ARM®

Low-Power Leadership

Software Design for Cortex™-M Microcontroller

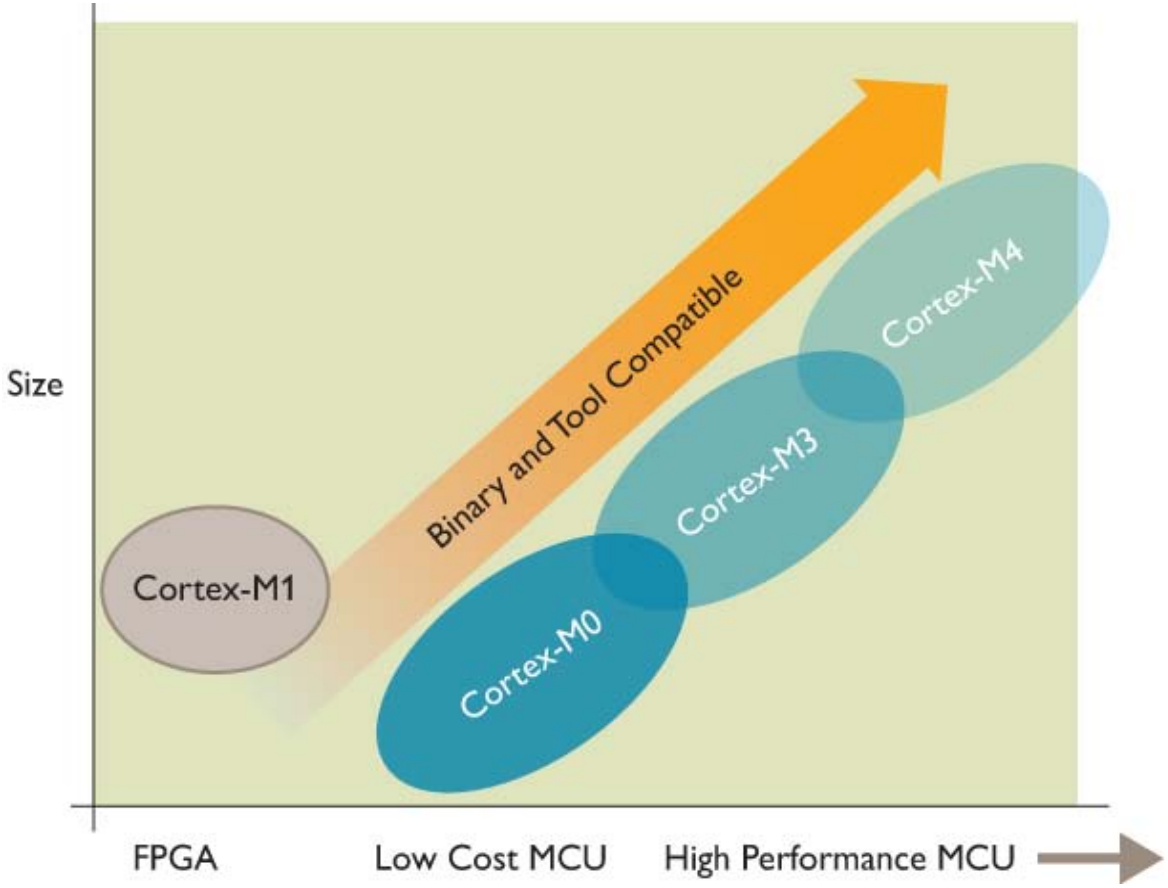
Reinhard Keil
Director MCU Tools

Agenda

- Cortex-M Processor Overview for Microcontrollers
  - Cortex-M0: Easy to use 32-bit processor in an 8/16-bit footprint
  - Cortex-M3: De-facto industry standard 32-bit Microcontroller
  - Cortex-M4: New Cortex-M Processor with DSP and Floating Point Unit
- Software Concepts
  - CMSIS: Cortex Microcontroller Software Interface Standard
  - Super-Loop: Complete Application runs in an Endless Loop
  - RTOS: Resource Control for Time-Critical Applications
  - MPU: Memory Protection Unit for improve Software Security
  - UML: Graphical Modeling Language for better Software-Architecture
- Middleware
  - Why is Middleware Required
  - Selected Middleware Vendors along with middleware offering

Cortex-M Processor Family

- **Seamless embedded architecture**
  - Spanning cost and performance points



ARM Cortex-A Series:
Applications processors for feature-rich OS and user applications

ARM Cortex-R Series:
Embedded processors for real-time signal processing and control applications

ARM Cortex-M Series:
Deeply embedded processors optimized for microcontroller and low-power applications

Spanning the application range

- **Traditional 8/16/32-bit classification obsolete**
  - Seamless architecture across all applications
  - Every product optimised for ultra low power systems

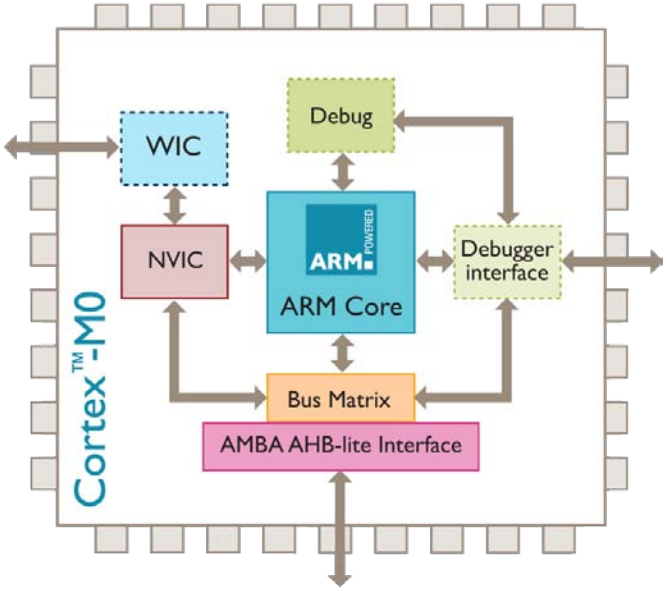
Cortex-M0	Cortex-M3	Cortex-M4
“8/16-bit” applications	“16/32-bit” applications	“32-bit/DSC” applications
Lowest cost Optimised connectivity	Performance efficiency Feature rich connectivity	SIMD/DSP Instructions Floating Point Unit (optional) High Performance MCU

Cortex-M processor industry adoption

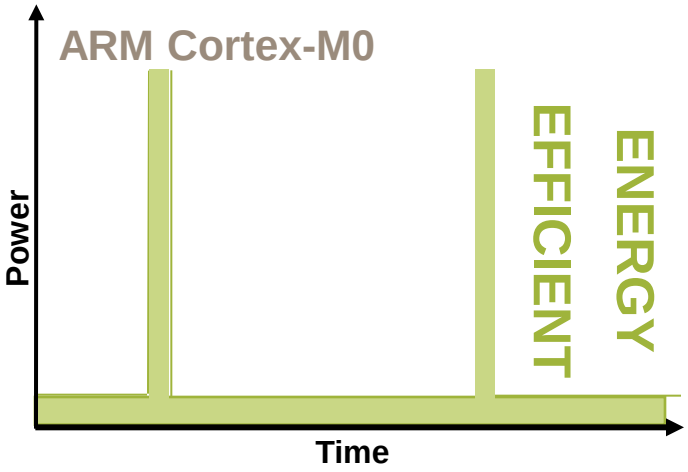
- **ARM Cortex-M3 processor momentum continues**
  - 35+ licensees in applications from MCU, SoC, wireless sensor nodes
  - 140% CAGR in units shipped by vendors
- **ARM Cortex-M0 processor announced in 2009**
  - 13 licensees already in MCU, mixed-signal and FSM replacement
- **New Cortex-M4 processor just announced**
  - Available for licensing now

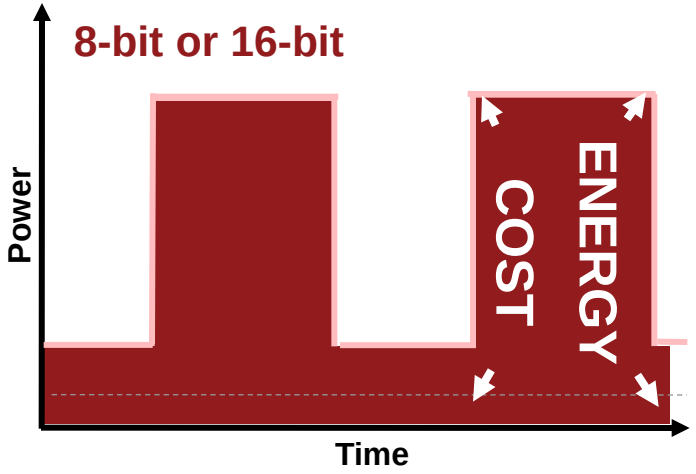
ARM Cortex-M0 Processor

- The smallest ARM processor ever
  - A third of the area of ARM7TDMI-S™ at comparable performance
- Architected for ultra low power
  - Power management unit via Wake-up Interrupt Controller

