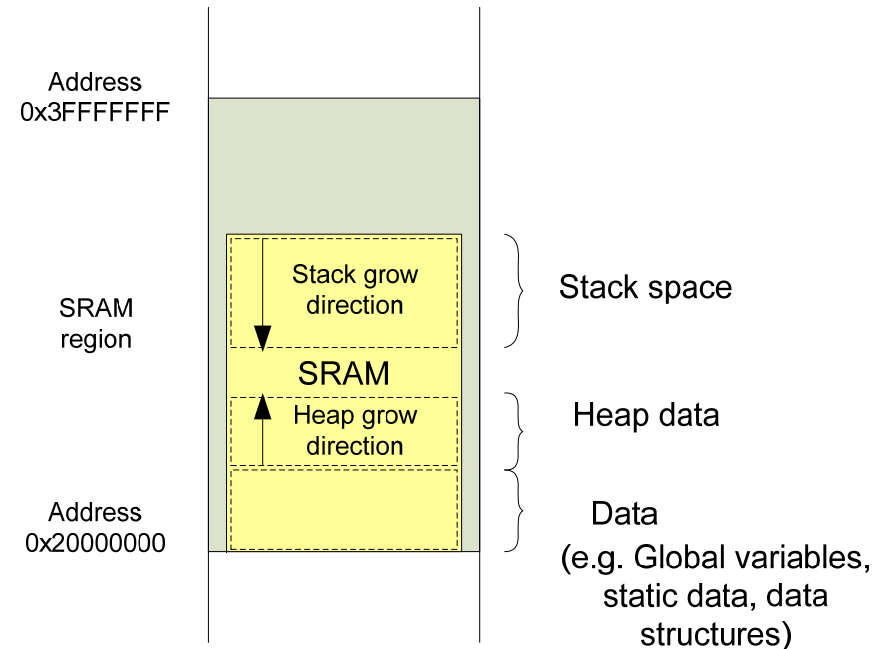
- Beim ersten „PUSH“ wird der SP zunächst dekrementiert, dann das Datum gespeichert.
- Wenn das SRAM z.B. von 0x20000000 bis 0x20007FFF liegt, dann setzt man den MSP auf 0x20008000, so dass der Stack vom Ende des SRAMs her anfängt.



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

# Stack und Heap

- Werden Daten dynamisch erzeugt (z.B. malloc), dann werden diese Daten auf dem „Heap“ (dt. Halde) gespeichert.
- Der Heap beginnt i.d.R. nach den statischen Daten und wächst nach oben!



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

## Stack und Heap (2)

- Beim Anlegen von Projekten mit der µVision werden in der Datei „startup\_NUC1xx.s“ die Größen für Stack und Heap vorgegeben.
  - Voreinstellung Stack: 1024 Byte
  - Voreinstellung Heap: 0 Byte
- Der „Linker“ sorgt dann dafür dass der SP entsprechend gesetzt wird.
  - Kann z.B. in der MAP-Datei (Projekt.map) eingesehen werden (oder im Simulator/Debugger).

# Beispiel: Berechnung der Fakultät

- Fakultät kann rekursiv berechnet werden:  
 $n! = n \cdot (n-1)!$
- Rekursive C-Funktion
- Heap durch Pointer „ptr“ benutzt
- Beispiel  $n = 6$ :  
 $c = 720 = 6!$   
 $b = 0x12345678$

```
#include <stdint.h>
#include <stdlib.h>

uint32_t fakultaet(uint32_t n) {
    if(n>1)
        return n*fakultaet(n-1);
    else
        return n;
}

int main (void) {

    uint32_t a, b, c;
    int *ptr = (int *)malloc(sizeof(int));

    *ptr = 0x12345678;

    a = 6;
    c = fakultaet(a);

    b = *ptr;

}
```

