

# **Introduction to Computer Interfacing and Embedded Systems**

# What is ECE 315 all about?

---

- ECE 315 is titled “Computer Interfacing” but the course actually covers material from a broader range of areas:
  - ❖ Microcomputers and hardware interfaces
  - ❖ Software for real-time embedded systems
  - ❖ Computer communications interfaces
- This course considers the design and debugging of systems that involve the interaction of microcomputer ***hardware***, embedded ***software***, and ***communications interfaces***.
- ECE 315 is intended to prepare students for ECE 492, the Computer Engineering Design Project.

# Prerequisites for ECE 315

---

- 1) A first course in ***microcomputer architecture***, such as:  
ECE 212 – Introduction to Microprocessors, or  
ECE 311 – Computer Organization and Architecture, or  
CMPUT 229 – Computer Organization and Arch. I, or  
Permission of the instructor
  
- 2) An intermediate course in ***C/C++ programming***, such as:  
CMPUT 201 – Practical Programming Methodology, or  
ECE 220 – Programming for Electrical Engineering, or  
Permission of the instructor

# Other courses that are related to ECE 315

---

- 1) A survey course in ***operating systems***, such as:  
CMPUT 379 – Operating Systems Concepts
  
- 2) A survey course in ***computer communications***, such as:  
CMPUT 313 – Computer Networks, or  
ECE 487 – Data Communications Networks

The key required concepts from operating systems and computer communications will be covered in ECE 315.

# What is an “Embedded System”?

---

- **Personal Computers** (PCs) are the most obvious form of programmable digital computer. There is a screen for outputting text and images to a human user, and a keyboard and mouse (or touch-screen equivalents) for inputting characters and selection decisions from menus.
- However, the number of PCs and tablets in an advanced economy is dwarfed by at least a factor of 10 by the much larger number of embedded systems.
- An **embedded system** is a computer system that is used to monitor and control a product or engineering system. An embedded system often has only a simplified and/or application-specific user interface. A conventional PC or tablet user interface might be absent in such a system.

# Industrial Process Control

---

- Process control systems are widely used to allow a relatively small number of human operators to control a complex industrial process (e.g., oil refinery, natural gas processing plant, pulp mills, power generation plant, car assembly line) with high consistency, efficiency, and safety.
- A wide variety of technologies are used to implement modern *process control systems* (PCSs):
  - Ladder logic, *Programmable Logic Controllers* (PLCs)
  - *Remote Terminal (or Telemetry or Telecontrol) Units* (RTUs)
  - *Proportional-integral-derivative* (PID) controllers
  - *Supervisory Control and Data Acquisition* (SCADA) systems
  - *Distributed Control Systems* (DCSs)
  - The “Internet of Things”, Edge (or Fog, or Mist) Computing
- The focus in this course will be on software-programmed, networked embedded microcomputers, which are widely used in modern PCSs.

# Ex: Allen-Bradley Micro850 PLC

---

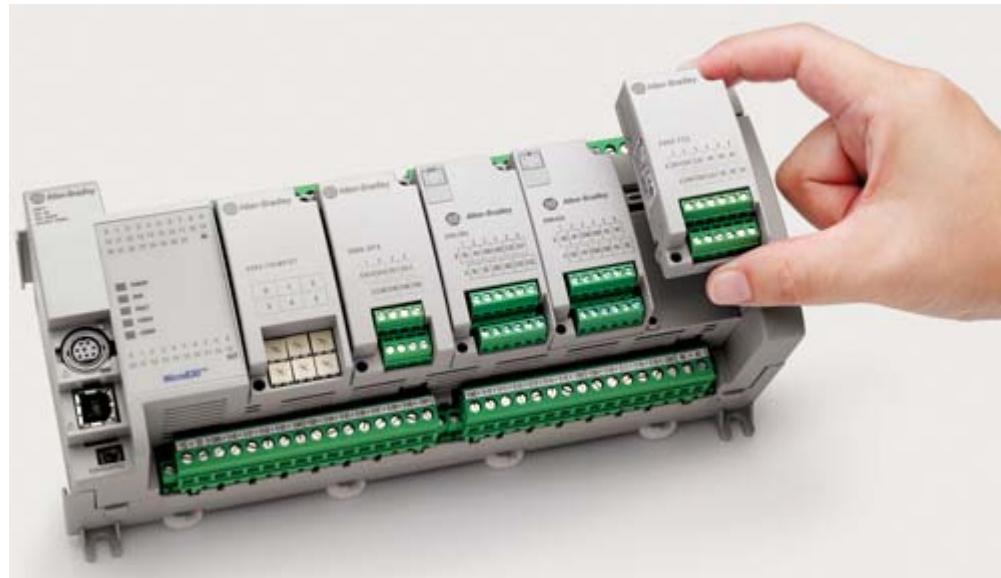


Photo courtesy of Rockwell Automation

- A typical high-end *Programmable Logic Controller* (PLC)
- Highly customizable with expansion I/O modules, terminal blocks, etc.
- Can be expanded to handle up to 132 digital input/output signals
- Can be programmed using three standard PLC methods: (1) ladder diagrams, (2) function blocks, and (3) structured text.

# Supervisory Control and Data Acquisition Systems

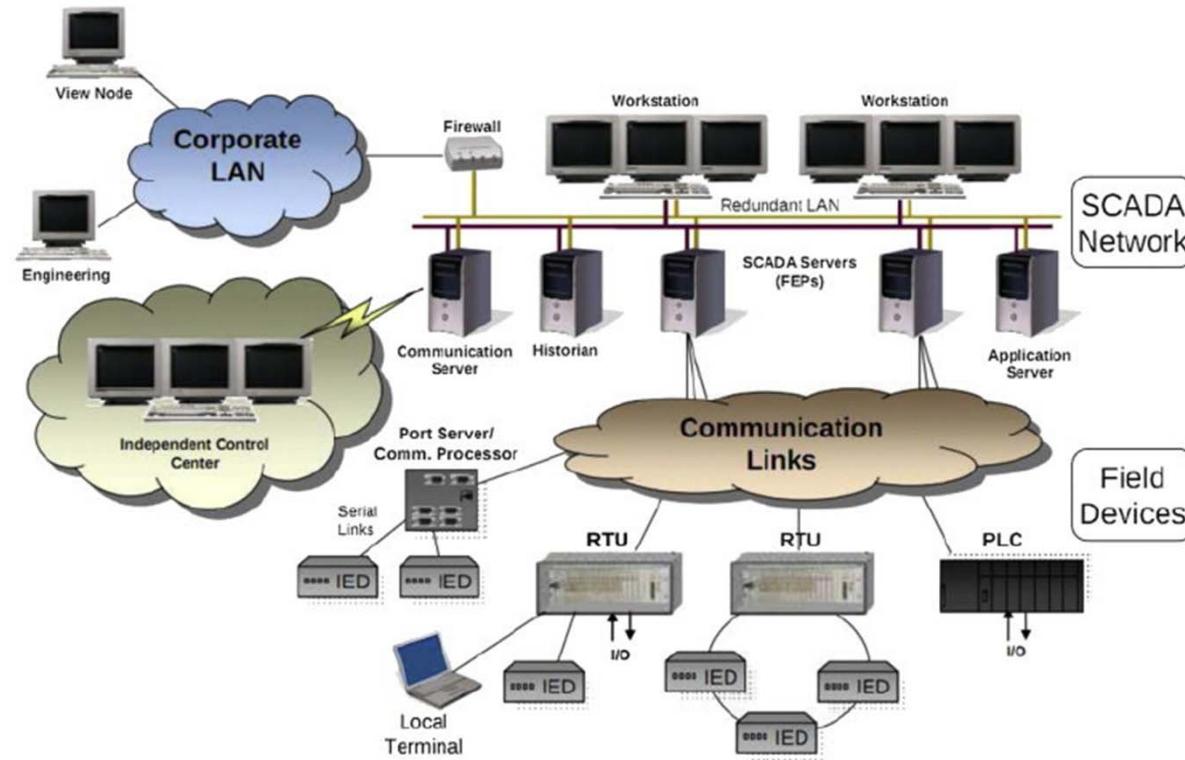


Photo courtesy of Dr. Helge Janicke, De Montfort Univ.

- SCADA systems are typically geographically distributed systems that coordinate and supervise *remote terminal units* (RTUs) and *programmable logic controllers* (PLCs) over a communications network.
- SCADA systems play a major role in Alberta industry (e.g., oil & gas)

# The Internet of Things (IoT)

---

- “The ***Internet of Things*** (IoT) is the network of physical devices, vehicles, home appliances and other items embedded with electronics, software, sensors, actuators, and network connectivity which enables these objects to connect and exchange data.” [Wikipedia, Jan. 8, 2017]
- The term was reportedly introduced by Kevin Ashton in a presentation that he gave at Proctor & Gamble in 1999.
- There is nothing particularly new about the Internet of Things: at present it is a fashionable term that describes a technology that has been emerging for many years already.
- The current popularity of the Internet of Things is serving a useful purpose in that it highlights both the growing *opportunities* as well as the *risks* of the increasingly pervasive use of networked embedded systems.

# Other embedded computing technologies

---

There is a somewhat confusing variety of terminologies:

- Sensor networks, mechatronics
- The Internet of Things, smart distributed infrastructure
- Cloud computing vs. edge computing
- Fog computing, mist computing
- Ubiquitous computing (ubicomp), pervasive computing
- Smartphone, smart city, smart grid, smart home, etc.
- Ambient Intelligence

New terms come into fashion all the time.

# The Microcomputer “Brain”

---

- Lying at the heart of an embedded system is a microcomputer ( $\mu$ C) that contains a ***Central Processing Unit*** (CPU), different kinds of ***memory***, and ***input/output subsystems***. The CPU fetches, decodes & executes software instructions.
- A ***microprocessor*** ( $\mu$ P) is a digital system that contains a CPU and closely related subsystems such as an ***interrupt handling subsystem*** and a ***cache memory***.
- A ***microcontroller unit*** (MCU) is a miniaturized system-on-a-chip (SoC) that contains, on a single semiconductor integrated circuit (IC), one (or more) ***microprocessor(s)*** and ***supporting subsystems*** (e.g., timers, interfaces) that are programmed to implement the desired embedded system.

# Where do we find Embedded Systems?

---

- In ***consumer electronics***: TVs, cell phones, electronic games, WiFi routers, entertainment systems, cameras, ...
- In ***residences***: microwave ovens, refrigerators, thermostats, high-efficiency furnaces, security systems, solar panels, ...
- In ***automobiles***: used to control the ignition timing, the fuel injectors, electrical power generation and distribution, fault detection and diagnosis, “black box” event recording, anti-lock braking, entertainment systems, anti-theft, wireless locking & ignition, maps & navigation, autonomous driving, ...
- In ***industry***: infrastructure (e.g., communications, electrical power, banking system, stock markets), numerical controlled tools, factories, sawmills, refineries, oil fields, gas plants, ...

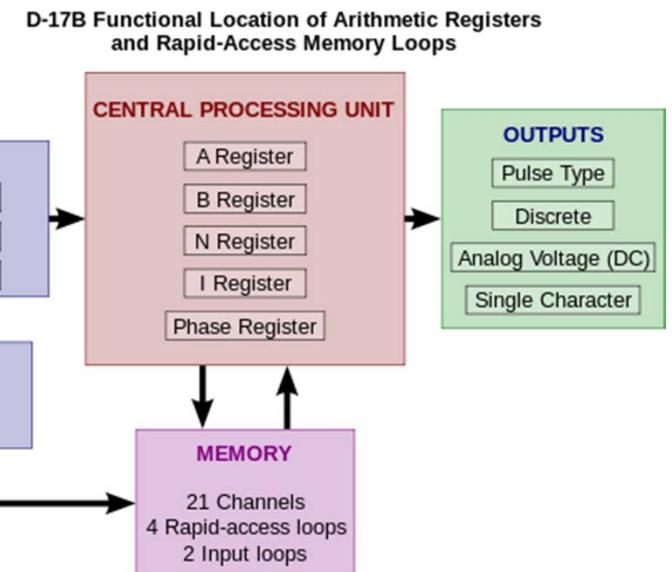
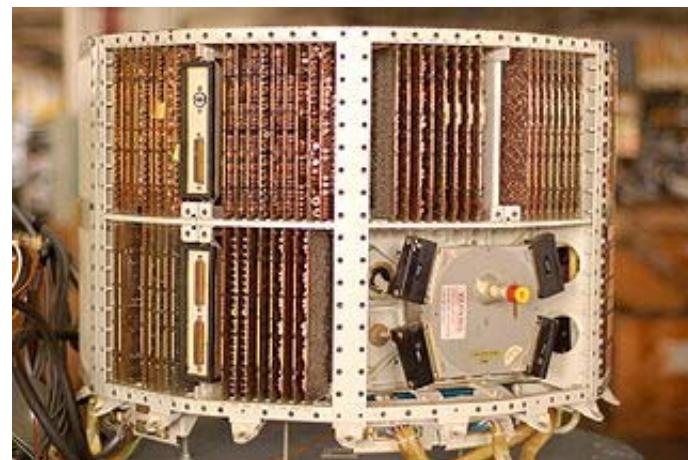
# Why Use Embedded Systems?

---

- Embedded systems provide high-speed, ***software-programmable*** sensor monitoring and signal processing, data processing, actuator control, communications capabilities, etc.
- Computerization using an embedded system permits the use of ***sophisticated control algorithms*** that can provide greater efficiency, productivity, reliability, flexibility, upgradeability, safety, and profitability (both for the vendors and users).
- Embedded systems can compensate for the inaccuracies of lower-quality sensors and actuators to ***provide an effectively higher-quality system at a lower total cost***.
- Hardware and software companies continue to push for new applications of embedded systems to grow their markets.

# Early Embedded Systems (1)

- 1962: Autonetics / North American Aviation D-17B military computer used to implement the guidance system for the Minuteman I ICBM. Weighing 28 kg, the D-17B contained 1521 discrete transistors, 6280 diodes, 1116 capacitors, 504 resistors, and dissipated 250 W at a clock frequency of 345.6 Hz.



Data and images from [www.wikipedia.org](http://www.wikipedia.org)

Copyright © 2020 by Bruce Cockburn

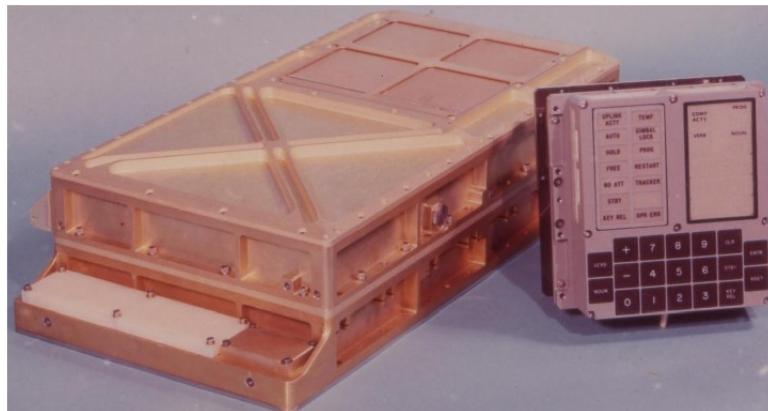
1-14

# Early Embedded Systems (2)

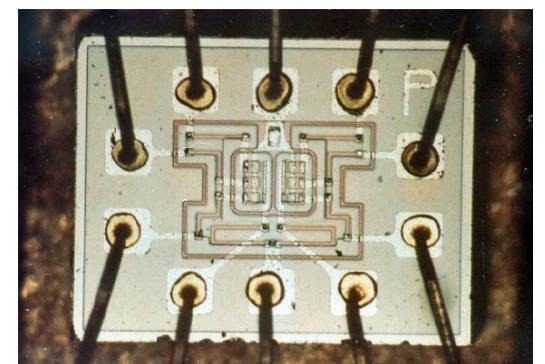
- 1966: MIT Instrumentation Laboratory / Raytheon Apollo Guidance Computer (AGC), used to control the Apollo spacecraft. Weighing 32 kg, the AGC contained 2800 integrated circuits (dual 3-input NOR gates) and dissipated 55 W with a 4-phase 1.024-MHz clock.



Apollo Command & Service Modules



AGC Computer and DSKY User Interface



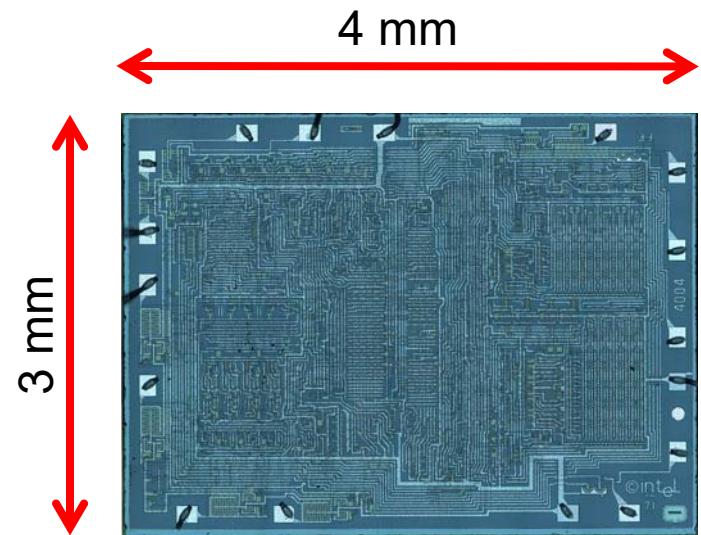
RTL Dual 3-input NOR IC

Data and images from [www.wikipedia.org](http://www.wikipedia.org)

# Early Embedded Systems (3)

- 1970: The Busicom 141-PF calculator was designed, in collaboration with Intel Corp. to use a 4-bit microprocessor, the 4004. The 4004 was the first commercially successful microprocessor IC. To implement the full embedded system, the 4004 was supported with a 256-byte ROM & 4-bit I/O port (the 4001), a 40-byte RAM (the 4002) and a 10-bit parallel shift register (the 4003). The 4004 contained 2300 transistors and used a 740 kHz clock to execute from 46300 to 92600 instructions per second.

Unicom 141P, a version of the Busicom 141-P



Intel 4004 µP in 10-µm PMOS

Data and images from [www.wikipedia.org](http://www.wikipedia.org)

# The Irving Oil Refinery in St John, NB

---



- Canada's largest oil refinery, capable of processing 300,000 barrels of oil per day
- Computer control is required to maximize production efficiency and ensure safety.



Photos from the Globe and Mail  
Copyright © 2020 by Bruce Cockburn

# Keyera's Gas Complex in Rimby, AB

---

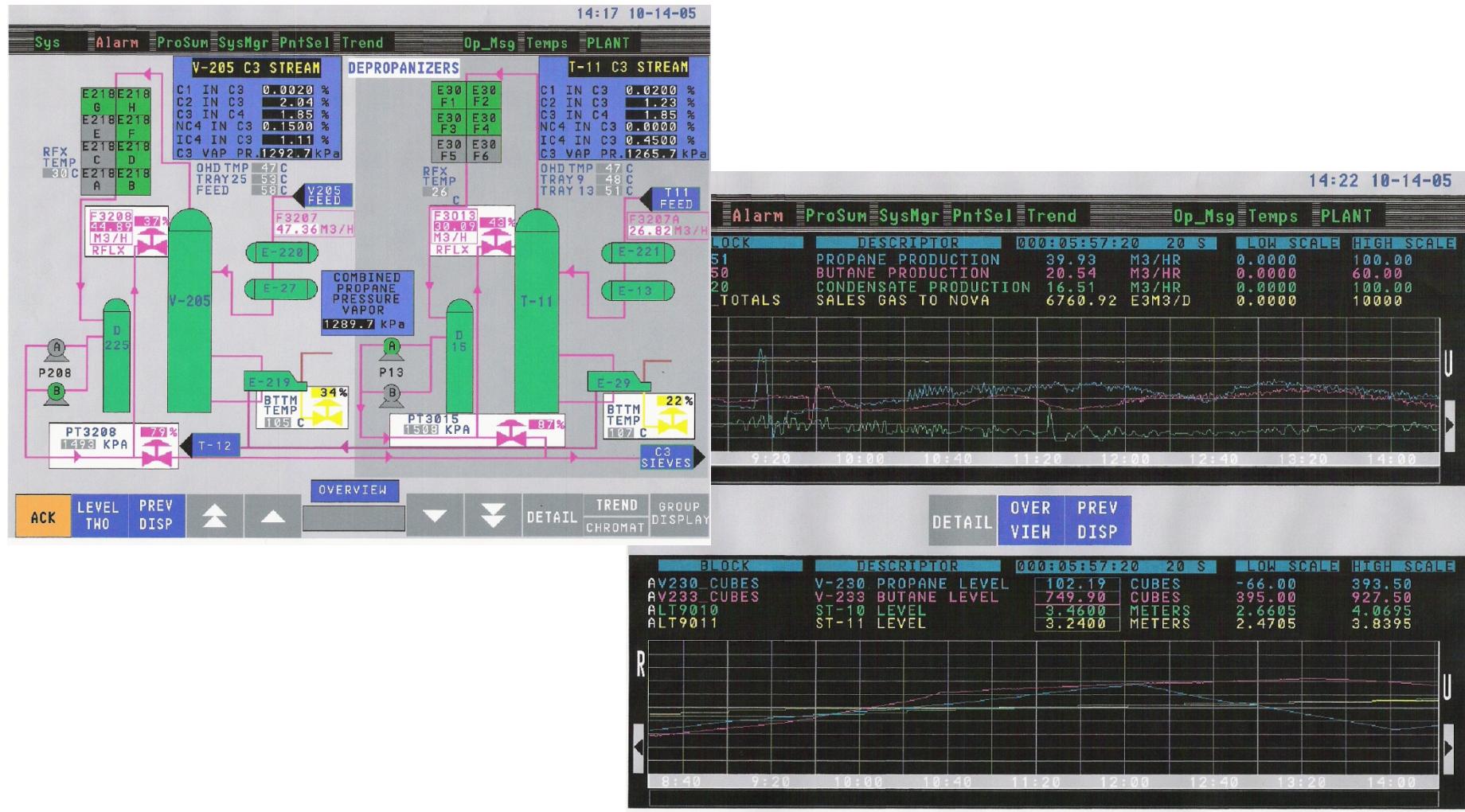


Photo courtesy of Keyera Corp.

Copyright © 2020 by Bruce Cockburn

1-18

# Views of a Control Console at the Gas Plant



Photos courtesy of Keyera Corp.

Copyright © 2020 by Bruce Cockburn

# Apple iPhone 7 Teardown

---



Photos from TechInsights (Ottawa, ON)

Copyright © 2020 by Bruce Cockburn

1-20

# Apple iPhone 7 Main Printed Circuit Board (PCB)

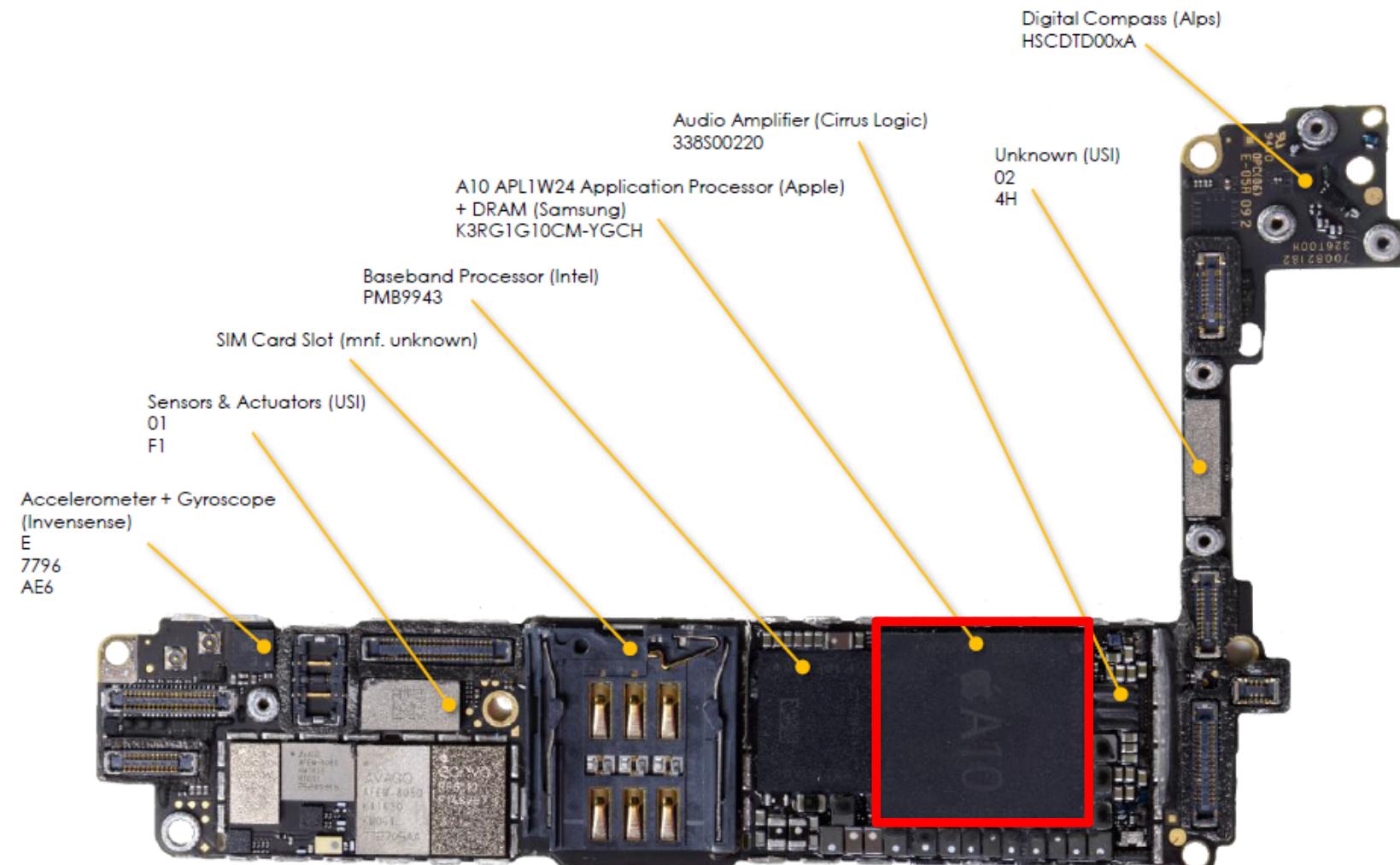


Photo from TechInsights-ChipWorks

# Apple A10 Application Processor



Photo from Tech Grapple

- Two 2.34-GHz 64-bit CPUs
- Also, two low-power CPUs
- Six-core graphics processor
- TSMC 16-nm FinFET CMOS
- 125 mm<sup>2</sup> die area
- 3.3 billion transistors

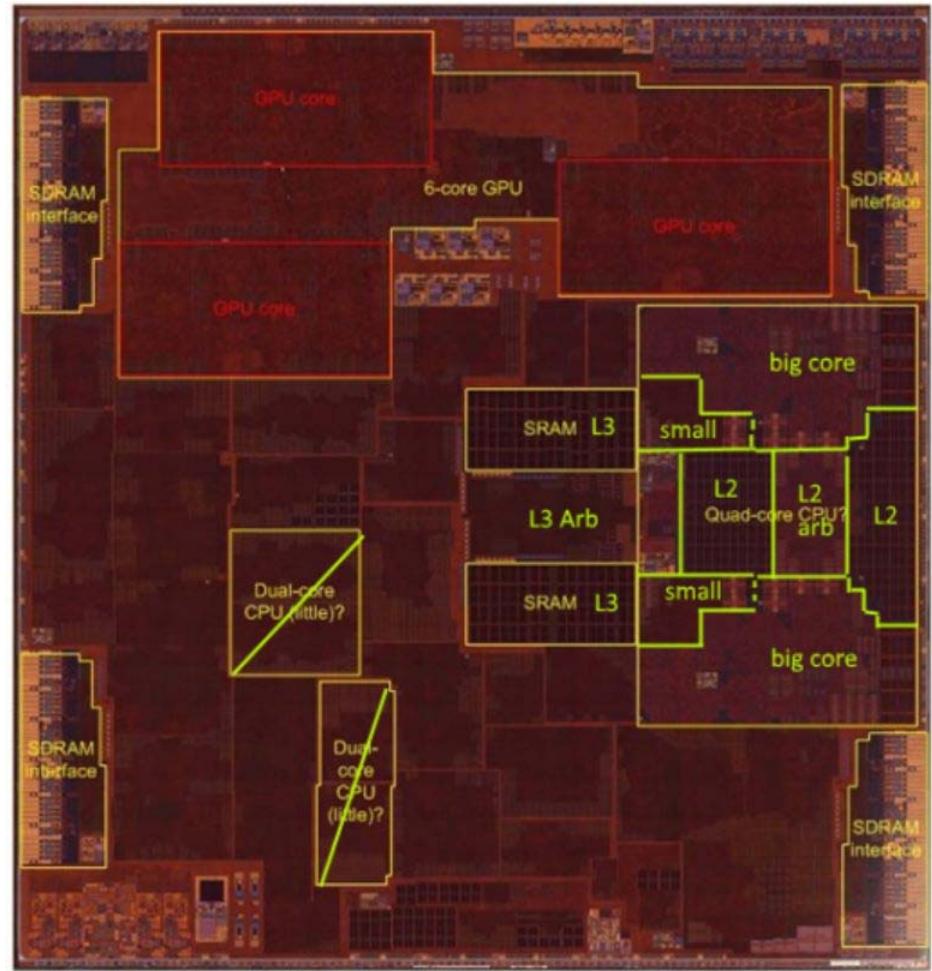
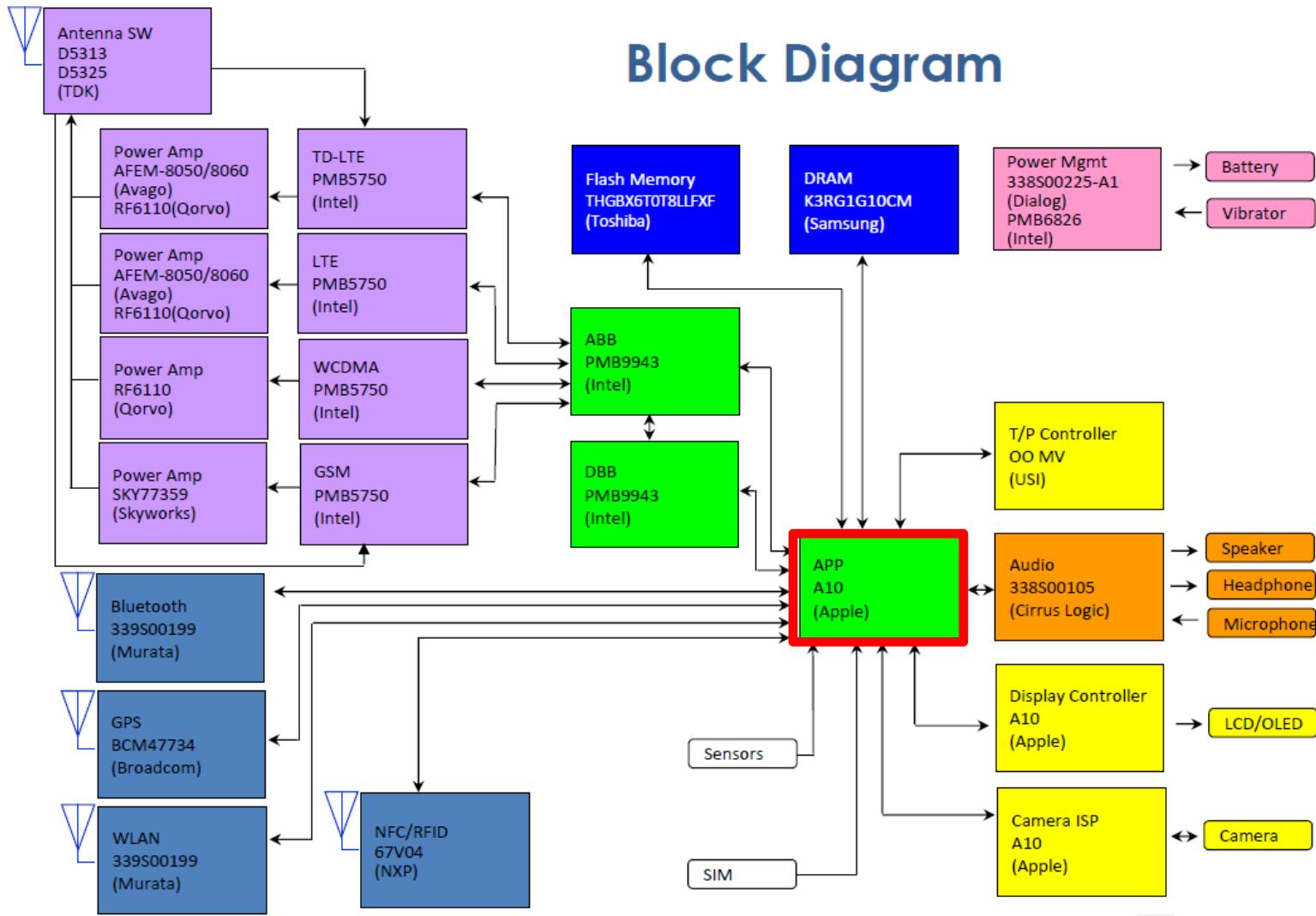


Photo from TechInsights-ChipWorks

# Apple iPhone 7 System Architecture



TechInsights CONFIDENTIAL. All content © 2016.  
TechInsights Inc. All rights reserved.

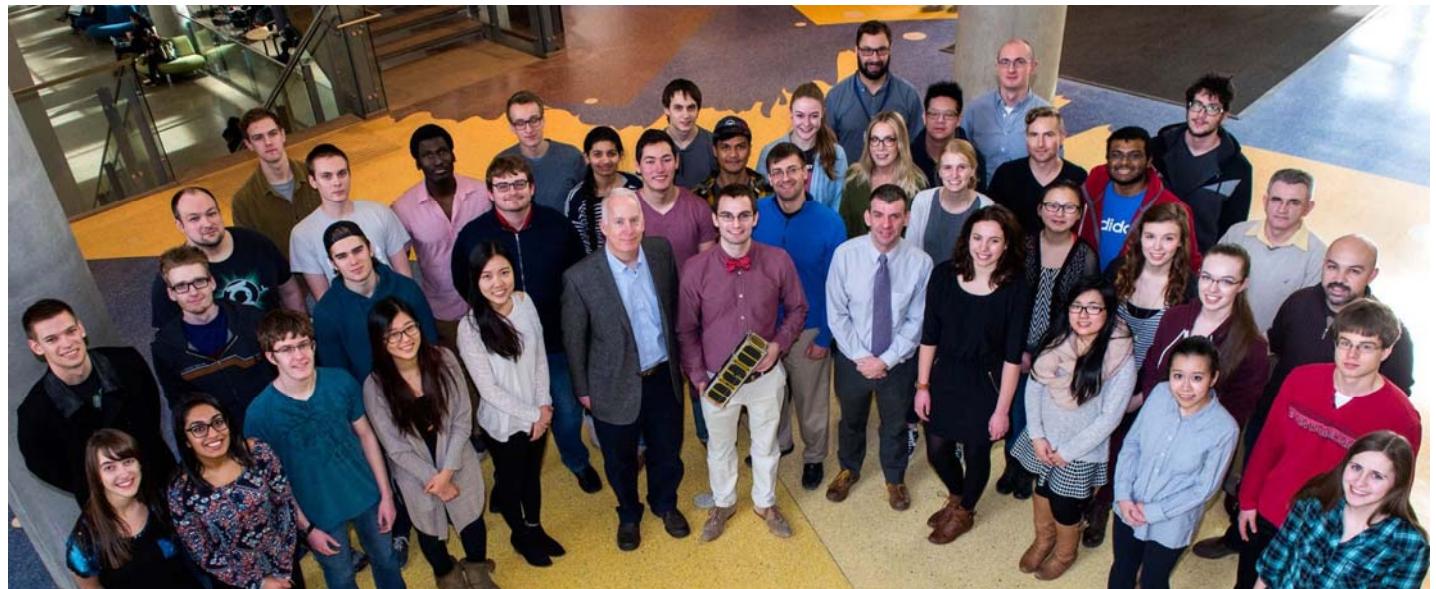
TECHINSIGHTS chipworks

# “Ex-Alta 1” CubeSat Satellite Designed at the U of A

---

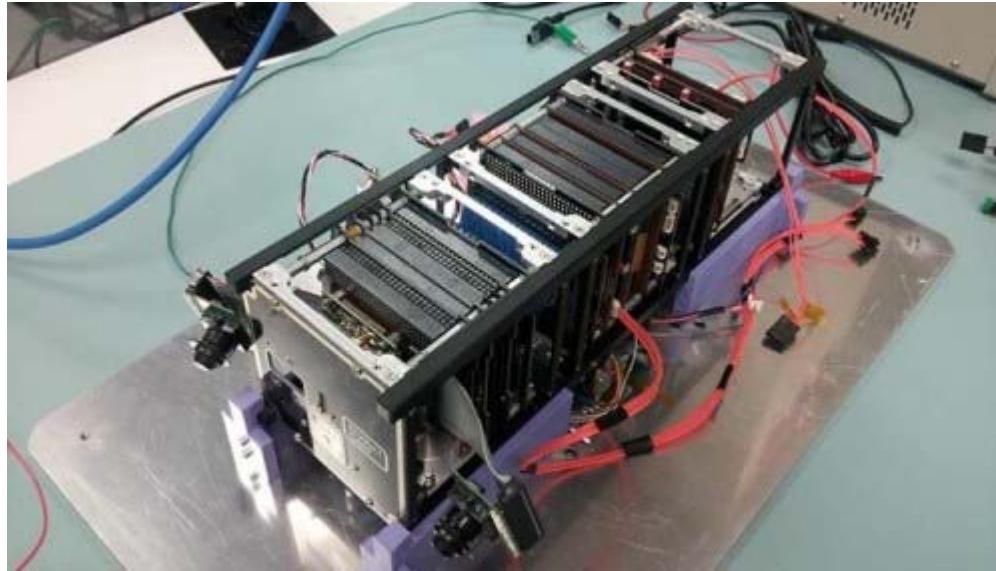


- A  $10 \times 10 \times 30 \text{ cm}^3$  cube satellite
- Launched to the Int'l Space Station May 26, 2017. Ejected into orbit May 28, 2017.
- The orbit decayed on Nov. 14, 2018
- Tested an experimental digital fluxgate magnetometer (Dept. of Physics, U of A)
- Participated in a multipoint space plasma physics experiment



# Ex-Alta 1 Embedded System Details

---

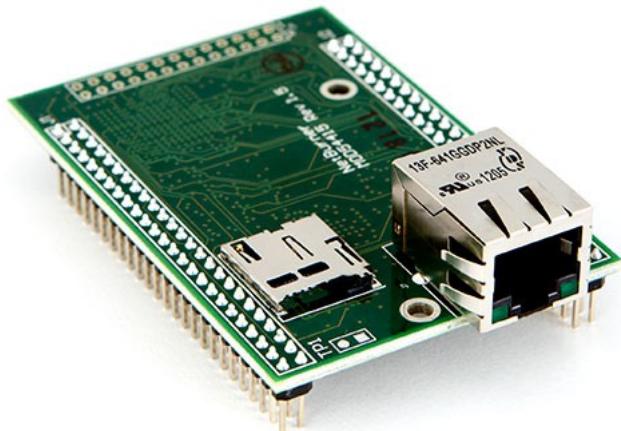


- The basic architecture is a 3U CubeSat platform (Innovative Solutions in Space, Delft, the Netherlands).
- Microcomputer: PC104 class (9.0 cm × 9.6 cm PCB) NanoMind A721D (GomSpace A/S, Aalborg East, Denmark) with 8 to 40-MHz 32-bit ARM7 CPU
- Memory: 2 MB RAM, 4 MB code flash, 4 MB data flash, 2 Gbyte SD card
- Comms.: Two I2C busses at 400 kbit/s using 68-byte transmit & receiver buffers
- Software environment: FreeRTOS and NanoMind software libraries

# The ECE 315 Embedded System

---

- The NetBurner MOD5441X “Ethernet Core Module” is a modern (from 2013) 32-bit microcomputer based on the 250-MHz Freescale MCF54415 microcontroller unit (from 2011). This System-on-a-Chip (SoC) has a 250-MHz, 32-bit ColdFire V4  $\mu$ P plus a large number of supporting subsystems: SDRAM controller, 10/100 Mbps Ethernet controller, 10 serial UARTs, 4 DMA-supported SPI interfaces, 8 12-bit ADCs, 2 12-bit DACs, 42 digital I/Os, etc.