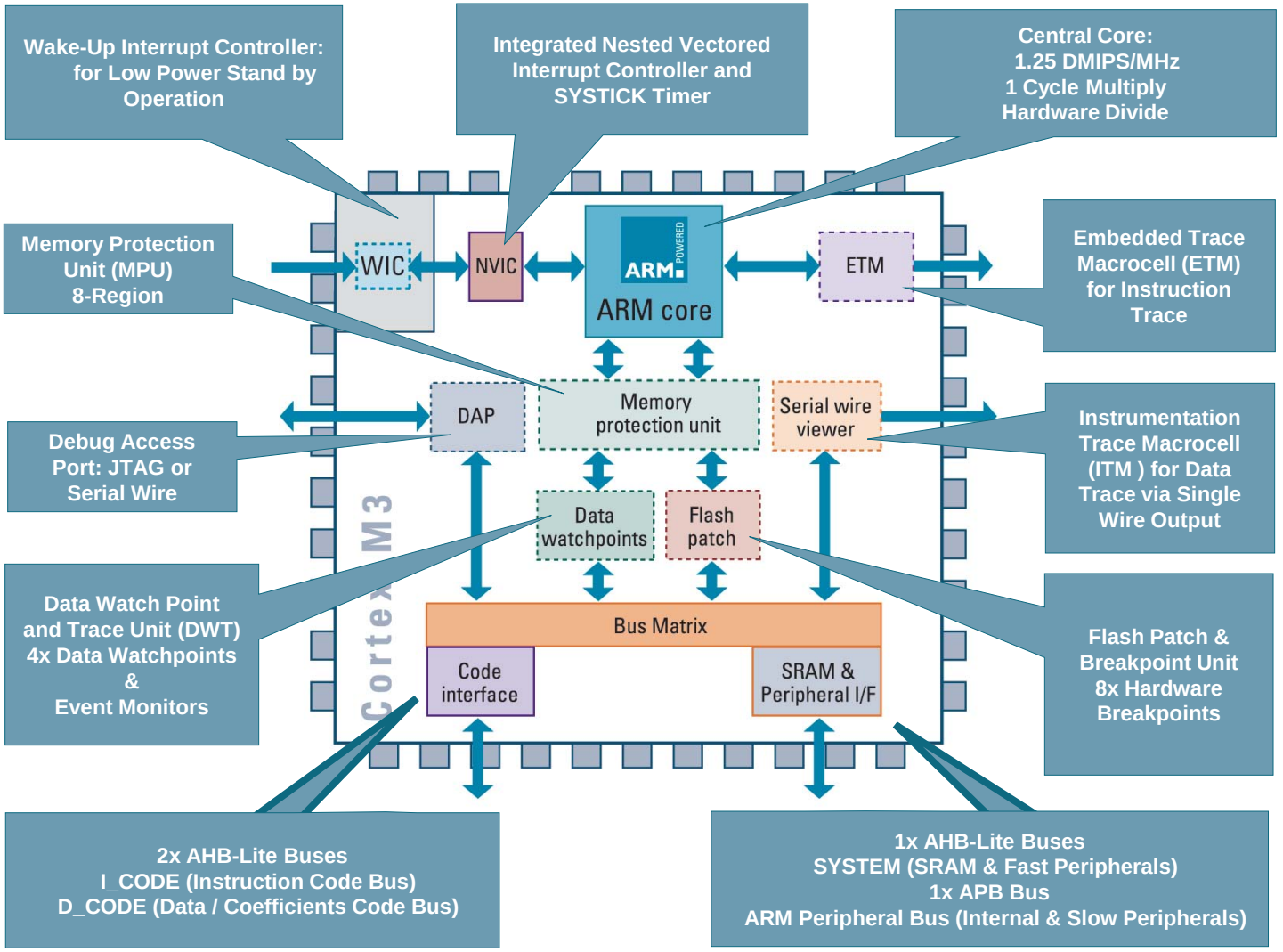


Cortex
Low-Power Leadership from ARM®



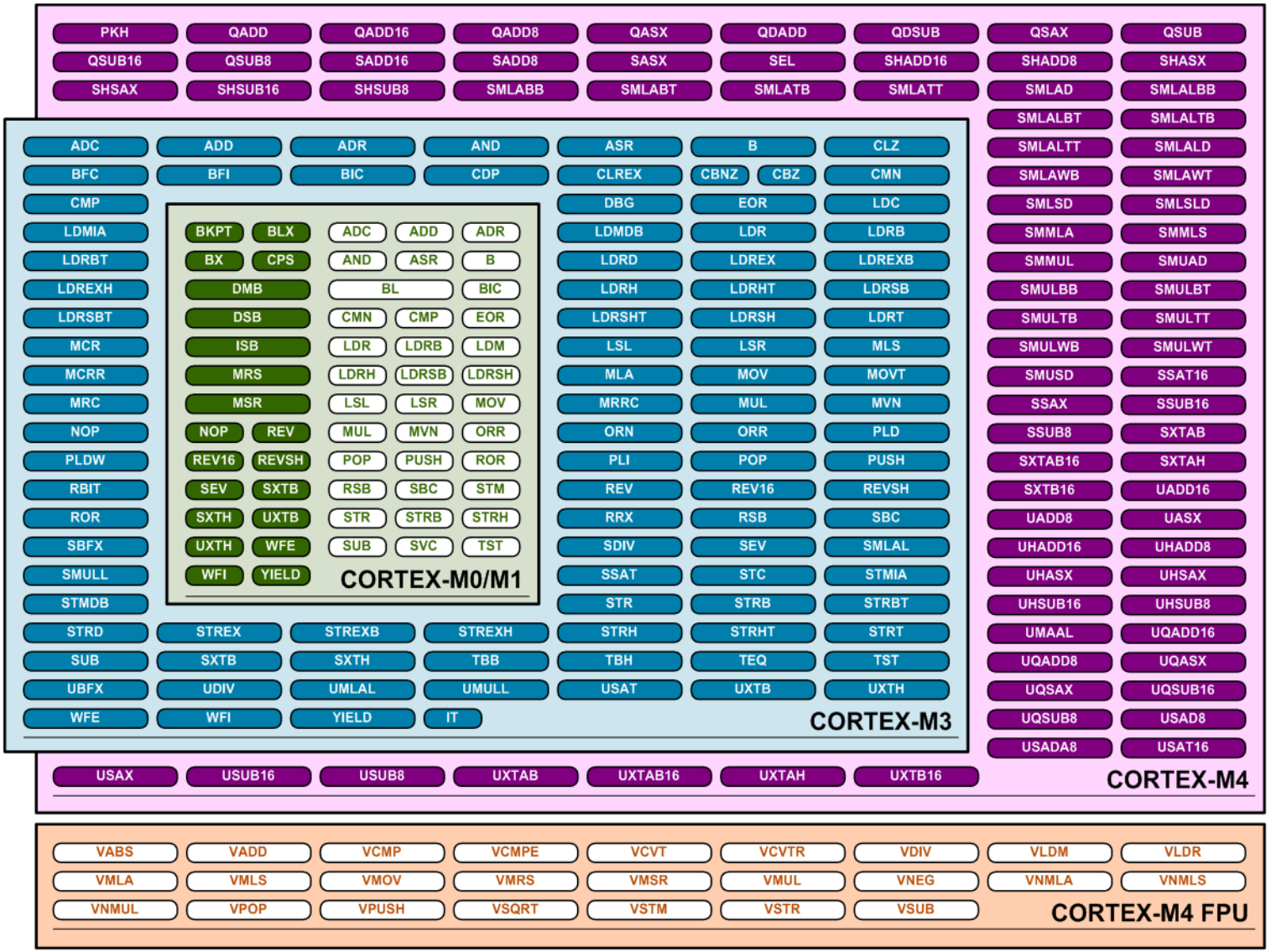


Cortex-M3 Processor Overview

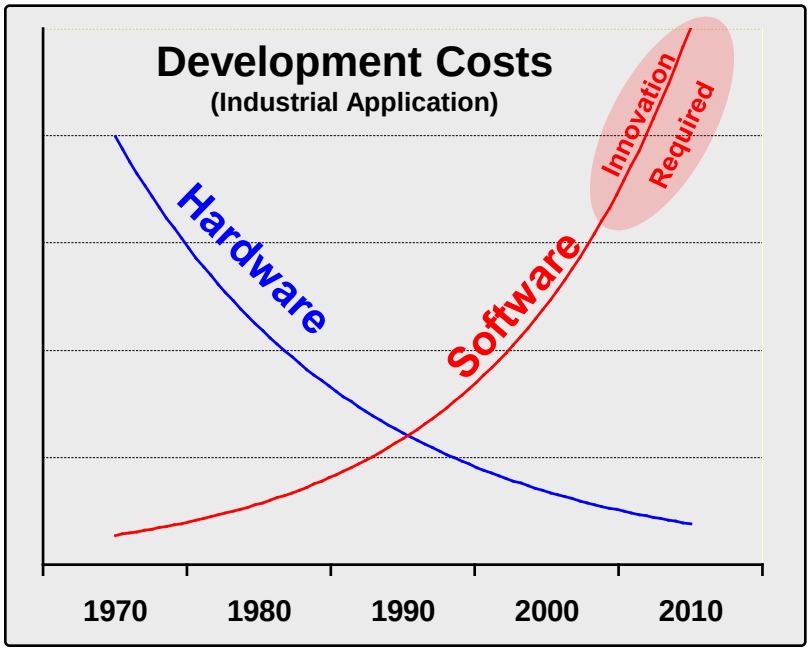


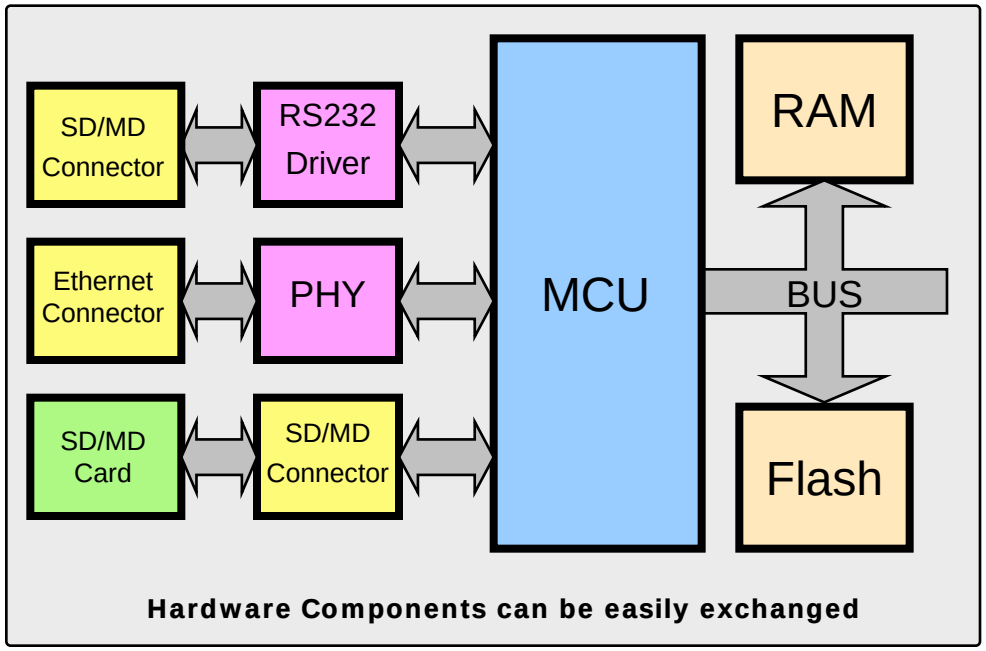
optional blocks, please consult your silicon manufacturers data sheet