# Speicheraufteilung für Beispiel (nur SRAM)

Adresse	Segment
0x2000_00A0 – 0x2000_049F	Stack, 1024 Byte
0x2000_0080 – 0x2000_009F	Heap, 32 Byte
0x2000_001C – 0x2000_007F	.bss (nicht-init. statische Daten, 96 Byte)
0x2000_0000 – 0x2000_001B	.data (init. statische Daten, 28 Byte)

SP zeigt auf 0x2000\_04A0

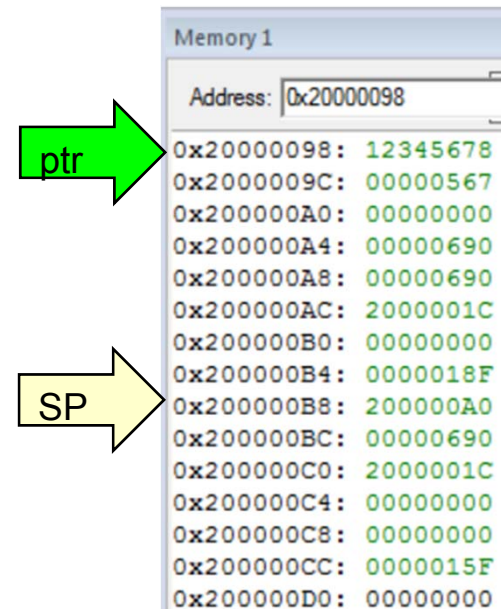
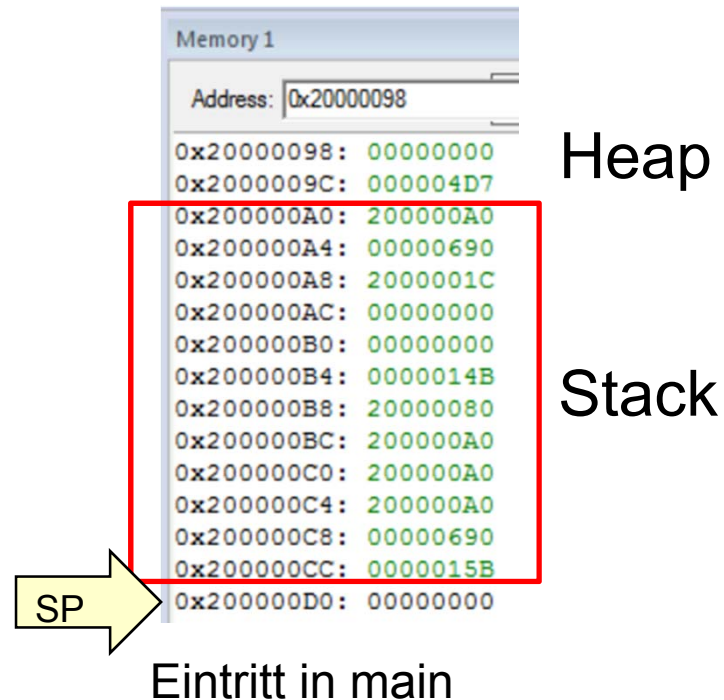


# Verkleinerung des Stack

Adresse	Segment
0x2000_00A0 – 0x2000_00CF	Stack, 48 Byte (12 Words)
0x2000_0080 – 0x2000_009F	Heap, 32 Byte (8 Words)
0x2000_001C – 0x2000_007F	.bss (nicht-init. statische Daten, 96 Byte)
0x2000_0000 – 0x2000_001B	.data (init. statische Daten, 28 Byte)