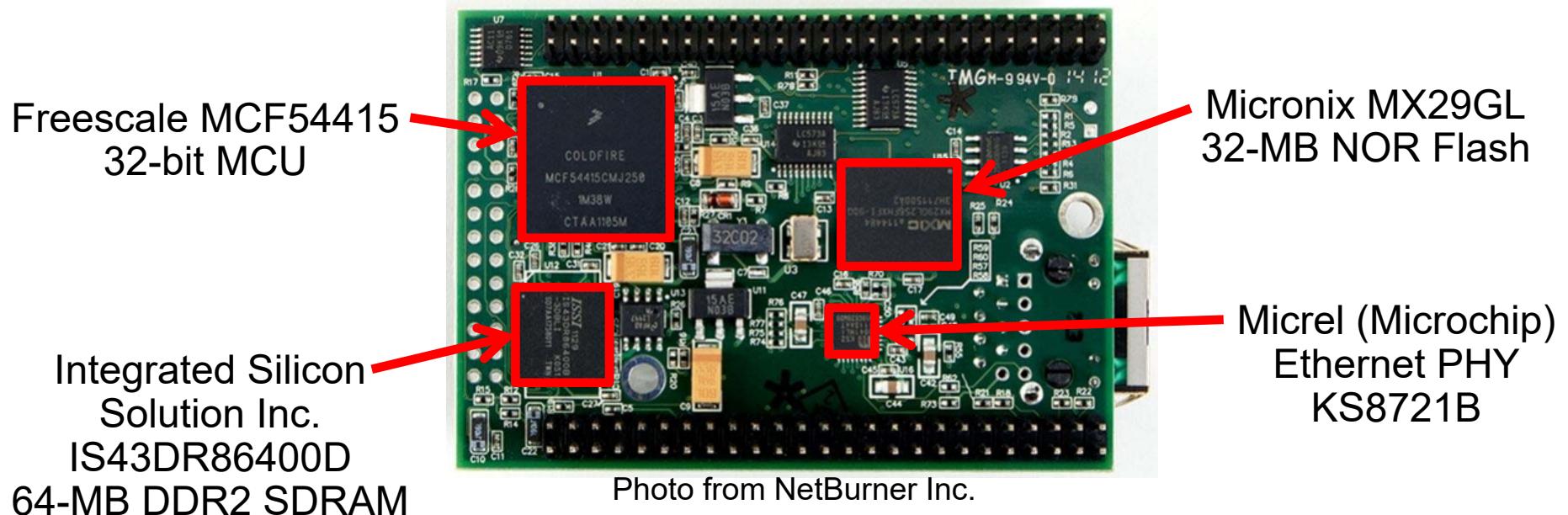
NetBurner MOD54415-100IR  
Ethernet Core Module



NetBurner MOD-DEV-70CR  
Development Board

# The NetBurner MOD54415-100IR Module

- The daughter card is the MOD54415-100IR Ethernet Core Module,
- The MOD54415-100IR is an embedded controller that has a 32-bit CPU (in the MCF54414 Microcontroller Unit), 32-Mbyte flash memory, 64-Mbyte SDRAM, and a 10/100 Mbps Ethernet interface.
- It also has numerous programmable peripheral interfaces.



# MCF5441X Microcontroller Architecture

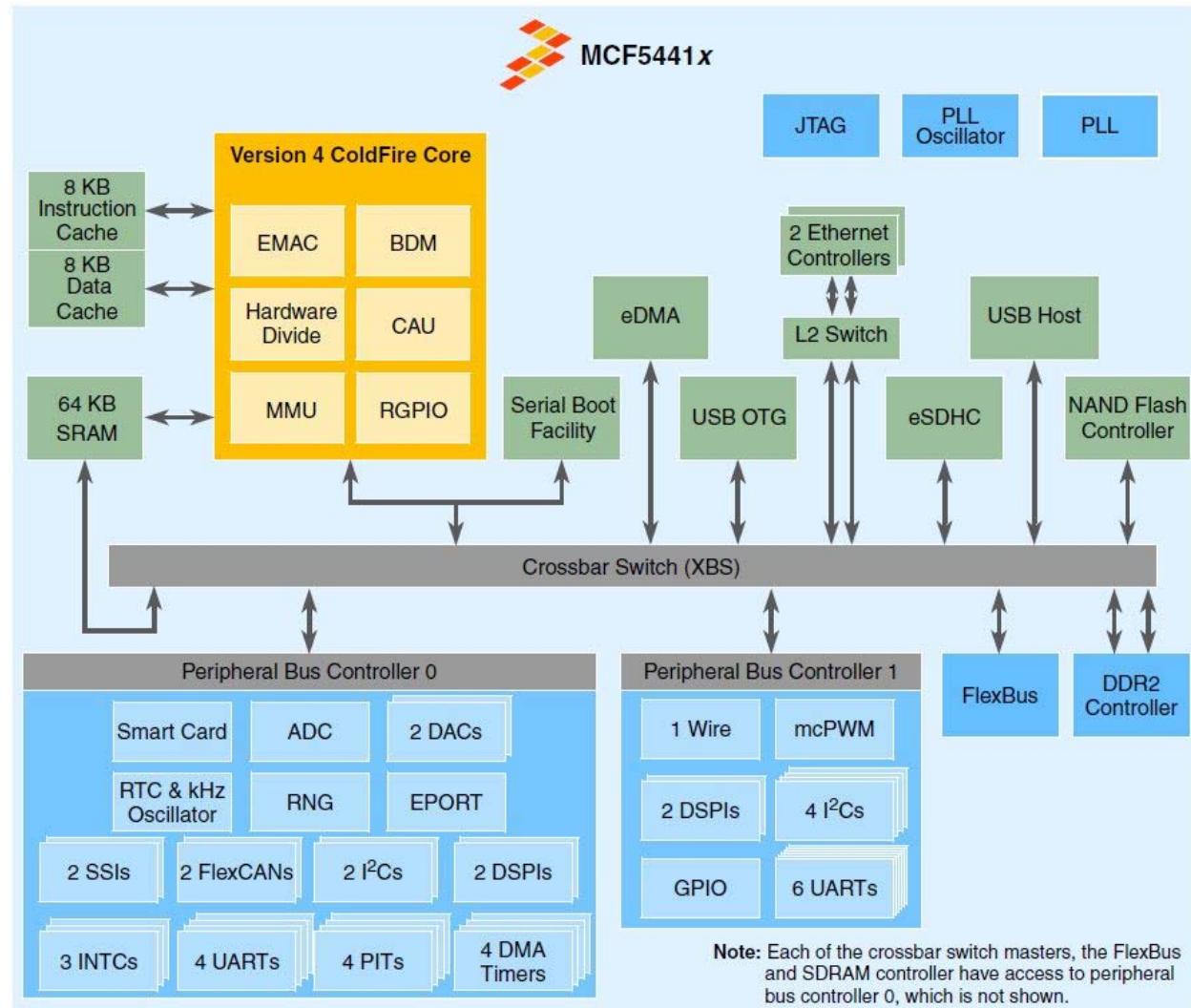


Diagram courtesy of Freescale Semiconductor Inc.

# What are “Real-time Embedded Systems”?

---

- A ***real-time embedded system*** is an embedded system that is fast enough to react to changes in the environment so that the surrounding system that it is responsible for controlling is indeed controlled correctly, safely and efficiently.
- An embedded system may fail to operate “in real time” for several possible reasons:
  - The processor is too slow for the required workload.
  - The software was inefficiently designed, or has become overburdened over time due to upgrades.
  - The real-time requirements of the system have been increased over time to become excessively demanding.

# Hard Real Time versus Soft Real Time

---

- It is common practice to distinguish between hard real time and soft real time embedded systems.
- A ***hard real-time embedded system*** is an embedded system where violations of real-time constraints would definitely be unacceptable. Constraint violations could cause injury or death to humans, equipment damage, loss of material, or serious financial loss.
- A ***soft real-time embedded system*** is an embedded system where occasional violations of the real-time constraints would not be desirable, but could be tolerated. For example, a vending machine or an Automated Teller Machine (ATM) can tolerate small delays with only minimal occasional frustration to human users.

# Performance Measures for Embedded Systems

---

- ***Algorithm correctness:*** Were the intended algorithms correctly applied to the system inputs and stored data to produce the desired system state changes and outputs?
- ***Response time:*** Did the embedded system respond fast enough to input changes (from external signals & user commands) with the appropriate updated outputs?
- ***Resource efficiency:*** Was the consumption of electrical power, natural gas, water, other chemicals, communication bandwidth, etc. minimized when accomplishing the work?
- ***Initial cost:*** How much did it cost the customer to buy and/or build and then install the embedded system?

# Other Important Performance Measures

---

- ***Predictable or deterministic response time:*** Is the response time sufficiently predictable (that is, have a small variance) as well as being acceptably fast?
- ***Well-structured design:*** Was the system architecture (both software and hardware) designed to minimize the engineering costs, to minimize the introduction of design errors, and to minimize the costs of implementing future engineering change orders (ECOs)?
- ***Total lifetime operating cost:*** How much will it cost to operate the system over its expected lifetime, including maintenance, upgrade & repair costs, training costs for personnel, recycling & disposal costs, etc.?

# Safety Critical Systems

---

- Embedded systems are increasingly placed in control of systems that could cause unacceptable damage and loss in the event of technical failure. Examples: medical equipment, transportation systems, nuclear reactors & weapons systems.
- In *safety critical design* the system is designed so that, for all of the expected failure scenarios, the overall system will either continue to operate (perhaps with reduced functionality) or be shut down safely by the embedded system in such a manner that damage and loss to people, to equipment, and to the environment are avoided or at least minimized.
- In a *fail-safe design*, a serious failure will cause the system to be shut down and placed in a safe state.
- A *fault-tolerant system* is a system that can recover from a wide variety of failures and continue operating. Typically such a system has duplicated or redundant hardware.

# Joint Human-Computer Control

---

- Embedded systems are increasingly used to enhance and/or to assist in the control of systems along with humans. Ex:
  - Power steering, cruise control, antilock brakes in cars
  - Fly-by-wire in aircraft; autopilot in ships and vehicles
- Special care must be paid to ensure that the assistance provided by an embedded system will never worsen an operating situation (especially an emergency situation) in its interaction with human operators.
- The embedded system should clearly warn the human operator of dangerous situations. Embedded system control should be possible for the trained human operator to disable or override. There are few situations where an embedded system should be allowed to override a human operator.

# Why is Computer Interfacing important?

---

- Embedded systems must interact with an analog world.  
=> digital-analog & analog-digital interfaces are required
- Digital systems are constructed using digital subsystems.  
Few computer engineers design hardware from scratch.  
=> digital-digital interfaces between subsystems are required
- Software needs to initialize, control, & monitor the hardware.  
=> software-hardware interfaces (e.g., drivers) are required
- Software systems are constructed from software subsystems.  
=> software-software interfaces needed (e.g., function calls)
- Human users judge digital systems by the user interface.  
=> user interface design is critical to product success

# Typical Computer Interfacing Activities

---

- 1) Selecting software/hardware subsystems that can at least potentially work well with each other.
- 2) Providing appropriate hardware-hardware connections.  
Select standards, connectors, cabling, drivers, receivers, etc.
- 3) Configuring hardware interfaces using switches, jumpers, firmware, software, network settings, device drivers, etc.
- 4) Configuring software by selecting compatible software versions, invoking compilers and linkers with the correct parameters, enabling/disabling conditionally-compiled modules, calling drivers with the correct parameters, etc.
- 5) Designing any new “glue” hardware and software that might be required to get subsystems to interface correctly.

# **Review of Technical Background**

# The Nature of the Interfacing Problem

---

- Physical laws do not change
  - => Currents & voltages obey the laws of Kirchoff and Ohm.
  - => Signal propagation and noise calculations are the same.
  - => But digital & software systems don't obey simple laws!
- Hardware (H/W) is constantly evolving because of technological improvements and inter-company competition.
  - => System size & complexity increase rapidly with time.
  - => New features get added; old features are often retained.
  - => Multiple versions of the hardware must often co-exist.
- Software (S/W) flexibility is both an advantage and a danger.
  - => Complex H/W must be initialized & configured in S/W.
  - => S/W system complexity overwhelms any one designer.
  - => Multiple versions of software must often be maintained.
  - => Software invariably contains design defects (“bugs”).

# Typical Mechanisms at Hardware Interfaces

---

- 1) *Analog signal conditioning*  
e.g. buffers/drivers, noise rejection filters, anti-aliasing filters
- 2) *Analog-to-digital* and *digital-to-analog* signal conversion  
e.g. Analog inputs from sensors need to be converted into digital form to permit digital signal processing (DSP).  
e.g. Digital outputs need to be converted into voltage or current signals to control *actuators* in the analog world.
- 3) *Digital modulation* of an analog signal  
e.g. Digital signals must be encoded as modulated analog signals to propagate well over communication channels.  
*Modem* = modulator (transmitter) + demodulator (receiver)
- 4) *Timing recovery, demodulation & decoding* a modulated signal. Present in digital communication receivers.

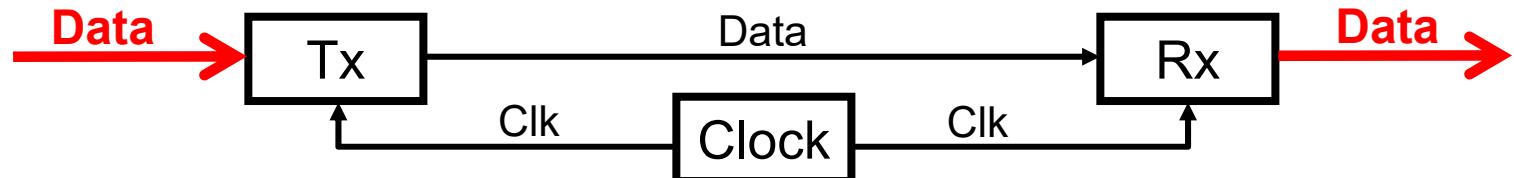
# Typical Mechanisms at Hardware Interfaces

---

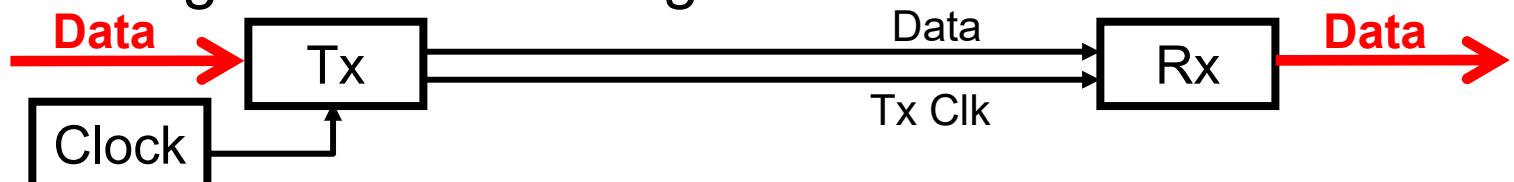
- 5) *Digital signal conversion, conditioning and isolation*  
e.g. Voltage level conversion, voltage-current conversion  
e.g. Optical isolation transceivers, isolation transformers
- 6) *Impedance matching (to avoid unwanted signal “ringing”)*  
e.g. Parallel termination resistance at the far end of a line  
e.g. Series termination resistance at the near end of a line
- 7) *Synchronization, framing, handshaking, error detection*  
e.g. Start and stop bits in RS-232C  
e.g. Request-to-send and clear-to-send handshaking  
e.g. Parity bit (or checksum) generation at transmitter  
e.g. Parity bit (or checksum) re-generation and checking  
at receiver. Possibly error correction at receiver.
- 8) *Data buffering and flow control to handle rate fluctuations*  
e.g. Hardware buffers inside transmitters and receivers

# Synchronous Digital Interfaces

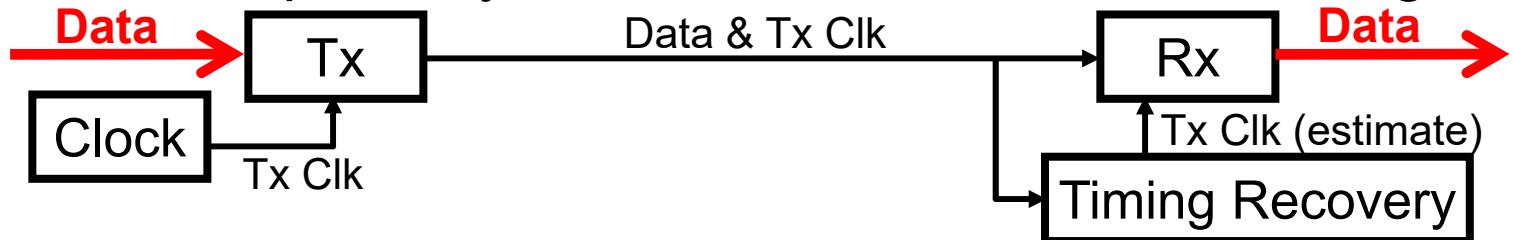
- 1) Transmitter (Tx) and receiver (Rx) clock signals are both directly derived from the same physical clock signal.



- 2) Transmitter clock is sent to the receiver as a separate signal along with the data signal.



- 3) Receiver clock is synchronized with a clock signal that is encoded along with the transmitter's data signal.  
There is no separately transmitted transmitter clock signal.



# Asynchronous Digital Interfaces

---

- The transmitter and receiver sides use physically distinct and unrelated clock signals. The clock frequencies may be nominally the same. More commonly, the raw receiver clock is some multiple (say 16x) of the transmitter clock.
- For example, if the receiver clock is 16x the transmitter clock, then the raw receiver clock can be divided by 16, and given one of 16 possible phases that is (at least initially) closest to the estimated phase of the transmitter clock.
- Thus the original transmitter clock can be approximated by the divided-down receiver clock. But this approximate clock will drift away in phase due to any frequency offset between the transmitter and divided-down receiver clocks.

# Semisynchronous Digital Interfaces

---

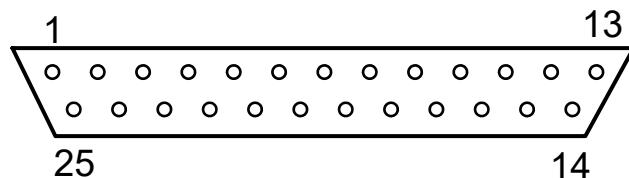
- Like in a synchronous interface, there is a common clock.
- Time is measured out as a number of whole clock periods.
- Communication events (e.g., read & write operations on a bus) take a *variable number of clock periods* to complete.
- The number of clock periods can be determined either by (1) handshake signals “on-the-fly” at the time of the event, or (2) by some fixed number that is programmed in the hardware.
- Hence we get much of the *timing flexibility* of an asynchronous interface, while retaining many of the *simplifications of synchronous (i.e., clocked) design*.
- Most modern digital interfaces are semisynchronous.

# Serial Hardware Interfaces

---

- An interface through which the *data bits* are transmitted *one bit at a time* in each direction.

Ex. RS-232C



- Pin 1: ground (GND)
- Pin 2: *transmit data* (TxD)
- Pin 3: *receive data* (RxD)
- Pin 4: request to send (RTS)
- Pin 5: clear to send (CTS)
- Pin 6: data set ready (DSR)
- Pin 20: data terminal ready (DTR)

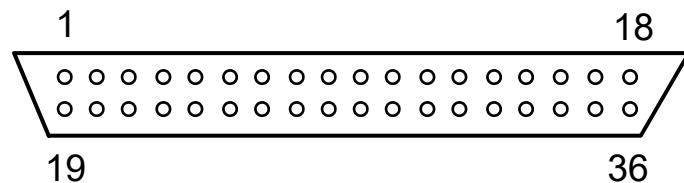
- Note: A smaller 9-pin version was widely used in PCs.

# Parallel Hardware Interfaces

---

- An interface through which *multiple data bits* are transmitted *at the same time, in parallel*, over different connections.

Ex. Centronics printer I/F



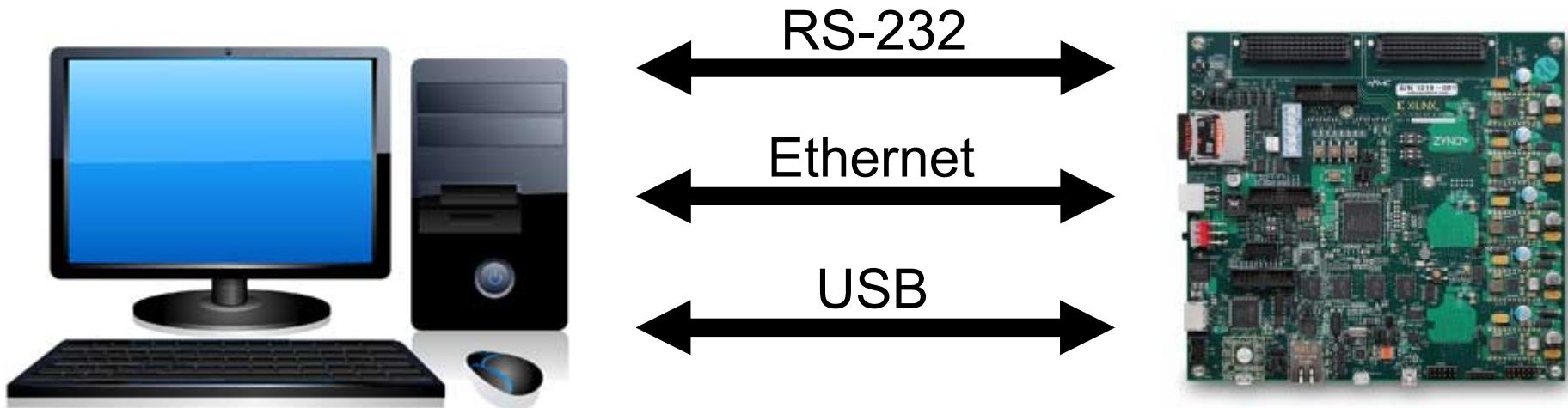
Pin 1: data strobe  
Pins 2-9: 8 *data bits*  
Pin 10: acknowledge  
Pin 11: busy  
Pin 12: paper end  
Pin 13: select  
Pin 17: ground  
Pin 18: +5 V  
Pin 32: fault detected

# Examples of Hardware Interfaces

---

	Synchronous	Asynchronous	Semisynchronous
Serial	USB peripheral	RS-232C terminal	ATM telecom
Parallel	6800 µP bus	GPIB instrument	ColdFire bus (ECE 315 system)

# Embedded Software Development (1)



*Host system* contains:

- full-featured O.S.
- Internet connection
- IDE
- source version control
- (usually) a cross compiler

*Target system* contains:

- boot loader
- ASCII interface
- monitor for debugging
- (often) file system
- (rarely) native compiler

# Embedded Software Development (2)

---

- Embedded systems range widely in performance and features (e.g., simple 4-bit microcontrollers, complex 32-bit microcomputers, microprocessor + FPGA on one chip).
- The variety and complexity (both hardware and software) of embedded systems tends to make embedded software development a challenging and hence expensive task.
- A variety of different software tools must be used during embedded software development: source code editors, compilers, build automation tools, debuggers, monitors, etc.
- Unless the tools are designed in a consistent way, each tool will tend to have a different user interface. Going from tool to tool will thus require mental “mode switching”, which is tiresome leading to lower programmer productivity and bugs.

# Integrated Development Environments (IDEs)

---

- An *integrated development environment (IDE)* is a single application that executes on the host system (e.g., personal computer or workstation) where software is being designed.
- A typical IDE (e.g., CodeWarrior, Eclipse) provides a complete set of software development tools (e.g., source code editor, compiler, build automation tool, version control system, debugger, etc.) that have a ***consistent user interface*** in order to maximize programmer productivity and to increase the quality (e.g., reduce the number of bugs, increase the maintainability) of the resulting software.
- The complexity of an IDE can require a considerable initial learning effort. However, increased productivity should result once the “learning curve” has been climbed.

# Automated Version Control Systems (1)

---

- Both software and hardware design activities generate a large number of computer files containing design details.
- In general, humans are terrible at keeping track of a large number of details. Also, misunderstandings among the members of a design team can easily lead to design errors.
- Only certain versions of different design files will correspond to consistent (and potentially working) versions of the entire system that were developed at one particular time.
- A ***version (or revision) control system*** is a computerized file library (or repository) that keeps track of the design files such as documentation, source files for software, source files for hardware, configuration files, test plans and scripts, design bug reports, engineering change orders (ECOs), etc.

# Automated Version Control Systems (2)

---

- There are many available version control systems:
  - RCS, CVS, Perforce, Subversion, Git, etc.
- These systems keep track of the same kinds of information:
  - Name, version number and date for all versions of a file
  - When, why and by whom each file was first created
  - Designer name and reasons for each version of a file
  - The current version of all files
  - Sets of consistent versions of design files
  - File status: *active* and up-to-date, *checked-out* by one (and possibly more) designers; *frozen*; *obsolete*
  - When and by whom a file was “checked out” for change
  - *Etc.*

# Automated Version Control Systems (3)

---

- Typical commands in version control systems:
  - *create* a new file with a given name
  - view the version (or revision) *history* of a file
  - obtain a copy of a *specific version* of a file
  - *check out* (and possibly *lock*) a file for modifications
  - *identify the designers* who have checked out a file
  - *check in* (and possibly *unlock*) one modified file
  - *check in* a consistent related group of modified files
  - *merge* multiple checked-out files into one file (tricky!)
  - *release* a stable and consistent set of design files
  - *roll back* to a previous consistent system design version
  - *Etc.*

# Assembly Language vs. Compiled Code

---

- There was a long tradition of using assembly language programming in embedded systems. This was done in the belief that compilers produced inefficient code, and that an expert programmer could take better advantage of the features of the CPU-specific machine language.
- However, assembly language code is rarely used today.
  - Compiler technology is much better today.
  - Programmer productivity (even for experts) is much too low in assembly language compared to compiled code.
- Assembly language might be seen today in a few places, e.g.
  - Context switch and other interrupt service routines
  - High-performance graphics and signal processing S/W

# Selection of a Programming Language

---

- Many factors influence the choice of programming language:
  - ❖ Language features: enhanced productivity & safety
  - ❖ Higher-level languages usually increase both human productivity and system safety
  - ❖ Compatibility with the existing investment in software
  - ❖ Compiler & software support for the hardware platforms
  - ❖ Availability of kernels, hardware drivers, TCP/IP stacks
  - ❖ The preferences & experience of the S/W designers
- *There is no perfect programming language.* Other factors (e.g., the software development processes and practices in actual use) typically play the most important role in determining the success of a software development project.

# Software Engineering

---

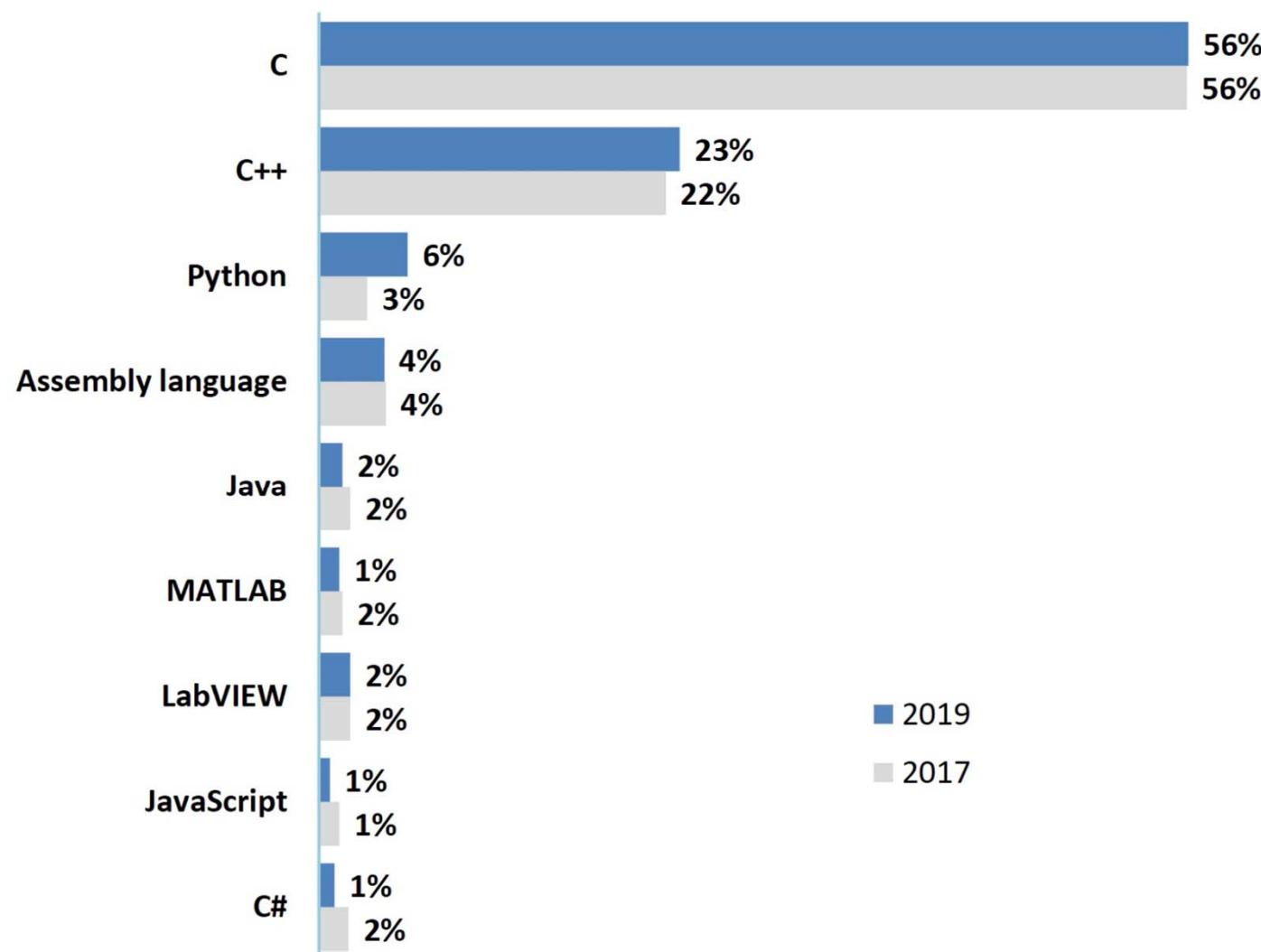
- “*Software engineering is the application of engineering to the design development, implementation, testing and maintenance of software in a systematic method.*” [Wikipedia]
- There is no one best software engineering methodology, but some proven systematic approach must be adopted in the development of software. The risks and costs of problems in undisciplined software development are too great to ignore.
- At a minimum, there should be documentation to cover the software requirements and specifications, coding standards, commenting standards, design reviews for software changes, software defect (bug) reporting and resolution, etc.
- Use the features in the language (e.g., the freedom to name identifiers and to include meaningful comments) to make the code self-documenting or at least easier to understand.

# The C Programming Language

---

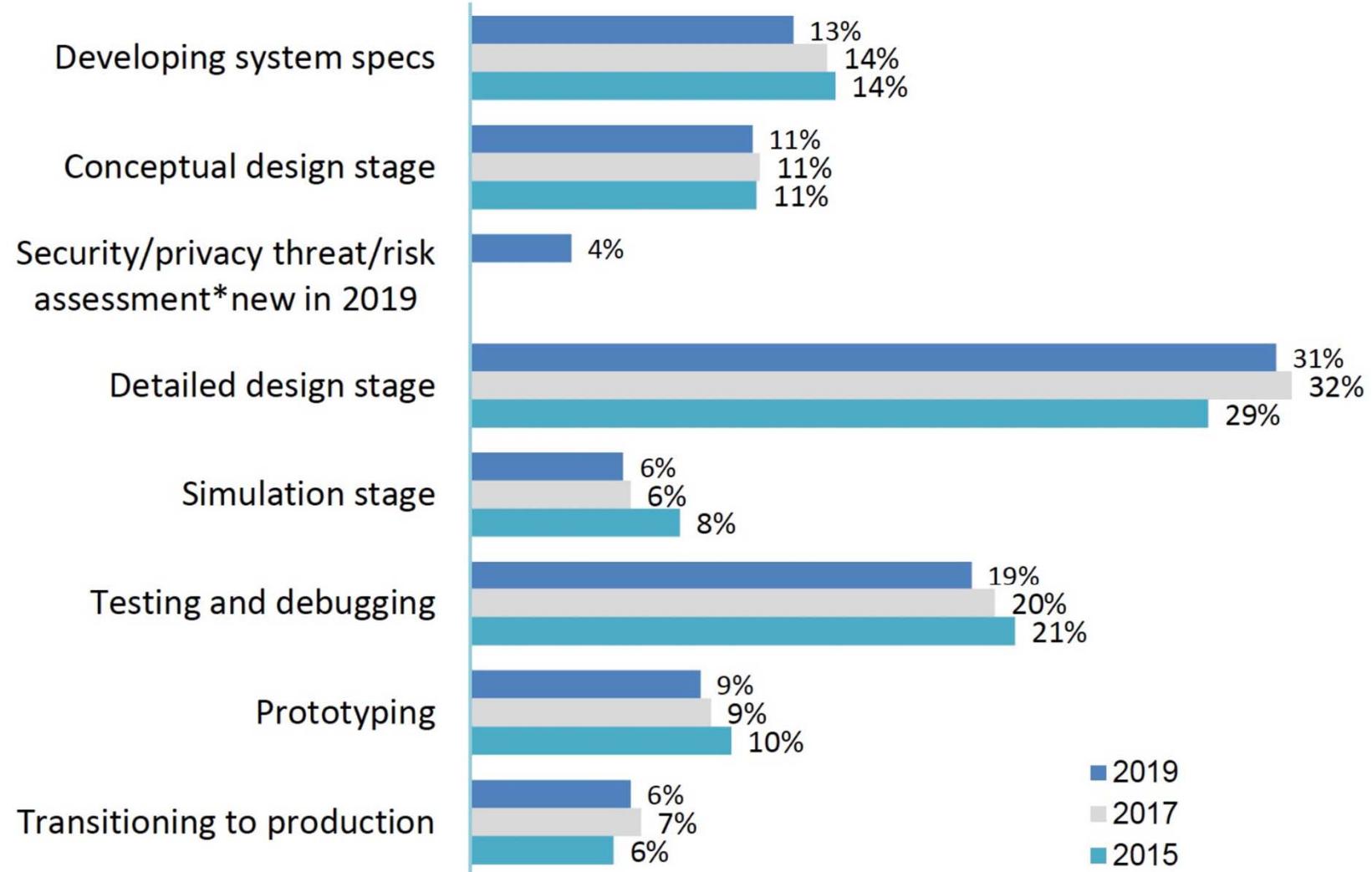
- The C programming language is one of the most widely used languages. Most computer architectures have C compilers.
- Developed by Dennis Ritchie at Bell Laboratories over 1969-1973, C was intended to provide both high-level language constructs (e.g., loops, pointers, I/O) as well as fairly direct access to memory structures, such as words, bytes and bits.
- The flexibility of C is accompanied by many potential hazards.
- Historically, embedded systems have migrated from assembly language to the widespread use of C as the best compiled (but still relatively low-level) language alternative.
- C's dominant position in embedded systems is likely to diminish over time (there are safer languages available now) but the existing C code base will remain very large.

# The Top Languages for Embedded Systems



Courtesy of AspenCore's 2019 Embedded Markets Study

# Time Spent on the Design Stages



Courtesy of AspenCore's 2019 Embedded Markets Study

# Some Pros and Cons of C

---

## ***Pros***

- Widely supported across microcomputer architectures
- Used to implement the popular Unix/Linux operating syst.
- The largest existing code base for embedded systems

## ***Cons***

- C missed out in progress in language design since 1970
- C syntax is compact and “powerful”, but also hazardous
- Poor support for data encapsulation & information hiding
- Haphazard rules for exporting symbol links across files
- Dynamic memory allocation and recycling can be tricky
- Pointers are the source for many serious design errors . . .