Cortex-M Instruction Set



Software Complexity – The Challenge

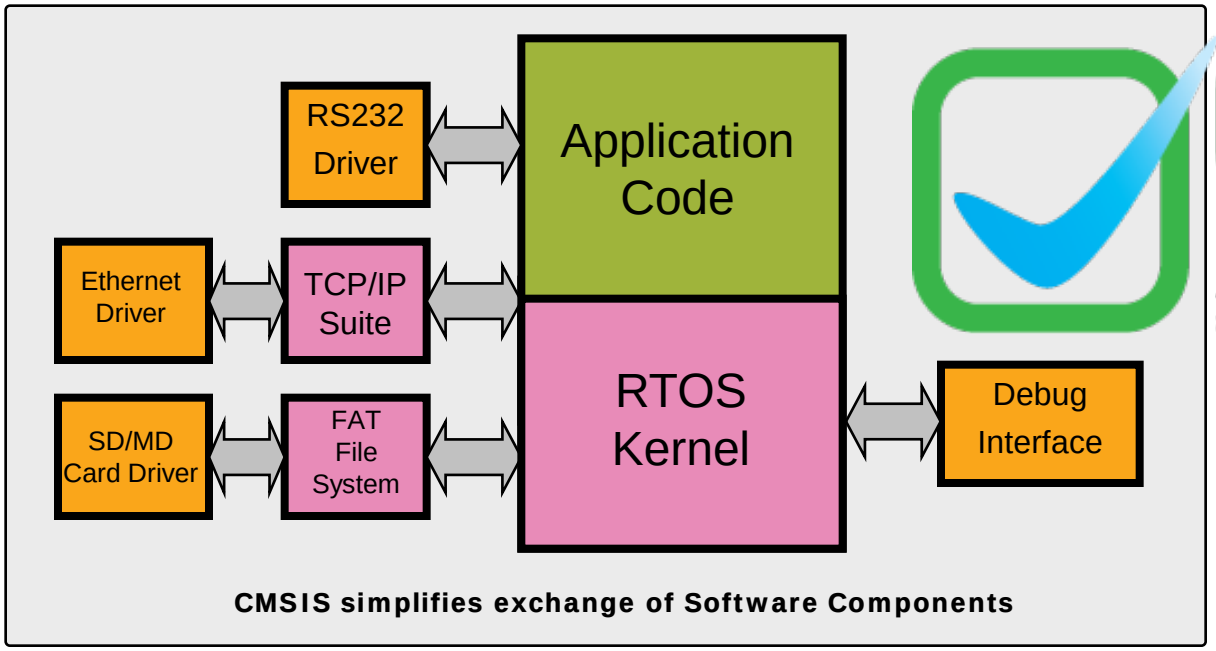




- Well-known issues that drive software costs
  - Increasing product requirements that are implemented by software
  - Hardware problems tend to become compensated by software
- Up to now software components cannot be easily exchanged

A Microcontroller Software Interface Standard is Required!

CMSIS simplifies Code Re-Use



CMSIS COMPLIANT
ARM® Cortex™ Microcontroller Software Interface Standard

RAISONANCE

hitex DEVELOPMENT TOOLS

code_red

IAR SYSTEMS

Micrium Empowering Embedded Systems

KEIL™ Tools by ARM

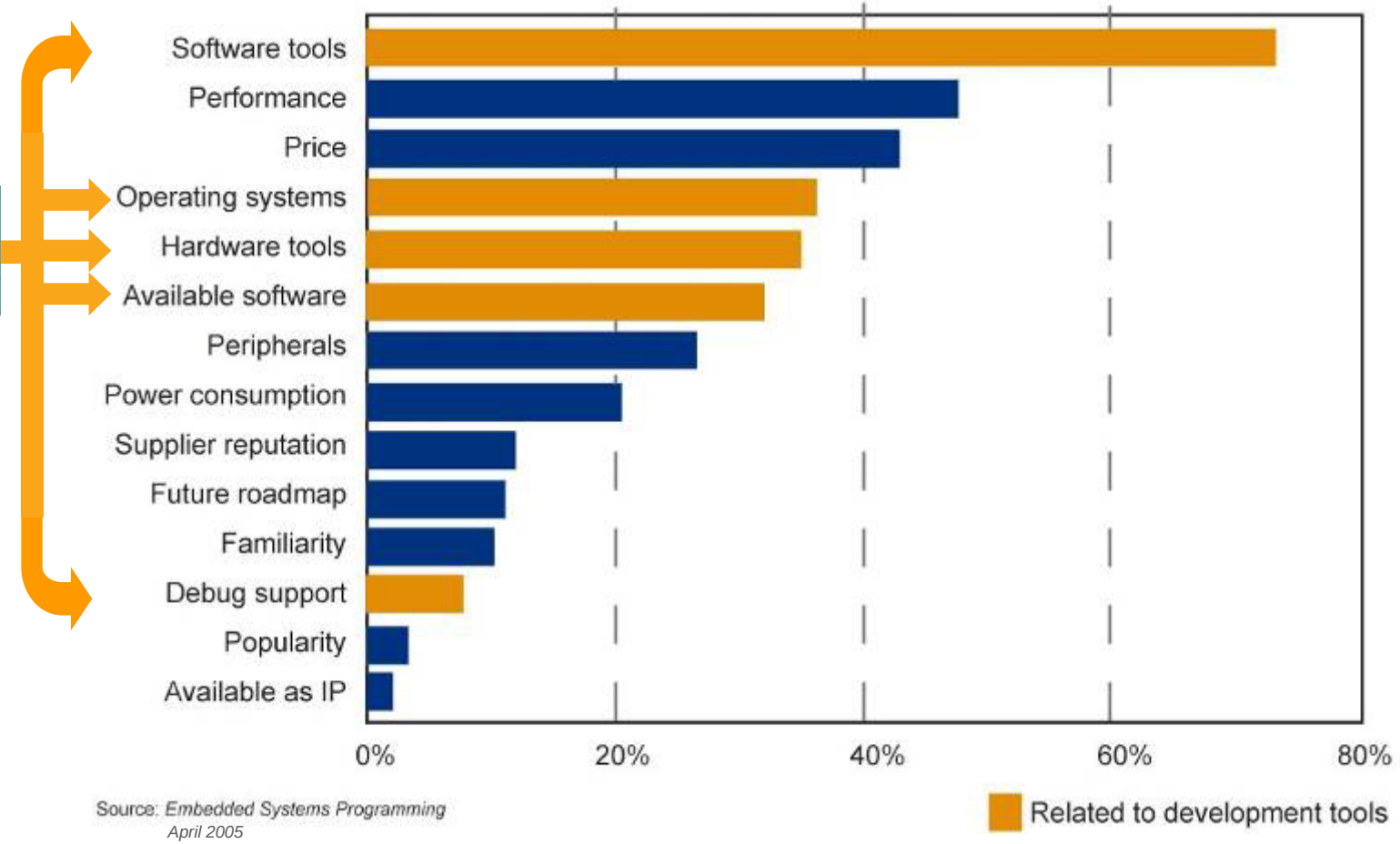
SEGGER

Altium

The **Cortex Microcontroller Software Interface Standard (CMSIS)** enables deployment of software components to physical Microcontroller Devices

Standardization - driven by software reuse

- #1 factor in choosing a processor is the software development tools available for it



Source: *Embedded Systems Programming*
April 2005

Software Concepts for Embedded Applications

- CMSIS: Cortex Microcontroller Software Interface Standard
- Super-Loop: Complete Application runs in an Endless Loop
- RTOS: Resource Control for Time-Critical Applications
- MPU: Memory Protection Unit for improve Software Security
- UML: Graphical Modeling Language for better Software-Architecture

What is CMSIS?

- **CMSIS - Cortex Microcontroller Software Interface Standard**
  - Abstraction layer for all Cortex-M processor based devices
  - Developed in conjunction with silicon, tools and middleware partners

- **CMSIS Peripheral Access Layer defines**
  - Consistent layout of all peripheral registers
  - Vector definitions for all exceptions and interrupts
  - Functions to access core registers and core peripherals
  - Device independent interface for RTOS Kernels
  - Debug channel (for printf-style + RTOS Kernel)

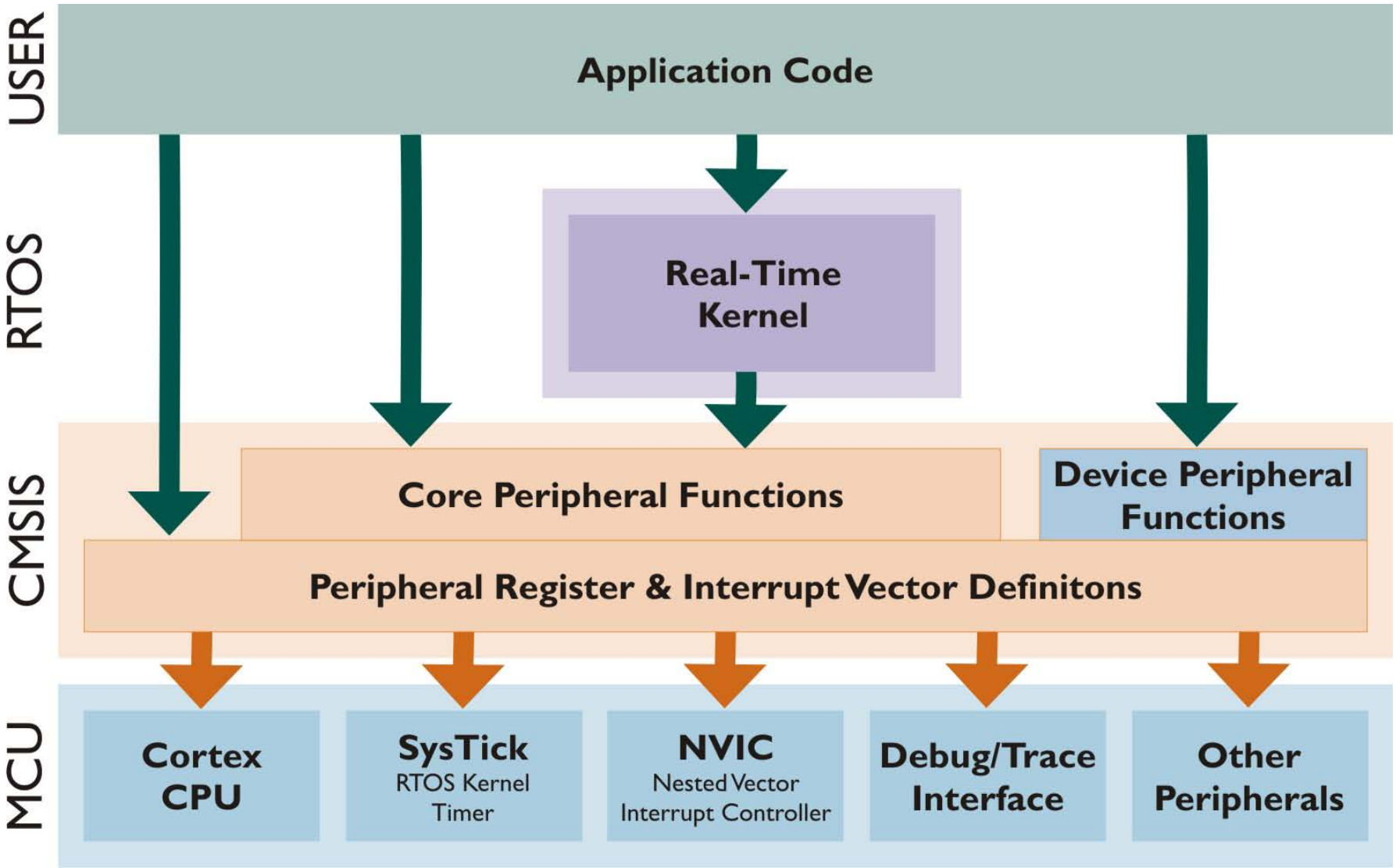
- **CMSIS compliant software components allow**
  - Easy reuse of example applications or template code
  - Combination of software components from multiple vendors