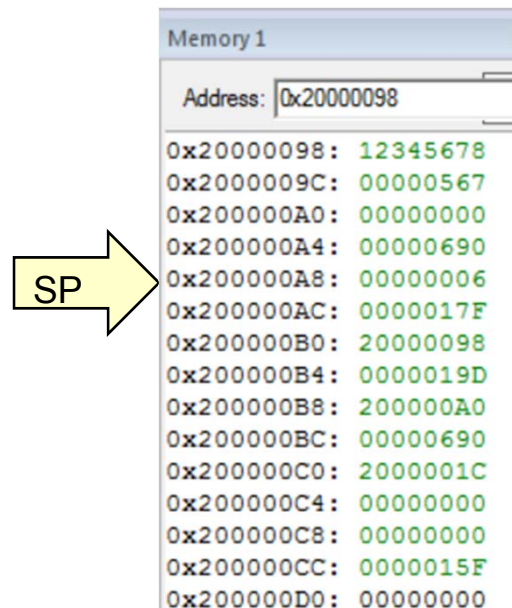
SP zeigt auf 0x2000\_00D0

# Veränderung des Stack während der Laufzeit



Vor Aufruf von „fakultaet“:  
SP wird durch Eintritt in Main um  
6 Words dekrementiert,  
„ptr“ zeigt auf 0x2000\_0098

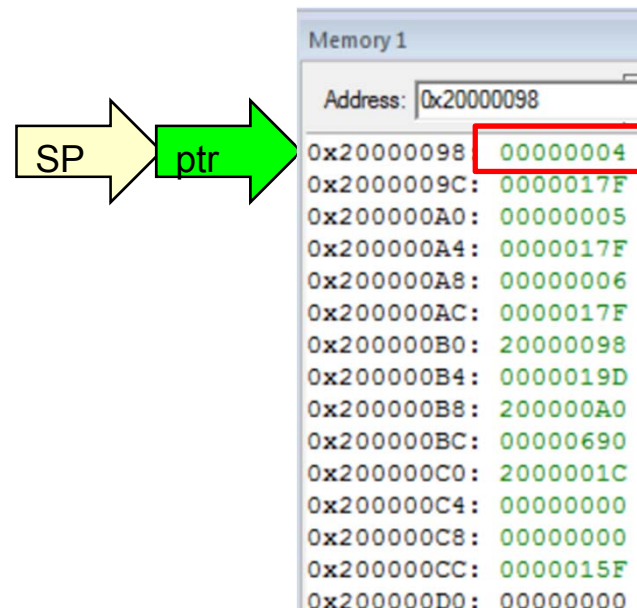
# Stack überschreibt Heap (Stack Overflow)



A yellow arrow labeled 'SP' points to the first line of the memory dump.

Memory 1	
Address:	
0x20000098:	12345678
0x2000009C:	00000567
0x200000A0:	00000000
0x200000A4:	00000690
0x200000A8:	00000006
0x200000AC:	0000017F
0x200000B0:	20000098
0x200000B4:	0000019D
0x200000B8:	200000A0
0x200000BC:	00000690
0x200000C0:	2000001C
0x200000C4:	00000000
0x200000C8:	00000000
0x200000CC:	0000015F
0x200000D0:	00000000

Nach 1. Rekursion,  
Jede Rekursion benötigt  
2 Words auf dem Stack



A yellow arrow labeled 'SP' and a green arrow labeled 'ptr' both point to the first line of the memory dump. The value '00000004' at address 0x20000098 is highlighted with a red box.

Memory 1	
Address:	
0x20000098:	00000004
0x2000009C:	0000017F
0x200000A0:	00000005
0x200000A4:	0000017F
0x200000A8:	00000006
0x200000AC:	0000017F
0x200000B0:	20000098
0x200000B4:	0000019D
0x200000B8:	200000A0
0x200000BC:	00000690
0x200000C0:	2000001C
0x200000C4:	00000000
0x200000C8:	00000000
0x200000CC:	0000015F
0x200000D0:	00000000

Nach zwei weiteren  
Rekursionen:  
SP = ptr  
Folge: Datum auf das ptr zeigt  
wird überschrieben, somit  
b = 0x00000004

# Stack und Heap: Überläufe

- Überläufe bei Stack und Heap sind Laufzeitprobleme: Programm „stürzt“ ab, Ursache unklar. Solche Probleme sind schwer zu analysieren.
- Stack:
  - Ausreichend groß machen.
  - Wie groß ist die Aufruftiefe bei Funktionen?
  - Insbesondere rekursive Funktionen können problematisch werden.
- Heap: Bei dynamischer Speicherallokation Bedarf abschätzen.