# Pointers in C

---

- A **pointer** is a variable that stores the **base address** of data (e.g., variables, arrays, strings, data structures, software “objects”) or a piece of code (e.g., functions) in memory.
- The flexibility of pointers is useful for forming & accessing data structures, and for passing complex parameters. However, *pointer errors can easily lead to serious bugs and crashes*.
- Pointers are declared much like all other variables. Pointers can be typed; however, the type can be effectively overridden.

```
int var1;      /* variable var1 has type int */
int *Ptr1;     /* pointer Ptr1 has type (int *) */
int * Ptr1;    /* same as previous line */
char *s;       /* s has type (char *) */
int * Ptr1, Ptr2; /* Ptr1 is ptr of type (int *) */
                  /* Ptr2 is var of type int */
int *Ptr1, *Ptr2; /* two ptrs of type (int *) */
void *Ptr1, *Ptr2; /* two typeless pointers */
```

# Creating Pointer Values in C

---

- The value of a newly declared variable pointer is not defined. *Many program bugs are caused by attempts to use pointers that have not yet been assigned valid pointer values.*
- The predefined constant **NULL** is a pointer value that points to nothing. In C++, the value 0 is often used instead of **NULL**.
- The **reference operator &** extracts the **base address** of any given software structure (e.g., function, variable, pointer).

```
int var1, *Ptr1, **Ptr2;  
Ptr1 = &var1;      /* Ptr1 points to var1 */  
Ptr2 = &Ptr1;      /* Ptr2 points to Ptr1 */
```

- Pointer values can also be derived from other pointers, or they can be returned as output values from functions.

```
int *Ptr1, *Ptr3;  
Ptr1 = Ptr2+1; /* Ptr2 must hold an (int *) value */  
Ptr3 = functionThatReturnsIntPtr( 1, 2, var1 );
```

# Dereferencing Pointers in C

---

- The **dereferencing operator** \* is used to access the data or software code that is being pointed to by a pointer. Ex:

```
int var1, var2, *Ptr; /* create 2 vars & one ptr */
var1 = 6;             /* initialize var1 */
Ptr = &var2;           /* Ptr points to var2 */
*Ptr = 3;             /* load var2 with value 3 */
var2 = var1 + *Ptr;   /* var2 now holds value 9 */
void (* Fptr)( char, int); /* create function ptr */
Fptr = &NameOfFunction1; /* use the & operator */
*Fptr( 'g', 3 );       /* execute the function */
Fptr = NameOfFunction1; /* this also works in C */
*Fptr( 'h', 6 );       /* execute the function */
Fptr = NULL; /* NULL should never be dereferenced */
*Fptr( 'k', 7 ); /* will crash the task or worse */
```

# Arrays in C

---

- Arrays group variables or data structures of the same type.
- Arrays are accessed (inside compiled code) using pointers.  
*Nothing checks if the pointer goes outside the array limits.*

```
int *Ptr;  
  
int ar[100];           /* create array of 100 ints */  
for (i = 0; i < 100; i++)  
    ar[i] = 2 * i;      /* load next array element */  
Ptr = &ar[0];          /* point to first array element */  
for (i = 0; i < 100; i++) {  
    Ptr = &ar[i]        /* point to next array element */  
    *Ptr = 2 * i;        /* load next array element */  
}  
Ptr = ar;   /* Also legal: point to 1st array elem */  
for (i = 0; i < 100; i++) *(Ptr++) = 2 * i;
```

# Example using Pointers

---

```
#include <stdio.h>

int *functionThatReturnsIntPtr(int *aPtr, int *bPtr) {
    if (*aPtr > *bPtr)
        return aPtr;
    else if (*aPtr < *bPtr)
        return bPtr;
    else
        return NULL;
}

int main() {
    /* Best to declare all variables at the start */
    int var1[4] = {4, 5, 6, 7};
    int *Ptr1, **Ptr2, *Ptr3, *Ptr4;

    Ptr1 = &var1[0];      /* Ptr1 = var1; also works */
    Ptr2 = &Ptr1;
    Ptr3 = Ptr1 + 2;      /* inside var1 array */
    Ptr4 = functionThatReturnsIntPtr(Ptr1, Ptr3);

    return 0;
}
```

# Pointer Arithmetic and Operators in C

---

- A typed pointer can be manipulated arithmetically.
  - can *add* or *subtract integers* to/from a pointer
  - can *subtract* (but not add) two pointers of the same type
  - can *compare* two pointers of the same type for equality (e.g., `Ptr1 == Ptr2` and `Ptr1 != NULL`).
- The addition (+) and subtraction (-) operators are modified for pointers to advance the pointer forward or backward along an array of identically typed variables or data structures. The integer value corresponds to the number of array elements. For example, `* (Ptr1+1) = *(Ptr2-3);`
- Pointers can be incremented *after* they are used (e.g., `Ptr2++`) as well as *before* they are used (e.g., `++Ptr1`). Pointers can also be decremented before or after they are used. For example: `*(--Ptr1) = *(Ptr2--);`

# Character Arrays and Strings in C

---

- We can create an array of characters (`char`'s).
- A *string* is an array of `char`'s that is terminated with the ASCII null character '\0'. The terminating null character is not displayed when a string is sent to an output device. It is used by string processing functions so that they can recognize the end of the string without having to be informed of its length.
- The compiler automatically adds the terminating character to *string constants*. For example,

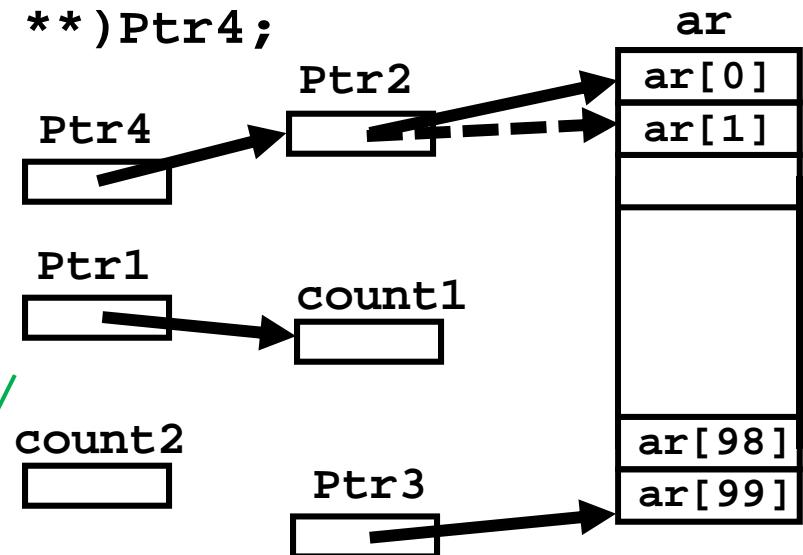
```
printf("hello, " );           /* string has 8 char's */  
printf("world\n");           /* string has 7 char's */
```

- String input functions automatically append the \0 terminator.

```
char studentName [50];    /* create char buffer */  
scanf ("%s", studentName ); /* 49 chars maximum */
```

# Examples that Use Pointers in C

```
int *Ptr1, *Ptr2, *Ptr3;      /* type is (int *) */
void *Ptr4;                  /* same as void * ptr4 */
int count1, count2, ar[100];
Ptr1 = &count1;
Ptr2 = ar;                   /* same as Ptr2 = &ar[0] */
Ptr3 = &ar[99];
Ptr4 = (void *)&Ptr2; /* Ptr4 contains addr of Ptr2 */
count1 = 1024;
*(Ptr2++) = 512;
*Ptr3 = *Ptr1 + ar[0] + **(int **)Ptr4;
/* *Ptr1 contains 1024 */
/* ar[0] contains 512 */
/* (int **)Ptr4 refs Ptr2 */
/* Ptr2 now refs ar[1] */
/* ar[1] has unknown value */
/* **(int **)Ptr4 is unknown */
/* ar[99] contains 1536 */
/* if ar[1] contained 0 */
```



# Casting Pointers in C (1)

---

- Typed variables & pointers, together with run-time type checking, provide a powerful way of avoiding or detecting software design errors.
- However, to obtain greater efficiency, flexibility and portability (with some loss in safety) it is common for embedded system software to override type checking, especially for pointers.
- The C programming language allows the types of variables and pointers to be *cast* into a more general type.
- To avoid type checking, pointer values to typed objects can be assigned to a pointer that is of type (**void \***).
- General functions can be designed to manipulate (**void \***)'s. However, a (**void \***) cannot be dereferenced. We must first cast the (**void \***) to a pointer to a typed object (e.g., an **int \***) and then dereference it to obtain the object (e.g., an **int**).

# Casting Pointers in C (2)

---

```
#include <stdio.h>

void printAnInteger(void *argPtr) {
    printf("Integer = %d", *(int *)argPtr);
}

void printAString(void *argPtr) {
    printf("Character = %s", (char *)argPtr);
}

int main() {
    int var1 = 2017;
    char str2[] = "2017";
    void *argPtr;
    void (* funPtr)(void *)

    funPtr = printAnInteger;
    argPtr = (void *)&var1;
    *funPtr( argPtr ); /* typeless argument is passed */

    funPtr = printAString;
    argPtr = (void *)str2;
    *funPtr( argPtr ); /* typeless argument is passed */

    return 0;
}
```

# Some Common Dangers in C Code

---

- ***Using a pointer*** variable that has ***not yet been initialized*** or has been ***corrupted***. This problem typically leads to bus errors or segmentation faults followed by system crashes.
- ***Buffer overflow***: Adding elements into an buffer (implemented as an array) beyond the buffer's capacity. This can cause system crashes as well as security breaches. This problem can occur using standard functions, like **`strcpy`** and **`gets`**, which rely on the presence of a terminating \0 or EOF. Use **`strncpy`** and **`fgets`** instead.
- ***Type checking can be easily overridden*** in C using casting and **`void`** pointers. For example, a 32-bit **`double`** integer can be stored at one address and then read out and (only the upper byte) used as an 8-bit **`char`**.

# Software Objects in C++ (1)

---

- Programming languages are provided with features that increase programmer productivity and that avoid errors.
- A software **object** is a data structure that gathers together *private* and/or *public state variables* and the functions (also called *member functions* or *methods*) that must be used when operating on private variables.
- Software objects are often a convenient way of modelling the behaviour of physical objects, hardware subsystems, complex data structures, peripheral interfaces, etc.
- Private variables and implementation details can be safely hidden using software objects.
- New object instances are initialized in a consistent way.
- Unneeded objects are recycled in a consistent way.

# Software Objects in C++ (2)

---

- Objects are instantiated from **classes** that define the state variable types and functions.
- Classes can be organized into inheritance hierarchies so that commonalities and differences among the classes can be clearly expressed and possibly exploited.
- Each object instantiation has its own separate set of instances of the state variables.
- An object **constructor function** is called automatically when an object is created and initialized. This guarantees that object initialization is done consistently and safely.
- An object **copy constructor function** is called automatically when an object is passed into a function as a parameter. The C.C.F. ensures that the passed copy has its own dynamically created subobjects & pointers.
- A **destructor function** is called automatically when an object is deleted. This ensures that any resources attached to the object (e.g., dynamically allocated memory) are safely recycled.

# Software Objects in C++ (3)

---

```
class Register8 {
public:
    Register8();           // constructor function
    // Copy constructor and destructor fcns not used here.
    //Register8( const Register8 &obj ); // A copy
    // constructor is required if the class objects contain
    // pointer variables and dynamically allocated objects.
    //~Register8(); // destructor not used in this class
    void setBit( int );
    void clearBit( int );
    void enableOutput();
    void disableOutput();
private:                  // here all vars are private
    bool register[8];
    bool outputEnable;
};

Register8::Register8() { // constructor function
    outputEnable = 0;      // init all private vars
    for (i=0; i<8; i++) register[i] = 0;
}
```

# Software Objects in C++ (4)

---

```
void Register8::setBit( int b ) {
    if ((b > -1) && (b < 8)) register[b] = 1;
}

void Register8::clearBit( int b ) {
    if ((b > -1) && (b < 8)) register[b] = 0;
}

int Register8::getBit( int b ) {
    if ((b > -1) && (b < 8))
        return register[b];
    else
        return -1;           // error return code
}

void Register8::enableOutput() {
    outputEnable = 1;
}

void Register8::disableOutput() {
    outputEnable = 0;
}
```

# Software Objects in C++ (5)

---

```
int main()
{
    Register8 redReg;           // Create 3 Register8
    Register8 yellowReg;        // objects. They will
    Register8 greenReg;         // be initialized.

    redReg.setBit(2);           // Operate on redReg
    redReg.setBit(4);
    redReg.enableOutput();

    yellowReg.setBit(7);         // Operate on yellowReg
    yellowReg.enableOutput();

    if ( redReg.getbit(4) )      // should check for -1
        greenReg.setBit(5);
    else
        greenReg.setBit(7);
    greenReg.enableOutput();

    return 0;
}
```

# Global vs. Local Variables

---

- A ***global variable*** is a variable that is accessible by any function, unless the variable is “masked” by a local variable with the same name.
- A ***local variable*** is a variable that is defined within the scope of the `main()` function or any other function.

```
int sysTime; /* Global variable declaration */
/* sysTime is readable by all functions unless masked */
int main()
{
    int addTime( int x ); /* Local function prototype */
    /* Additional lines of code can appear here */

    int addTime( int x ) /* Local function definition */
    {
        int y;          /* Local variable declaration */
        y = x + sysTime; /* Can access a glob variable */
        return y;
    }                      /* Variable y is now recycled */
    return 0;             /* Return value 0 means no error */
}
```

# Global Variables Can Be Dangerous

---

- All functions can access a global variable, which is both a convenience and a potential hazard.
- The convenience is that global variables can be accessed very fast, with no overhead due to parameter passing.
- *The main hazard of global variables is that they are a path for complicated and potentially unwanted side-effects.* The hazards of such interactions are even harder to predict and understand in multitasking systems (described later).
- Duplicate global variable names can arise when merging software projects, which can easily lead to errors.
- In general, **avoid using global variables** if at all possible.
- The only safe situation for using a global variable is to store a global parameter that is declared in a well-documented header file, and only possibly read by other routines. Note that C does not prevent multiple writers to a global variable.

# Memory Allocation for Objects in C

---

- Variables, data structures and other objects require memory space, which must be allocated and managed during the lifetime of the software system.
- C takes an old-fashioned, low-level approach that gives more direct control (and responsibility) to the programmer.
- Most modern programming languages (e.g., Java) avoid “raw pointers” and explicit dynamic memory allocation.
- ***Static memory allocation:*** Memory space is allocated to the object for the lifetime of the software system.
- ***Automatic memory allocation:*** Memory space is allocated automatically when the object is instantiated and de-allocated when the context of the object is exited.
- ***Dynamic memory allocation:*** Memory space is requested as required from the operating system. Allocated memory must be returned to the O.S. when it is no longer required.

# Static Memory Allocation in C

---

- **Static memory allocation:** Memory is allocated to the object for the lifetime of the software system.
- Static memory allocation is handled by the compiler tool chain. Sufficient memory space is assigned in a suitable region of the memory map (e.g., near the program code).
- Typical examples:
  - **External objects**, which are declared outside of all the functions. These are visible across all source files.
  - An **external object** that is also declared to be **static** is unknown outside of the one source file.

```
static int temperature;
```
  - An **internal object**, which is one that is defined inside a function, that is declared to be **static**. The object's value (or state) is preserved across function calls.

# Register Variables in C

---

- The registers in the central processing unit (CPU) are normally used to hold intermediate values or variables that have very limited scope.
- CPU registers offer the fastest possible implementation for a variable, but the number of registers is quite limited.
- To create fast code, modern compilers attempt to implement as many variables as possible as CPU registers.
- In some applications, a small number of variables might be very heavily used. In such cases, the programmer may wish to force the use of CPU registers for some variables to obtain the greatest possible performance.
- The register keyword can be used in C programs as a *strong hint* to the compiler to use a register for a variable.

**register long totalcount;**

# Automatic Memory Allocation in C

---

- **Automatic memory allocation:** Memory is allocated automatically when a new object is instantiated inside a context, and de-allocated later when the context is exited.
- Implemented using a hardware-supported stack. All modern CPUs provide one or more stacks for this purpose.
- Typical examples:
  - *Function arguments* (passed by value into a local copy)
  - *Local or internal objects* declared inside functions
  - *Local objects* declared inside brace-delineated blocks
- **Stack overflow** is the failure when there is an attempt to store more data in a stack than the stack has room for.
- Common causes: (1) infinite recursion, (2) storing an overly large object or too many objects onto the stack.

# Dynamic Memory Allocation in C

---

- **Dynamic memory allocation:** Memory is requested as required from the operating system. Allocated memory must be returned to the O.S. when it is no longer required.
- The **heap** is a region of memory that is used by the O.S. to provide space to implement all of the dynamic objects.
- `malloc( N )` allocates  $N > 0$  bytes from the heap and returns a pointer to the base addr. of that region in memory.

```
int *array = (int *)malloc( 20 * sizeof(int) );
int *array = malloc( 20 * sizeof(int) ); // also
// Note! malloc returns a (void *) if the
// bytes are allocated; otherwise, NULL
```

- Use `free( ptr )` to recycle a dynamic object back to the heap

```
free( array );
free( &array ); // same effect
free( &array[0] ); // same effect
```

# Potential Hazards of Dynamic Memory (1)

---

- Dynamic memory allocation in C assumes that the programmer will avoid errors during both the allocation and de-allocation steps. Such errors can have serious effects.
- ***Using an uninitialized pointer variable:*** A pointer variable must be properly initialized to point to an object with allocated memory; otherwise, serious system failures will likely occur as a result of the subsequent bus error or segmentation fault (or the overwriting of data or code).
- ***Failure to check for allocation failure:*** When `malloc()` fails to be allocated all of the requested bytes, a NULL pointer is returned. This possibility must be checked for and dealt with safely or else the program will go on to use a NULL pointer value and likely cause a bus error or segmentation fault, very likely leading to a system crash.

# Potential Hazards of Dynamic Memory (2)

---

- **Failure to call free() to recycle dynamic memory:** If a dynamic object is not freed after the object is no longer required, the memory space allocated to that object is effectively removed from the heap. This situation is called a **memory leak**. Over time memory leaks will degrade system performance and eventually cause failure.
- **Calling free() more than once for the same pointer value:** This could cause a new object to be corrupted.
- **Using a pointer to a dynamic object that has already been freed:** This is called a *dangling pointer* bug. Be very careful when a pointer to a dynamic object is assigned to other pointer variables! Freeing the object using one of the pointers will invalidate the pointer value in all of the other pointers that still point to the (now recycled) dynamic object.

# Dynamic Memory Allocation in C++

---

- C++ introduces new functions that offer safer and more convenient dynamic memory allocation than those in C.
- The `new` function returns a pointer to memory from the heap for a new instance of the object class and also automatically calls the object's constructor function (if one exists).

```
TypeName *typeNamePtr;  
typeNamePtr = new TypeName;
```

- The `delete` function calls the object's destructor function (if one exists) and then deallocates the memory to the heap.

```
delete typeNamePtr;
```

- The `new []` and `delete []` functions must be used to allocate and deallocate an array of objects.

```
int *intArrayPtr;  
intArrayPtr = new int [100];  
delete [] intArrayPtr;
```

# Garbage Collection

---

- **Garbage collection** is a mechanism that is used in many software environments (e.g., Java, C#, Matlab, Python, but not C or C++) to automatically reclaim *garbage memory*, that is, memory space that is used to implement objects that are no longer in use and no longer required.
- Garbage collection is intended to eliminate potentially serious problems like memory leaks, dangling pointer bugs, double free bugs, etc. It can also be integrated along with algorithms that reduce memory fragmentation.
- However, garbage collection is challenging to implement in a way that does not impact system performance. This is the reason why garbage collectors are uncommon in embedded systems, especially real-time embedded systems.
- Garbage collection can co-exist in a system that also provides manual dynamic memory allocation.

# Smart Pointers in C++

---

- C++ does not have built-in garbage collection.
- Also, the `new` and `delete` functions still leave open the possibility of serious pointer-related problems
- A *smart pointer* is an abstract data type, introduced in C++98, that provides pointer functionality with enhancements that aim to avoid serious errors like initialization errors, dangling pointers, attempting to move a pointer to beyond its valid address range, etc.
- A smart counter can use a *reference counter* to determine the number of users of a dynamically allocated region of memory. Only after the reference count goes to zero will the memory be released back to the heap using a destructor function.
- Different smart pointers were defined in C++98, 11 & 17.

# Microcomputer Concepts and the MCF54415 32-bit Microcontroller Unit

## *References:*

- Freescale Semiconductor, Inc., “ColdFire Family Programmer’s Reference Manual”, Doc. No. CFRM, Rev. 3, July 2005.
- Freescale Semiconductor, Inc., “MCF5441x Reference Manual”, Doc. No. MCF54418RM, Rev. 4, Jan. 2012.

Figures and tables from the above documents have been included in these course notes with the permission of Freescale Semiconductor for educational purposes in ECE 315 only. The original Freescale Semiconductor documentation should be consulted to ensure accuracy.

Freescale™ and ColdFire® are registered trademarks of Freescale Semiconductor, Inc.

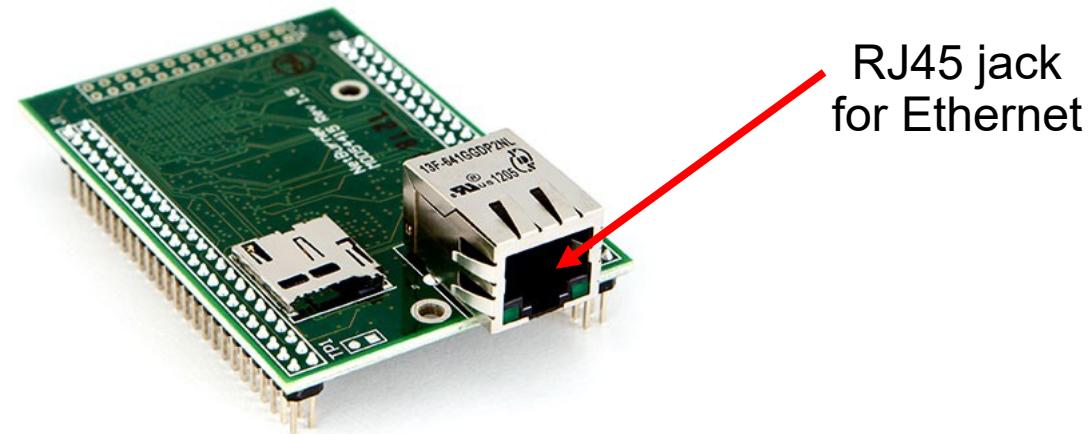
In December 2015, NXP Semiconductors N.V. completed its acquisition of Freescale Semiconductor.

# Microprocessor-related Terminology

---

- A **microprocessor** ( $\mu$ P) is a central processing unit (CPU) on one chip. The CPU is responsible for ***fetching***, ***decoding*** and ***executing*** binary machine language instructions that are stored in a memory.
- A **microcontroller unit** (MCU) is an *integrated circuit* (IC) that contains a microprocessor as well as other useful support circuits, such as timers, memory, input/output interfaces, direct memory access controllers, etc. The ECE 315 lab microcomputer is built using the Freescale MCF54415 MCU, which contains one Version 4 ColdFire 32-bit microprocessor.
- A **microcomputer** ( $\mu$ C) is a computer system that has been built around either a microprocessor or microcontroller unit chip. The ECE 315 lab microcomputer is the NetBurner MOD54415-100X.
- A **digital signal processor** (DSP) is a specialized microprocessor that has features (e.g., *fast multiply-accumulate instructions*, *parallel data signal paths*, etc.) that make it especially efficient at performing the kinds of numerically-intensive calculations that are required in digital signal processing (e.g., in digital filters and image processing).

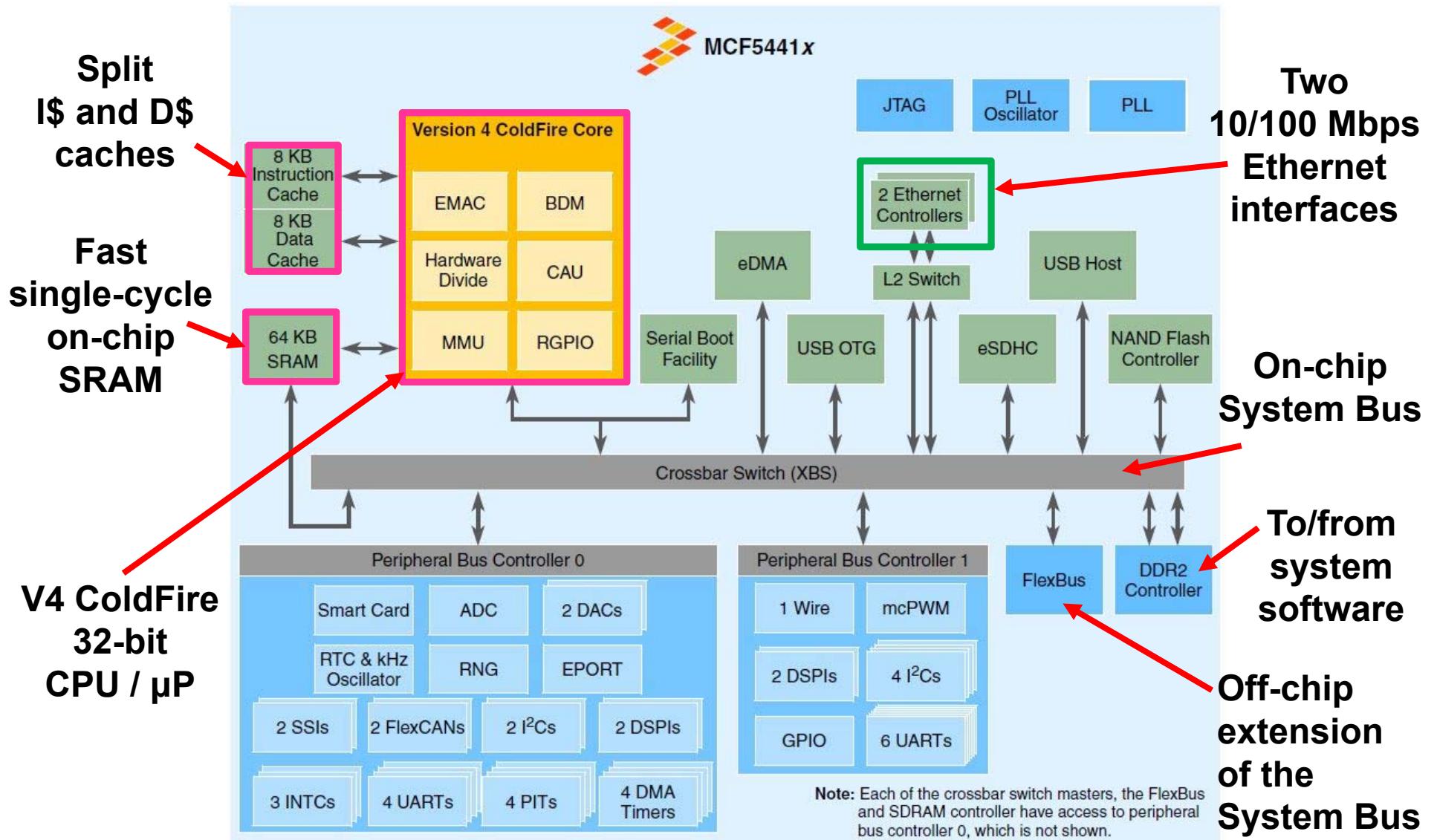
# The ECE 315 Microcomputer: NetBurner MOD5441X



NetBurner MOD54415-100IR  $\mu$ C  
“Ethernet Core Module”

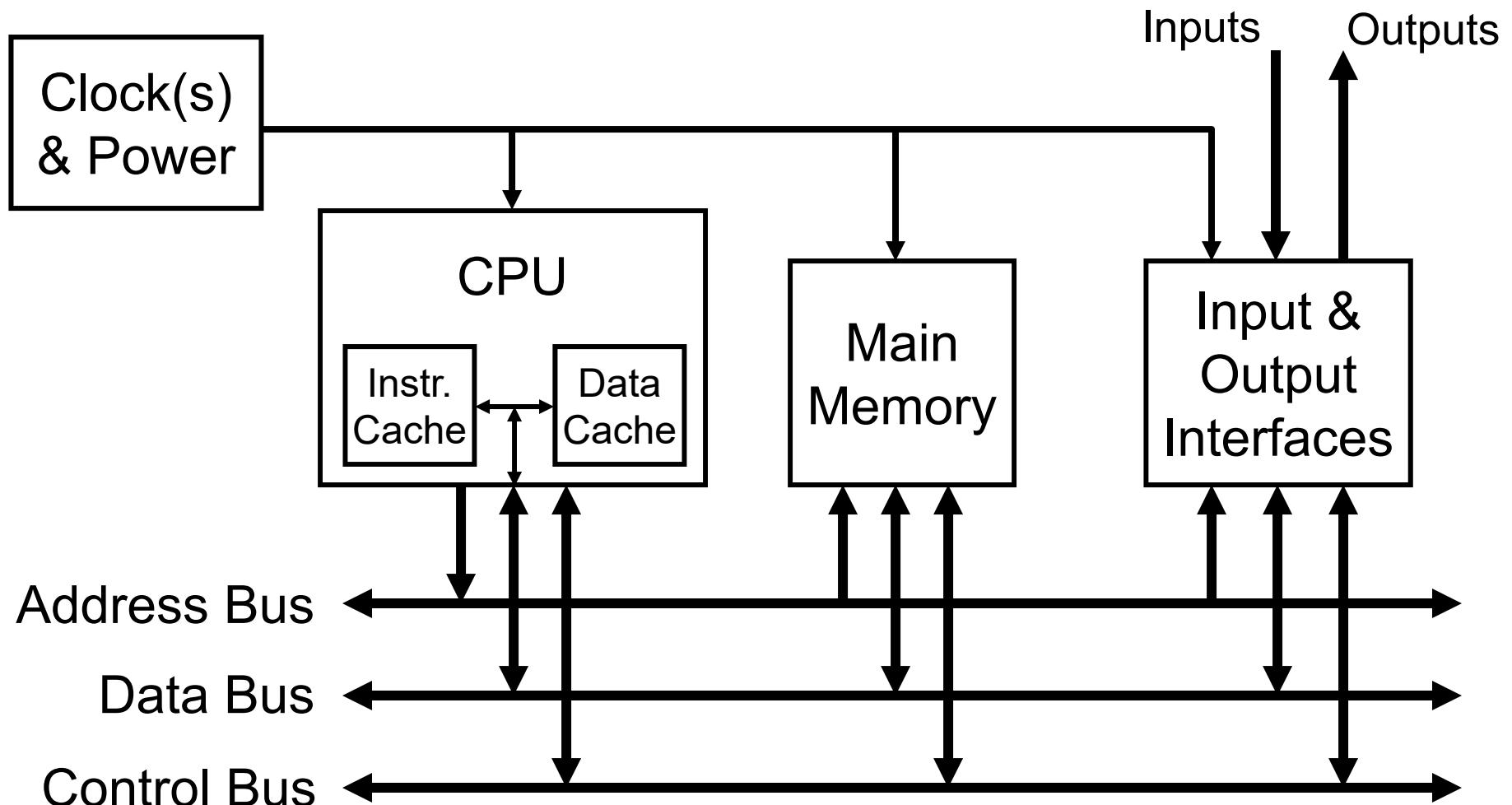
Freescale MCF54415 MCU  
< 1100 mW power at 250 MHz

# Architecture of the MCF5441x MCUs, x = 0,5,6,7,8



# Microcomputer Architecture Using a Microprocessor with Separate Internal Instruction and Data Caches

---



# The Instruction and Data Caches

---

- Cache memory is used in computers as a cost-effective way of *boosting performance* by speeding up memory accesses.
- Cache memory is a relatively *fast memory*, of relatively small capacity, that is *located physically close to the CPU*.
- The cache memory is used to store “local” copies of data and instructions that were retrieved previously from main memory as the software was executing on the CPU.
- Whenever the CPU fetches data or instructions, the cache is accessed to see if it contains a local, fast-access copy.
- The “*replacement rule*”, which is used to determine which data and instructions are evicted from (or retained in) the limited capacity cache, is tuned so that frequently used data and instructions are likely to be kept longer in the cache.

# The System Bus

---

The system bus is usually composed of three sub-busses:

- The “*address bus*” is used by the CPU to tell the rest of the microcomputer system which address it is using for the present *read*, *write*, or *read-modify-write* bus operation.
- The “*data bus*” is used to communicate information between the different parts of the microcomputer.
  - During *reads*, data travels from the addressed location (in either the memory or the input/output devices) to the CPU.
  - During *writes*, data travels from the CPU to the addressed location (in either the memory or the input/output devices).
  - The data bus “width” (in bits) determines the data size of the microcomputer system.
- The “*control bus*” contains various signals used to control and synchronize events in the microcomputer.

# Microcomputer Memory

---

- A memory holds binary information, which consists of both *program instructions* and *data*. (In memory, all information is stored as 0's and/or 1's. Hexadecimal notation is often used to make binary values easier for humans to read and write.)
- Information “words” are identified in a memory using an address, which is just a binary (often written in hex) number. The “word size” of a computer is usually determined by the number of wires in the data bus (i.e., the data bus size).
- A *random-access memory* (RAM) allows the words stored in the RAM to be both *read* and *written*.
- A *read-only memory* (ROM) allows the words to be read, but not changed. The contents of a ROM are *fixed*. ROM is usually implemented today using flash memory.

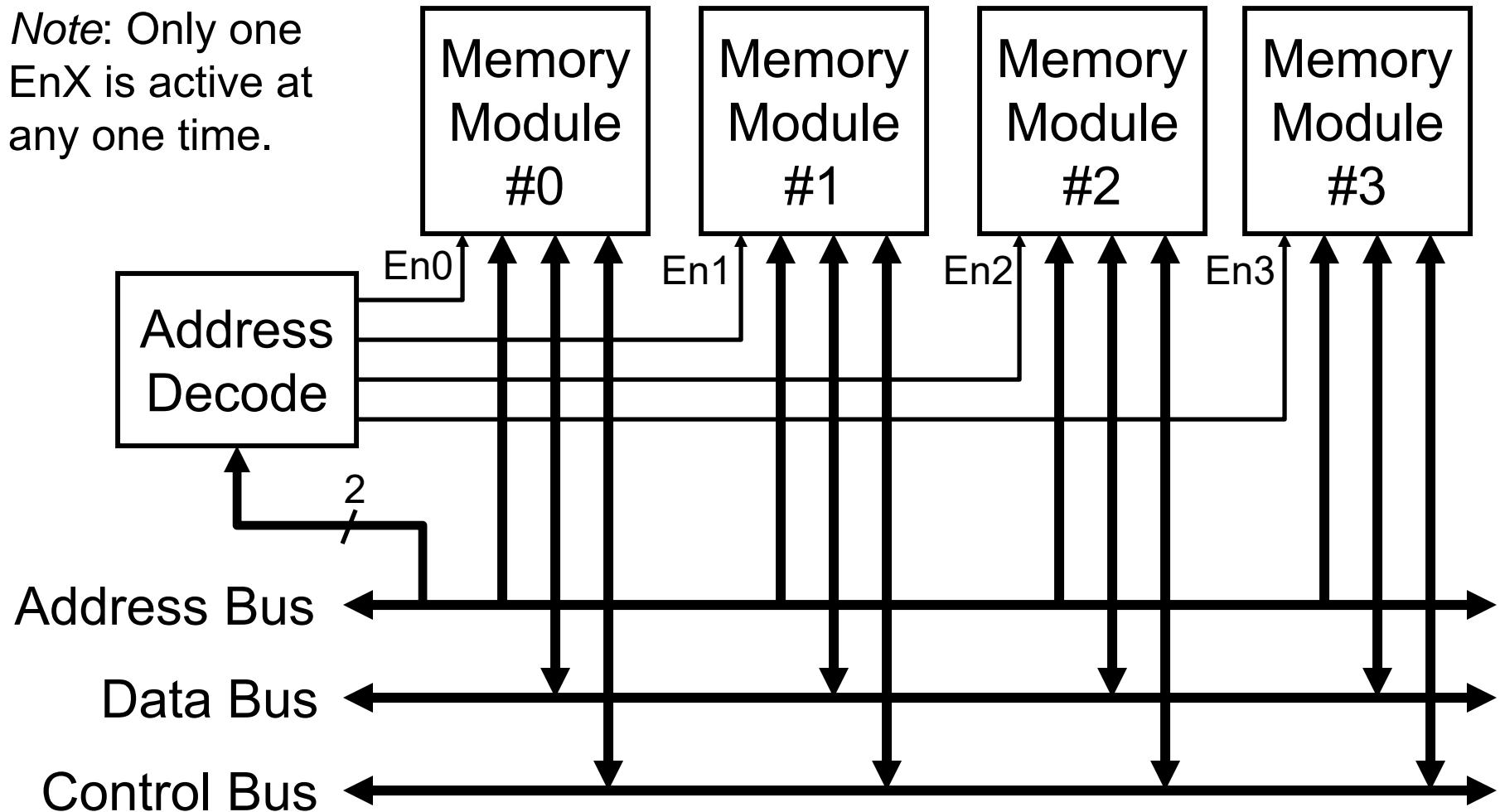
# Volatile vs. Nonvolatile Memory

---

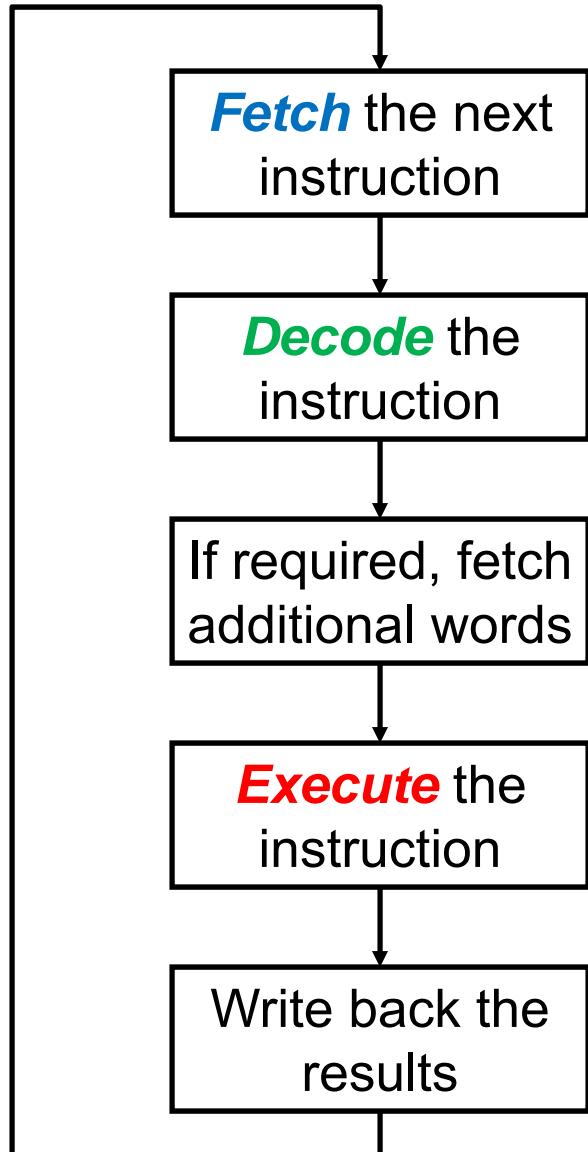
- Semiconductor memory is either *volatile* or *nonvolatile*.
- Volatile memory can retain stored data only as long as power is supplied.
  - *Static RAM* (SRAM) is fast but it is relatively power-hungry and expensive per bit. Smaller capacity than DRAM.
  - *Dynamic RAM* (DRAM) is cheap but relatively slow, and it requires refreshing to avoid losing stored data.
  - *Synchronous DRAM* (SDRAM) is a fast form of DRAM that uses a high-speed pipelined synchronous interface.
  - *Double data rate DRAM* (DDR<sub>x</sub>) is a fast form of SDRAM.
- *Nonvolatile memory* (NVM) can retain data even after power is lost. NVM is used to implement read-only memory (ROM).
  - *Flash memory* is currently the dominant NVM type.

# A Simple Memory Subsystem

Note: Only one EnX is active at any one time.