What CMSIS is Not

The **Cortex Microcontroller Software Interface Standard (CMSIS)** is not another complex software layer that forces vendors to identical features

- **CMSIS does not make Peripherals Equal**
  - Peripheral Access Layer defines common methods, not features
    - I/O, Timers, PWM, A/D, D/A, etc. can have different functions
- **CMSIS does not require immense Resources**
  - Peripheral Access Layer requires < 1KB code, 4Bytes variable.
    - Including device startup code
- **CMSIS does not prevent from direct Hardware Access**
  - Peripheral Registers are access via memory accesses
    - Each Peripheral has a struct that collects all registers
    - Device documentation should reflect this structure

CMSIS – Structure



CMSIS – Files for Peripheral Access Layer

Compiler Vendor-Independent Files:

- **Cortex-M Core Files (provided by ARM)**
  - core_cm3.h+core_cm3.c core_cm0.h+core_cm0.c
- **Device-specific Files (provided by Silicon Vendors)**
  - Register Header File (*device*.h)
  - System Startup File (system_*device*.c)
- **Compatible with all supported Compilers (ARM, IAR, GNU, Tasking, …)**

Compiler-Vendor + Device-Specific Startup File:

- **Device Specific Compiler Startup Code (provided by Silicon Vendors)**
  - startup_*device*.c

CMSIS Files are available via www.onARM.com:

- **Device Database that lists all available devices**
  - CMSIS Files can be downloaded

CMSIS – Example

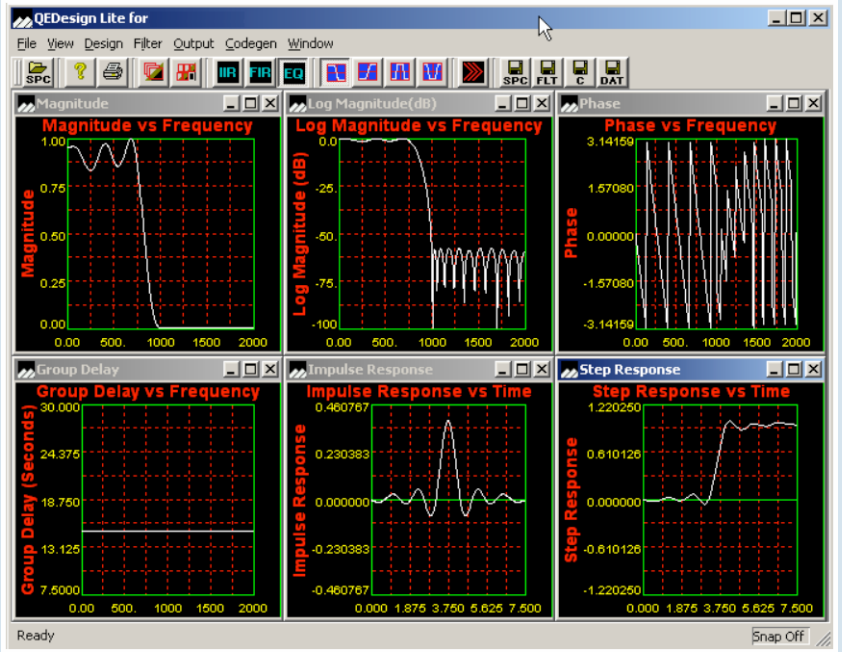
```
#include <device.h>                          // file name depends on device

void SysTick_Handler (void) {                // SysTick Interrupt Handler
   ;
}


void TIM1_UP_IRQHandler (void) {             // Timer Interrupt Handler
  ;
}

void timer1_init(int frequency) {            // set up Timer (device specific)
  NVIC_SetPriority (TIM1_UP_IRQn, 1);        // Set Timer priority
  NVIC_EnableIRQ (TIM1_UP_IRQn);             // Enable Timer Interrupt
}

// The MCU is initialized by CMSIS startup + system file
void main (void) {
  if (SysTick_Config (SystemFrequency / 1000)) { // SysTick 1mSec
       : // Handle Error
  }
  timer1_init ();                            // setup device specific timer
    :
}
```

Cortex-M4 CMSIS Extensions

- Cortex-M4 support available today in MDK-ARM and ARM Compiler
  - C Compiler intrinsic functions for Cortex-M4 extended Instructions
  - Optimized Floating Point Library using FPU CPU Instructions
  - Complete µVision Debugger support; including Instruction Set Simulation

... UNDER DEVELOMENT ...

- CMSIS - Expanded with Cortex-M4 Features (Intrinsic Functions)
  - Every CMSIS compliant C Compiler supports Cortex-M4 extensions
- Optimized Library using CMSIS
  - Designed to make DSP programming easy for MCU users
  - **General Functions**
    math, trigonometric, control functions (building blocks)
  - **Digital Filter Algorithms**
    for filter design utilities and DSP toolkits (MathLab, LabVIEW, etc.)



Super-Loop Design Pattern

- Application implemented as endless loop containing function calls
  - Implies a fixed order for function execution
  - Fits perfect for small embedded system; frequently choice for 8/16-bit MCU
  - Time-critical program portions implemented as interrupt service routine (ISR)
  - Communication via global variables; not data communication protocol
  - Cortex-M security (MPU, PSP, Thread Mode Privilege Level) not needed

```
void main (void) {
  Device_Initialization ();     // Configure & Initialize MCU

  while (1)  {                  // Endless Loop (the Super-Loop)
    Get_InputValues ();         // Read Values
    Calculation_Response ();    // Calculate Results
    Output_Response ();         // Output Results
  }
}
```

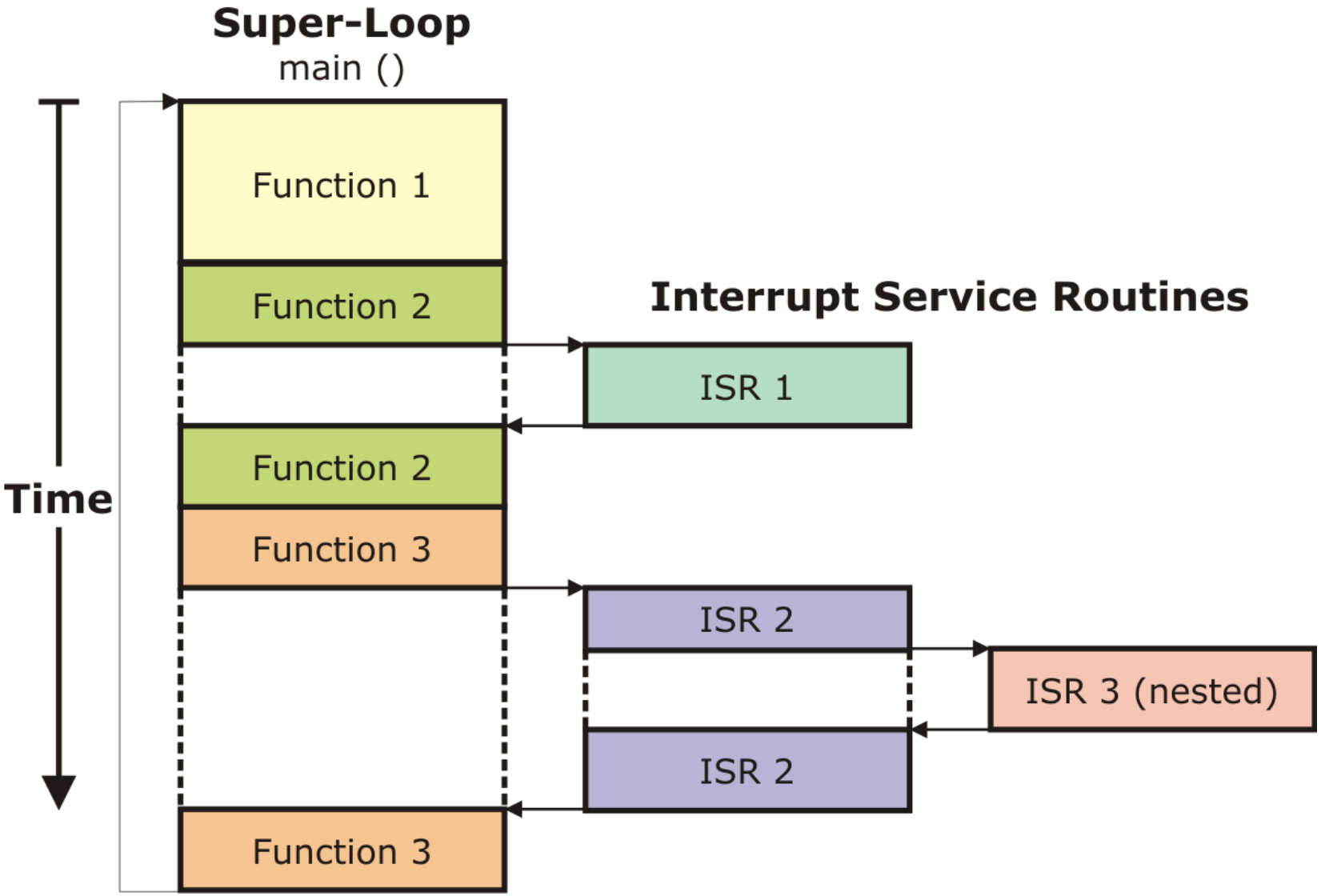
Super-Loop Example

Super-Loop with Cycle & Power-Saving

- Easy to expand for time synchronisation & power-down

```
uint32_t volatile ms10Ticks;                // Counter for 10 Milli-Second Interval

void SysTick_Handler (void) {               // SysTick Interrupt Service Routine
  ms10Ticks++;                              // Increment 10ms Counter
}

void WaitForTick (void)  {
  uint32_t curTicks;

  curTicks = ms10Ticks;                     // Save Current SysTick Value
  while (ms10Ticks == curTicks)  {          // Wait for next SysTick Interrupt
    __WFE ();                               // Power-Down until next Event/Interrupt
  }
}

void main (void) {
  SysTick_Config(SystemCoreClock / 100);    // Set SysTick: 10ms Interval
  Device_Initialization ();                 // Configure & Initialize MCU

  while (1)  {                              // Endless Loop (the Super-Loop)
    Get_InputValues ();                     // Read Values
    Calculation_Response ();                // Calculate Results
    Output_Response ();                     // Output Results
    WaitForTick ();                         // Synchronize to SysTick Timer
  }
}                                                       // RED: CMSIS Features
```

Super-Loop – Timing



Super-Loop – Data Exchange

```
struct { uint16_t msec; uint8_t sec; uint8_t min; uint8_t  hour; }  volatile clock;

void SysTick_Handler(void) {                  // SysTick ISR called every msec
  if (++clock.msec == 1000)  {                // increment ms, check overflow
    clock.msec = 0;
    if (++clock.sec == 60)  {                 // increment seconds, check overflow
      clock.sec = 0;
      if (++clock.min == 60)  {               // increment minutes, check overflow
        clock.min = 0;
        if (++clock.hour == 24)  {            // increment hour, check overflow
          clock.hour = 0;
        }
      }
    }
  }
}

void CheckAlert (void)  {                     // check for alarm at 12:59
  if (clock.min == 59 &&                      // check minute for 59
      clock.hour == 12)  {                    // check hour for 12
    Alert ();                                 // call Alarm Function
  }
}
```

Super-Loop – Data Exchange

```
struct { uint16_t msec; uint8_t sec; uint8_t min; uint8_t  hour; }  volatile clock;

void SysTick_Handler(void) {                  // SysTick ISR called every msec
  if (++clock.msec == 1000)  {                // increment ms, check overflow
    clock.msec = 0;
    if (++clock.sec == 60)  {                 // increment seconds, check overflow
      clock.sec = 0;
      if (++clock.min == 60)  {               // increment minutes, check overflow
        clock.min = 0;
        if (++clock.hour == 24)  {            // increment hour, check overflow
          clock.hour = 0;
        }
      }
    }
  }
}

void CheckAlert (void)  {                     // check for alarm at 12:59
  if (clock.min == 59 &&
      clock.hour == 12)  {
    Alert ();                                 // call Alarm Function
  }
}
```

Problem when SysTick ISR occurs at 11:59:59.999

Super-Loop – Data Exchange

- Problem Solution: disable interrupts
  - Ensure that this is only for short period:
    requires frequently re-write of application code
  - Don't be too clever: disabling a single high-priority ISR
    means that low-priority ISR may delay ISR execution

```
void CheckAlert (void)  {                    // check for alarm at 12:59
  int AlertFlag;

  __disable_irq();                           // disable all interrupts
  AlertFlag = (clock.min == 59 &&            // check minute for 59
               clock.hour == 12);            // check hour for 12
  __enable_irq();                            // enable all interrupts

  if (AlertFlag)  {
    Alert ();                                // call Alarm function
  }
}                                                    // RED: CMSIS Features
```

Super-Loop – Disadvantages

- Time-critical operations must be processed within interrupts (ISR)
  - ISR functions get complex and require long execution times
  - ISR nesting may create unpreticable execution time & stack requirement
- Data exchange Super-Loop ⇆ ISR via global shared variables
  - Application programmer must ensure data consistency
- A Super-Loop can be easily synchronized with the SysTick timer, but:
  - If a system requires several different cycle times, it is hard to implement
  - Split of time-consuming functions that exceed Super-Loop cycle
  - Creates software overhead and application program is hard to understand
- Super-Loop applications get easily complex; therefore hard to extend
  - A simple change may have unpreticable side effects; such side effects are time consuming to analyze due to lack of analysing features.

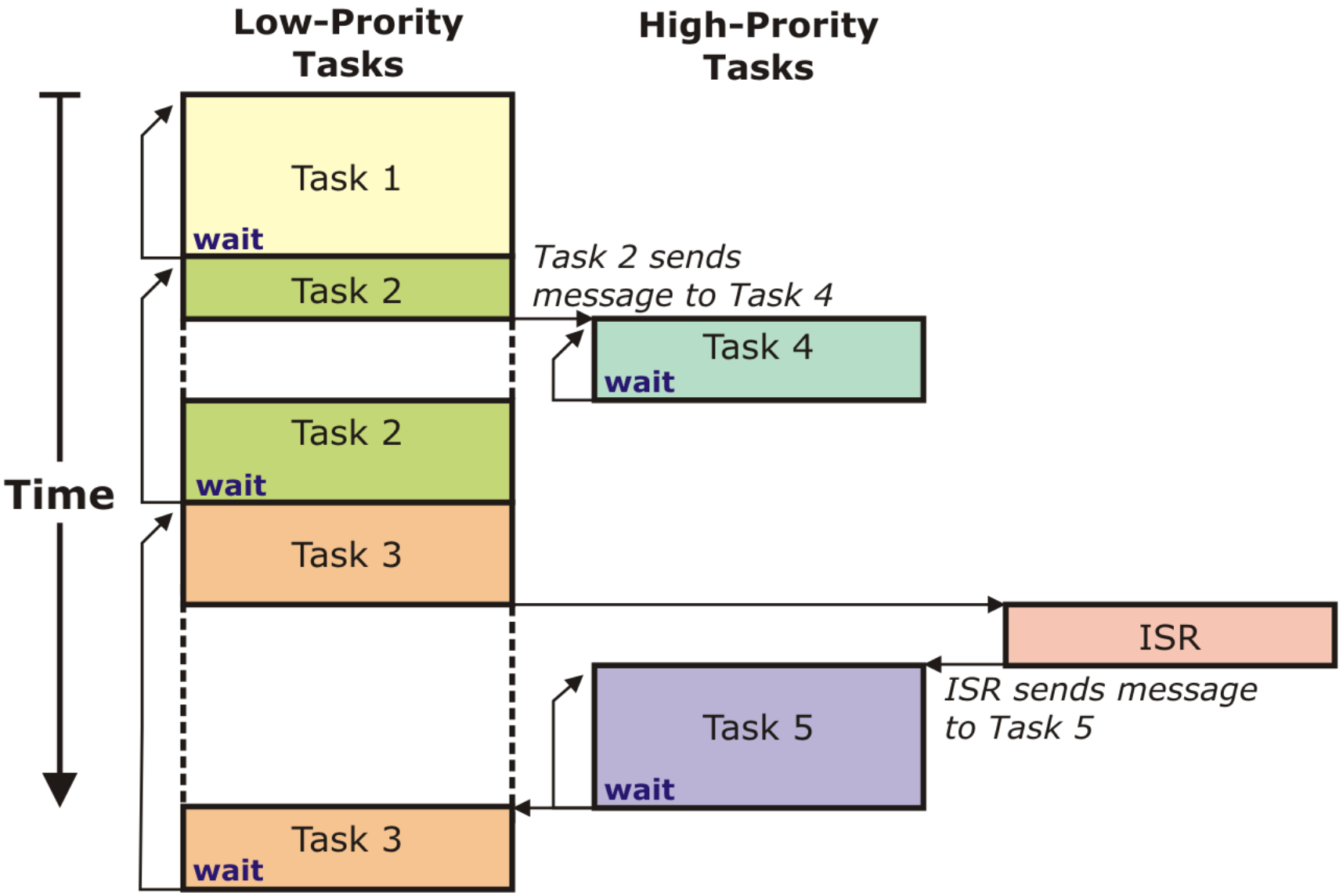
The simplicity of the Super-Loop concept has several serious disadvantages for today's applications. A Real-Time Operating System solves this problems.

RTOS Design Pattern

- A Real-Time Operating System (RTOS) controls resources
  - Applications separated into tasks (threads) that run independent
  - Extensive time control (time delay, time intervals, round-robin scheduling)
  - Deterministic execution times and control of task (process) scheduling
  - Inter-task communication (events, message), resource sharing (semaphore, mutex), and memory allocation features with message pools
  - Supports development with error checking, debug and test facilities

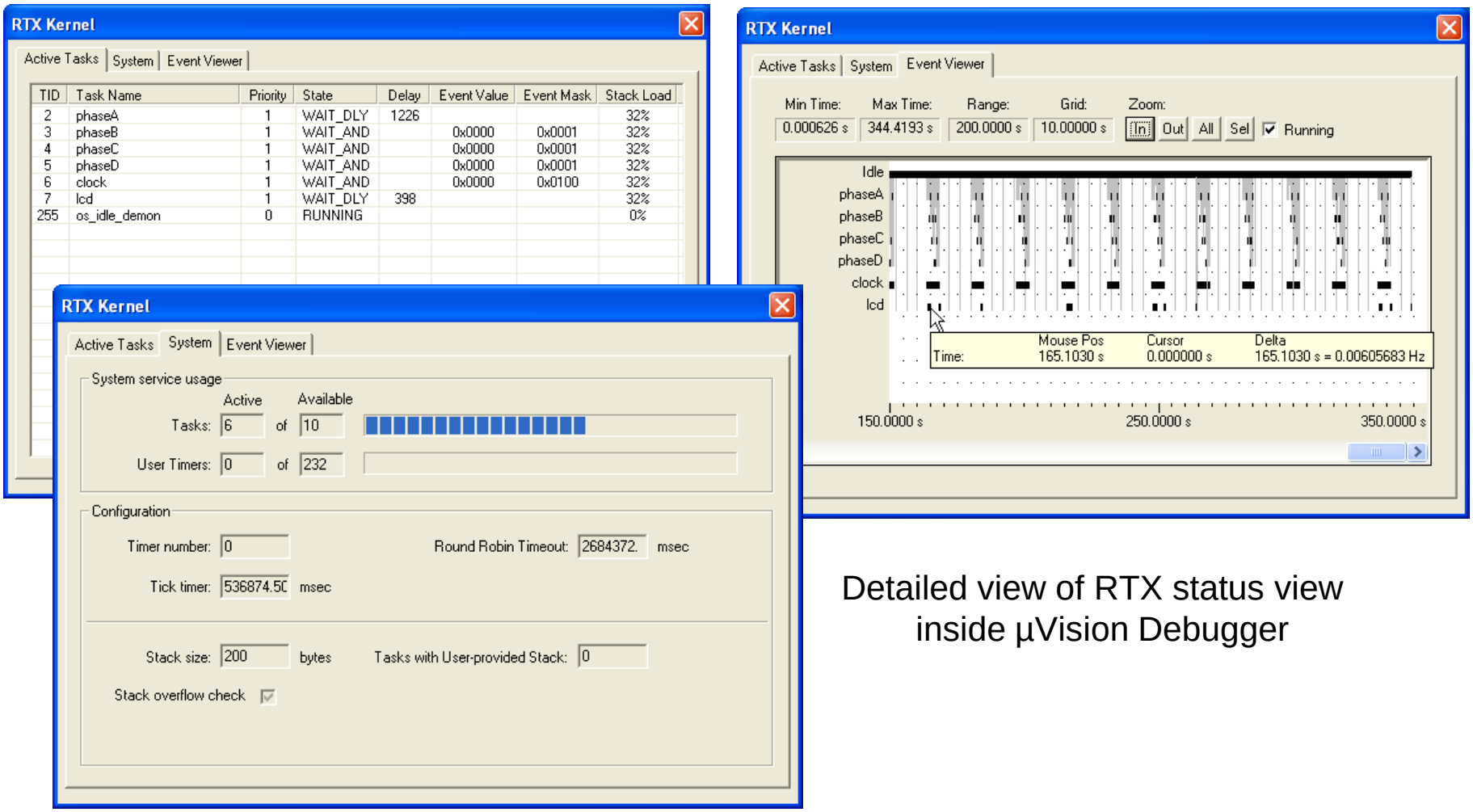
```
__task void clock (void) {
  os_itv_set(60Sec);              // Task waits on 60 second interval

  while (1) {                     // Endless loop (the task loop)
    if (time.min == 59 && time.hour == 12)  {
      os_evt_set(1, id_Alert);  // Set Event flag from task Alert
    }
    os_itv_wait ();               // wait for one minute
    if (++time.min == 60)  { time.min = 0; ++time.hour; }
  }
}
```