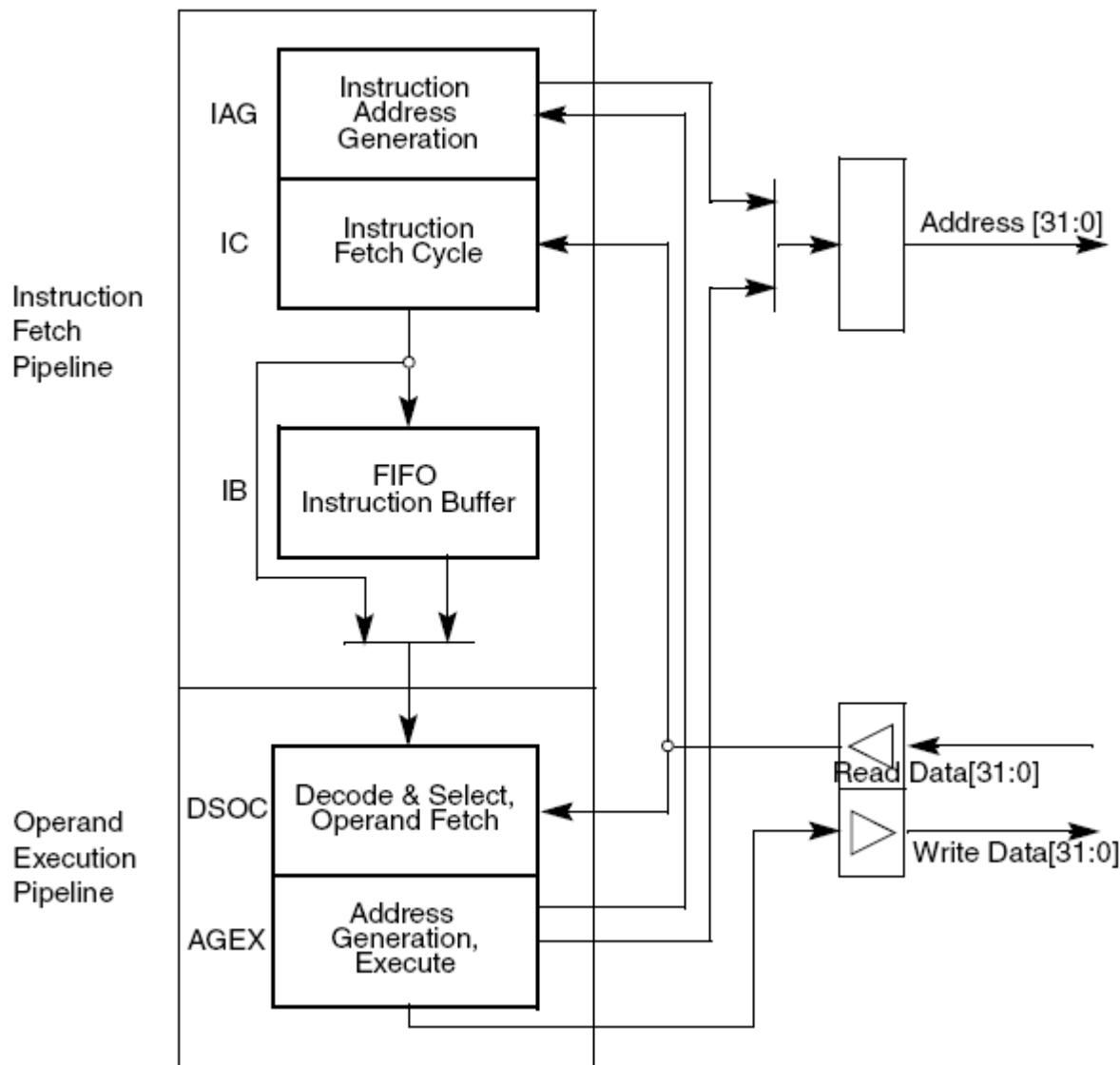


# The Instruction Fetch-Decode-Execute Cycle



- Place contents of PC on the address bus.
- Read first word of the present instruction off of the data bus (using a read bus cycle).
- Inspect the “operation code” in the first word of the instruction. Decide which kind of instruction is being performed.
- Update the PC, place PC contents on the address bus, and read additional data words (if necessary) using read bus cycles.
- Update all CPU registers (including the PC) according to the present register contents and the present instruction type.
- If necessary, update words in memory and I/O devices using write bus cycles.

# Instruction Prefetching



Many modern CPUs, such as the Freescale MCF54415, have the ability to “prefetch” instructions from slow DRAM and to store them in a fast *first-in first-out* (FIFO) instruction buffer.

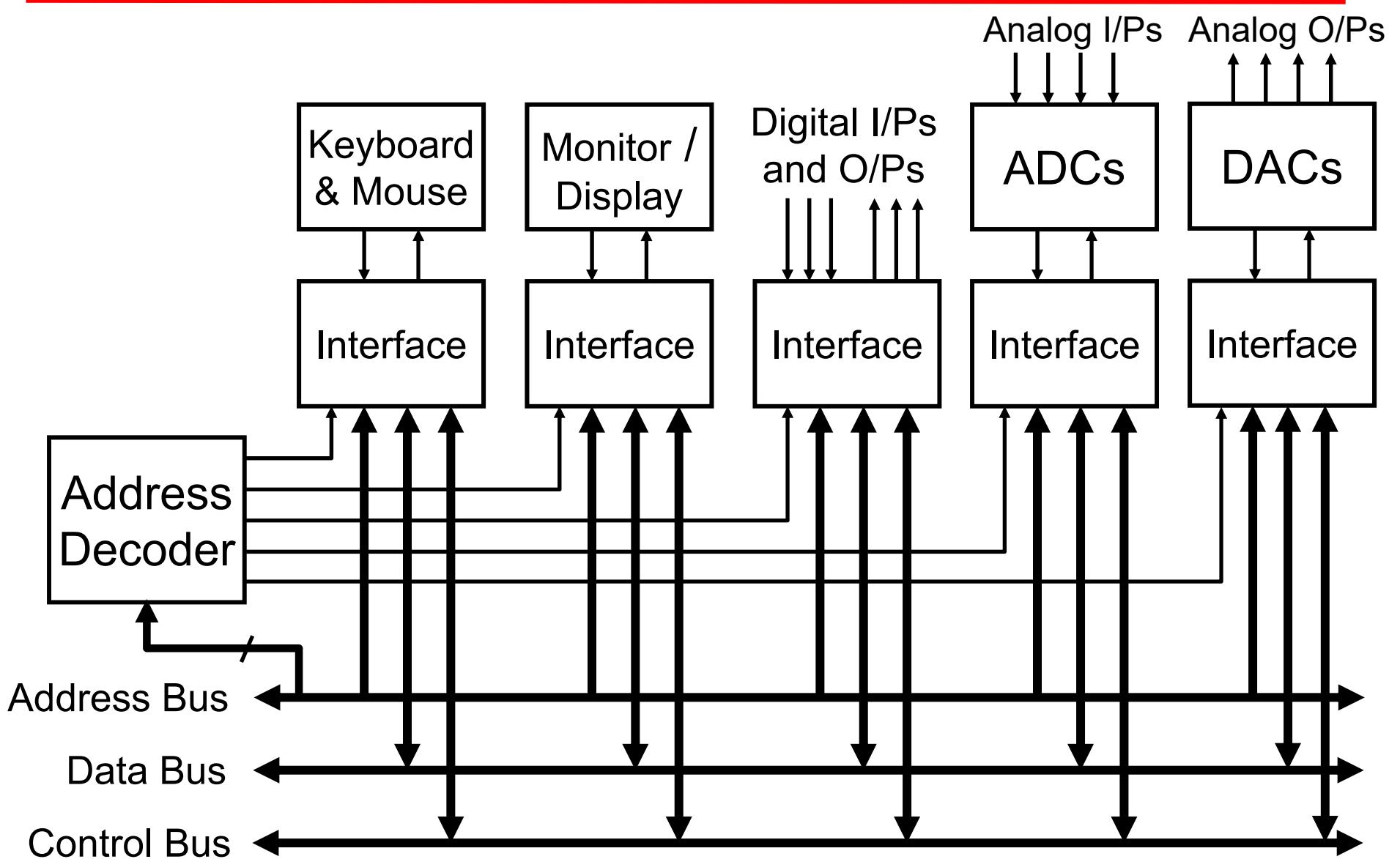
This is done to speed up the effective rate of instruction execution by reducing the average fetch time.

# Input and Output Devices

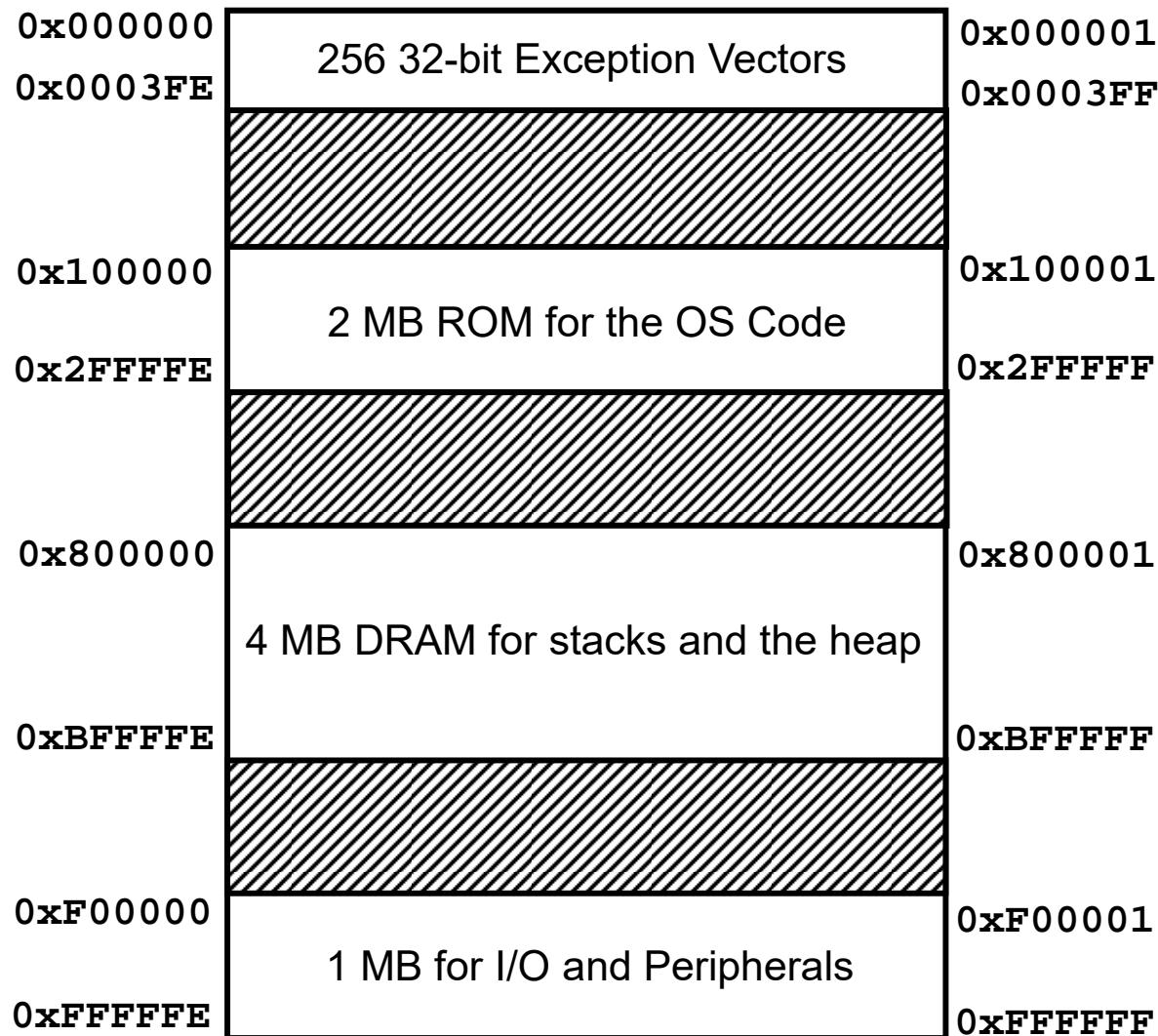
---

- To be able to do useful work, a microcomputer needs to exchange information with its environment.
- ***Input devices*** allow information to be transferred from the environment into the microcomputer.
  - *Keyboard*, for inputting ASCII-encoded symbols.
  - *Mouse*, for inputting positions and command selections.
  - *Analog-to-digital converters* (ADCs), for converting measurements of analog signals into digital quantities.
- ***Output devices*** allow information to be transferred from the microcomputer out to the environment.
  - *Terminal Monitor or Screen*, for displaying graphical data and ASCII (or EBCDIC or Unicode) encoded symbols.
  - *Printers*, for producing hardcopy text and graphical output.
  - *Digital-to-analog converters* (DACs), for producing software-programmable analog signals.

# Memory-Mapped I/O Interfaces



# Ex: A Simple 68000-style 16-MB Memory Map



# A Selection of (now obsolete) Motorola Interface Chips

---

MC6800 Family (8-bit data, *synchronous* system bus):

- MC6821 Peripheral Interface Adaptor (PIA)
- MC6840 Programmable Timer Module (PTM)
- MC6843 Floppy Disk Controller (FDC)
- MC6845 Cathode Ray Tube Controller (CRTC)
- MC6850 Asynchronous Communications Interface Controller (ACIA)

M68000 Family (8/16/32-bit data, *semisynchronous* system bus):

- MC68120 Intelligent Peripheral Controller
- MC68175 VME Bus Controller
- MC68454 Intelligent Multiple Bus Controller
- MC68488 General Purpose Interface Adaptor
- MC68681 Dual Asynchronous Receiver/Transmitter (DUART)
- MC68590/802 Ethernet Controller chip set

# Internal and External Interfaces

---

- When transistor budgets per IC were much smaller, it was necessary to implement most of the interface subsystems on additional ICs outside of the microprocessor IC.
- In fact, up until 1989/90, it was common to implement the floating-point instructions on a “*co-processor*” IC that was closely coupled with a separate microprocessor IC.
- As transistor budgets per economically manufacturable IC grew, more and more interface functions were brought onto the microprocessor IC, creating the microcontroller unit.
- Modern microcontroller units, such as the MCF54415, contain a large number of flexible *internal* (on-chip) interfaces. *External* (off-chip) interfaces can be accessed, if necessary, over an off-chip extension of the system bus.

# CISC versus RISC CPUs

---

- The first 4-, 8- & 16-bit µPs were designed as *Complex Instruction Set Computers* (CISCs), with a relatively large number of instruction types and addressing modes to allow expert programmers to obtain compact assembly language code and the fastest possible execution.
- A more modern design philosophy is the *Reduced Instruction Set Computer* (RISC). Compared to a CISC, a RISC has a smaller number of instruction types and addressing modes. RISC designs thus require simpler hardware that can be more easily pipelined for higher clock rates than CISC designs.
- In practice, modern CPUs claim to be RISCs, but CISC features are commonly present in most modern CPUs to preserve compatibility with earlier CISC products.

# The ColdFire Family of 32-bit CPUs

---

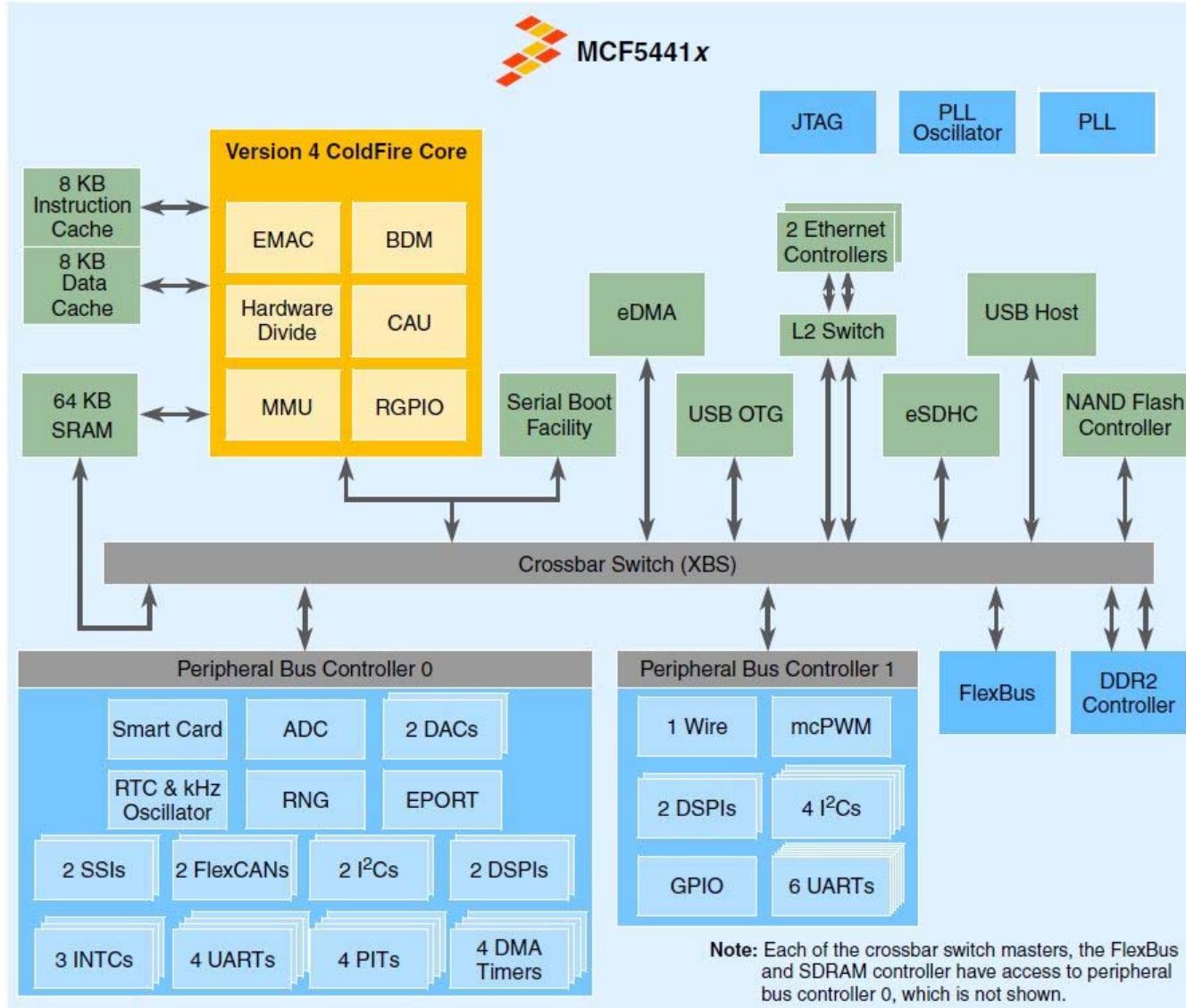
- In 1994 Motorola introduced the Version 2 ColdFire µP core to update its 68xxx and CPU32 -class 16/32-bit µPs.
- The ColdFire is not object code compatible with 68xxx and CPU32 processors. However, object code from the older processors can be translated automatically into code that can be executed directly on ColdFire µPs.
- The ColdFire has fewer instructions (e.g., no binary-coded decimal instructions) and fewer addressing modes available for many instructions compared to the earlier 68xxx µPs.
- In 2004 the Semiconductor Products Sector of Motorola was spun off as Freescale Semiconductor, Inc. The ColdFire family continues to be developed by Freescale (now a part of NXP Semiconductor).

# The ECE 315 Microcontroller Unit (MCU)

---

MCF54415 = 250-MHz V4 ColdFire µP + 16-Kbyte I/D cache +  
64-Kbyte fast on-chip dual-ported SRAM +  
64-channel DMA controller +  
Synchronous DDR2 DRAM controller +  
dual 10/100-Mbps Fast Ethernet Controllers (FEC) +  
up to ten serial UARTs + USB 2.0 host interface +  
CAN interfaces + I<sup>2</sup>C interfaces + DSPI interfaces +  
up to 87 general-purpose input/output pins +  
watchdog timer + four 32-bit DMA timers +  
four programmable periodic interrupt timers +  
dual 4:1 muxed 12-bit ADCs and dual 12-bit DACs +  
6 programmable chip selects + 5 IRQ inputs +  
much more (see the MCF5441X Reference Manual)

# Architecture of the MCF5441x MCUs



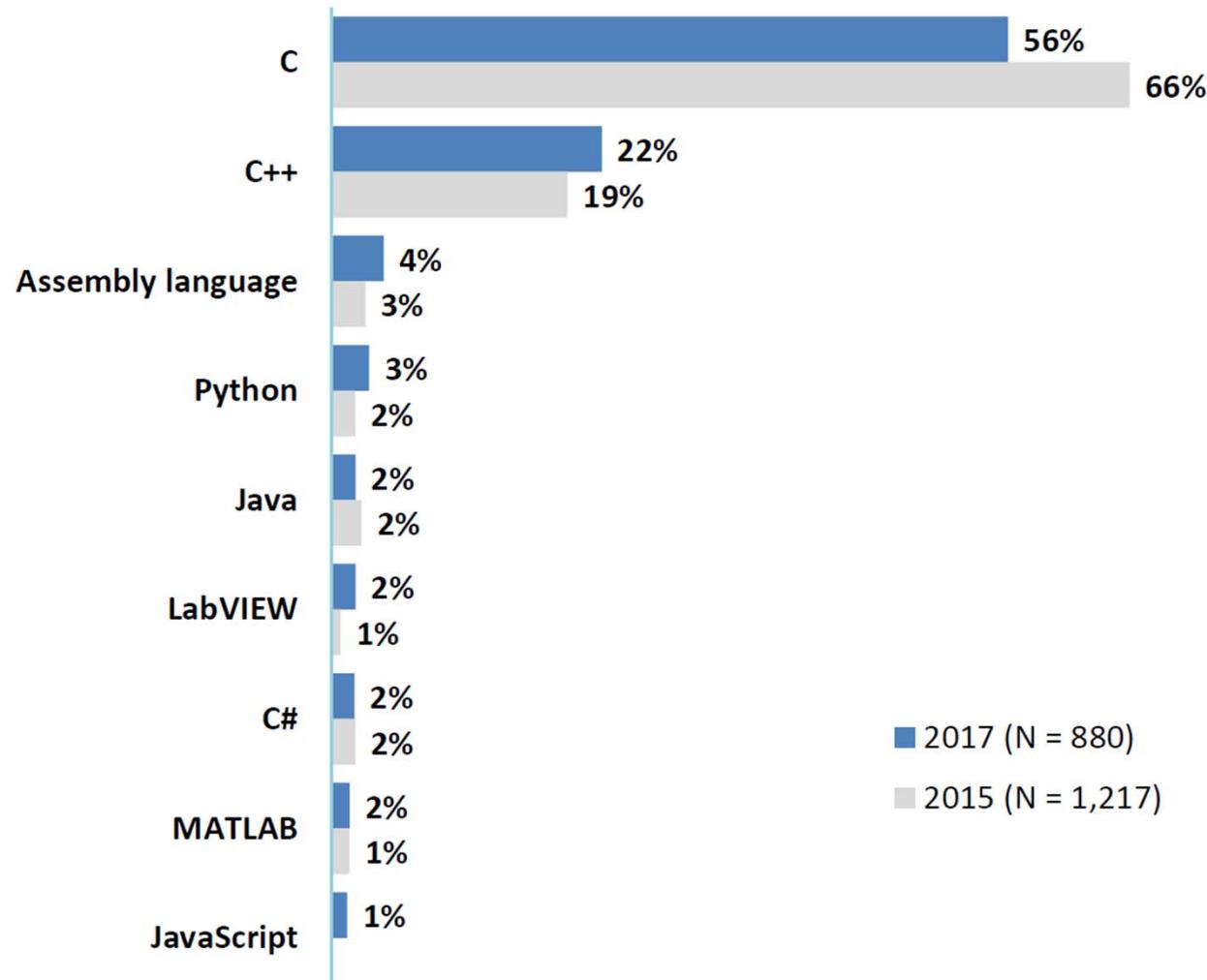
Copyright © 2012 by Freescale Semiconductor, Inc.

# Assembly Language vs. High-Level Language

---

- To maximize the effectiveness of the earliest generations of microprocessors, all programming was done in assembly language.
- As microprocessors became faster and software systems became larger, software was increasingly developed in high-level programming languages like C and C++.
- *Software designers are far more productive if they can use compiled high-level programming languages.*
- Software designed for the MCF5441x 32-bit microcontroller is likely to be entirely programmed in a high-level language like C. *Assembly language will only be used in the most time-critical portions of the operating system, and perhaps in some interrupt service routines.*

# Languages Used to Implement Embedded Systems



Courtesy of AspenCore's 2017 Embedded Markets Study

# User and Supervisor Modes

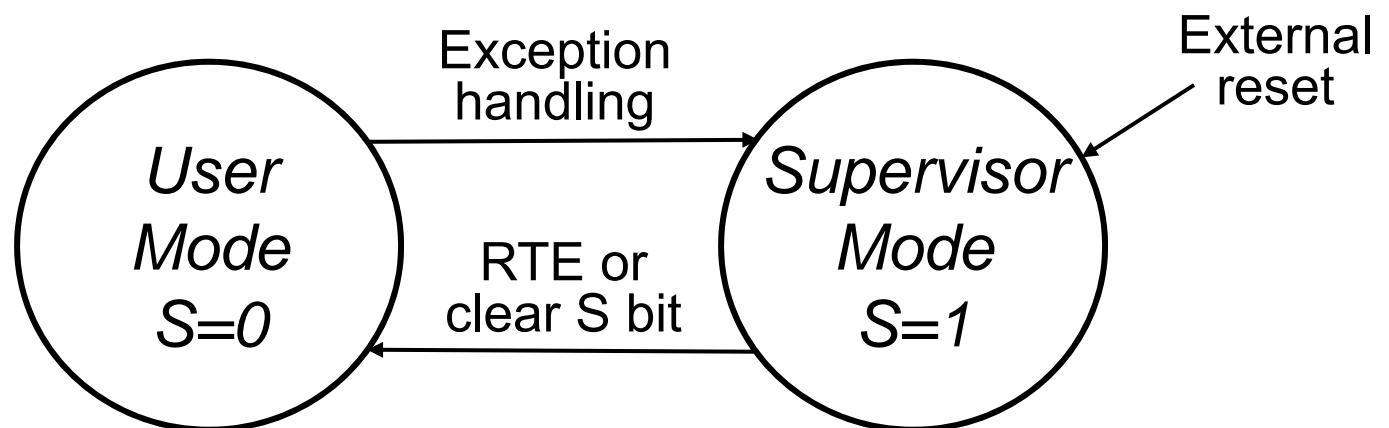
---

- The 68xxx, CPU32 and ColdFire microprocessor families provide two different execution modes, called “User Mode” (S bit in the CPU Status Register is 0) and “Supervisor Mode” (S bit is 1).
- ***Supervisor Mode*** is intended for trusted operating system code and exception handling routines, while ***User Mode*** code is intended for ordinary software.
- Supervisor Mode uses a different stack pointer, can access a greater number of CPU registers, and can use some additional “privileged” instruction types.
- User Mode code requires little or no modification when moving to different processors within the family, but Supervisor Mode code typically needs some changes.

# Transitions Between the User and Supervisor States

---

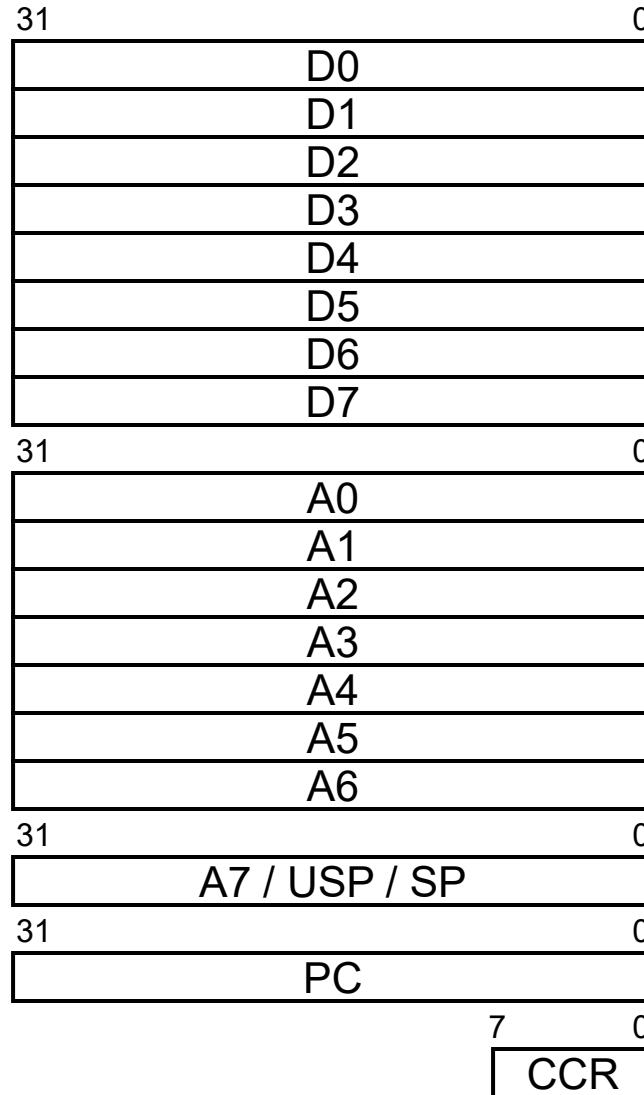
- When a CPU is reset by asserting the external hardware reset, it will start executing instructions in supervisor mode, with the S bit in the status register (SR) set to 1.
- All of the software could be executed in supervisor mode, but it is often desirable to restrict the privileges of at least some processes/tasks to user mode.



# User Mode Programming Model (68xxx & ColdFire)

Can program the CPU as if it were a true 32-bit micro.

Interface logic handles transfers (1, 2 or 4 bytes) over the external 16-bit data bus.



Eight 32-bit Data Registers

Dn.L => all 32 bits

Dn.W => lower 16 bits

Dn.B => lower 8 bits

Seven 32-bit Address Registers

An.L => all 32 bits

An.W => sign-extend the lower 16 bits

User Stack Pointer

Program Counter

Condition Code Register

# Supervisor Mode Programming Model

---

- In supervisor mode, most of the registers are the same registers as in user mode: D0-D7, A0-A6, PC, CCR
- The stack pointer (A7 or SP) is a different register. The new register is the ***Supervisor Stack Pointer***, which can be identified using either A7, SP or SSP. The ***User Stack Pointer*** can still be accessed using USP.
- An extension of the CCR, the 16-bit ***Status Register*** (SR), can be accessed by supervisor mode programs.
- A 32-bit ***Vector Base Register*** (VBR) can be used to define a new base address for the Exception Vector Table.
- Eleven more 32-bit registers (CACR, ACR0-7, RAMBAR, MMUBAR) and 8-bit ASID control the CPU configuration.

# CPU Registers for Supervisor Mode Only

---

31:24	23:16	15:8	7:0	Mnemonic
—		Status Register		SR
		Supervisor/User A7 Stack Pointer		A7
		User/Supervisor A7 Stack Pointer		OTHER_A7
		Vector Base Register		VBR
		Cache Control Register		CACR
		Access Control Register 0		ACR0
		Access Control Register 1		ACR1
		RAM Base Address Register		RAMBAR1

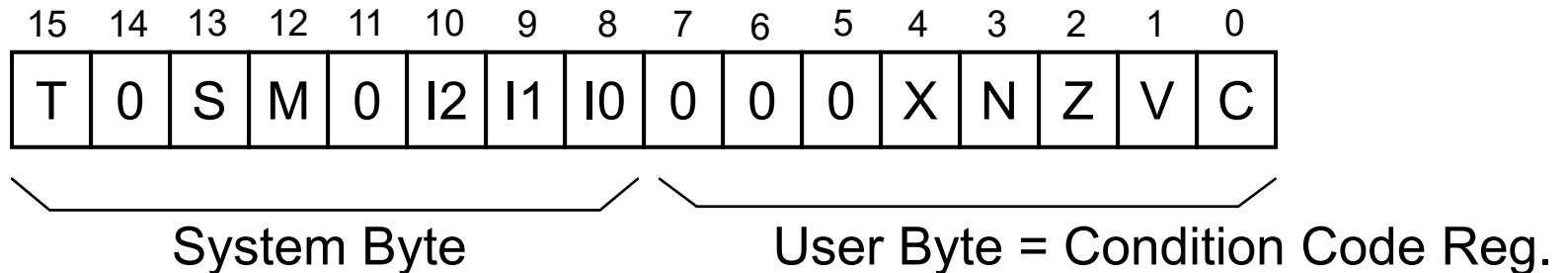
Copyright © 2008 by Freescale Semiconductor, Inc.

If S bit == 1, then A7 == SSP and OTHER\_A7 == USP

If S bit == 0, then A7 == USP and OTHER\_A7 == SSP

Note: OTHER\_A7 is not visible to the software.

# ColdFire Status Register



- Bit 15: T => Trace mode enable/disable on every single instruction
- Bit 13: S => Supervisor mode enable/disable
- Bit 12: M => Master/interrupt state: cleared by interrupt exception, and can be set by the RTE or Move to SR instructions
- Bits 10,9,8: I2,I1,I0 => Interrupt Priority Mask
- Bit 4: X => Carry out bit for extended precision arithmetic
- Bit 3: N => Negative result obtained
- Bit 2: Z => Zero result obtained
- Bit 1: V => Arithmetic overflow detected
- Bit 0: C => Carry out bit (borrow out for subtraction)

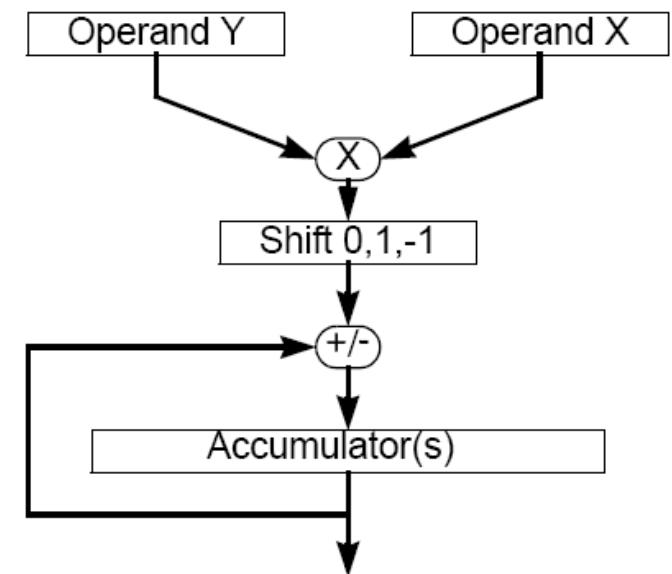
# Common Multiply-Accumulate (MAC) Calculations

Four-tap Finite Impulse Response (FIR) Filter:

$$y(i) = \sum_{k=0}^3 b(k)x(i-k) = b(0)x(i) + b(1)x(i-1) + b(2)x(i-2) + b(3)x(i-3)$$

N-tap Infinite Impulse Response (IIR) Filter:

$$y(i) = \sum_{k=1}^{N-1} a(k)y(i-k) + \sum_{k=0}^{N-1} b(k)x(i-k)$$



# Enhanced Multiplier-Accumulate (EMAC) Unit

---

- To support customer requirements for high-speed MAC, some of the ColdFire processors have been provided with high-speed, pipelined MAC hardware that is used by the multiple instructions and new instructions that move data to and from the new MAC registers.
- The MCF54415 has an Enhanced Multiply-Accumulate (EMAC) unit that supports high-speed MAC operations optimized for 32-bit operands.
- The EMAC has a four-stage MAC pipeline, and provides four 48-bit accumulators (wider to support summations).
- New EMAC instructions are provided (see the Reference Manual for more details).

# EMAC Registers

---

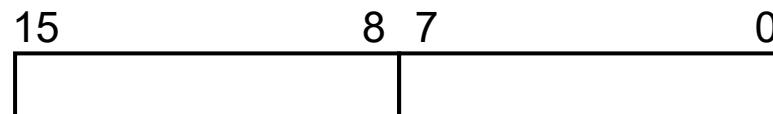
31:24	23:16	15:8	7:0	Mnemonic
MAC Status Register				MACSR
MAC Accumulator 0				ACC0
MAC Accumulator 1				ACC1
MAC Accumulator 2				ACC2
MAC Accumulator 3				ACC3
Extensions for ACC0 and ACC1				ACCext01
Extensions for ACC2 and ACC3				ACCext23
MAC Mask Register				MASK

Copyright © 2008 by Freescale Semiconductor, Inc.

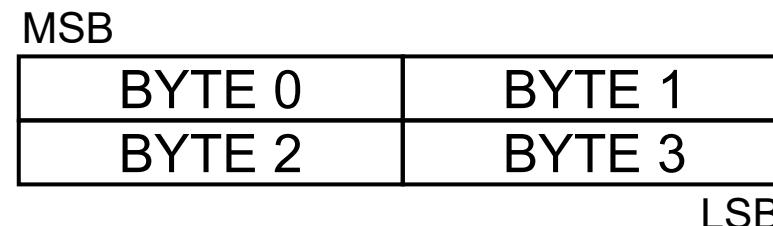
The MASK register is used to optionally constrain the address in MAC instructions to simplify accesses to data stored in circular queues (discussed later in the course).

# Formats of the Native Data Types in Memory

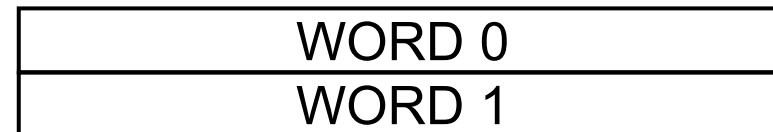
1) Bits



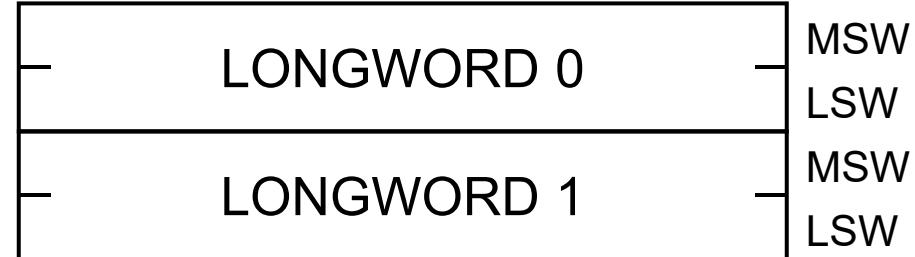
2) Bytes (8 bits)



3) Words (16 bits)



4) Longwords (32 bits)



*Note:* The Binary-Coded Decimal (BCD) native data type that was supported in the M68000 and CPU32 is not present in the ColdFire.

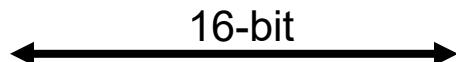
# Aside: Byte Address Order Conventions

---

## Big Endian

(IBM, Motorola/Freescale, HP, Sun)

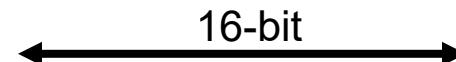
MSB	LSB
BYTE 0	BYTE 1
BYTE 2	BYTE 3
BYTE 4	BYTE 5
BYTE 6	BYTE 7



## Little Endian

(Intel, AMD, early ARM CPUs)

MSB	LSB
BYTE 1	BYTE 0
BYTE 3	BYTE 2
BYTE 5	BYTE 4
BYTE 7	BYTE 6



MSB				LSB
BYTE 0	BYTE 1	BYTE 2	BYTE 3	
BYTE 4	BYTE 5	BYTE 6	BYTE 7	

MSB				LSB
BYTE 3	BYTE 2	BYTE 1	BYTE 0	
BYTE 7	BYTE 6	BYTE 5	BYTE 4	

# Format of Machine Language Instructions

---

- ColdFire machine language instructions are variable in length: they can occupy *one*, *two* or *three* whole adjacent words, depending on the instruction type and the operands.
- *Note:* The original CISC M68000 has instructions that contained from one to seven whole words. The ColdFire instructions have less variability in length. This simplifies the implementation of the CPU.
- As in the M68000, ColdFire instructions are “word-aligned” in memory: i.e., they must start on even addresses.
- The first word in an instruction is the *operation word*. The left-justified *operation code* (“opcode”) in the operand word is from 4 to 16 bits long.

# ColdFire Opcode Map

---

Bits 15–12	Hex	Operation
0000	0	Bit Manipulation/Immediate
0001	1	Move Byte
0010	2	Move Longword
0011	3	Move Word
0100	4	Miscellaneous
0101	5	ADDQ/SUBQ/Scc/TPF
0110	6	Bcc/BSR/BRA
0111	7	MOVEQ/MVS/MVZ
1000	8	OR/DIV
1001	9	SUB/SUBX
1010	A	MAC/EMAC instructions/MOV3Q
1011	B	CMP/EOR
1100	C	AND/MUL
1101	D	ADD/ADDX
1110	E	Shift
1111	F	Floating-Point/Debug/Cache Instructions

Copyright © 2008 by Freescale Semiconductor, Inc.

Copyright © 2020 by Bruce Cockburn

3-36

# Addressing Modes

---

- The source and destination operands of machine language instructions can be specified in many different ways called “addressing modes”.
- ***Simple addressing modes*** are available for specifying operands in terms of data values, register contents, or the contents of specified memory locations.
- ***Compound addressing modes*** are used to specify the operands in terms of two or more components. This is convenient for accessing common data structures.
- Some addressing modes are implied by the instruction type, and do not need to be encoded separately.

# ColdFire Addressing Modes

---

Addressing Modes	Syntax	Mode Field	Reg. Field	Data	Memory	Control	Alterable
Register Direct Data Address	Dn An	000 001	reg. no. reg. no.	X —	— —	— —	X X
Register Indirect Address Address with Postincrement Address with Predecrement Address with Displacement	(An) (An)+ -(An) (d <sub>16</sub> ,An)	010 011 100 101	reg. no. reg. no. reg. no. reg. no.	X X X X	X X X X	X — — X	X X X X
Address Register Indirect with Scaled Index and 8-Bit Displacement	(d <sub>8</sub> ,An,Xi*SF)	110	reg. no.	X	X	X	X
Program Counter Indirect with Displacement	(d <sub>16</sub> ,PC)	111	010	X	X	X	—
Program Counter Indirect with Scaled Index and 8-Bit Displacement	(d <sub>8</sub> ,PC,Xi*SF)	111	011	X	X	X	—
Absolute Data Addressing Short Long	(xxx).W (xxx).L	111 111	000 001	X X	X X	X X	— —
Immediate	#<xxx>	111	100	X	X	—	—

Copyright © 2008 by Freescale Semiconductor, Inc.

# Simple Addressing Modes

---

## 1) Immediate Data

#<value>

Depending on the operand size specified by the instruction the data is obtained from:

.B => the low order byte in the operation word

.W => the next word following the operation word

.L => the next longword following the operation word

## 2) Quick Immediate Data

#<imm3> or #<imm8>

Similar to immediate data, except that the value of the operand is restricted to either 3 bits or 8 bits, depending on which quick instruction is used.

Instruction execution is a bit faster because the operand is embedded in the operation word.

# Simple Addressing Modes (cont'd)

---

- 3) Absolute Short Address** (0xNNNN).W  
The included 16-bit integer is sign-extended to 32-bits to produce the address of the operand in memory.
  
- 4) Absolute Long Address** (0xNNNNNNNN).L  
The two included words are concatenated to form the 32-bit address of the operand in memory.
  
- 5) Data Register Direct** Dn  
The operand is contained in the specified data register.
  
- 6) Address Register Direct** An  
The operand is contained in the specified address register.

# Indirect Addressing Modes

---

- 7) **Address Register Indirect** (An)  
Operand is stored in memory at the address contained in the given address register An.
- 8) **Address Register Indirect with Postincrement** (An)+  
Same as address register indirect, except that the contents of An are incremented at the end of instruction execution by 1, 2 or 4 if the operand size is byte, word or long word, respectively.
- 9) **Address Register Indirect with Predecrement** -(An)  
Same as address register indirect, except that before any operands are selected by the instruction, the contents of An are decremented by 1, 2 or 4 if the operand size is byte, word or long word, respectively.

# Compound Addressing Modes

---

## 10) Address Register Indirect with Displacement

( $<\text{d16}>$ , $\text{An}$ )

Similar to address register indirect, except that a 16-bit 2's-complement displacement  $<\text{d16}>$  is added to the address retrieved from the given address register  $\text{An}$  to give the address of the operand in memory.

## 11) Address Register Indirect with Scaled Index and 8-bit Displacement

( $<\text{d8}>$ , $\text{An.s}$ , $\text{Xn.s} * \text{scale}$ ) where scale = 1, 2, 4 or 8

Similar to address register indirect, except that an 8-bit 2's-complement displacement  $<\text{d8}>$  and the (possibly scaled) contents of an index register ( $\text{Xn.s} = \text{An.W}$ ,  $\text{An.L}$ ,  $\text{Dn.W}$  or  $\text{Dn.L}$ ) are added to the address from the given register  $\text{An}$  to give the address of the operand in memory.

# Compound Address Modes (cont'd)

---

## 12) Program Counter with Displacement ( $<\text{d16}>$ ,PC)

The given 16-bit 2's-complement displacement  $<\text{d16}>$  is sign-extended to 32 bits and then added to the contents of the program counter to obtain the memory address of the operand.

## 13) Program Counter with Scaled Index and 8-bit Displacement

( $<\text{d8}>$ ,PC, $Xn.s * \text{scale}$ ) where scale = 1, 2, 4, or 8  
Similar to program counter with displacement, except that the displacement  $<\text{d8}>$  is 8-bit and the contents of an index register ( $Xn.s = An.L$ ,  $An.W$ ,  $Dn.L$  or  $Dn.W$ ) are added to the program counter to obtain the memory address of the operand. The  $.W$  index register values are sign-extended to 32 bits before being added in.

## 14) Implied Register Addressing Mode

---

- Some instructions imply that certain CPU registers contain the source and/or the destination operand(s).  
JSR, BSR, RTS, RTE, etc.
- In such instructions, there is no need to encode the operands explicitly as the addressing mode.
- Motorola/Freescale view these instructions as using an “Implied Register” addressing mode.
- Possible implied CPU registers:  
PC, SSP, USP, SR

# Eight Broad Classes of ColdFire Instructions

---

- 1) Data Movement
- 2) Integer Arithmetic
- 3) Logical Operations
- 4) Shift Operations
- 5) Bit Manipulation
- 6) Program Control
- 7) System Control
- 8) Cache Maintenance

*Note:* Not all of the ColdFire instructions are listed in the following pages.  
Consult the ColdFire Family Programmer's Reference Manual for the full list.

# 1) Data Movement Instructions

---

MOVE.s <ea1>,<ea2>	General-purpose data movement, s = B, W or L
MOVEA.s <ea>,<Ax>	Move operand into an address register, s = W, L
MOVEM.L <ea1>,<ea2>	Move multiple registers to/from memory region
MOVEQ.L #<data>,<Dx>	Move 8-bit data to data register
MOVE.W CCR,<Dx>	Move zero-padded CCR to a data register
MOVE.B <ea>,<CCR>	Move 8-bit data into the CCR
MOVL <ea1>,<ea2>	Special MOVE instructions for the 8 EMAC regs. MACSR, ACC0/1/2/3, ACCext01/23 and MASK
MOVCLR.L ACCn,<Rm>	Rm <- ACCn in EMAC, then clear ACCn
LEA.L <ea>,<Ax>	Load effective address (i.e., initialize a pointer)
PEA.L <ea>	Push a pointer onto the A7 stack
LINK Ay,#<disp>	Create a stack frame to hold local variables
UNLK An	Remove a stack frame and restore frame pointer

## 2) Integer Arithmetic Instructions

---

ADD.L <ea1>,<ea2>	Add, where one operand must be a data register
ADDA.L <ea>,Ax	Add source operand to an address register
ADDI.L #<data>,Dx	Add 32-bit immediate data to a data register
ADDQ.L #<data>,Dx	Add a value from 1 to 8 to a data register
ADDX.L Dy,Dx	Add $Dy + Dx + X$ bit and then store sum in Dx.L
Subtract instructions are also available, with SUB replacing ADD above	
MULU.s <ea>,Dx	Multiply (unsigned) to Dx.L, s = W or L
DIVU.W <ea>,Dx	Divide (unsigned) Dx.L by <ea>.W and store the quotient in Dx.W and remainder in Dx.L[31:16]
DIVU.L <ea>,Dx	Divide (unsigned) Dx.L by <ea>.L and then store the quotient in Dx.L. Discard the remainder.
REMUL.L <ea>,Dw:Dx	Find remainder (unsigned) of Dx.L divided by <ea>.L and then store in Dw.L

Signed multiply, divide and remainder instructions are available using MULS, DIVS and REMS, respectively.

## 2) Integer Arithmetic Instructions (cont'd)

---

CMP.s <ea>,Dx	Compare with data register, s = B, W or L
CMPA.s <ea>,Ax	Compare with address register, s = W or L
CMPI.s #<data>,Dx	Compare data with data register, s = B, W or L
CLR.s <ea>	Clear, with s = B, W or L
NEG.L Dx	Negate Dm: compute (#0 – Dx.L) and store in Dx.L
NEGX.L Dx	Compute (#0 – Dx.L – X bit) and store in Dx.L
EXT.W Dx	Sign extend Dx.B to Dx.W
EXT.L Dx	Sign extend Dx.W to Dx.L
EXTB.L Dx	Sign extend Dx.B to Dx.L

## 2) Integer Arithmetic Instructions (cont'd)

---

Additional arithmetic instructions are available for the enhanced multiply-accumulate (EMAC) unit that is present in the MCF54415:

“Multiply Accumulate”

MAC.W Ry.b,Rx.c<scale>,ACCn

MAC.L Ry,Rx<scale>,ACCn

Ry and Rx are address and/or data registers

Upper (U) or lower (L) word specified by “b” and “c”

Optional <scale> is “<<1“ or “>>1” or “”

ACCn is one of ACC0, ACC1, ACC2 or ACC3

ACCn <= ACCn + scaled product of Rn.b x Rx.b

“Multiply Accumulate with Load” (same as MAC, but also do Rw.L <= <ea>)

MAC.s Rn.b,Rm.c<scale>,<ea>,Rw,ACCx

MAC.s Rn.b,Rm.c<scale>,<ea>&,Rw,ACCx (also AND <ea> with MASK)

“Multiply and subtract scaled product from accumulator”

Uses same syntax as MAC, but with MSAC instead

### 3) Bit-wise Logical Instructions

---

AND.L <ea>,Dx	Bitwise AND of <ea> and Dx, then store in Dx
AND.L Dy,<ea>	Bitwise AND of Dy and <ea>, then store in <ea>
ANDI.L #<data>,Dx	Bitwise AND of immed. data and Dx
OR.L <ea>,Dx	Bitwise OR of <ea> and Dx, then store in Dx
OR.L Dy,<ea>	Bitwise OR of Dy and <ea>, then store in <ea>
ORI.L #<data>,Dx	Bitwise OR of immed. data and Dx
EOR.L <ea>,Dx	Bitwise XOR of <ea> and Dx, then store in Dx
EOR.L Dy,<ea>	Bitwise XOR of Dy and <ea>, then store in <ea>
EORI.L #<data>,Dx	Bitwise XOR of immed. data and Dx
NOT.L Dx	Bitwise negate/flip contents of data register Dx

## 4) Shift Operations

---

ASL.L Dy,Dx	Arithmetic shift left; 0's inserted at the LSB
ASL.L #<data>,Dx	Arithmetic shift left; 0's inserted at the LSB
ASR.L Dy,Dx	Arithmetic shift right; sign extended at the MSB
ASR.L #<data>,Dx	Arithmetic shift right; sign extended at the MSB
LSL.L Dy,Dx	Logical shift left; 0's inserted at the LSB
LSL.L #<data>,Dx	Logical shift left; 0's inserted at the LSB
LSR.L Dy,Dx	Logical shift right; 0's inserted at the MSB
LSR.L #<data>,Dx	Logical shift right; 0's inserted at the MSB
SWAP.W Dx	Swap the two 16-bit words in a 32-bit register

*Note:* The M68000 and CPU32 rotate instructions (e.g., ROR, ROL) are not provided in the ColdFire instruction set.

## 5) Bit Manipulation Instructions

---

BSET.s Dy,<ea>	Set bit in <ea> at bit position given in Dy, s = B,L
BSET.s #<data>,<ea>	Set bit in <ea> at bit position #<data>, s = B,L
BCLR.s Dy,<ea>	Clear bit in <ea> at bit position given in Dy, s = B,L
BCLR.s #<data>,<ea>	Clear bit in <ea> at bit position #<data>, s = B,L
BCHG.s Dy,<ea>	Flip bit in <ea> at bit position given in Dy, s = B,L
BCHG.s #<data>,<ea>	Flip bit in <ea> at bit position #<data>, s = B,L
BTST.s Dy,<ea>	(Z bit in CCR) <= complement of specified bit
BTST.s #<data>,<ea>	(Z bit in CCR) <= complement of specified bit
BITREV.L Dx	Bitwise reverse the contents of Dx.L
BYTEREV.L Dx	Byte-wise reverse the contents of Dx.L
FF1.L Dx	Load Dx.L with offset to first 1 bit from MSB position

# 6) Program Control Instructions

---

*Returns:*

RTS

Return from subroutine

RTE

Return from exception handling routine (*Privileged*)

*Unconditionals:*

BRA <label>

Branch always using 8 or 16-bit displacement

BSR <label>

Branch to subroutine using 8 or 16-bit displacement

JMP <label>

Jump to instruction at <label> in program

JMP <ea>

Jump to instruction at <ea>

JSR <label>

Jump to subroutine at <label> in program

JSR <ea>

Jump to subroutine at <ea>

NOP

No operation (safely waste 3 core clock cycles of time)

TPF

Two-word NOP, with no pipeline synchronization

TPF.W #<data>

Four-word NOP, with no pipeline synchronization

TPF.L #<data>

Six-word NOP, with no pipeline synchronization

# 6) Program Control Instructions (cont'd)

---

## *Test Operand:*

- |            |  |
|------------|--|
| TST.s <ea> | Update CCR bits Z and N according to <ea>; s = B,W,L   |
| TAS.B <ea> | Update CCR bits Z and N according to the byte at <ea>. After that update, TAS.B sets bit #1 of the byte at <ea>. <i>Important:</i> TAS.B performs an indivisible read-modify-write operation on the bus: ideal for flag & semaphore updates. |

## *Conditionals:*

- |               |   |
|---------------|---|
| Bcc.B <label> | Branch using 8-bit displacement if condition “cc” is true; otherwise, continue with the next instruction  |
| Bcc.W <label> | Branch using 16-bit displacement if condition “cc” is true; otherwise, continue with the next instruction |
| Scc.B Dx      | Set Dx to 0x1 if condition “cc” is true; otherwise, clear to 0x0  |

The 16 possible values of the condition “cc” are given on the next slide.

## 6) Program Control Instructions (cont'd)

---

The 16 possible values of the condition “cc” are:

T	true	1	
F	false	0	
EQ	equal	Z=1	
NE	not equal	Z=0	
PL	plus	N=0	
MI	minus	N=1	
CC	carry clear	C=0	(HS, high or same, unsigned operands)
CS	carry set	C=1	(LO, low, unsigned operands)
VC	overflow clear	V=0	
VS	overflow set	V=1	
LT	less than	Destn. < Source	(signed operands)
GT	greater than	Destn. > Source	(signed operands)
LE	less than or equal	Destn. <= Source	(signed operands)
GE	greater than or equal	Destn. >= Source	(signed operands)
LS	low or same	Destn. <= Source	(unsigned operands)
HI	high	Destn. > Source	(unsigned operands)

# 7) System Control Instructions

---

*Moves involving the Condition Code Register (non-privileged):*

MOVE.B CCR,Dx

MOVE.B Dy,CCR      *Note: Upper byte of SR is not changed*

MOVE.B <ea>,CCR    *Note: Upper byte of SR is not changed*

*Privileged instructions (supervisor mode only):*

MOVE.W SR,Dx      *Note: Unimplemented bits read as 0's*

MOVE.W Dy,SR

MOVE.W <ea>,SR

MOVEC.L Ry,Rx      Move longword between control registers

STRLDSR #<data>   Push SR onto stack, and then SR <= #<data>

RTE                  Return from exception handling routine

HALT                Halt processor core

STOP #<newSR>   Load SR; stop execution and wait for interrupt

## 7) System Control Instructions (cont'd)

---

### *Trap-generating:*

TRAP #<4bitval>	Force one of 16 possible traps
ILLEGAL	Force an ILLEGAL exception

### *Debug:*

PULSE	Sends code to Processor Status (PST) output port
WDDATA.s <ea>	Drive debug data to DDATA output port, s = B,W,L
WDEBUG.L <ea>	(Privileged) Execute loaded debug command

## 8) Cache Maintenance Instructions

---

**CPUSHL** “Push and possibly invalidate cache line”

If data is valid and modified, push specified cache line; invalidate cache line if programmed in Cache Control Register (CACR) (this synchronizes the CPU pipeline).

The MCF54415 contains an 8-Kbyte instruction cache memory to speed up instruction execution. The cache can be configured using the CACR into the following modes:

- Instruction cache
- Write-through data cache
- Split instruction/data cache

CPUSHL is a way of invalidating a single cached instruction. It is faster than starting a full cache invalidation sequence. CPUSHL is a privileged instruction.

# Subroutines

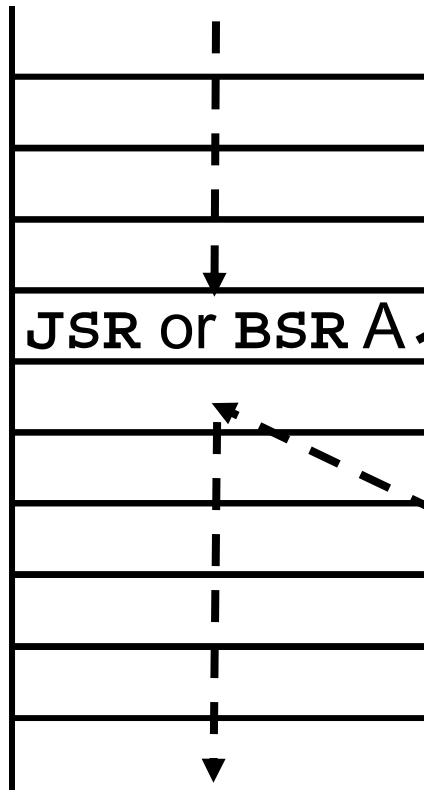
---

- A subroutine is a short program segment that can be called by any other program, including itself.
- Subroutines are typically used to encapsulate frequently used code.
- Subroutines provide more efficient use of memory since the enclosed code does not have to be repeated in several places in other program(s).
- Subroutines promote the re-use of known-good code, which increases programmer productivity.
- Subroutines cost a small extra amount of execution time when compared to “in-line” code.

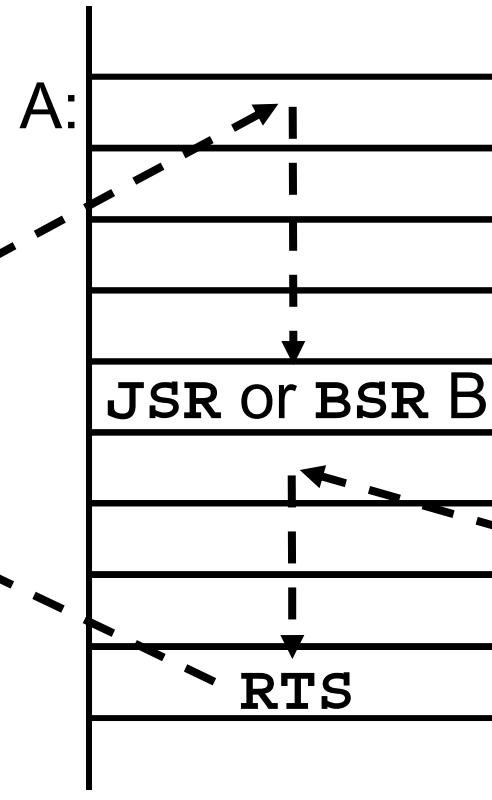
# Execution Pattern for Subroutine Calls

---

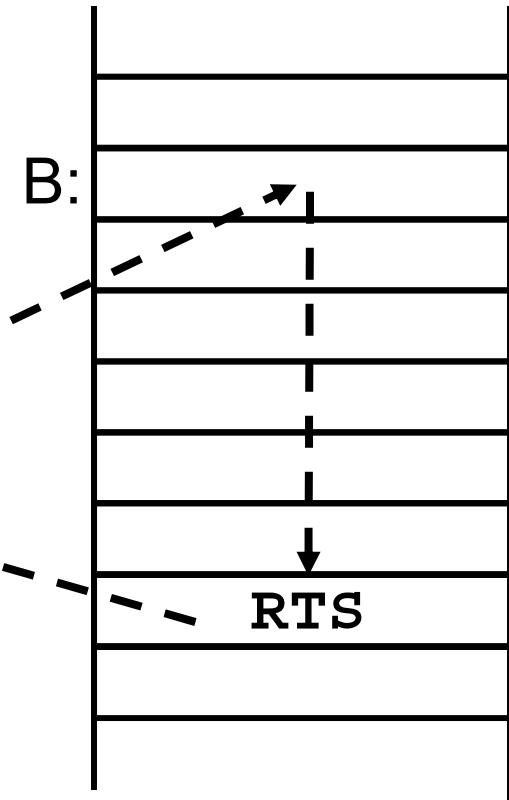
Main Program



Subroutine A



Subroutine B



# Parameter Passing

---

- Input and output parameters can be passed into and out of a subroutine using a number of different techniques:
  - CPU registers
  - memory locations in the heap (not on the stack)
  - memory locations on the stack
  - as values embedded among the instructions

# The Normal Processing State

---

- In normal instruction processing, the next instruction that will be executed by the CPU is the one that is pointed to by the Program Counter (PC) once the end of the current instruction has been reached.
- This PC value is determined either:
  - Implicitly: The PC is incremented automatically past the end of the current instruction to point to the first word of the next instruction in memory.
  - Explicitly: The PC is changed to any desired address. This address may be given as an absolute address (as in jumps), or as a branch displacement that must be added to the implicitly determined PC value.

# Exceptions, Traps, and Interrupts

---

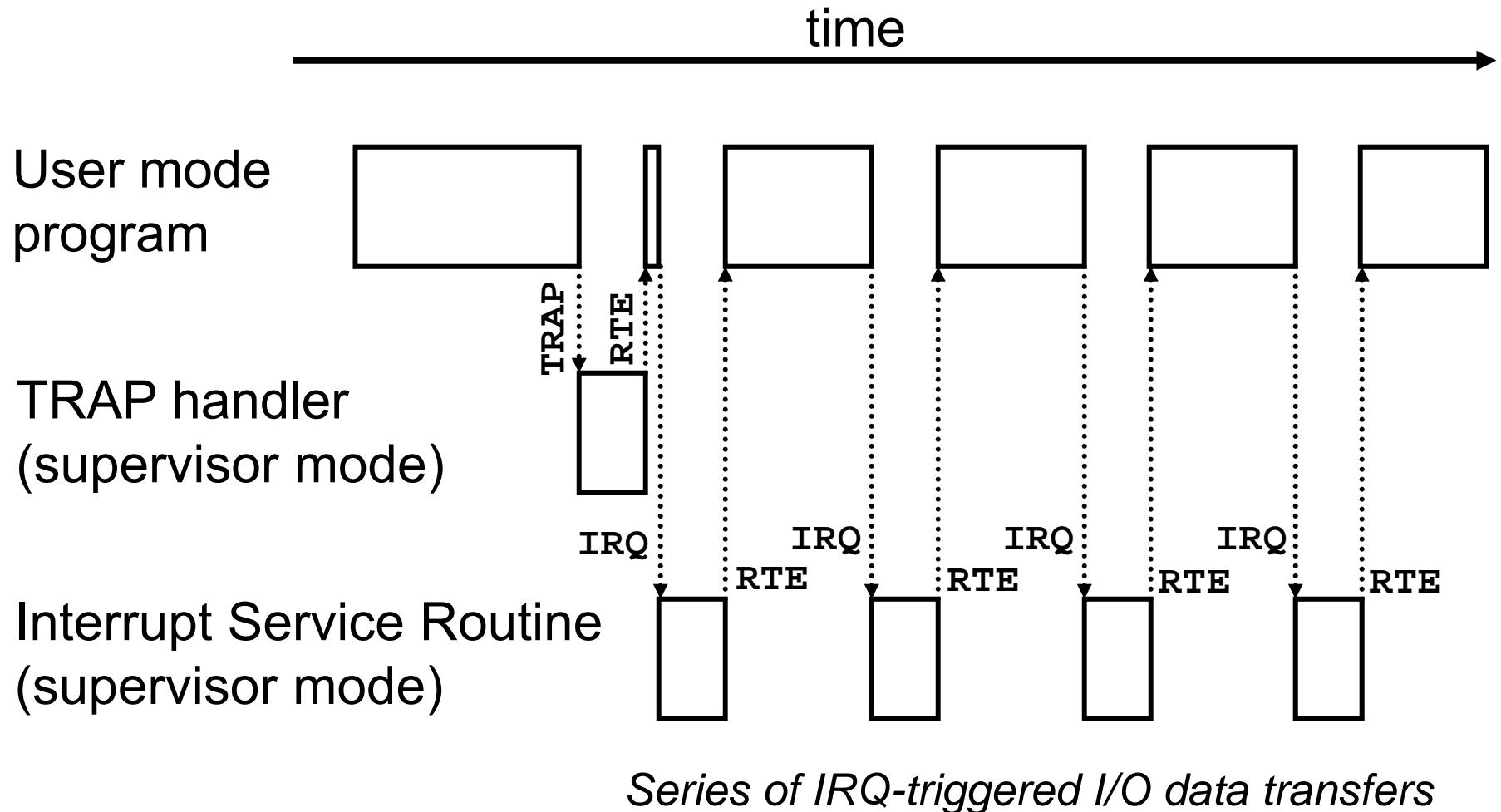
- It is often convenient to have a way of diverting the CPU out of the normal processing state so that it can deal promptly with special situations. Freescale (following Motorola) calls these situations “exceptions” and “interrupts”.
  - 1) Typical internally-caused exceptions (also called traps):
    - software traps: e.g., I/O routines, OS routines
    - arithmetic exceptions: e.g. divide-by-zero
    - trace mode for program debugging
  - 2) Typical externally-caused exceptions (also called interrupts):
    - hardware interrupts: e.g. I/O hardware, timers
    - bus error
    - hardware reset signal

# Exception Processing (cont'd)

---

- Exceptions are dealt with by subroutine-like programs called “exception handling routines”. Sometimes we will also refer to “trap handlers” and “interrupt service routines”. These routines all execute in Supervisor Mode.
- Exceptions can occur during both normal program instruction execution, as well as during the execution of exception handling routines.
- The supervisor stack is used to store the return address of the program that was interrupted when the CPU was required to temporarily halt executing the current program, and then start executing the appropriate exception handling routine.
- Once an exception handling routine has finished, execution is returned to the interrupted program (or routine) by the **RTE** (return from exception) instruction.

# Interrupt-driven I/O Execution Pattern



# Interrupt Priority Levels

---

- Interrupt exceptions can be generated by events in peripheral devices (via peripheral interface chips) in the microcomputer or by other subsystems in the microcontroller chip itself (e.g., FEC, DMACs, timers in the MCF54415).
- The designer of the microcomputer must have grouped the possible interrupts into seven Interrupt Priority Levels.
- The “Interrupt Priority Mask” (I3-I0) in the SR records the level of the interrupt currently being handled. If no interrupt is currently active, then I3-I0 = 0b000.

*Level 7: (Highest priority)*

- A non-maskable interrupt: When such an interrupt occurs, the associated exception handling routine will be called even if another exception handling routine is being executed.

*Levels 6-1: (Six maskable levels, in decreasing priority)*

- An interrupt at any of these priorities is serviced by the CPU only if there is no other active interrupt at the same or at a higher priority level.

# The Interrupt Mask Bits

---

<b>Interrupt Mask Value</b>	<b>Levels Disabled (Masked)</b>
<b>I<sub>2</sub> I<sub>1</sub> I<sub>0</sub></b>	
1 1 1	Levels 1 - 6
1 1 0	Levels 1 - 6
1 0 1	Levels 1 - 5
1 0 0	Levels 1 - 4
0 1 1	Levels 1 - 3
0 1 0	Levels 1 - 2
0 0 1	Level 1
0 0 0	All levels enabled

# User vs. Autovectored Interrupts

---

- *User Interrupts:*
  - Initiated by events in peripheral subsystems, such as timers, interface chips, etc.
  - An “exception vector number” that indicates to the CPU where in memory to find the exception handling routine. In the M68000 and CPU32, the external device supplies this number during the IACK cycle.
- *Autovectored Interrupts:*
  - Also initiated by events in peripheral subsystems.
  - Interrupt handling hardware automatically determines which of 7 possible “levels” an active interrupt should be associated with.
  - The resulting level determines one of the 7 available autovectored interrupt handling routines to use.

# Exception Vectors

---

- An “exception vector” contains the long-word (4-byte) starting address of an “exception handling” or “interrupt service” routine in memory.
- The RESET exception vector is actually *two* vectors:
  - The address of the RESET handling routine.
  - The address of the first supervisor stack pointer.
- M68000, CPU32 and ColdFire microprocessors store 256 exception vectors together in an “Exception Vector Table”.
  - The table occupies the first 1024 bytes in the MC68000.
  - The table is pointed to by the contents of the “vector base register (VBR)” in the CPU32 and ColdFire. The table does not have to occupy the first 1024 bytes in memory (more flexibility).
  - In the ColdFire, the lowest 20 bits of the VBR are 0. This forces the Exception Vector Table to be aligned on 1-Mbyte boundaries.

# ColdFire Exception Vector Table

Vector Number(s)	Vector Offset (Hex)	Stacked Program Counter	Assignment
0	0x000	—	Initial stack pointer
1	0x004	—	Initial program counter
2	0x008	Fault	Access error
3	0x00C	Fault	Address error
4	0x010	Fault	Illegal instruction
5	0x014	Fault	Divide by zero
6–7	0x018–0x01C	—	Reserved
8	0x020	Fault	Privilege violation
9	0x024	Next	Trace
10	0x028	Fault	Unimplemented line-a opcode
11	0x02C	Fault	Unimplemented line-f opcode
12	0x030	Next	Debug interrupt
13	0x034	—	Reserved
14	0x038	Fault	Format error
15–23	0x03C–0x05C	—	Reserved
24	0x060	Next	Spurious interrupt
25–31	0x064–0x07C	—	Reserved
32–47	0x080–0x0BC	Next	Trap # 0-15 instructions
48–63	0x0C0–0x0FC	—	Reserved
64–255	0x100–0x3FC	Next	User-defined interrupts

"Fault" refers to the PC of the instruction that caused the exception; "Next" refers to the PC of the next instruction that follows the instruction that caused the fault.

Copyright © 2008 by Freescale Semiconductor, Inc.

Copyright © 2020 by Bruce Cockburn

3-70

# Interrupt/Exception Priorities

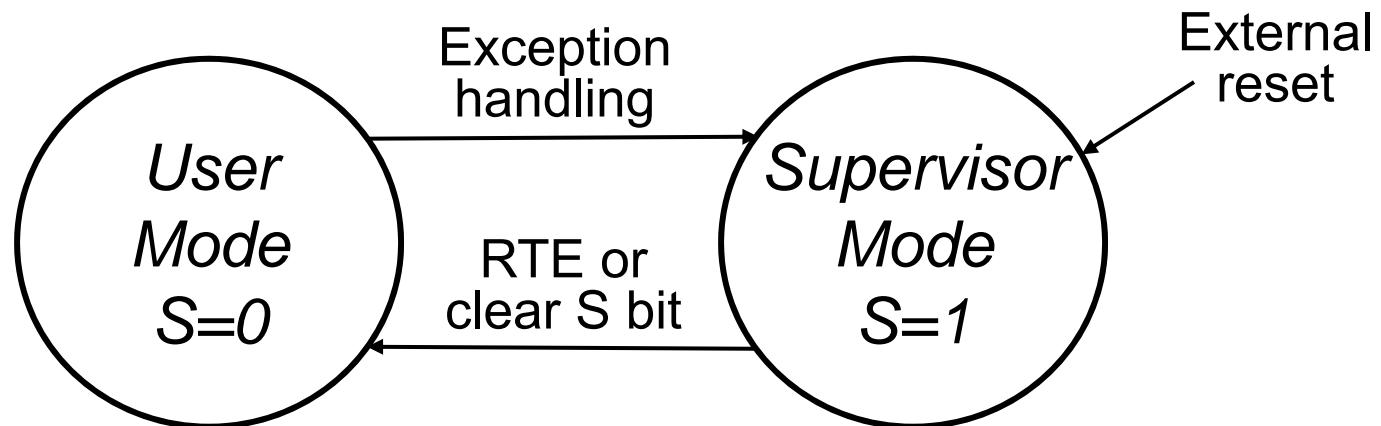
---

- The MCF54415 contains three interrupt controller modules, denoted by INTC0 (highest priority), INTC1 and INTC2 (lowest priority).
- Each INTC $x$  can handle up to 64 interrupt/exception sources
  - 7 external sources (up to 5 of them can be interrupt signals)
  - 173 fully-programmable interrupt/exception sources altogether
- The *Interrupt Acknowledge* (IACK) cycle is handled inside the MCF54415 by the two interrupt controllers (an IACK cycle does not appear on the external bus, as it did in the M68000 and CPU32). Interrupt requests and exceptions must be “cleared” in the hardware by the Interrupt Service and Exception Handling Routines.
- All exceptions after the first 63 in the Exception Vector Table are organized into 7 priority “Levels”, with each level containing 9 “priorities”.
- Within each level, the one fixed-level interrupt source has a midpoint priority. Eight additional interrupt sources can be mapped in software to four higher-than-midpoint and four lower-than-midpoint priorities.

# Transitions Between User and Supervisor States

---

- The handling of an exception causes a transition to Supervisor Mode, which is the mode used by all exception handling routines.
- Exception handling routines usually end with an **RTE** instruction, which restores the pre-existing mode.
- A supervisor mode program can also clear the S bit to enter User Mode. This can be done with a **MOVE to SR**.



# Automatic Exception Processing Sequence

---

## **Step 1:** (Save the old SR, and change the processor state)

- Make an internal copy of the SR.
- Enter Supervisor Mode by setting the S bit in the SR.
- Disable tracing by clearing the T bit(s). There is only one T bit in the 68000 and ColdFire; there are two T bits in the CPU32.
- For interrupts of a sufficiently high priority and for reset exceptions, update the interrupt priority mask bits (I2, I1, I0) in the status register. In the ColdFire, also clear the M bit (Master/Interrupt) in the SR.

## **Step 2:** (Get the appropriate Exception Vector Number)

- For interrupts, get the appropriate exception vector number during an “Interrupt Acknowledge Cycle” (IACK). During an IACK cycle, the interrupt priority is placed on the address bus.
- For all other exceptions, use internal CPU logic to determine the corresponding 8-bit Exception Vector Number.

# Automatic Exception Processing Sequence

---

## **Step 3:** (Save the current processor state on the stack)

- Create an exception stack frame on top of the supervisor stack.
- Load the SR and PC values, and possibly other information, into the new stack frame.
- The ColdFire has a single exception frame format.
- In the M68000 and CPU32, there were several frame formats.

## **Step 4:** (Start executing the exception handling routine)

- Multiply the 8-bit exception vector number by four to get the offset of the 32-bit Exception Vector into the vector table.
- Go into the exception vector table (in the M68000 the table base address is 0x0; in the CPU32 and ColdFire, the base address is in the VBR), retrieve the Exception Vector, and load it into the PC.
- If no other higher-priority exception is pending, resume the normal instruction fetch-decode-execute cycle.
- In the ColdFire, interrupt handling is disabled for the next instruction so that STRLDSR can be used to change the interrupt mask.

# The Reset Exception

---

- A reset exception is caused by a hardware reset signal originating from outside the microcontroller. (The ColdFire, unlike the 68xxx and CPU32, does not have a software RESET instruction for producing reset exceptions.)
- The reset exception is used for system initialization and recovery from catastrophic failure. No CPU state information is saved on the supervisor stack.
  - S bit is set
  - Trace mode is disabled (T bits cleared)
  - M bit is cleared
  - Interrupt Priority Mask is set to **0b111**
  - Vector Base Register (VBR) is cleared
  - SP is loaded with vector 0 at address **\$000**
  - PC is loaded with vector 1 at address **\$004**
  - First instruction of the system initialization routine is fetched, decoded and executed

# Memory Map Organization in the MCF54415

Table 1-2. System Memory Map per Boot Mode

Address Range <sup>1</sup>	Boot Source		
	FlexBus	Flash Controller	Serial Boot
0x0000_0000 0x0000_FFFF	FlexBus	NAND flash controller <sup>2</sup>	Internal SRAM <sup>2</sup>
0x0001_0000 0x00FF_FFFF			
0x1000_0000		FlexBus	FlexBus
0x3FFF_FFFF			
0x4000_0000	SDRAM controller		
0x7FFF_FFFF			
0x8000_0000 0x8BFF_FFFF <sup>3</sup>	Internal SRAM backdoor <sup>3</sup>		
0x8C00_0000 0x8FFF_FFFF	Rapid GPIO		
0x9000_0000	Reserved		
0xBFFF_FFFF			
0xC000_0000 0xDFFF_FFFF	FlexBus		
0xE000_0000 0xFFFF_FFFF	Peripheral bus controller 1 <sup>4</sup>		
0xF000_0000 0xFFFF_FFFF	Peripheral bus controller 0 <sup>4</sup>		

MCF5441x Reference Manual, Section 1.8

Boot software for bootstrapping the system: i.e., hardware configuration and testing, loading the operating system, etc.

1 Gbytes of SDRAM for the operating system, application software, and application data

Fast (on-chip) 64-Kbyte SRAM is mapped into this range. Locations can be accessed by the CPU and by one other bus master at the same time.

Noncacheable external peripherals

Noncacheable internal peripherals

# **Software Concepts for Embedded Systems**

# Software Development

---

- Software development is often both expensive and risky.
- The total expenses of software development and the significant risk of failure (surprisingly common in large projects) are not fully appreciated by many managers.
- Software is often impossible to specify completely at the start of a project. Software requirements are typically incomplete and often change as the project progresses.
- Software systems can be arbitrarily complex, and this complexity can easily overwhelm the understanding and capabilities of teams of even expert software designers.
- Turnover rates are high in the software industry. Expertise is lost as designers leave. New designers must be trained and are (at first) more likely to introduce software errors.

# Software Engineering

---

- "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [IEEE Std-610.12-1990]
- To increase the productivity of the software designers, and improve the reliability of the resulting software systems:
  - Adopt proven software engineering methods.
  - Program in a well-designed *high-level language*. In the embedded systems industry, C is dominant.
  - Use assembly language only when efficiency and/or fast execution speed are essential.
  - *Avoid designing the entire system from scratch.*
  - Re-use and adapt existing known-good software.
  - Build a new software system on top of an existing reliable operating system or kernel (e.g., MicroC/OS).
  - Grow a working simple prototype into the final system.

# Divide-and-Conquer Design Strategy

---

- A standard strategy for managing complex engineering design problems is to *partition the design into loosely interacting, simpler sub-systems or modules*.
- The simpler sub-systems can then be designed and verified separately. The resulting sub-system solutions can then be combined to solve the original design problem.
- In software systems, the sub-systems could include:
  - operating system or kernel (usually from a 3<sup>rd</sup> party)
  - the file system (often from a 3<sup>rd</sup> party)
  - messaging system (usually part of the O.S.)
  - exception and/or interrupt handling routines
  - network interface and other device drivers
  - library routines, functions, object classes, etc.
  - webserver, other servers, custom forms, etc.
  - custom application tasks / processes / threads

# Aspects of Modular Design

---

- What makes a design have appropriate modularity?
- There are no hard rules. Here are some guidelines:
  - **Keep modules fairly short** and **easy to understand**: they should contain only one or two pages of code.
  - Modules should have **high cohesion**: each module is concerned with closely related functions and data.
  - Modules should enclose and **hide information details** from other modules that don't need those details.
  - Modules should be **loosely coupled**: there should only be relatively few & simple inter-module dependencies.
  - Only the **essential parameters** should be passed from one module to another.
  - **Global variables should not be used** to pass information between modules. A global variable, if one is required, should be updated by only one module.