RTOS – Timing



Tool support for RTX in MDK-ARM

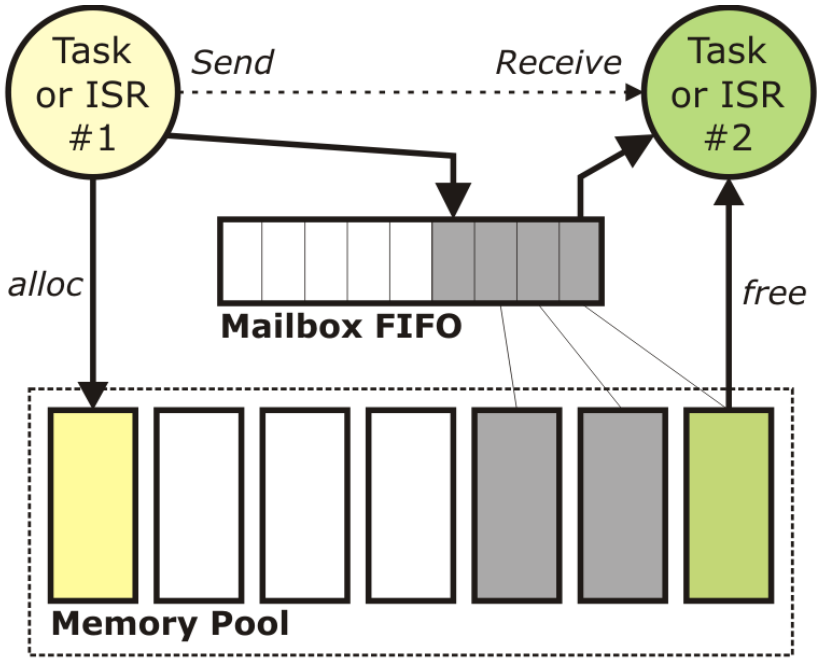
- µVision Debugger supports RTX **Kernel-aware debugging**



Detailed view of RTX status view inside µVision Debugger

RTOS Message Passing + Memory Pools

- Task or ISR can exchange data
  - A single data owner at a time
  - Mailbox FIFO for multiple messages
    On overruns: forces task switch or error notification
  - Deterministic time behavior (independent from current load)
  - Timeout prevents from application hang-up




```
os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
  void *msg;

  os_mbx_init (mailbox1, sizeof(mailbox1));
  msg = alloc();
// fill message content

  os_mbx_send (mailbox1, msg, 12);
}
```

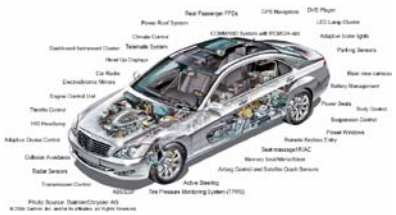
timeout

```
__task void task2 (void) {
  void *msg;
   ..
  os_mbx_wait (mailbox1, &msg, 100);
// process message content here

  free (msg);
}
```

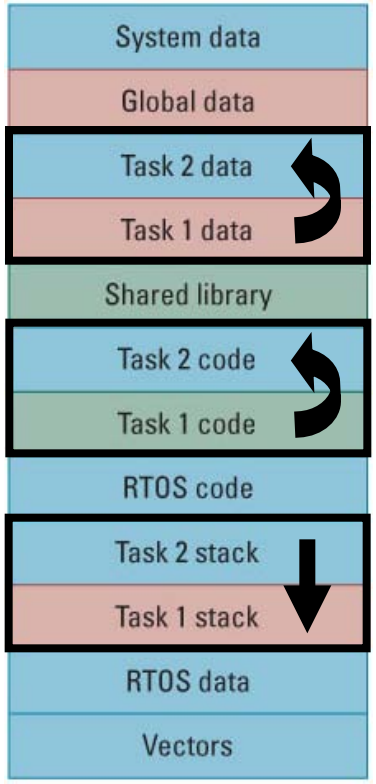
timeout

Memory Protection Unit (MPU)

- MPU provides access control for eight memory regions



- Zero Latency Memory Protection
  - 8 register-stored regions
  - Same regions used for instructions and data
  - Minimum region size 32 Bytes (max 4GB)
  - No address translation or Page Tables

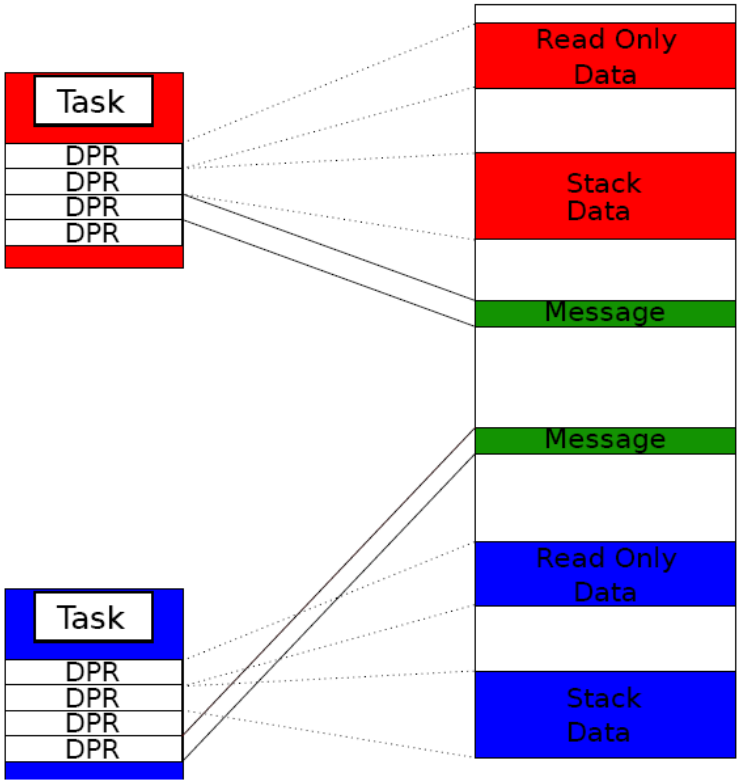


- Configured via memory-mapped control registers

Not available in Cortex-M0 / Cortex-M1

RTOS with MPU Support: PXROS-HR

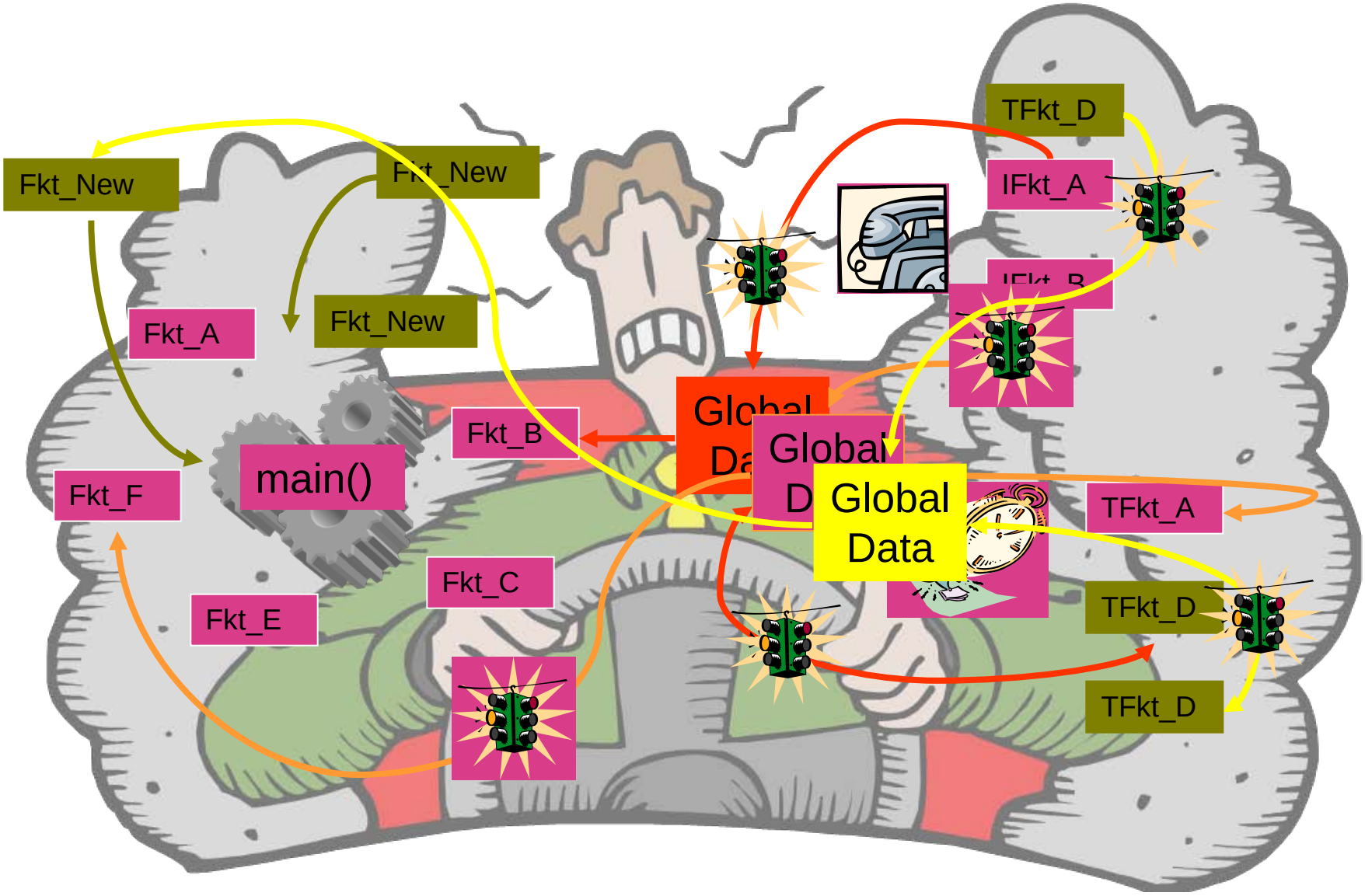
- Uses Cortex-M3 Memory Protection for IEC61508
- Protected transfer for communication objects