# Sources of Real-time Events

---

- Real-time events originate either externally or internally to the embedded system.
- ***Externally initiated events***, for example:
  - A signal has been received from outside the system, such as a *user input* or a *communications message*.
  - A *sensor* has detected a change in the environment.
  - An *actuator* has completed a commanded action.
  - An *alarm condition* (e.g., power failure) has occurred.
- ***Internally initiated events***, for example:
  - A *timer-triggered interrupt* has occurred.
  - A *direct memory access data transfer* has finished.
  - A *software-triggered exception* has occurred.

# General Kinds of Real-Time Specifications

---

- An embedded real-time system must act in a way that satisfies all the real-time specifications.
- Typical kinds of real-time specifications:
  - ***Maximum response time*** to different events
  - ***Maximum variability/jitter in the response time***
  - ***Accuracy of the repetition frequency***: e.g., an analog sensor signal may need to be sampled and converted at some specific frequency.
  - ***Maximum variability/jitter in repetition frequency***
  - ***Degradation behaviour*** when the workload exceeds system capacity and/or if the battery charge is nearly exhausted. E.g., can less important activities be safely deferred (possibly by skipping over code or disabling interrupts) so that important events will be handled?

# The Important Role of Idle Time in Real-time Systems

---

- ***Idle time*** is CPU execution time when no real-time work is being performed by the software.
- Idle time can be used to perform low-priority work (e.g., background preemptive testing, measuring the amount of idle time by incrementing an idle instruction counter, memory defragmentation). When idle time is used for low-priority work it is sometimes called ***background time***.
- Sufficient idle time must be present in real-time systems:
  - ❖ For *externally initiated events*, idle time provides the necessary excess CPU capacity so that *worst-case bursts of event-handling workload can be handled* within the maximum response time and determinism (i.e., max. response time variability) specifications.
  - ❖ For *internally initiated events*, idle time provides flexibility so that the effects of *internal sources of timing variability* (e.g., variability in the execution time of compiled software, variability caused by cache memory and virtual memory, variability caused by DMA activity) *can be absorbed and effectively hidden*. The timing for internally initiated events should be determined by H/W timers.

# Programming on Bare Metal

---

- For the simplest embedded systems, it may be desirable to design the software system as a ***standalone program*** that runs directly on the microcomputer hardware without the presence of a kernel or operating system. This is often called programming on “***bare metal***”.
- The main alternative to a bare metal software design is one that runs on the microcomputer hardware along with generic kernel or operating system software.
- ***Main advantage of programming on bare metal:*** Very efficient operation with the simplest possible software.
- ***Disadvantages:*** The software will not leverage the features in available kernels or operating systems. It may be harder to add new functionality later. Leads to a less modular and less flexible S/W design. Harder to do divide-and-conquer.

# Software Architectures for Embedded Systems

---

## ***Single-threaded Architectures*** (often bare-metal)

- Foreground-background systems
- Sleep mode with interrupts
- One non-preemptive loop (no interrupts)
- One non-preemptive loop with interrupts
- Multiple non-preemptive loops with interrupts
- etc.

## ***Multitasking Architectures*** (using a kernel or operating system)

- Round-robin time slicing
- Periodically scheduled state-driven code
- Cooperative multitasking
- Preemptive multitasking
- etc.

# Foreground-Background Systems

---

- The simplest microcomputer-based systems might be programmed using one S/W thread executing in a single nonpreemptive loop, omitting a kernel or operating system.
  - The main loop can meet less challenging timing constraints through normal instruction execution.  
=> ***background processing***  
*Drawback:* A long main loop might cause overly slow or overly variable response time to input events. You will not likely get hard real-time performance.
  - Interrupt service routines provide faster response for critical events (assuming interrupt hardware is present)  
=> ***foreground processing***
- Foreground-Background systems are widely used in less demanding applications (e.g., appliances, simple games)

# Sleep Mode with Interrupts

---

- It may be possible and desirable to keep the microcomputer in a ***low-power sleep state*** most of the time. Stretching the lifetime of a battery-based power supply is typically a key priority. Hard real-time performance is not required.
- The microcomputer is ***woken up by external interrupt signals*** that exceed a specified interrupt priority level.
  - All of the “work” is done by the interrupt service routines. There is no background processing.
  - The MCF54415 provides a privileged STOP instruction that allows the software to load the SR with a new interrupt priority mask before putting the controller into one of three possible stop modes: *wait*, *doze* & *stop*.
  - Only sufficiently high interrupts will wake up the CPU.

# One Nonpreemptive Loop (1)

---

- Also called “***static nonpreemptive scheduling***”.
- A number of input sources or devices need to be polled. The required polling frequencies may be different.
- A ***single processing loop*** polls all of the inputs & devices in some fixed order. The loop iterates at a frequency that is at least as fast as the fastest required polling frequency.
- **The worst-case (longest possible) execution time of the software in the loop must be less than the loop period.**
- The looping frequency is best determined by a ***hardware timer***. Don’t rely on the instruction execution times to determine the looping frequency.
- A waiting sub-loop at the end of the main loop safely uses up excess time at the end of each iteration of the main loop.

# One Nonpreemptive Loop (2)

---

```
initialization_steps();

static int flag = 0;

while (1) {

    /* wait for the timer ISR to set flag to 1 */

    while ( !flag ) {}

    flag = 0;      /* cleared for next iteration */

    /* Start executing the nonpreemptive loop */

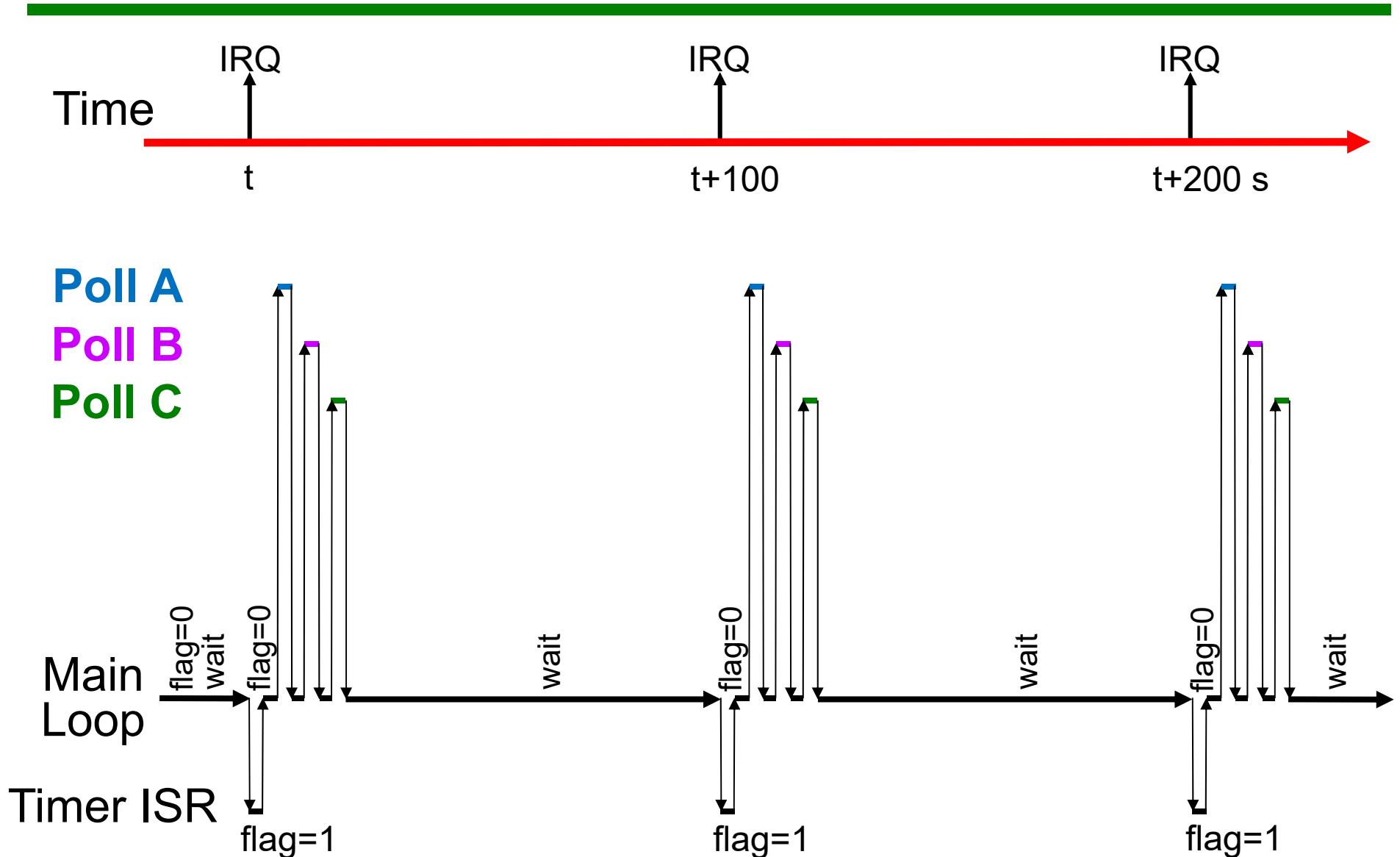
    poll_A();

    poll_B();

    poll_C();

} /* end while */
```

# One Nonpreemptive Loop (3)



# One Nonpreemptive Loop (4)

---

- ***Advantages:***
  - Simplicity
  - Guaranteed minimum polling frequency
  - Guaranteed maximum response time
- ***Disadvantages:***
  - Relatively poor determinism and possibly slow response when handing external events.
  - Some devices may be polled faster than what is necessary and/or preferred.
  - The single loop has relatively poor cohesion. This can be counteracted by using modular polling subroutines for each polled input and device.

# One Nonpreemptive Loop with Interrupts (1)

---

- Enhance a single nonpreemptive loop with a few interrupts to ensure that hardware-triggered events are serviced promptly and deterministically.
- Each ***interrupt service routine*** (ISR) should be ***short***. For example, an ISR might access a few hardware registers and transfer data to/from a buffer in memory. ***Detailed data processing should be done using software in the main loop, not the ISR.*** The ISR can enable data processing by signalling to a flag or semaphore that allows software outside the ISR to proceed later (within a brief delay).
- Keep the processor lightly loaded so that the worst-case execution of ISRs does not compromise the real-time performance of the main processing loop.

# One Nonpreemptive Loop with Interrupts (2)

---

```
initialization_steps();

static int flag = 0;

while (1) {

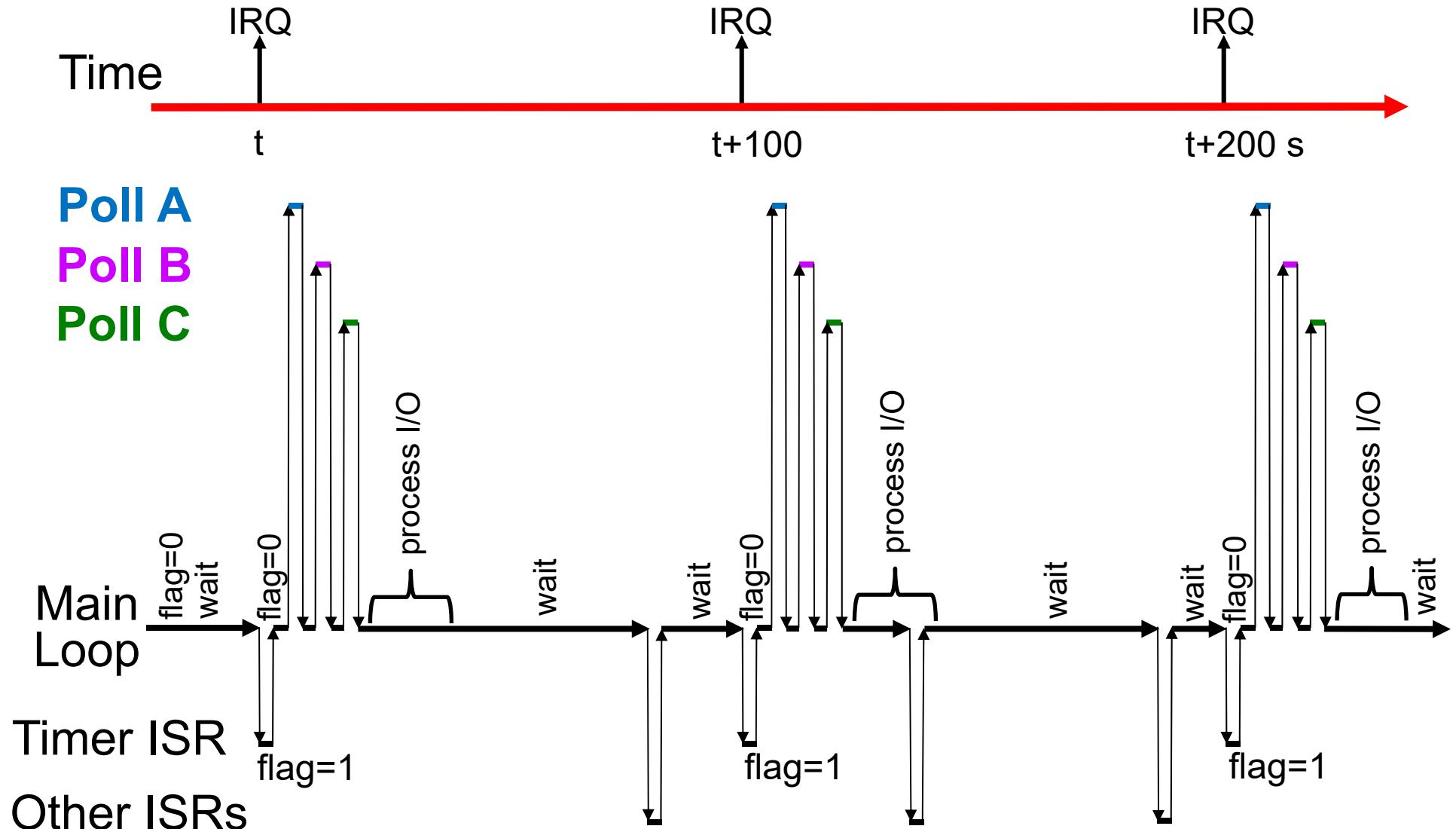
    /* wait for the timer ISR to set flag to 1 */
    while ( !flag ) {}

    flag = 0;

    /* Start executing the nonpreemptive loop */
    poll_A();
    poll_B();
    poll_C();
    process_data_inputs_from_other_ISRs();
    transfer_data_outputs_to_other_ISRs();

} /* end while */
```

# One Nonpreemptive Loop with Interrupts (3)



# One Nonpreemptive Loop with Interrupts (4)

---

- ***Advantages:***
  - Simplicity
  - Guaranteed minimum polling frequency
  - Predictable maximum response time for both polled devices and external events.
- ***Disadvantages:***
  - Some devices may be polled faster than necessary, which is wasteful of the processor's time.
  - The modularity of the single processing loop is reduced if the other IRQs generate work in the loop.
  - Frequent execution of multiple ISRs can reduce the determinism provided by each ISR to external events.

# Multiple Nonpreemptive Loops with Interrupts (1)

---

- Use ***two or more nonpreemptive loops*** that iterate at different, harmonic periods. Note: the multiple loops must be implemented using ***one software thread***. Use interrupts to provide fast and deterministic response to external events.
- Executions of the multiple loops should be enforced using a ***single hardware timer*** to ensure timing accuracy.
- ***Keep the ISRs short***. Most data processing must be done outside the ISRs, in the one main software thread.
- ***Keep the processor lightly loaded*** so that even the worst-case execution of ISRs does not compromise the real-time performance of the multiple processing loops.
- Each polled input or device is assigned to the one loop that iterates just fast enough to meet its real-time constraints (frequency and/or maximum response time).

# Harmonic Loop Periods

---

- When multiple nonpreemptive loops are executing in a real-time system, it becomes very difficult to predict the execution time behaviour and to guarantee that all the real-time response time constraints will be met.
- These problems are greatly reduced if the loop periods are **harmonic**, that is, each loop period is a whole multiple of **all** shorter loop periods.
- The loop periods 10 ms, 50 ms and 200 ms are harmonic. Note that  $50 = 5 \times 10$ ,  $200 = 20 \times 10$ , and  $200 = 4 \times 50$ .
- The loop periods 10 ms, 25 ms and 133 ms are not harmonic. (Note: 10 ms, 20 ms and 120 ms are harmonic).
- A single hardware timer should be responsible for producing the timing of all of the harmonic loop periods.

# Example with Two Nonpreemptive Loops (1)

---

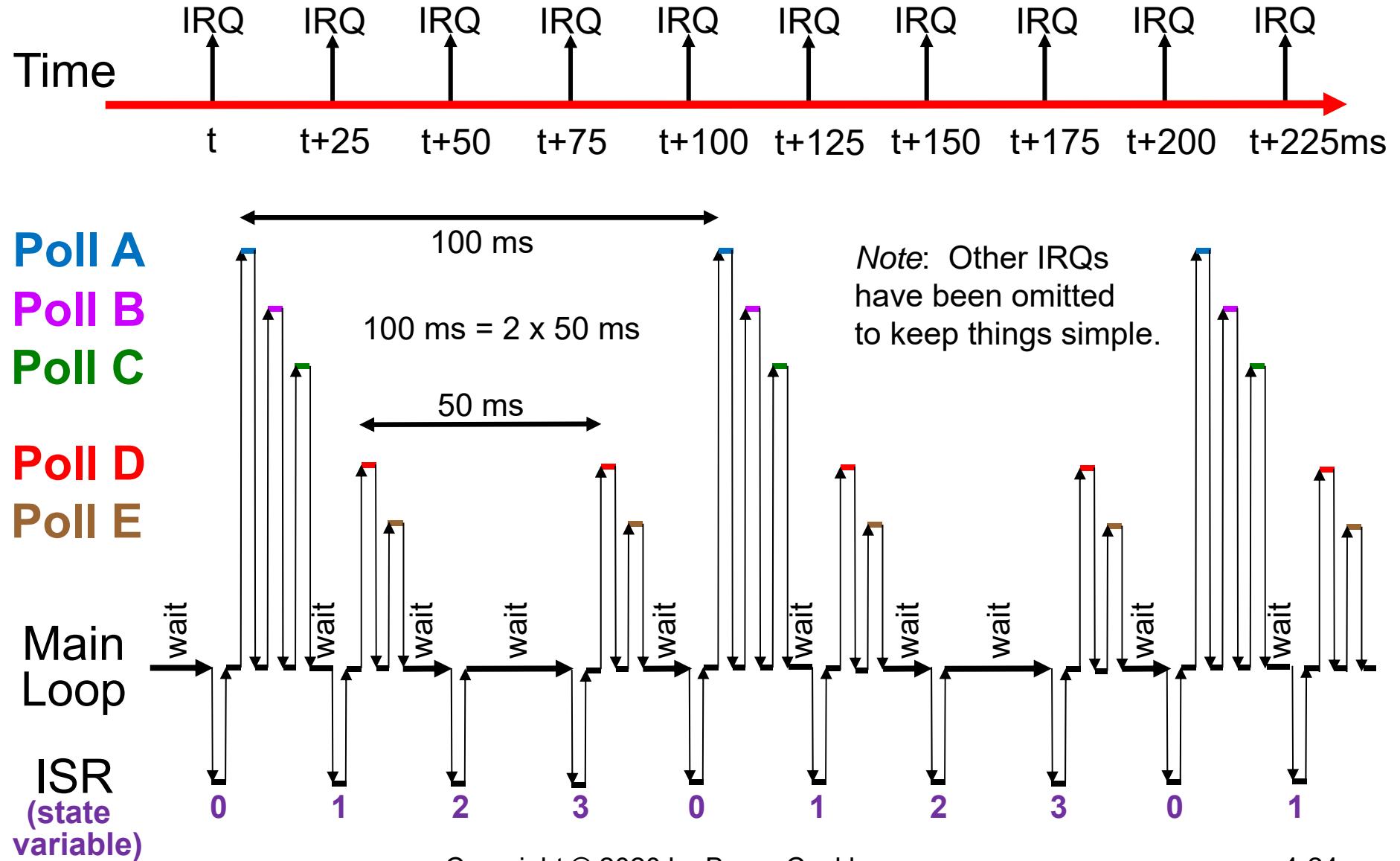
```
void function do_100_ms_loop() { poll_A(); poll_B(); poll_C(); }
void function do_50_ms_loop() { poll_D(); poll_E(); }

initialization_steps();
enum state_type { ST0=0, ST1, ST2, ST3 } state;
static int state = ST0; /* 100 ms list executes first */
static int flag = 0;

while (1) {
    /* wait for the timer ISR to set flag to 1 */
    while ( !flag ) {}

    switch ( state ) {
        case ST0: do_100ms_loop();      state = ST1; break;
        case ST1: do_50ms_loop();       state = ST2; break;
        case ST2: /* no loop here */   state = ST3; break;
        case ST3: do_50ms_loop();       state = ST0; break;
    } /* end switch */
    flag = 0;
} /* end while */
```

# Example with Two Nonpreemptive Loops (2)



# Meeting Real-time Constraints

---

## Multiple Nonpreemptive Loops with Interrupts (cont'd)

- ***Advantages:***
  - Guaranteed minimum polling frequency for each input or device
  - Guaranteed maximum response time for polled inputs and devices
  - External events are handled promptly and deterministically using hardware-triggered interrupts.
- ***Disadvantages:***
  - Some devices may be polled faster than necessary, which is wasteful of the processor's time.
  - The one software thread has less cohesion because it now handles more than one loop. A state variable keeps track of the timer ISR calls and the active loop.

# Tasks

---

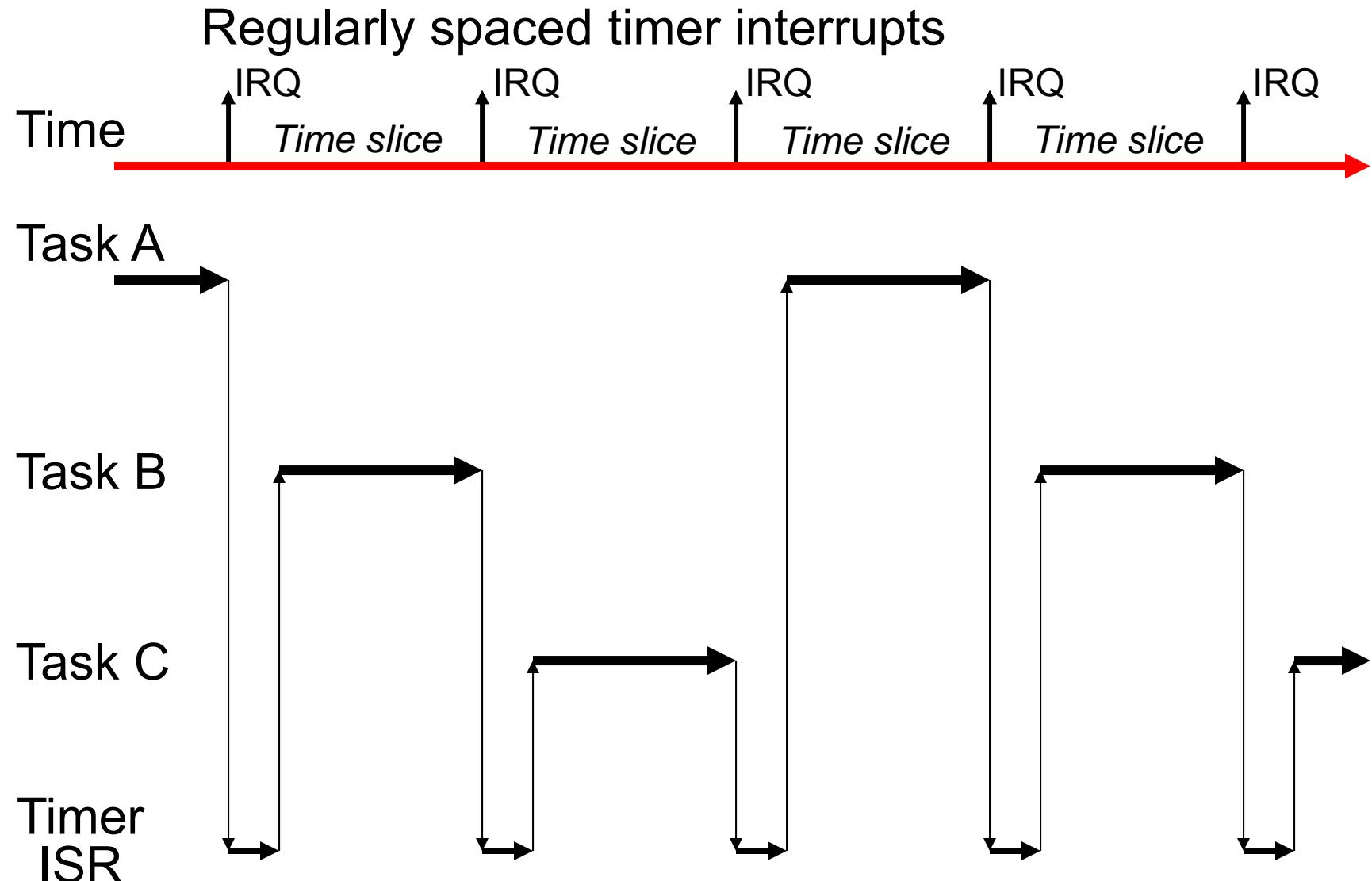
- A **task** is an executing program together with the current context of the task at the present program instruction.
- The **context** of a task has several components:
  - the contents of all *CPU registers*
  - the contents of the task's *stack* in RAM
  - the contents of any other *allocated RAM locations*
  - the state of any other *allocated system resources*
- A task can be stopped at one time and restarted later if the task context is saved in a special data structure, which is often called a **task control block** (TCB).
- The task control block will only have space to contain the CPU registers; the other parts of the context are left alone (but other tasks must be prevented from changing them).

# Partitioning Software into Multiple Tasks

---

- An effective way of developing a software system is often to design it as a collection of interacting tasks.
- Each task is written as if it has exclusive access to the CPU.
- Each task provides *one coherent service* in the system, and should therefore have a *relatively simple structure*.
- The tasks interact with each other using synchronization primitives (e.g., semaphores), messages, shared data structures, etc. to produce the desired system behaviour.
- System software is provided for stopping and restarting tasks. Usually only one task can run at a time on one CPU. (*Note:* processors are now widely available that provide hardware-controlled “multithreading” or “hyperthreading” that allows multiple tasks to appear to run on one CPU core.)

# Strictly Round-Robin Time Slicing



# Context Switching

---

- In a round-robin multitasking system, the timer ISR is part of a body of system software called the *kernel*.
- The timer Interrupt Service Routine must first decide if the currently executing task should be suspended.
- If not, then the ISR executes a “**return from exception**” to allow the same task to keep on running with another slice.
- If yes, then the ISR causes a “**context switch**” as follows:
  1. the contents of all CPU registers are loaded into the task’s Task Control Block (TCB).
  2. A new task is selected from a queue of ready-to-run tasks (actually, a queue of TCBs).
  3. The CPU registers are loaded from the register values that were saved previously in the new task’s TCB.
  4. The ISR executes a “return from exception” instruction, and the new task resumes executing where it left off.

# Is Time Slicing a Real-time Kernel?

---

- The time slicing multitasking architecture effectively creates multiple virtual CPUs, with one such CPU for each task.
- Time slicing is a simple way of *sharing one CPU* for the execution of *multiple non-hard-real-time software tasks*.
- The different tasks can be allocated different proportions of the available time on the one actual CPU. For example, some tasks can be allocated a greater fraction of the available equal-duration time slices per second.
- To provide some timing flexibility (at the cost of extra kernel complexity), tasks might have the ability to give up the CPU early before the end of their allocated slice.
- Real-time events (internal and external) can be handled by interrupt service routines in conjunction with tasks.

# Periodically-Scheduled State-Driven Code

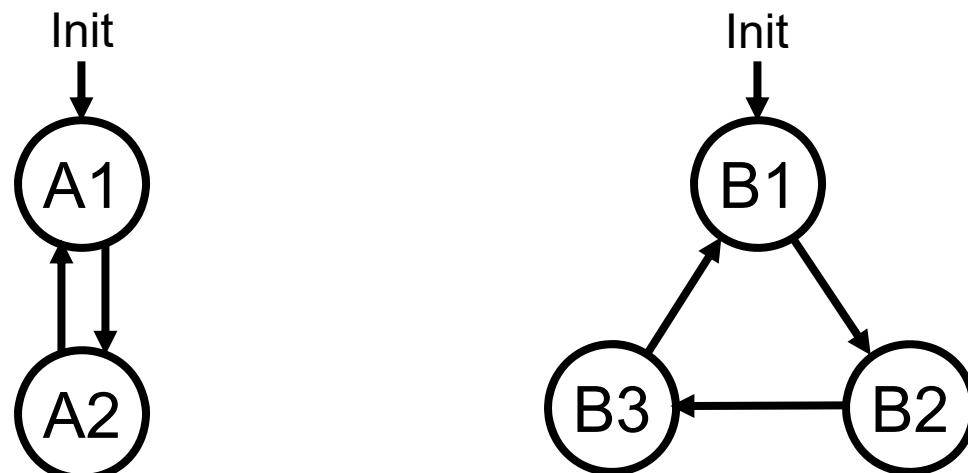
---

- In some applications, it may be possible to partition each task into a collection of interconnected *states*.
- Each state corresponds to a code segment that needs to be executed in that state (e.g., control and input/output actions).
- After a state code segment has been executed, the task gives up the CPU to the kernel. The delay until the execution of next state of a task can be fixed or changeable.
- Different tasks advance from state to state at predictable rates (e.g., some tasks at 10 ms intervals, other tasks at 100 ms intervals, others at 500 ms intervals, etc.). The periods are harmonic to make execution behaviour more predictable.
- When no task is running, non-state-driven tasks and/or a low priority ***idle task*** safely use up the remaining CPU time.

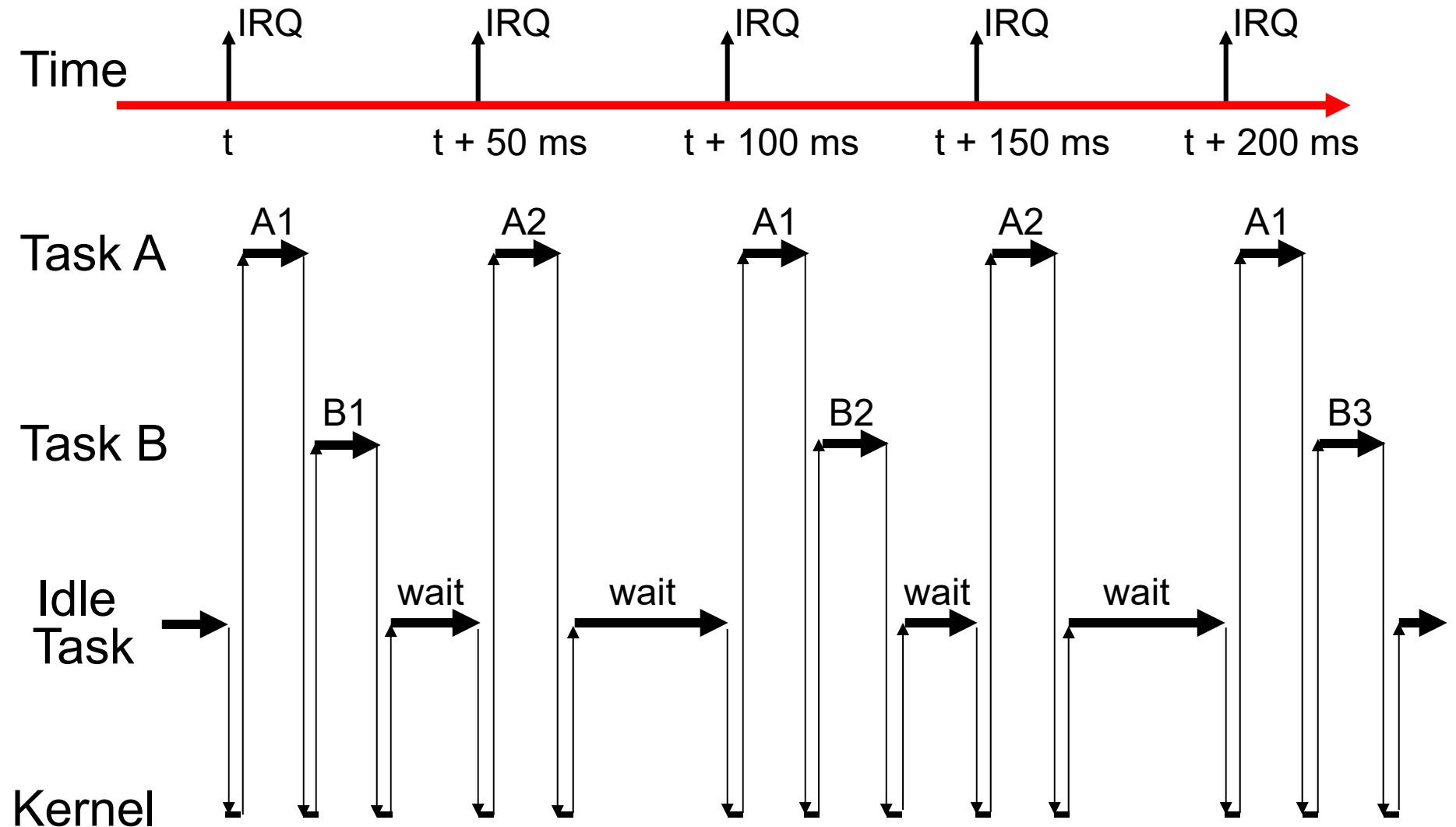
# Example of State-Driven Code (1)

---

- Consider a system partitioned into two state-driven tasks.
- Task A is to be scheduled to run every 50 milliseconds.
- Task B is to be scheduled to run every 100 milliseconds.
- The idle task is to consist of a simple "busy wait" loop that executes “no operation” (NOP) instructions.
- The state transition graphs for Tasks A & B are as follows:



## Example of State-Driven Code (2)



# Cooperative Multitasking (1)

---

- In **cooperative multitasking**, once a task starts to run on the CPU, it can continue to execute on the CPU until it decides to allow another task to execute.
- The tasks must interact cooperatively to ensure that the system behaves correctly.
- Task priorities are not used. However, prioritized hardware interrupts may still be present.
- TinyOS is an example of a successful cooperative multitasking operating system.
- How to introduce some idle time? One could introduce an idle task that busy waits (executes “no operation” or NOP instructions in a loop) until a hardware timer signals the start of the next looping interval.

# Cooperative Multitasking (2)

---

## ***Advantages:***

- Potentially very efficient. The number of context switches is minimized. One task can “hog” the CPU for as long as it needs.
- The operating system software is very simple.

## ***Disadvantages:***

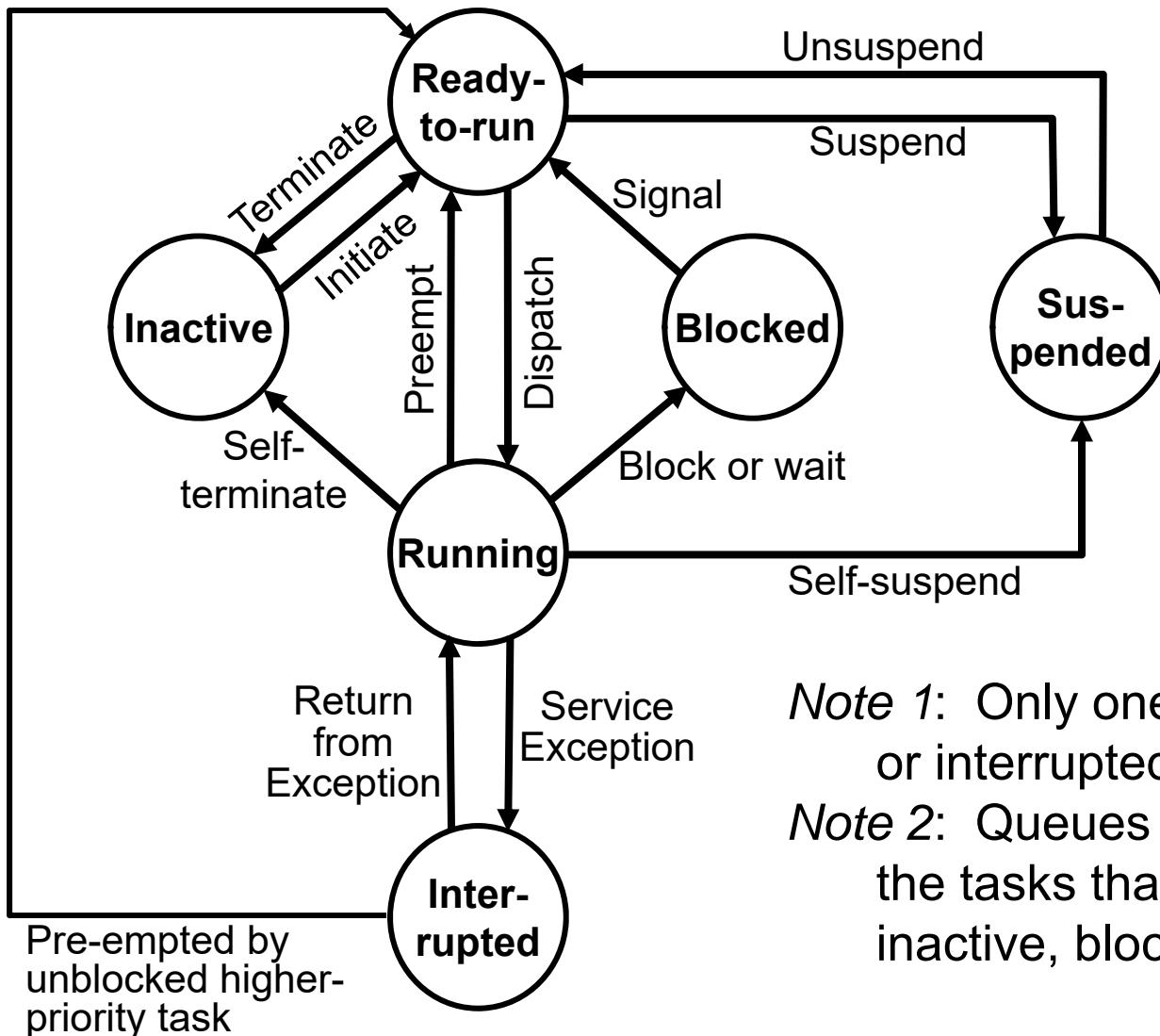
- The tasks have relatively poor modularity since the tasks must be designed to cooperate efficiently. One task can “hog” the CPU, preventing other tasks from running.
- Very hard to control or give guarantees on real-time response. ISRs can respond promptly, but the timing of subsequent task software is hard to predict.

# Preemptive Multitasking (1)

---

- In ***preemptive multitasking***, each task is assigned a ***priority*** that determines which among the one or more competing, ready-to-run tasks should be allowed to run on the CPU.
- **The highest-priority ready-to-run task always gets the CPU.** The operating system enforces this rule.
- The priority of each task is assigned to it when the task is first created. Usually, task priorities are fixed. Often task priorities are unique: each priority has at most one task.
- In some cases the priority of a task might be possible to change, but this leads to a situation where it becomes much harder to predict the real-time behaviour of the system.
- Idle time (e.g.,  $\geq 30\%$ ) must be present to provide timing flexibility. The idle time is safely consumed using by a lowest-priority idle task that is always ready-to-run.

# Process States in a Typical Preemptive Multitasking Environment



**Note 1:** Only one task is running or interrupted at any one time.  
**Note 2:** Queues are used to record the tasks that are ready-to-run, inactive, blocked, and suspended.