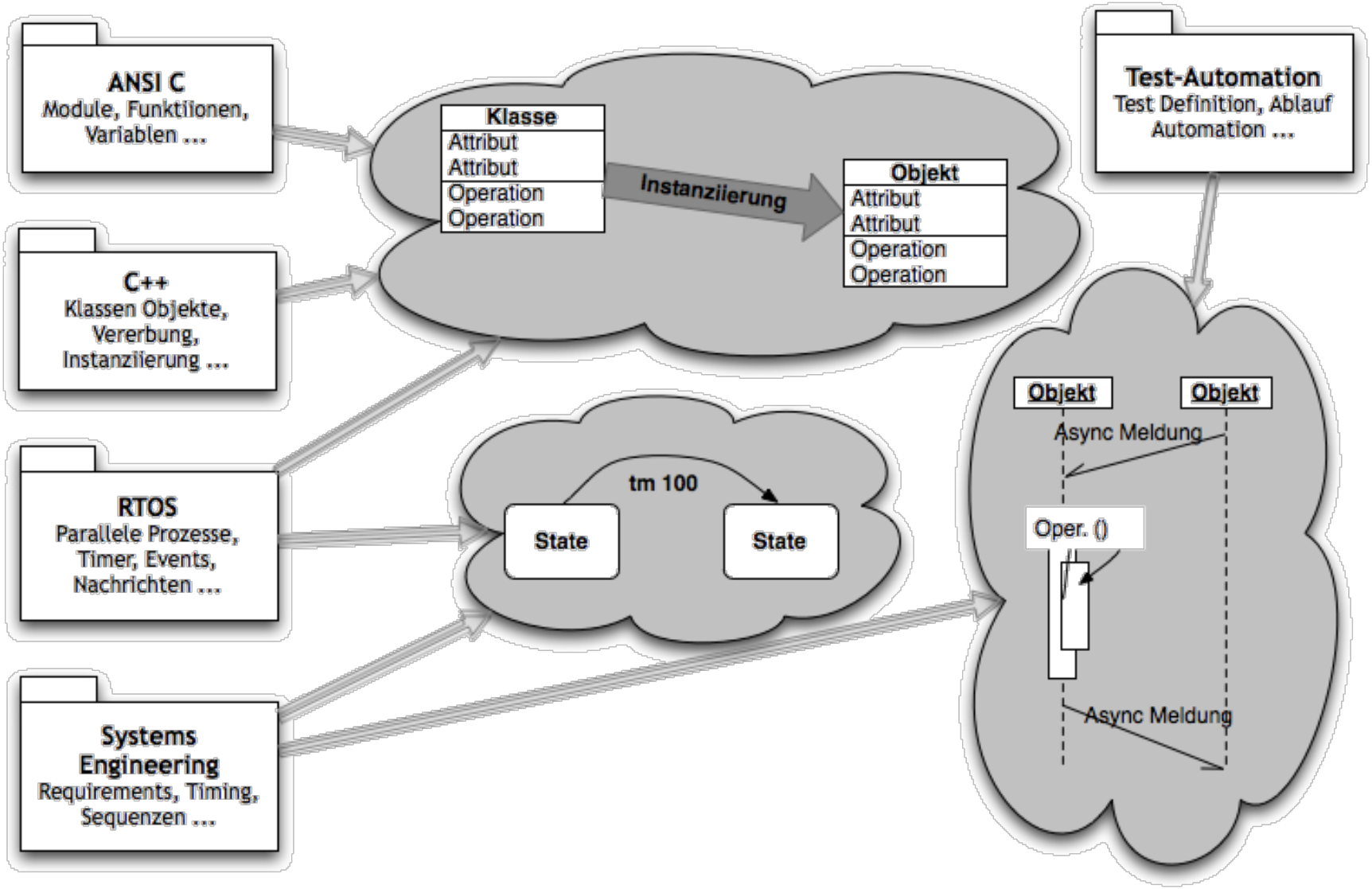
- Each module (Task) has its own protection register context
- PXROS-HR manages the memory protection
- Access violation lead to a trap

  $\Rightarrow$ Adaptive error handling

When does Software Design Crash?

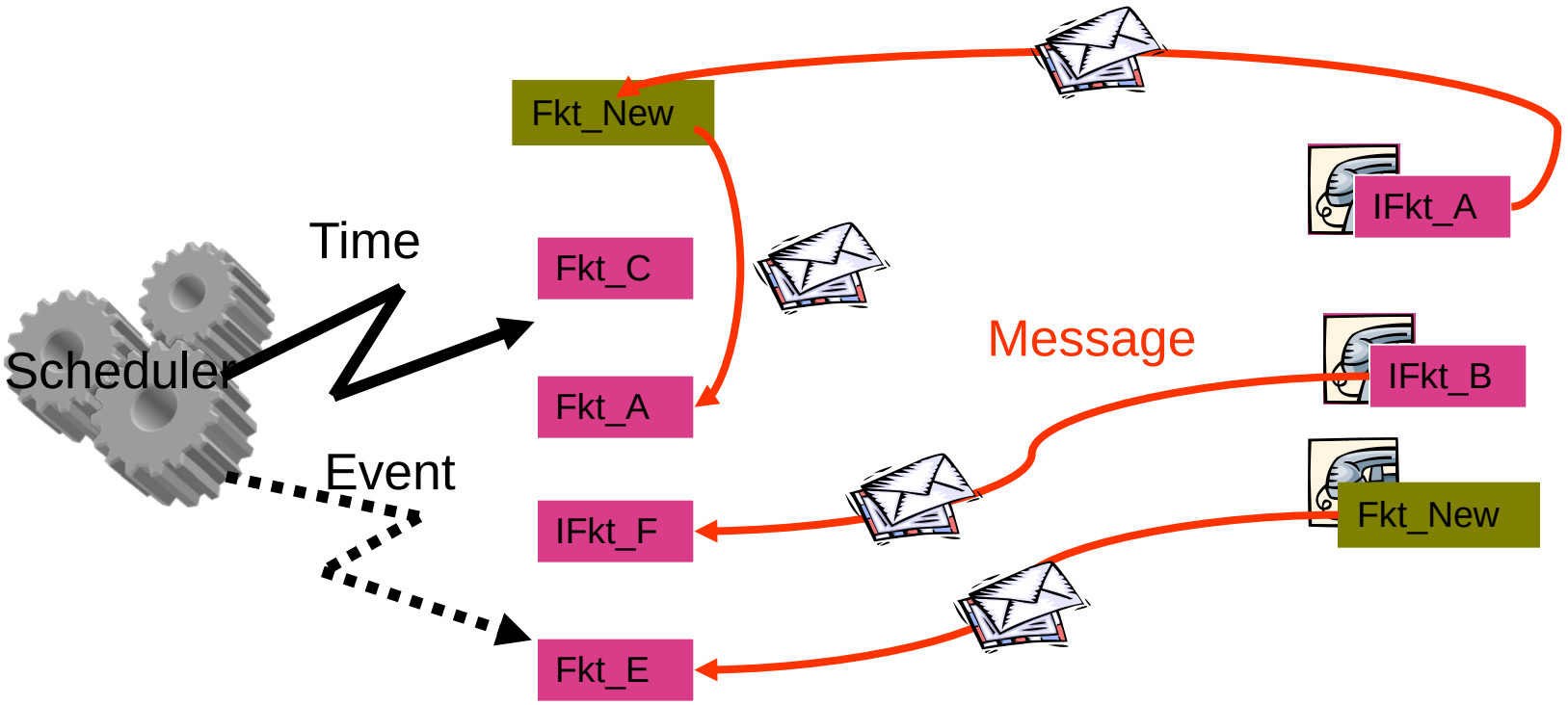


Engineering with UML

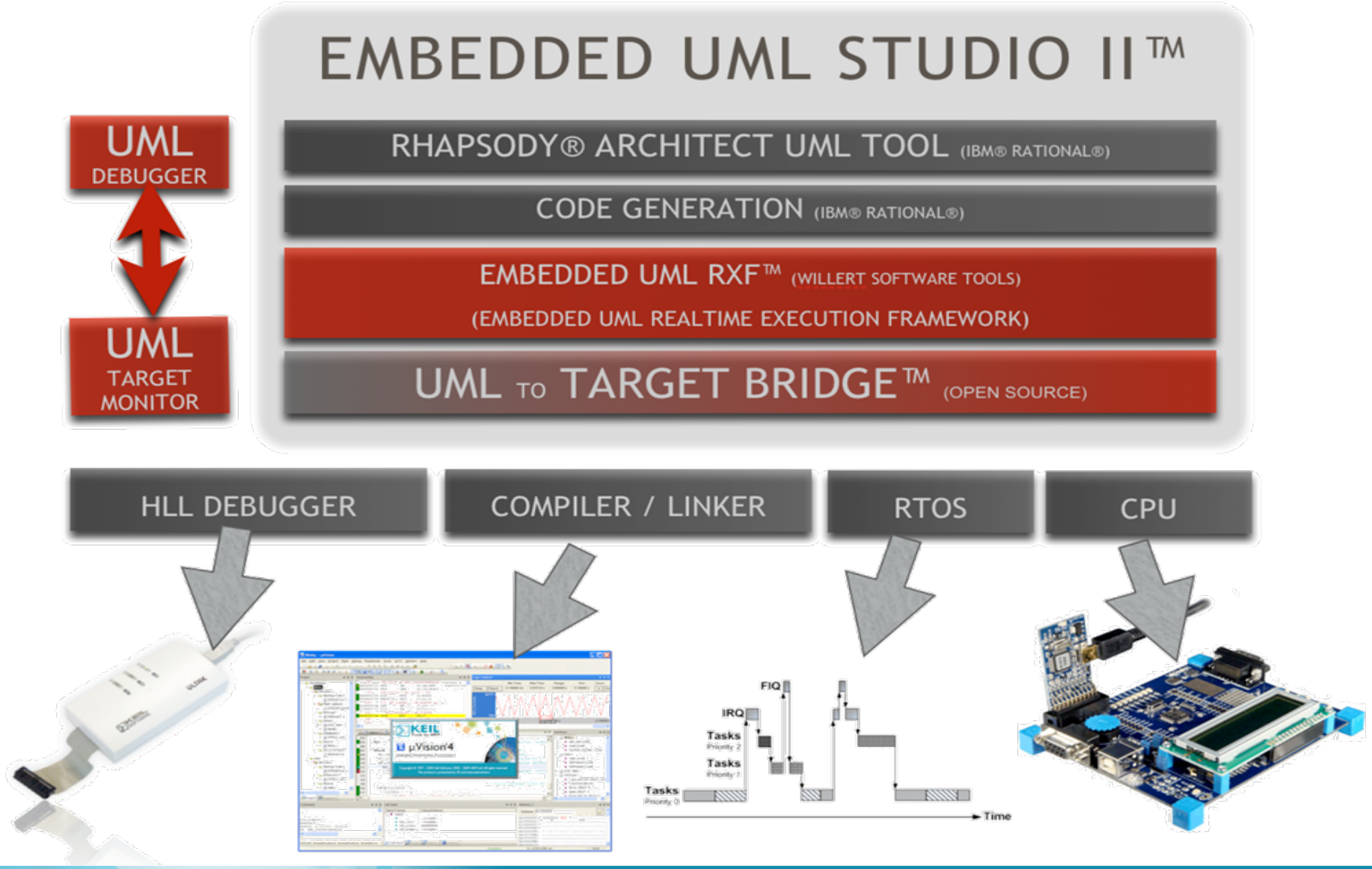


UML Forces Clean RTOS Design

- Timing: state charts allow timing specification
- Behaviour: state charts
- Dataflow: sequence diagram
- Priorities



Software Design with UML

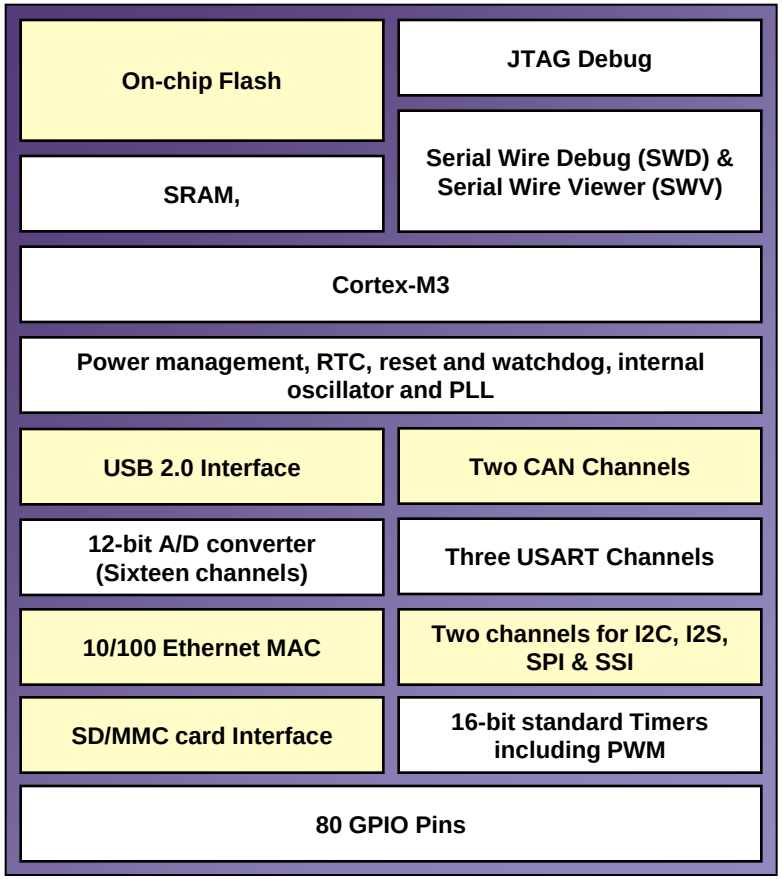


Middleware for Embedded Applications

- Why is Middleware Needed?
- Middleware Vendors (Selection)
- Typical Middleware Components

Today's Microcontroller Selection

- Microcontroller have
  - Processor
  - On-chip Memory
  - Interrupt System
  - Rich peripheral set
    - I/O Pins, Timers, PWM
    - A/D and D/A converters
    - UART, SPI, I2C
    - Complex communication peripherals (CAN, USB, Ethernet)



Block Diagram of a Standard Microcontroller

Embedded Connectivity Challenges

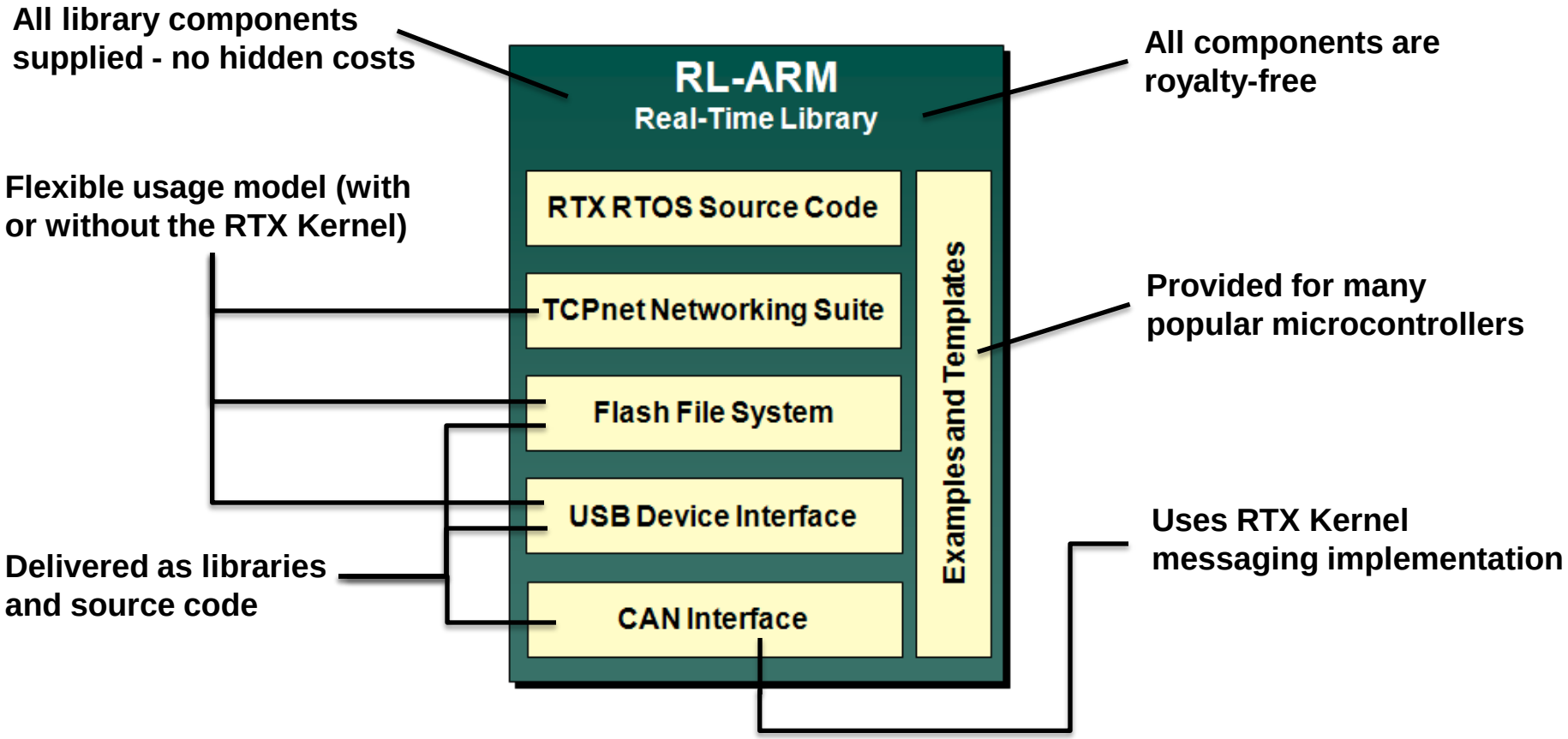
- Embedded devices are used everywhere
  - Need to support many different interfaces…
    - CAN, USB, SD/MMC, Ethernet
  - …and different protocols
    - HTTP, FTP, SMTP...
- Customers demand ease of use
  - Today's embedded devices need to support plug and play compatibility
- Developers need more functionality
  - Ability to support a wide range of interfaces
  - Need better development and debug tools for this task

Middleware Vendors (Selection)

- **CMX** www.cmx.com
  RTOS, TCP/IP, Flash File System, CANopen
- **HighTec** www.hightec-rt.com
  RTOS with MMU support, Monitor, TCP/IP, Flash File System
- **Keil/ARM RL-ARM** www.keil.com/rl-arm
  RTOS including collection of Middleware components: TCP/IP Suite, Flash File System, USB, CAN
- **Micrium** www.micrium.com
  RTOS along with rich set of middleware components
  TCP/IP, Bluetooth, USB Device/OTG/Host, Graphic Library
- **Quadros** www.quadros.com
  RTXC RTOS, TCP/IP, USB, file systems
- **SEGGER** www.segger.com
  Graphic Library, RTOS, TCP/IP, USB, File System, Boot Loader, etc.
- **Thread-X** www.rtos.com
  Graphic Library, RTOS, TCP/IP, USB, File System, etc.

What is RL-ARM?

- A collection of resources for solving these challenges
  - Middleware components created and used by ARM engineers



RL-ARM: TCPnet Networking Suite

- Add network support to your projects quickly and easily
  - Libraries support common network protocols
  - Supplied with templates and examples ready to port to any target
  - Take advantage of standard networking applications

Email, SMTP

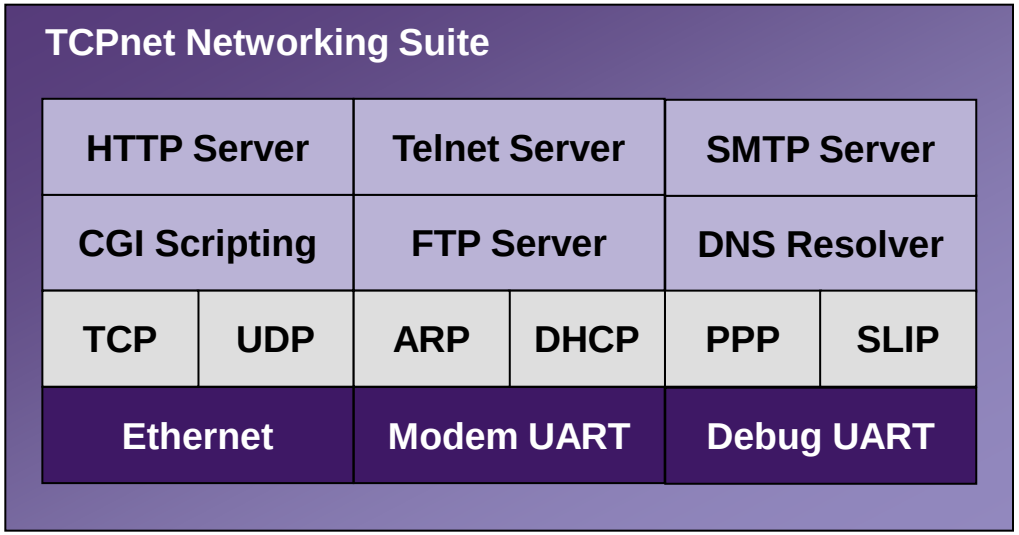
Modem, PPP

Remote Access, Telnet

Serial, SLIP

Web interface, HTTP

ARP, IEEE 802.xx network



END

Thank you