# Task States (cont'd)

---

**Running:** The one process that is actively executing instructions on the CPU (unless interrupted).

**Interrupted:** The one running process has been stopped temporarily to allow an interrupt service routine to run.

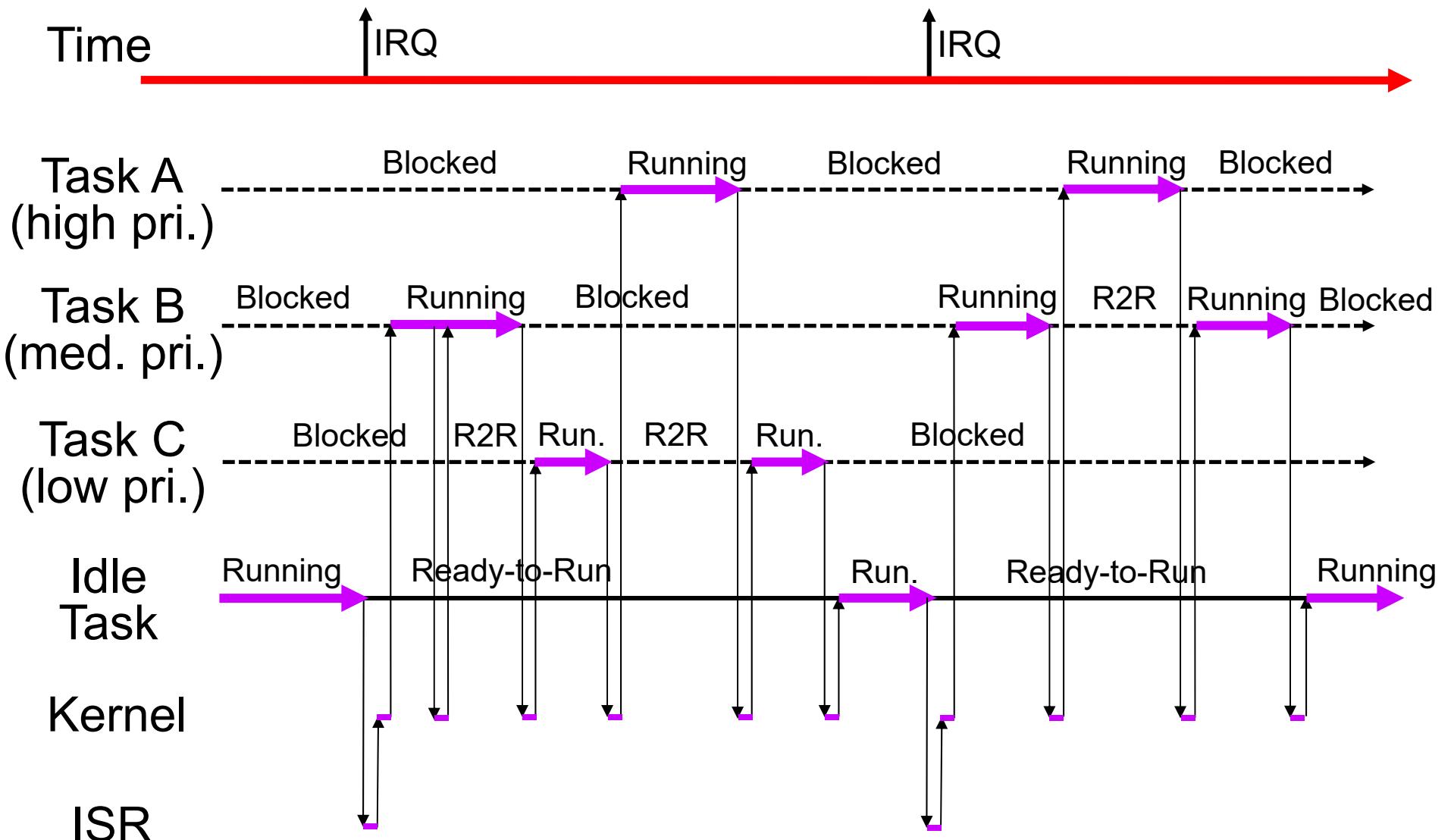
**Ready-to-run:** The process is available to start executing instructions as soon as the CPU becomes available.

**Blocked:** The process is waiting for a message to arrive, or is waiting for a flag or semaphore to become free.

**Suspended:** The process either suspended itself, or was suspended by another process.

**Inactive:** The process has been defined, but has either never been initiated, or was terminated later on.

# Preemptive Multitasking with Fixed Priorities



# Determinism: Predictability of Execution Timing

---

## **The highest priority task:**

- Not directly affected by the activities of any other task
- Delayed for short times by interrupt service routines

## **Other relatively high priority tasks:**

- Affected by the relatively few higher priority tasks
- Not affected by the execution of lower priority tasks
- Delayed for short times by interrupt service routines

## **Middle priority and lower priority tasks:**

- Affected by the execution of all higher priority tasks
- Not affected by the execution of lower priority tasks
- Delayed for short times by interrupt service routines

## **The lowest priority task (the Idle task):**

- Affected by the execution of all other tasks
- Delayed for short times by interrupt service routines

# Setting the Task Priorities

---

- In pre-emptive multitasking, the task priorities can be **fixed** (or static) from the time of task creation, or they can be **changeable** (or dynamic) as the tasks execute.
- **Fixed task priorities are generally preferred** because this assumption makes system behaviour much more predictable.
- In preemptive multitasking with fixed priorities, the priorities should be assigned to tasks using the **rate monotonic scheduling** (RMS) rule: that is, tasks that must execute more frequently or that have tighter deadlines when they are enabled should be assigned higher priorities (*lower priority numbers* in MicroC/OS).
- A few **safety-critical** or other **mission-critical** tasks might be given higher priorities than less critical tasks.

# Preemptive Multitasking

---

## ***Advantages:***

- Rate monotonic scheduling ensures that tasks that face tighter deadlines will have higher priority in getting time on the CPU. All real-time constraints will likely be met provided the system has sufficient idle time (e.g.,  $\geq 30\%$ ).
- Some of the context switching time of time slicing multitasking is avoided. The highest priority task can run efficiently, without interference, to its next blocking point.
- Inter-task interactions are minimized, which increases modularity and simplifies system design and analysis.

## ***Disadvantages:***

- Low-priority tasks, by being slow to post to synchronization primitives (e.g., semaphores) or by hogging shared resources, can block higher priority tasks from running.

# Related “Multi-” Terminology

---

Some definitions from the website “techtarget.com”:

- **Multiprogramming** is “the interleaved execution of two or more programs by a processor”.
- **Multitasking** is “the management of programs and the system services they request as tasks that can be interleaved (while running on the same CPU)”
- **Multithreading** is “the management of multiple execution paths through the computer or of multiple users sharing the same copy of a program”.
- **Multiprocessing** is “the coordinated processing of programs by more than one computer processor”.

# Tasks vs. Processes vs.Threads

---

- In larger computers systems (say those with full-scale operating systems, like Unix), it is more common to use the terms “process” and “thread” instead of “task”.
- Like a task, a **process** is associated with a program and a context. A process may also be associated with allocated memory regions, pointers, open files, sockets, etc.
- A **thread** (or light-weight process) is similar to a process, except that some resources may be shared with other active threads as part of a single process (e.g., files, memory space, memory pointers, input/output resources). Each thread has enough state information to be stopped and restarted independently (e.g., CPU register contents, its own stack space).

# Operating Systems

---

- An ***operating system*** is software infrastructure that provides services to human users and other software modules, while ensuring efficient and reliable access to computer resources.
- Services to human users include: login shells, shell scripts, a graphical user interface (GUI), a file system, access to input/output devices, access to the Internet, etc.
- Services to software modules include: system calls for accessing computer resources, inter-process synchronization and communication, time slicing and priorities to share the CPU's time among multiple active processes/tasks, etc.
- An operating system shields users and other software from much of the complexity of the computer system.

# Real-Time Operating Systems

---

- A ***real-time operating system (RTOS)*** is one that needs to provide services in a timely manner so that the computer system can interact effectively with on-going events and processes that are occurring in the physical environment.
- ***Hard real-time constraints*** are constraints with relatively inflexible and demanding maximum response times. Failure to meet hard real-time constraints is unacceptable. Predictability in the response times is often very important. Specially-developed operating systems are required.
- ***Soft real-time constraints*** are constraints that require response times that can be met successfully by many conventional operating systems (e.g., Unix/Linux and Windows) provided the system load is not excessive.

# Real-Time Kernels

---

- A ***kernel*** provides the most basic services that would be expected in a multitasking operating system:
  - task/process creation, initiation, termination
  - task/process suspension and unsuspension
  - (possibly) timer-controlled time slicing
  - (possibly) process priorities and pre-emption rules
  - exception and interrupt handling mechanisms
  - basic inter-process communication services
  - mechanisms for protecting critical sections
  - inter-process synchronization features
- A small-scale system may have an operating system that is really just a kernel on its own.
- A ***real-time kernel*** is one that is intended to be used in systems that must meet hard real-time constraints.

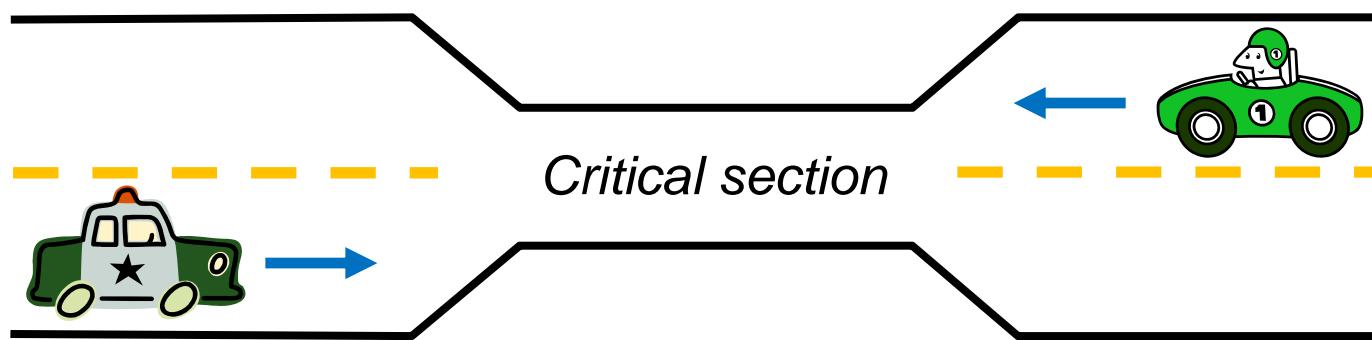
# Critical Sections

---

- It is common to have shared data structures that require the execution of several instructions in order to finish updates from one consistent state to a new consistent state.
  - shared data structures in the operating system (e.g., TCBs, synchronization primitives, I/O queues)
  - shared application data structures (e.g., two or more pointers need to be updated in a consistent way)
  - multiple registers in shared system hardware
  - etc.
- In a multitasking environment, there is the danger that shared data structures will be corrupted if a context switch occurs in the middle of a data structure update operation.
- A ***critical section*** is a section of code (e.g., data structure update) that must be executed to completion without being interrupted by task context switches or IRQs that could change/corrupt data that is updated in the critical section.

# Example: A Single-Lane Bridge

---



- Only one car can be on the bridge at a time. The bridge is like a critical section, and the cars are like two tasks.
- However, the analogy is not quite correct because the two cars could be moving at exactly the same time, whereas in a multi-tasking system on a single-threaded CPU, two tasks are never executing at the same time.
- One task will always arrive first at a critical section.

# Basic Strategies for Protecting Critical Sections

---

1. Disable all interrupts, allowing only one task to run
  - *Advantage:* Simple
  - *Disadvantage:* Too disruptive to the system?
2. Disable only those interrupts that could possibly threaten the critical section
  - *Advantages:* Simple and less disruptive than disabling all IRQs.
  - *Disadvantage:* Sometimes hard to determine those interrupts.  
The situation may change as new interrupts are introduced into a system that is modified and updated over time.
3. Disable multitasking, to allow only one task to run
  - *Advantage:* Simple
  - *Disadvantage:* All tasks are disturbed, which is disruptive to system. Critical sections must still be protected from interrupts.
4. Use semaphores to protect critical sections
  - *Advantage:* Minimizes the number of tasks that are affected.
  - *Disadvantage:* Critical sections within the semaphore functions must be protected from interrupts by disabling interrupts.

# Disabling all Interrupts

---

- (1) Mask out all interrupts (& disable multitasking) at the start of the critical section.
- (2) Restore interrupts (& re-enable multitasking) at the end of the critical section.

Example using a ColdFire processor (supervisor mode only):

```
MOVE.W  SR,-(SSP)    /* save SR on supervisor stack */
ORI.L   #0x0700,SR   /* disable IRQs */
/* critical section appears here */
MOVE.W  (SSP)+,SR    /* restore SR from supervisor stack */
```

*Advantages:*

- Simple to implement. Use a macro in high-level code.
- Fast to execute on the CPU

*Disadvantages:*

- Can be executed only in Supervisor Mode (uses SR,SSP)
- All non-level-7 interrupts in the system are affected
- Unnecessarily disruptive to the system?

# Using a Binary Flag to Protect a Critical Section

---

- A simple binary variable (often called a “flag”) can be used to enable or disable a task. Before proceeding into the critical section, the task must first consult the flag.
- If the flag is set to 1 (**green light**), the task can start executing the critical section.
- If the flag is cleared to 0 (**red light**), the task cannot enter the critical section. It must either do something else, or give up the CPU to another task and then try again later.
- The binary flag is usually implemented in a standard way as a ***binary semaphore*** (often called just a ***semaphore***).
- A binary semaphore is a special case of a more general software object, called a ***counting semaphore***.

# Synchronization Using Counting Semaphores

---

A counting semaphore is a software object with two parts:

- (1) an **integer count**, which is initialized to the number of available shared resources of a particular kind;
- (2) a **queue** of tasks that are blocked and waiting for the semaphore count to exceed a value of 0.

- “Count > 0” implies at least one resource is available. The value of count gives the number of available resources.
- “Count = 0” implies that no resources are available, and that no other tasks are blocked on the semaphore.
- “Count < 0” implies that no resources are available. The count’s absolute value gives the number of blocked tasks.

# Updating Counting (and Binary) Semaphores

---

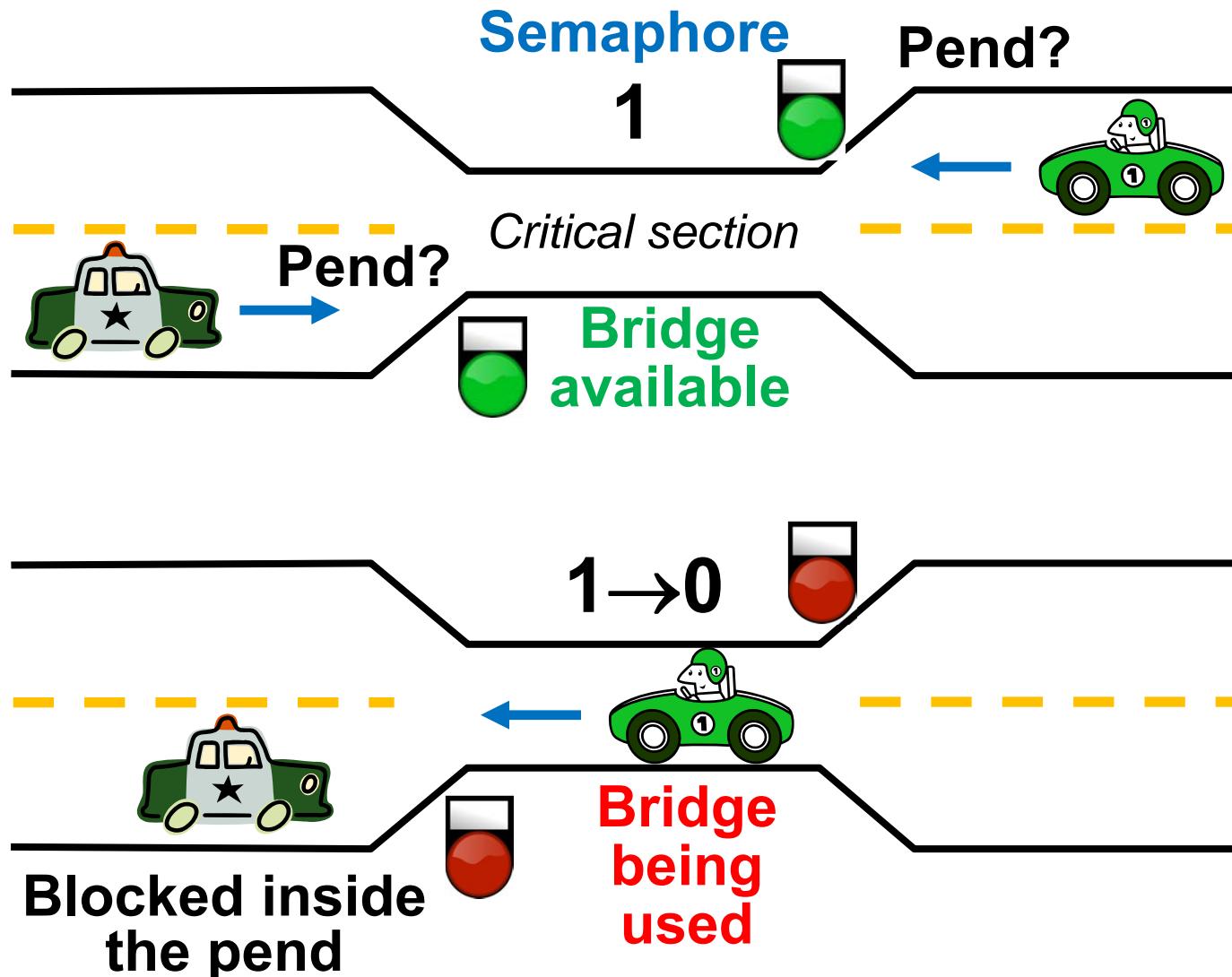
- **Pend** operation: A task needs exclusive use of a resource.
  - The count value is decremented by 1.
  - If the new count  $\geq 0$ , then the task is allocated one of the available resources from the shared pool.
  - If the new count  $< 0$ , then the task is switched off the CPU and is added to a queue of blocked tasks. A context switch occurs to a new running task.
- **Post** operation: A task has just finished using a resource.
  - The count value is incremented by 1.
  - If no task is blocked on the semaphore, simply carry on.
  - If tasks are blocked on the semaphore, then the highest priority blocked task is moved to the ready-to-run queue. A task context switch might occur as a result.
- *Note:* Both Pend and Post are critical sections that must be protected from corruption by ISRs. Lock out IRQs!

# Binary Semaphores

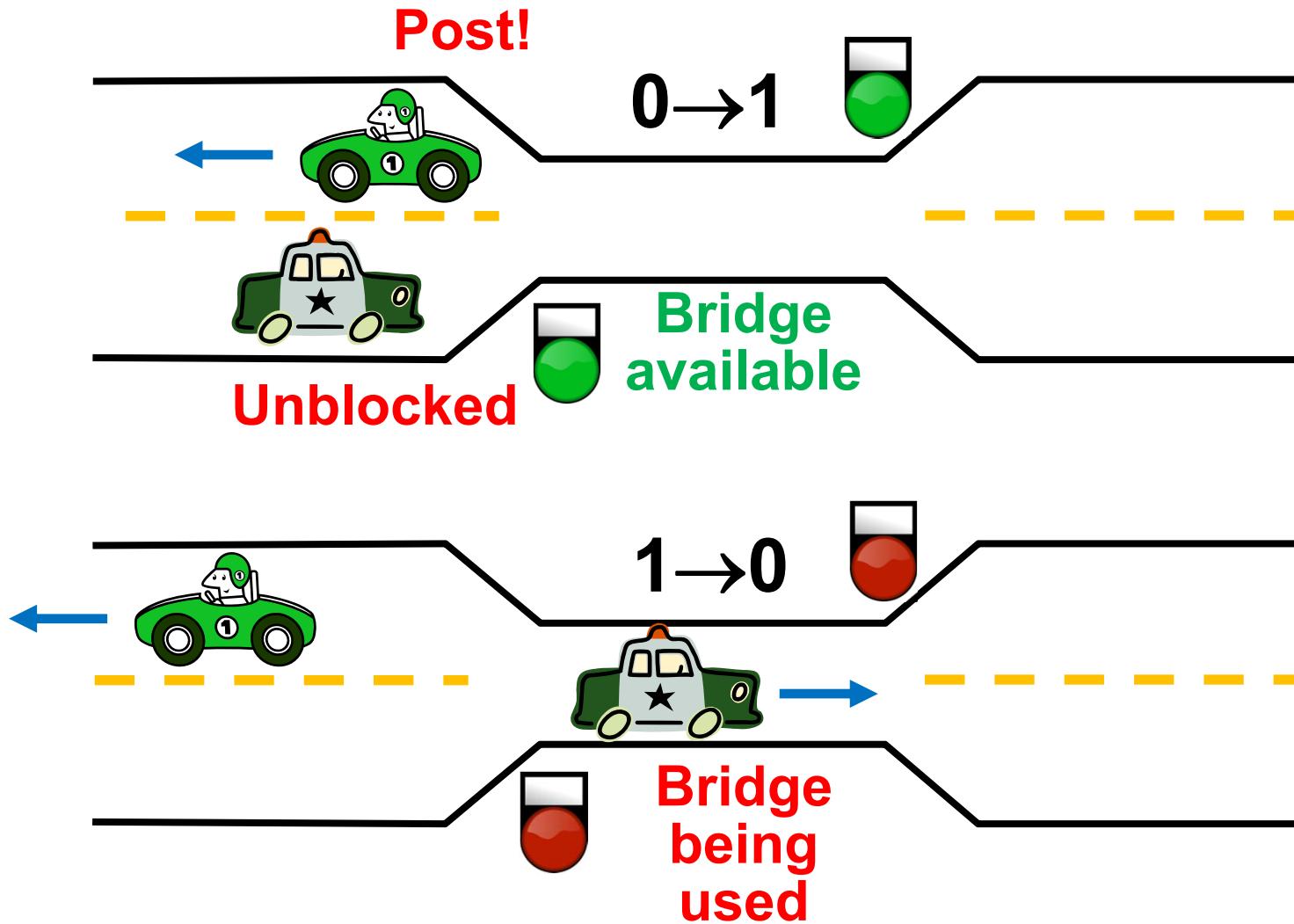
---

- A binary semaphore is a counting semaphore in which the count value is limited to 0 (**red light**) and 1 (**green light**).
- A binary semaphore can be used as a flag variable to enable the execution of a task at some point in the code.
- If the semaphore count is 0, then the task is blocked from executing (temporarily) when it does a "pend" operation on the semaphore. The kernel automatically schedules the next ready-to-run task, to allow the CPU to keep executing.
- If the semaphore count is 1, then the semaphore is decremented automatically to 0 by the "pend" function, and the task is allowed to enter the critical section.
- The semaphore count is incremented from 0 to 1 by a "post" operation to the semaphore, called by the task when it exits the critical section. *Note:* ISRs can also do a post.

# Back to the Single-Lane Bridge (1)

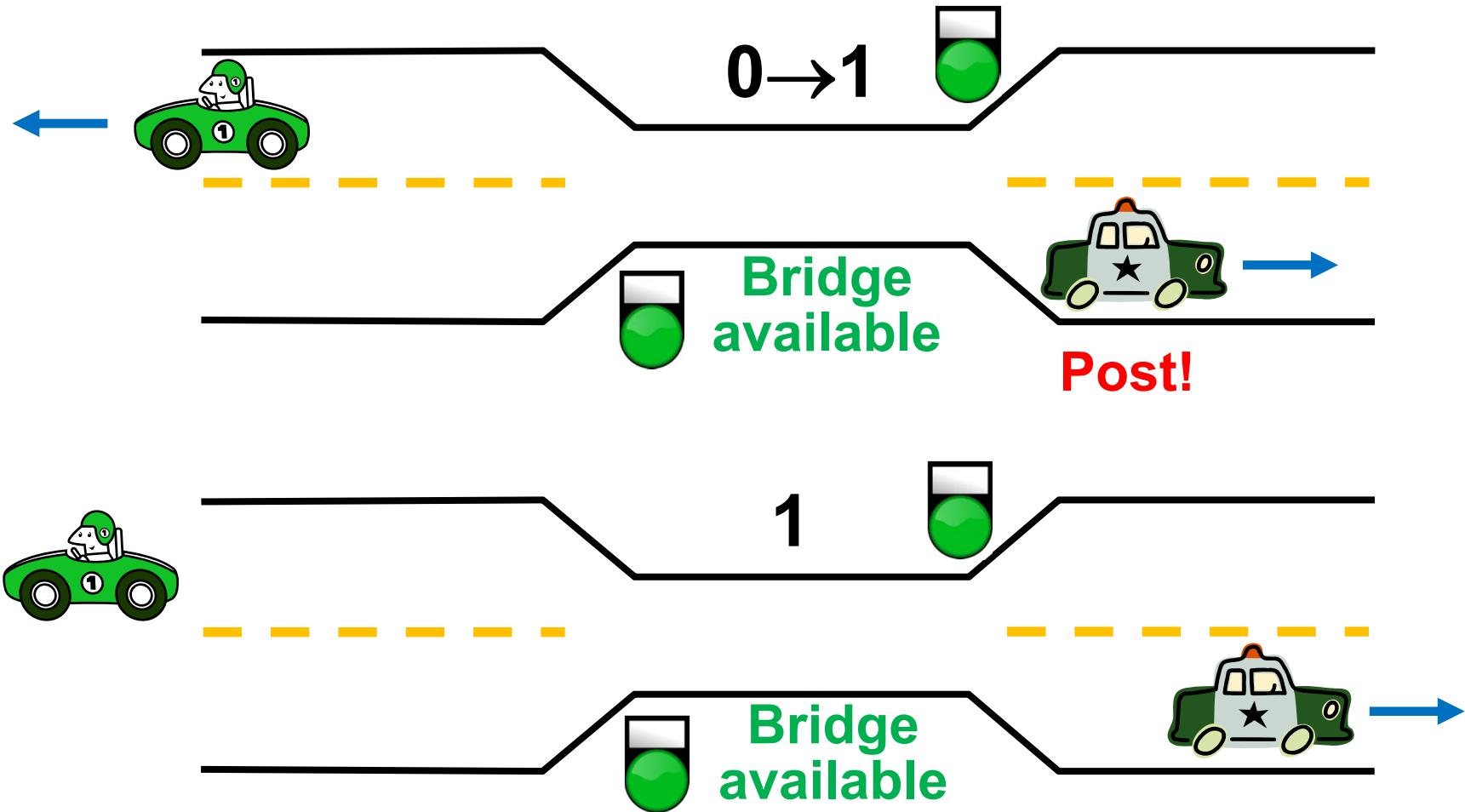


# Back to the Single-Lane Bridge (2)



# Back to the Single-Lane Bridge (3)

---



# Binary vs. Counting Semaphores (1)

---

- A **binary semaphore** is a **flag** that is used to synchronize the execution of two pieces of code (e.g., a task and an ISR).
  - The semaphore would typically be initialized to 0 to block a task from proceeding before some condition (e.g., some data has arrived to be processed) has been met.
  - The semaphore is set (posted or signalled) to 1 to unblock the task after the condition for execution is met.
  - For example, a message handling task might be blocked (when it pends on the semaphore) up until the time when all of the bytes in a new message have been fully transferred from an input hardware interface into a data structure by an interrupt service routine (which posts the semaphore when the data transfer is completed).

# Binary vs. Counting Semaphores (2)

---

- A **counting semaphore** is typically used to keep track of the number of available resources in a finite pool of resources.
  - The semaphore count is initialized to the size of the pool. All resources are initially available.
  - Each time that a resource is assigned and becomes busy, the count is decremented by 1 (pend); each time a resource is returned, the count is incremented by 1 (post)
- A counting semaphore can be used to implement a **binary semaphore**. The post and pend routines are the same, but the initial value of the count in a binary semaphore will be restricted to 0 (blocked) or 1 (unblocked).

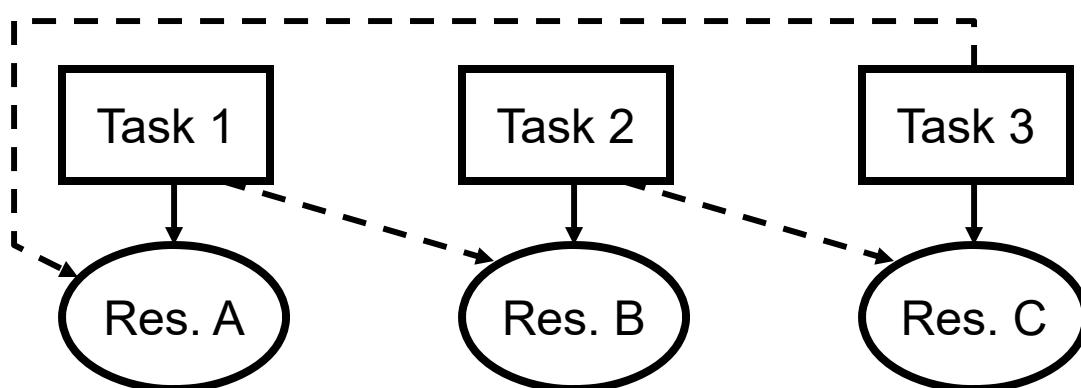
# Synchronization Using Message Passing

---

- Tasks (e.g., one **producer** task and one **consumer** task) can synchronize themselves to exchange data safely.
- Messages can be exchanged as follows:
  - (1) The **producer** task **sends** a message to the consumer task once newly produced data becomes available.
  - (2) The **consumer** task calls a **block-and-receive** function. If the message is immediately available, then the consumer task can proceed and process the new data (which may be enclosed or pointed to in the message). If the message has not yet been received, then the consumer task is switched off the CPU and is added to a queue of blocked tasks. When the next message arrives for the blocked consumer task, the consumer task is moved to the ready-to-run queue.

# Deadlock

- Multitasking systems can be vulnerable to a condition called “**deadlock**”, where two or more tasks become permanently blocked from running because they are waiting for resources, which are held by other tasks, to become available.
- Coffman’s four conditions for deadlock to be possible:
  1. Resources can be assigned exclusively to tasks.
  2. One task can request two or more resources.
  3. A task cannot be forced to release a resource.
  4. Two or more tasks form a circular chain where each task is waiting for a resource that is held by the next task in the chain.



# Watchdog Timers

---

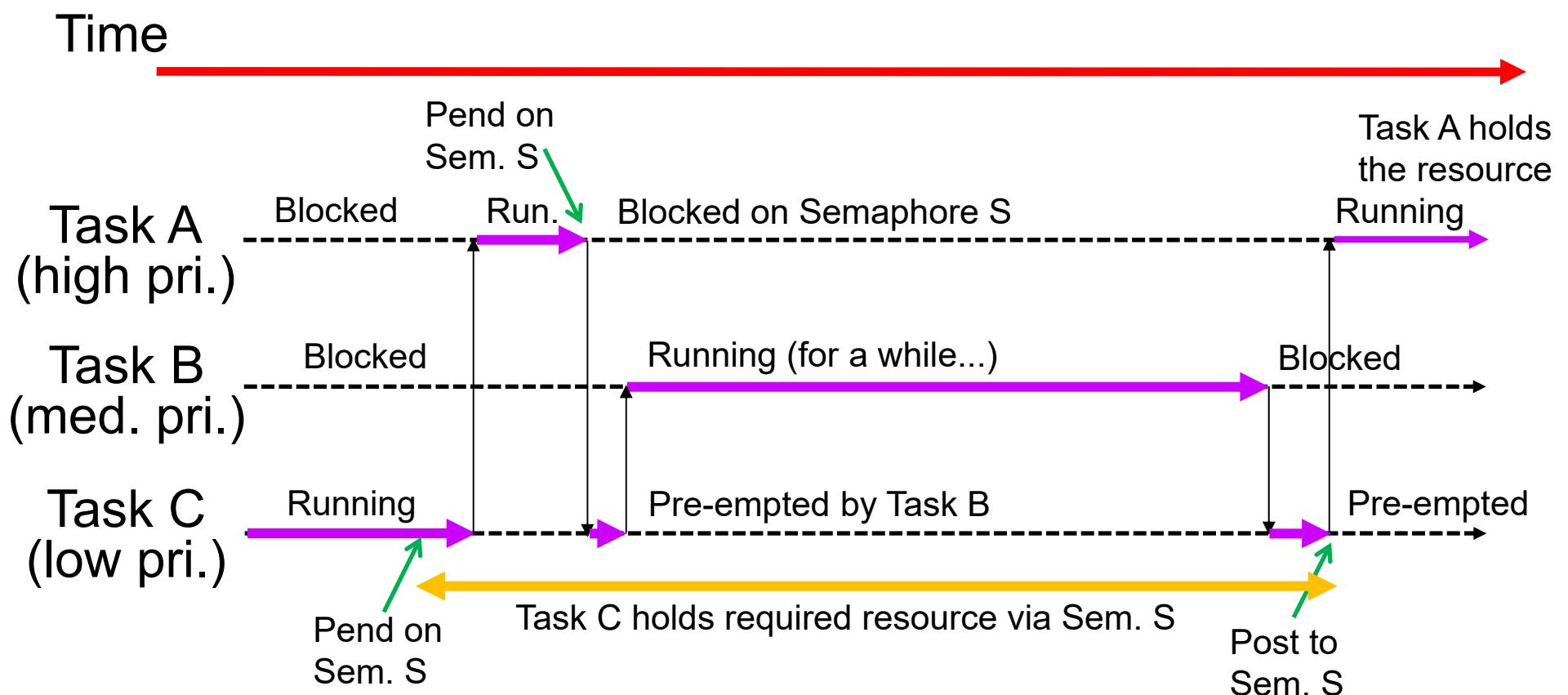
- A watchdog timer provides a mechanism that allows an embedded system to, in many cases, recover from a catastrophic system failure, such as a memory leak, multitasking deadlock, or radiation-induced flipped bits.
- A ***watchdog timer*** is a hardware timer that counts down from some starting value at some frequency. If the count ever reaches zero, a hardware reset signal is activated which reboots the system. Rebooting will clear up many errors.
- The software system includes a ***watchdog timer service routine*** that is scheduled to run sufficiently frequently (e.g., a low priority task at a slow repetition rate). This routine re-loads the watchdog timer with the starting value each time that it executes. In a healthy embedded system, the watchdog timer value will never be allowed to reach zero. The H/W reset will only happen after serious system failure.

# Priority Inversion (1)

---

- It is possible for a low-priority task to begin holding exclusive control of a resource (e.g., using a semaphore), and then later on a higher-priority task may start executing and then attempt (but fail) to gain access to the same resource that is currently controlled by the low-priority task. Execution may then return to the low-priority task when the high-priority task is blocked.
- This situation is called ***priority inversion*** because the low-priority task has blocked the execution of a higher priority task.
- The situation can be made even worse if a middle-priority task (possibly doing unrelated work) pre-empts the low-priority task, thereby blocking both the low-priority task and the high-priority task for some possibly long period of time.

# Priority Inversion (2)



Note: Task A is prevented from executing by Task C (which controls access to the required resource using Sem. S). Even if Task C has a short critical section, Task B may pre-empt Task C and thus add further delay to Task A.

# Solutions to the Priority Inversion Problem

---

- It is complicated to find safe ways of forcing a low-priority task to give up exclusive access to a resource in a critical section. The task may already have made partial changes.
- A common strategy for solving the priority inversion problem is to *temporarily raise the priority of the low-priority task to an even higher priority than that of any high-priority task that gets blocked on the semaphore* that protects the critical section. This allows the low-priority task to finish using the critical section quickly so that the high-priority task will not be delayed too long. Medium-priority tasks cannot interfere.
- MicroC/OS-II provides *mutual exclusion semaphores (mutexes)* which are binary semaphores that allow a high priority level to be reserved for automatic use by a low-priority task that happens to block a higher-priority task.

# Disadvantages of Multitasking

---

- Time is wasted performing the context switches between running tasks.
- Memory space is required to store the kernel code.
- Critical sections and shared resources must be protected. Synchronization mechanisms must be provided.
- New problems, such as deadlock and priority inversion, can occur when multiple tasks require exclusive control of multiple shared resources. Precautions must be taken to avoid / detect such problems.
- Debugging multitasking software can be tricky because it may be difficult to understand and/or recreate the complex interactions between the tasks that led to a problem.

# Figures of Merit for Real-Time Kernels

---

The properties and quality of a real-time kernel are measured using many different *figures of merit* (i.e. desirable properties):

**Features:** A kernel that has more features should make it easier and cheaper to implement the new system because the amount of required new software will be minimized.

**Code size and Scalability:** Many microcomputer systems provide a very limited amount of memory, hence a more compact kernel and/or a more customizable kernel is better.

**Real-time response:** The maximum amount of time that will elapse between the instant when an interrupt occurs and the time that the corresponding interrupt service routine starts executing. Faster response is better.

# Figures of Merit (cont'd)

---

**Time for a context switch:** The time for one task to be suspended, and for execution to begin in another task or interrupt service routine. Faster is better.

**Determinism:** The predictability of the time between when the system receives an interrupt or receives a command and the time when external actions and/or internal updates actually begin. Greater determinism/predictability is better.

**Certification:** Has the quality of the kernel been certified by a reputable organization according to some recognized standard? Certification is better.

**Portability:** The ease with which the kernel, and the overlying software, can be modified so as to function properly on a different microcomputer.

# **Introduction to the MicroC/OS RTOS for the NetBurner MOD54415**

# What is MicroC/OS ?

---

MicroC/OS is a real-time operating system (RTOS), also called a “kernel”, for microcomputers, which has the following features:

- *pre-emptive multitasking* and *unique priorities per task*
- many typical *kernel services* including task management, time delays, semaphores, mutual exclusion semaphores, event flags, mailboxes, message queues, etc.
- *portability*, because it is written in standard ANSI C
- *scalability*, because kernel features can easily be included or left out using conditional compilation
- *determinism*, which means that the response times for most kernel services are predictable and do not depend on the number of tasks that are currently active

# A Brief History of MicroC/OS

---

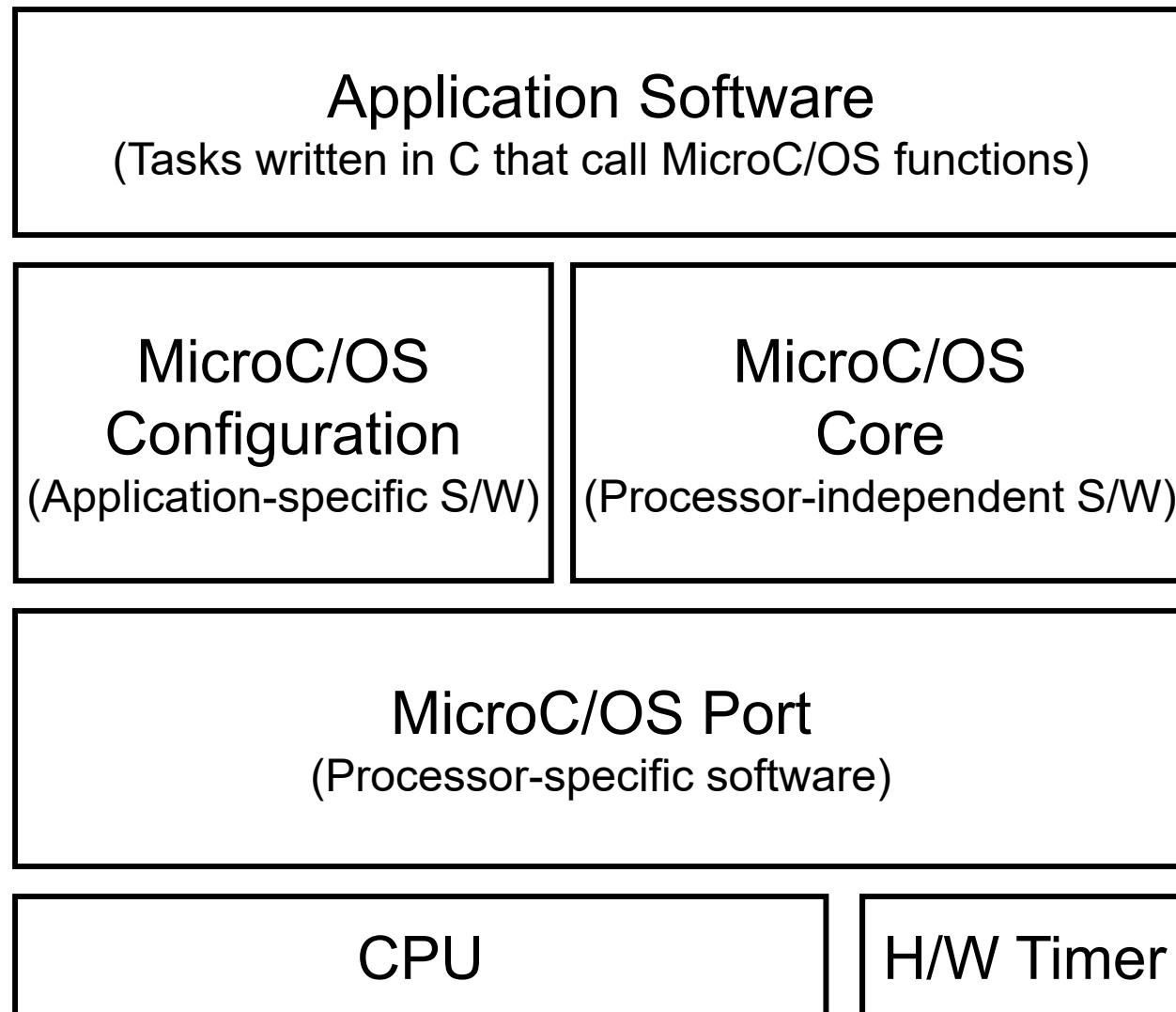
- Designed by Jean J. Labrosse in the early 1990s.
- The first version, called  $\mu$ COS or MicroC/OS, was released into the public domain in the summer of 1992.
- A new version was released along with a book titled  *$\mu$ C/OS, The Real-Time Kernel*, (CMP Books, 1992).
- Enhanced version released along with a book titled *MicroC/OS-II, The Real-Time Kernel*, 2nd ed., (CMP Books, 2002).
- MicroC/OS-II and -III are distributed by Micrium Inc. MicroC/OS-II can be used royalty-free by educational institutions for teaching & research. *Note:* A license is required to include the kernels in commercial products.
- The NetBurner boards that we use in ECE 315 use a modified version of the original royalty-free MicroC/OS.

# MicroC/OS Variations

---

- Free “ports” (that is, software implementations that have been modified to work on a different computer) of MicroC/OS are available for a wide variety of different CPUs/microcomputers and IDEs/compilers.
- The version of MicroC/OS that we have for the NetBurner MOD54415 lab microcomputer is slightly different from the version described in the 2002 book by Jean Labrosse.
- The process of editing source C/C++ code, compiling it, linking together a system “build”, downloading the build to flash memory on the MOD54415, and then executing the new build has been automated within the Eclipse IDE. Such automation increases productivity.
- The function call sequence for initializing the MicroC/OS environment has been simplified, and some MicroC/OS function calls have been dropped and new ones added.

# System Architecture Using MicroC/OS



# The MicroC/OS Core

---

- The core of MicroC/OS provides services that can be invoked by user programs by calling functions in C.
- User software is assumed to be partitioned into 1 to 63 tasks, which are written in C. Each task is usually structured in the form of an infinite loop that blocks on system calls.
- Each task is associated with:
  1. A “task priority”, from 1 to 63, where 1 is the highest priority and 63 is the lowest priority  
63 (the lowest priority) is reserved for the **IdleTask**.  
38, 39, 40, 45, 50, 51, 52 & 63 are reserved in our NetBurner implementation of MicroC/OS. *User tasks should only use priorities in the range 53 to 62.*
  2. Program code that defines the task behaviour.
  3. A private stack region in memory.
  4. An optional private task-specific data structure, which can store a name, floating-point register contents, etc.

# MicroC/OS Core (cont'd)

---

- MicroC/OS and MicroC/OS-II always create a special **IdleTask**, which is run on the CPU when there is no other task that is on the ready-to-run queue.
- MicroC/OS-II optionally creates a **StatisticsTask**, which is capable of gathering useful system performance data.
- The NetBurner version of MicroC/OS does not have the **StatisticsTask**, but similar capabilities are provided by the OS functions **OSDumpTasks** and **OSDumpTCPStacks**.
- **At any given time, the task with the lowest priority that is also ready-to-run will be using the CPU.** If no user task is currently ready-to-run, then either the **IdleTask** (or the **StatisticsTask**, if present) will be running on the CPU.
- In MicroC/OS, the first user task “UserMain” (1) initializes the kernel, (2) initializes the TCP/IP stack, (3) starts the HTTP server, (4) creates an initial set of user tasks, and (5) carries on executing, usually at a low frequency. UserMain might be used later to shut down the system.

# Reserved NetBurner MicroC/OS Priorities

---

- The NetBurner version of a networked MicroC/OS environment reserves priority 63 for the IdleTask, and 0 for the highest priority task (reserved and not available).
- The following additional priorities are also reserved:

```
#define MAIN_PRIO (50) // for the UserMain task  
#define HTTP_PRIO (45) // for web server task  
#define TCP_PRIO (40) // for TCP layer task  
#define IP_PRIO (39) // for IP layer task  
#define ETHER_SEND_PRIO (38) // for UDP sends
```

- User tasks must use other priorities. In the lab system, priority 51 is used by the LED task, and 52 controls the seven-segment display. *Priorities 53 to 62 are available.*
- See C:\Nburn\include\constants.h for priority definitions.

# Application Software for MicroC/OS

---

- The application software gains access to MicroC/OS by including specific global include files.
- It is useful to define symbols for various constant values so that meaningful names, not obscure numbers, can be used later.
- The hidden main() procedure loads up the context switch TRAP vector, initializes MicroC/OS system data structures, creates the default **UserMain()** start-up task, and starts multitasking.
- Code before the UserMain task allocates storage for the stacks of any other tasks, and space for any other “global” data structures (e.g., variables, pointers, semaphores, message queues, etc.) that will be required. These objects will be accessible by all tasks. Code must be provided for all of the user tasks, which are not executing at first.
- The UserMain task finishes **system initialization**, finishes **initializing the global data structures**, and “creates” **the other tasks** in the system. UserMain can carry on executing at a low scheduling frequency. It might be used later on to shut down the system in a safe way.

# “Hello World!” Example, Part 1

---

```
#include "predef.h"          // constants used during debugging
#include <stdio.h>           // standard input/output functions
#include <ctype.h>            // useful type definitions
#include <startnet.h>          // TCP_IP and HTTP start-up functions
#include <autoupdate.h>        // for downloading system builds to flash
#include <dhcpclient.h>        // create DHCP client to get IP address
#include <smarttrap.h>         // to allow smart traps
#include <taskmon.h>           // to access task monitor features
#include <NetworkDebug.h>       // to use networked debugger

extern "C" // required for plain C function calls from C++ code
{
    void UserMain(void *Pd); // declare main task function prototype
}

// define application name for NBEclipse
const char *AppName="Hello_World_application";
```

# “Hello World!” Example, Part 2

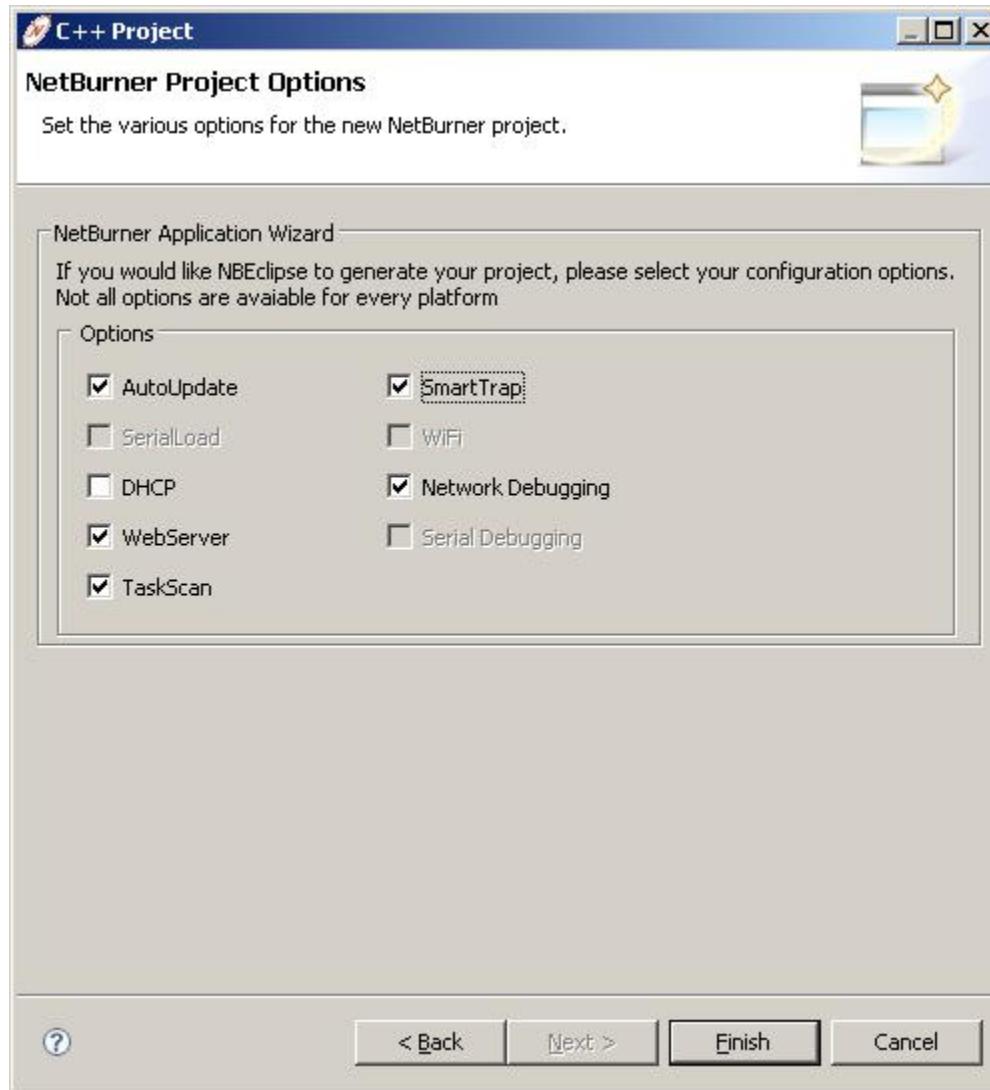
---

```
void UserMain(void *Pd) {                                // No input argument here
    InitializeStack();                                 // init the TCP_IP stack
    if (EthernetIP == 0) GetDHCPAddress();
        // get IP address if not already allocated statically
    OSChangePrio(MAIN_PRIO);                          // initialize task priority
    EnableAutoUpdate();                               // enable auto update to flash
    StartHTTP();                                    // start web server
    EnableTaskMonitor();                            // enable task scan monitor

#ifndef _DEBUG
    EnableSmartTraps();                           // use smart traps by default, or
#endif
#ifdef _DEBUG
    InitializeNetworkGDB_and_Wait();   // use networked GDB debugger
#endif

while (1) {                                              // loop forever
    iprintf("Hello World!\n"); // a version of printf
    OSTimeDly(TICKS_PER_SECOND * 1); // wait 1s per iteration
}
}
```

# NBEclipse Application Wizard Settings



# MicroC/OS System Files

---

- ucos.c** Generic MicroC/OS function definitions
- ucosmain.c** Start-up and debug functions, and the stack definition for the **IdleTask( )**
- ucosmcfc.c** ColdFire-specific MicroC/OS C functions, such as OSTaskCreate, Task Control Block init, and task stack checking code.
- ucosmcfa.s** ColdFire-specific functions in assembly language for task switching and the timer.
- main.c** Contains **main( )** function that does system initialization and creates **UserMain( )**.

# Creating MicroC/OS Tasks (1)

---

- The NetBurner port creates the `UserMain()` task.
- Additional application tasks can be “created” within `UserMain()` using the `OSTaskCreate` function.
- The code for the application tasks must be present in the source file before `UserMain()` so that it is compiled and ready to execute.

```
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <ucos.h>
#include <dhcpclient.h>

extern "C"      // Allow old-style C function prototypes
{
    void MyTask( void *Pd );
    void UserMain( void *Pd );
}
```

# Creating MicroC/OS Tasks (2)

---

```
asm( " .align 4 " );      // Go to next 4-byte-aligned address to
                           // ensure that the stack is 4-byte-aligned

// Global data structure definition for the new task's stack
DWORD MyTaskStack[USER_TASK_STK_SIZE]
    __attribute__( ( aligned( 4 ) ) );

void MyTask( void *Pdata )
{
    WORD data = *(WORD *)Pdata; // cast the passed parameter

    iprintf( "Data passed to MyTask(): %d\r\n", data );

    while ( 1 )
    {
        iprintf( "      Message from MyTask()\r\n" );
        OSTimeDly( TICKS_PER_SECOND * 1 ); // one second
    }

} // end of MyTask
```

# Creating MicroC/OS Tasks (3)

---

```
void UserMain( void *Pd )
{
    InitializeStack();
    OSChangePrio( MAIN_PRIO );
    if ( EthernetIP == 0 ) GetDHCPAddress();
    EnableAutoUpdate();

    WORD MyTaskData = 1234;

    if ( OSTaskCreate ( MyTask, (void *) &MyTaskData,
                        (void *) &MyTaskStack[USER_TASK_STK_SIZE],
                        (void *) MyTaskStack, MAIN_PRIO+1 ) != OS_NO_ERR )
    {
        iprintf( "**** Error creating MyTask\r\n" );
    }

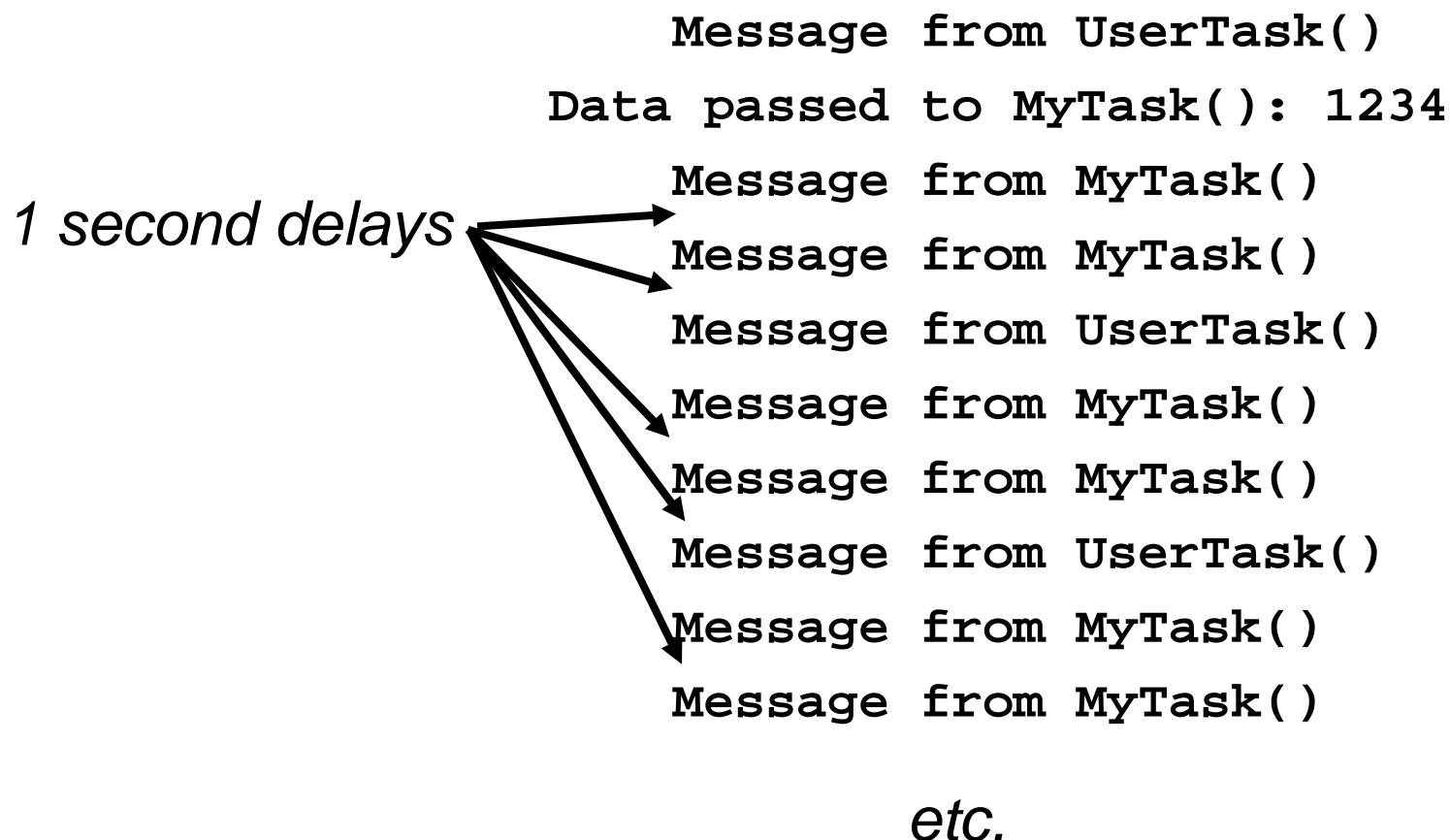
    while ( 1 ) // Could have !shutdown_system condition here
    {
        iprintf( "      Message from UserTask()\r\n" );
        OSTimeDly( TICKS_PER_SECOND * 2 ); // two seconds
    }

    // Could have system shut-down commands here
} // end of UserMain
```

# Creating MicroC/OS Tasks (4)

---

Resulting text output:



# MicroC/OS functions that can cause task blocking

---

*MicroC/OS calls:*

**OSTimeDly()**

**OSSemPend()**

**OSQPend()**

**OSMboxPend()**

*I/O System Calls:*

**select( )**

**read( )**

**write( )**

**gets( )**

**getchar( )**

**fgets( )**

*Network Calls:*

**accept( )**

**Also, creation  
of UDP packet**

## Functions that can cause task unblocking

*MicroC/OS calls:*

**OSTimeTick()** – called by the timer ISR

**OSSemPost()** – called by the running task or an ISR

**OSQPost()** – called by the running task or an ISR

**OSMboxPost()** -- called by the running task or an ISR

# Predefined Data Types in MicroC/OS

---

*Data types available in generic MicroC/OS and MicroC/OS-II:*

OS_TCB	Data type for Task Control Blocks
OS_EVENT	Data type for Event Control Blocks
OS_SEM_DATA	Data type used to return semaphore data

*Other symbolic type definitions in the NetBurner ports:*

BYTE	ColdFire 8-bit byte
WORD	ColdFire 16-bit word
DWORD	ColdFire 32-bit longword. Preferred storage type for fast access. Should be 4-byte aligned.
BOOL	Boolean
OS_SEM	Semaphore data structure
OS_FIFO	First-In First-Out queue data structure
OS_MBOX	Mailbox data structure
UINT8	Unsigned 8-bit integer
UINT16	Unsigned 16-bit integer
UINT32	Unsigned 32-bit integer

See C:\Nburn\includes\basictypes.h for all of the basic types.

# Event Control Blocks (ECBs)

- MicroC/OS-II uses Event Control Blocks to record information that is associated with objects that possibly have blocked tasks.
    - e.g. semaphores, mutual exclusion semaphores, mailboxes, queues
  - ECBs are not used in NetBurner MicroC/OS
  - User tasks do not normally look inside these ECBs, but tasks often need to handle pointers to ECBs when they call MicroC/OS-II functions.
  - Field .OSEventTbl[] contains 64 bit flags that record which tasks are currently blocked while waiting on the associated object.
  - Field .OSEventCnt gives the total number of tasks that are blocked waiting on the object.

# Function Return Codes in Netburner MicroC/OS

---

All of these symbolically-defined codes are of type BYTE.

OS_NO_ERR	No error occurred.
OS_TIMEOUT	Timeout expired.
OS_MBOX_FULL	Mailbox is full.
OS_Q_FULL	Queue is full.
OS_PRIO_EXIST	Task ID already in use.
OS_SEM_ERR	Invalid priority supplied.
OS_SEM_OVR	Over max. semaphore count.
OS_CRIT_ERR	Critical OS error.
OS_NO_MORE_TCB	No more TCBs available.

# Task Creation

---

```
BYTE OSTaskCreate( void ( * Ptask )( void * Ptakfunc ), void * Pdata,  
                  void * Pstacktop, void * Pstackbot, BYTE priority );
```

- Creates a new user task.
- "Ptask" points to the first instruction in the task.  
*Note:* This argument is just the name of the task function.
- "Pdata" points to an optional task-specific input argument.
- "Pstacktop" points to the top of the task's private stack.  
*Note:* The stack must be "4-byte-aligned" in memory.
- "Pstackbot" points to the bottom of the task's private stack.
- "priority" is a unique identifier from 0 to 63.
- Return code is OS\_NO\_ERR if task created successfully.
- Return code is OS\_PRIO\_EXIST if priority is already in use.

# Example of Task Creation

---

```
asm( " .align 4 " );      // go to next 4-byte aligned address

// Reserve memory for the task MyTask's stack
DWORD Pstacktop[USER_TASK_STK_SIZE]
    __attribute__( ( aligned( 4 ) ) );

void MyTask ( void * Pdata )
{
    // task code goes here
}

void UserMain ( void * Pd )
{
    // beginning part of task UserMain

    if (OSTaskCreate( MyTask, (void *) Pdata,
                      (void *) &Pstacktop[USER_TASK_STK_SIZE],
                      (void *) Pstacktop, MyTaskPriority )
        != OS_NO_ERR) {
        // handle task creation error
    }
    // rest of UserMain
}
```

# Task Deletion

---

```
void OSTaskDelete( void );
```

- Moves the calling task into the dormant state.
- A deleted task can be re-created by another task by calling OSTaskCreate.
- The code for the task, and usually also the task's stack area in memory, are left alone.

# Changing a Task's Priority

---

```
BYTE OSChangePrio( BYTE newpriority );
```

- Used to change the priority of the calling task.
- Can't be called directly by other tasks.
- Priority "newpriority" must not already be in use.
- Return code is OS\_NO\_ERR if successful.
- Return code is OS\_PRIO\_EXIST if newpriority is already assigned to an existing task.

# Task Delay Management Functions

---

**void OSTimeDIy( WORD ticks )**

- Block the calling task by "ticks" timer increments.
- The highest available ready-to-run task starts executing.
- The tick unit is defined from TICKS\_PER\_SECOND.
- The ECE 315 lab system has 200 ticks per sec.
- The largest possible delay is 65,535 ticks.
- There is no return code.

**void OSChangeTaskDIy( WORD priority, WORD newticks )**

- Cancel the time delay for another task, which has priority "priority" and that is currently waiting, and change it to the "newtick" delay.
- There is no return code.

# Semaphore Management Functions

---

BYTE **OSSemInit( OS\_SEM \* Psem, long count )**

- Initializes counting semaphore pointed to by "Psem".
- The initial long count value is set to "count".
- Returns pointer to semaphore data structure.
- Return code is **OS\_NO\_ERR** if successful.
- Return code is **OS\_SEM\_ERR** if the requested count is less than zero.

```
OS_SEM MySemaphore;      // create semaphore area  
  
OSSemInit( &MySemaphore, 0 ); // initialize sem
```

# Semaphore Management Functions

---

BYTE **OSSemPend( OS\_SEM \* Psem, WORD timeout );**

- Block and wait on the semaphore pointed to by "Psem".
- If the semaphore count is greater than zero, then the count is decremented by one and the task proceeds.
- If the semaphore count is zero or less, then the count is decremented and the task is moved to a blocked queue associated with "Psem", and the highest ready-to-run task begins executing.
- "timeout" sets the maximum block time in ticks.
- if timeout == 0, then an infinite wait is assumed.
- Return code is OS\_NO\_ERR if successfully unblocked before the timeout expired; otherwise get OS\_TIMEOUT.

```
OS_SEM MySemaphore;           // before all tasks
OSSemInit( &MySemaphore, 0 ); // in UserMain task

OSSemPend( &MySemaphore, 100 ); // in another task
```

# Semaphore Management Functions

---

```
BYTE OSSemPost( OS_SEM * Psem );
```

- Signal (increment) a semaphore pointed to by "Psem".
- The highest priority task waiting on semaphore "Psem" (if any) is moved to the ready-to-run queue.
- Return code is OS\_NO\_ERR if successful.
- Return code is OS\_SEM\_OVF if the value of the semaphore overflowed as a result of the post.

```
OS_SEM MySemaphore; // before all tasks
```

```
OSSemInit( &MySemaphore, 0 ); // in UserMain task
```

```
OSSemPost( &MySemaphore ); // in another task  
// or in an ISR
```

# Message Queue Management Functions

---

```
BYTE OSQInit( OS_Q * Pqueue, void **Pqstart, BYTE QSIZE );
```

- Initialize a circular queue of pointers to data elements.
- "Pqueue" points to an OS\_Q object that keeps track of the queue of tasks that are ever blocked on the queue.
- "Pqstart" is a pointer that points to an array of pointers to (void \*)'s. Each (void \*) points to a data element, such as a character array.
- Must previously have allocated memory for the array of (void \*)'s by declaring the array "void \* Pqstart [ QSIZE ];"
- "QSIZE" is the maximum number of pointers in the queue.

```
OS_Q MyQueue;                                // before all tasks
void * MyQueueArray[ QSIZE ];  
  
OSQInit( &MyQueue, MyQueueArray, QSIZE ); // UserMain
```

# Message Queue Management Functions

---

```
void * OSQPend( OS_Q *Pqueue, WORD timeout, BYTE * Perr );
```

- Block and wait on the queue pointed to by "Pqueue".
- "timeout" gives a maximum wait time in ticks.
- A "timeout" of 0 means that an infinite wait is possible.
- Return code pointed to by "Perr" is either OS\_NO\_ERR or OS\_TIMEOUT.
- The return value of the function is a pointer to the next data element at the head of the queue.
- If the queue was empty before the function call, then the return value is the predefined C constant NULL.

# Message Queue Management Functions

---

```
BYTE OSQPost( OS_Q * Pqueue, void * Pmsg );
```

- Deposit the message pointer "Pmsg" into the message queue pointed to by "Pqueue".
- OS\_NO\_ERR is returned if the message got posted successfully to the queue.
- Otherwise, the return code has value OS\_Q\_FULL if the message queue is full and it has no room to hold the new message pointer.

```
OS_Q MyQueue;           // in UserMain task
void *MyQueueArray[ QSIZE ];
OSQInit( &MyQueue, MyQueueArray, QSIZE );

char *Pmsg;             // in another task
Pmsg = "Hello World!";
OSQPost( &MyQueue, (void *) Pmsg );
```

# Temporarily Disabling Hardware Interrupts

---

`void USER_ENTER_CRITICAL( );`

- Sets the interrupt mask bits in the CPU status register to 111 to block all maskable interrupts. The previous mask bit pattern is saved on the supervisor stack.
- All MicroC/OS functionality is disabled.
- Each call to `USER_ENTER_CRITICAL` must be matched with exactly one subsequent call to `USER_EXIT_CRITICAL`
- This macro is used at the start of critical sections within many other MicroC/OS functions (e.g., `OSSemInit`, `OSSemPend`, `OSSemPend`, the message queue functions, etc.).

`void USER_EXIT_CRITICAL();`

- Restores the interrupt mask bits to what they were immediately prior to the previous call to `USER_ENTER_CRITICAL`.
- This macro is used within many MicroC/OS functions to end critical sections that were begun with `USER_ENTER_CRITICAL`.

# Temporarily Disabling Multitasking

---

`void OSLock( void );`

- Prevents task context switches and pre-emption according to task priority, but does not otherwise affect interrupt handling or access to MicroC/OS functions.
- Each call to OSLock must be matched with exactly one subsequent call to OSUnlock.

`void OSUnlock( void );`

- Restores normal multitasking.
- The call to OSUnlock will immediately trigger a context switch if a ready-to-run task other than the calling task now has the highest priority in the multitasking environment.

# MicroC/OS Configuration

---

- MicroC/OS-II can be configured readily at compile time using "switches" in files “OS\_CFG.h” and “includes.h”.
- Unnecessary object code for unused functions can be omitted from the system to minimize memory space.
- The NetBurner version of MicroC/OS has only three switches in "nburn/includes/predef.h"

```
#define UCOS_STACKCHECK 1      /* provide OSDumpTCBTasks() */
                                /* and OSDumpTasks() */
#define UCOS_TASKLIST    1      /* provide ShowTaskList() */
#define BUFFER_DIAG      1      /* provide ShowBuffers(),
                                /* GetBufferX and FreeBufferX */
                                /* see C:\Nbun\includes\buffers.h */
```

# Interrupt Handling in MicroC/OS

---

*Interrupt Service Routines* (ISRs) can be coded in assembly language using the following pattern:

- 1) Save all CPU registers on the interrupted task's stack
- 2) Increment the system variable OSIntNesting
- 3) If OSIntNesting == 1 then OSTCBCur->OSTCBStkPtr = SP
- 4) Clear the interrupting device
- 5) (optional) Re-enable interrupts to allow IRQ nesting
- 6) Execute the code that services the interrupt request. This code should execute quickly, and must not be blocked.
- 7) Call OSIntExit() to check for possible task switch after RTE
- 8) Restore all CPU registers
- 9) Execute an RTE instruction

# Interrupt Handling using the INTERRUPT Macro

---

## **INTERRUPT( ISR\_name, NewSRValue )**

- INTERRUPT wraps an Interrupt Service Routine (ISR) with the necessary code for saving the CPU registers at the start of the ISR, and then restoring the CPU registers at the end of the ISR.
- At the end of the ISR, the possibility of a context switch is also checked by MicroC/OS.
- The UserMain task must correctly load the corresponding exception vector entry with the address of the first instruction of the ISR.

Example:

```
// The vector value &eTPU_ISR must be written during initialization
// into the eTPU vector location in the Exception Vector Table.
INTERRUPT( eTPU_ISR, 0x0500 )
{
    // ISR code for the eTPU goes here
    // All interrupts at level 5 and below are masked out in this ISR
}
```

# ISR Execution Can Cause a Task Switch

---

- When an ISR executes, it can unblock tasks that were waiting on flags, mailboxes, queues, semaphores, etc.
- The originally interrupted task may now have a lower priority than one of the unblocked ready-to-run tasks.
- The function OSIntExit, which is called by code inserted by the INTERRUPT macro, checks for this possibility.
- OSIntExit either reloads and schedules the interrupted task (if it is still the highest priority task that can run), or it causes a context switch to the new highest priority ready-to-run task.

# **Serial Interfacing Using RS-232C, Ethernet, and SPI**

*Reference:*

- Freescale Semiconductor, Inc., “MCF5235 Reference Manual”, Doc. No. MCF5235RM, Rev. 2, July 2006.

Figures and tables from the above documents have been included in these course notes for educational purposes in ECE 315 only. The original documentation should be consulted to ensure accuracy.

Freescale™ and ColdFire® are registered trademarks of NXP Semiconductors N.V.

# Digital Signaling Modes

---

- ***Unbalanced or Single-ended Mode***
  - One wire carries the signal voltage that is defined with respect to a ground (0 volt) reference potential on a second wire or the shield.
  - *Advantages:* One ground wire can be shared for multiple signals.
  - *Disadvantages:* Noise that is added to either the signal or the ground reference (e.g., ground loop noise) can cause bit errors at the receiver.
- ***Balanced or Differential Mode***
  - Two wires carry the signal  $V^+$  (e.g., 0010) and the complemented signal  $V^-$  (e.g., 1101). A reference or ground wire is not required.
  - The receiver subtracts the two signals (e.g.,  $V^+ - V^-$ ), to recover the single-ended digital data waveform (roughly  $2 \times V^+$ ).
  - If  $V^+$  and  $V^-$  are corrupted by common mode additive noise, then the noise cancels out:  $(V^+ + V^{\text{noise}}) - (V^- + V^{\text{noise}}) = V^+ - V^- = 2 V^+$
  - Typical sources of additive noise: nearby signals, electric & magnetic fields produced by power supplies, motors, transformers, etc.
  - The noise rejection is even more effective if the wires are twisted.

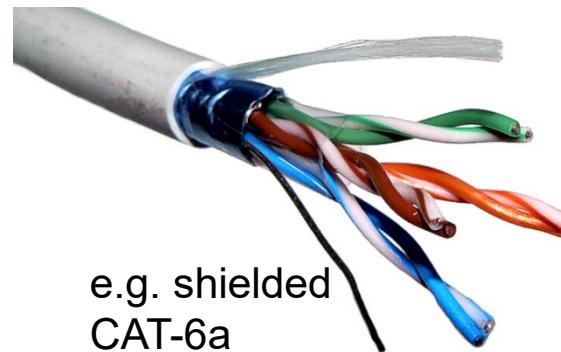
# Some Common Twisted Pair Cable Types

---



e.g. CAT-5e

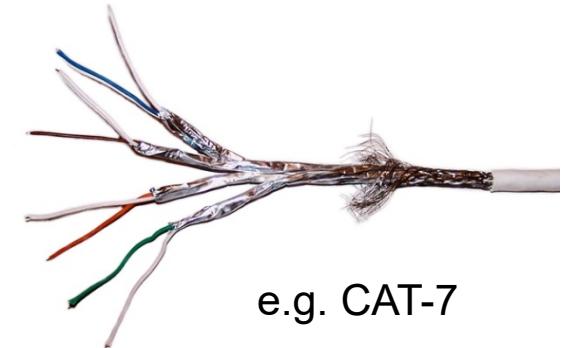
Unshielded  
Twisted  
Pair (UTP)



e.g. shielded  
CAT-6a

Foiled/Shielded  
Twisted Pair  
(**F/UTP**)

- **Cable has foil shield.**
- **TPs are unshielded.**



e.g. CAT-7

Fully Shielded  
Twisted Pair  
(**S/FTP**)

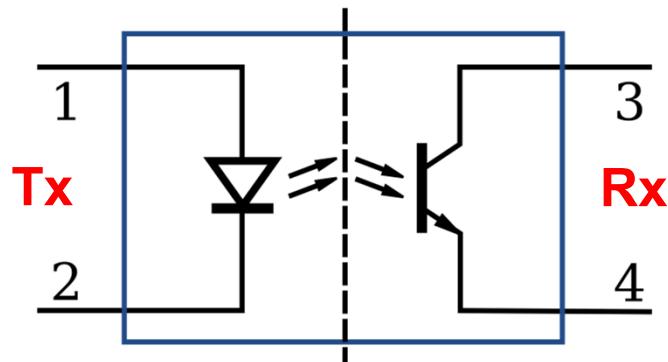
- **Cable is shielded.**
- **Foil for each TP.**

*Note:* The effectiveness of the noise rejection is increased if the twist rate per unit distance is different for each twisted pair (TP).

# Two Common Isolation Techniques

## Opto-isolator

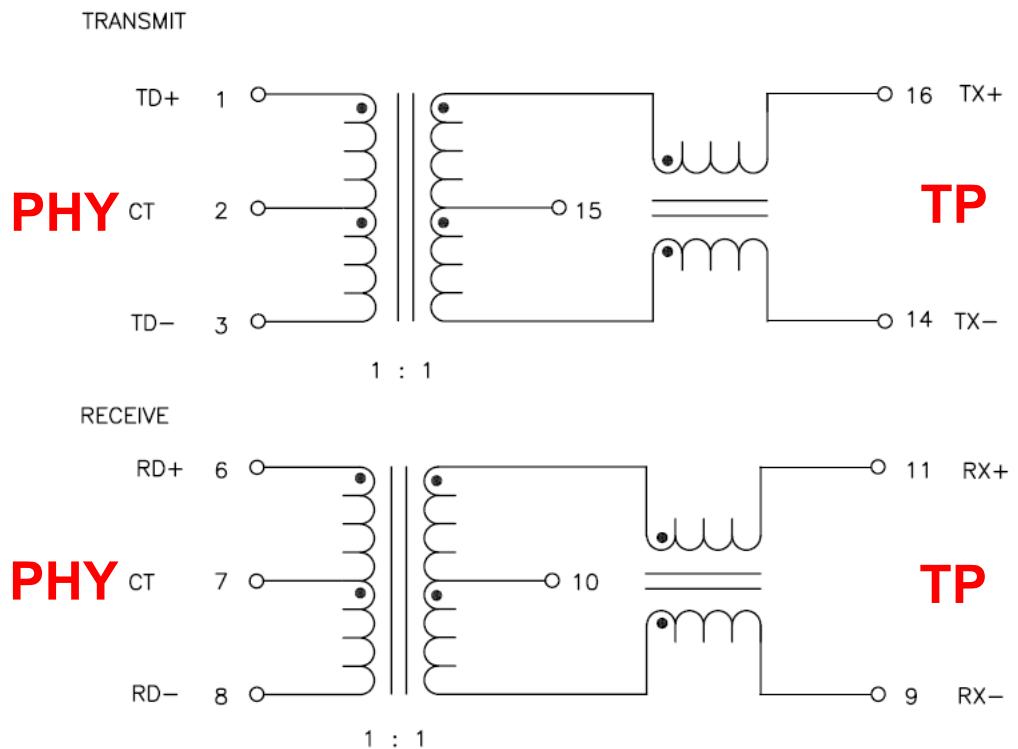
- Protects receiver from high voltage signals and noise
- Ground potentials can differ
- Relatively slow
- Audio applications
- Biomedical sensors



Courtesy Wikimedia.org

## Pulse Isolation Transformer

- Blocks DC offsets in signals
- Widely used for LAN signals
- E.g., 10/100Base-TX Ethernet



From Pulse Engineering H1102FNL Datasheet

# The RS-232C Serial Transmission Standard

---

- RS-232C was intended to be a ***low-cost asynchronous serial interface standard*** for connecting large central computers and remote terminal equipment to modems for the purposes of setting up data communication links over the public switched telephone network (PSTN).
- A minimal RS-232C connection requires only ***three wires***: (1) transmit data, (2) receive data, and a (3) ground return.
- RS-232C assumes that the data will be sent as ***7-bit or 8-bit character codes***, plus other “***overhead bits***” that provide timing, minimum character spacing, and (optionally) single-bit error detection using parity bits.
- The data signals are ***single-ended, unmodulated signals***.

# A Brief History of RS-232C

---

- Recommended Standard 232 Version C (RS-232C) was published by the Electronic Industry Association (EIA) in the U.S.A. in 1969 for use in modem connections. It has since become a ***de facto*** standard for connecting computer peripheral equipment.
- The EIA published a slightly revised specification, called “EIA 232D”, in 1987.
- The EIA, together with the Telecommunications Industry Association (TIA), produced yet another updated standard “EIA/TIA-232-E” in 1991.
- Most engineers continue to use the term “RS-232”.

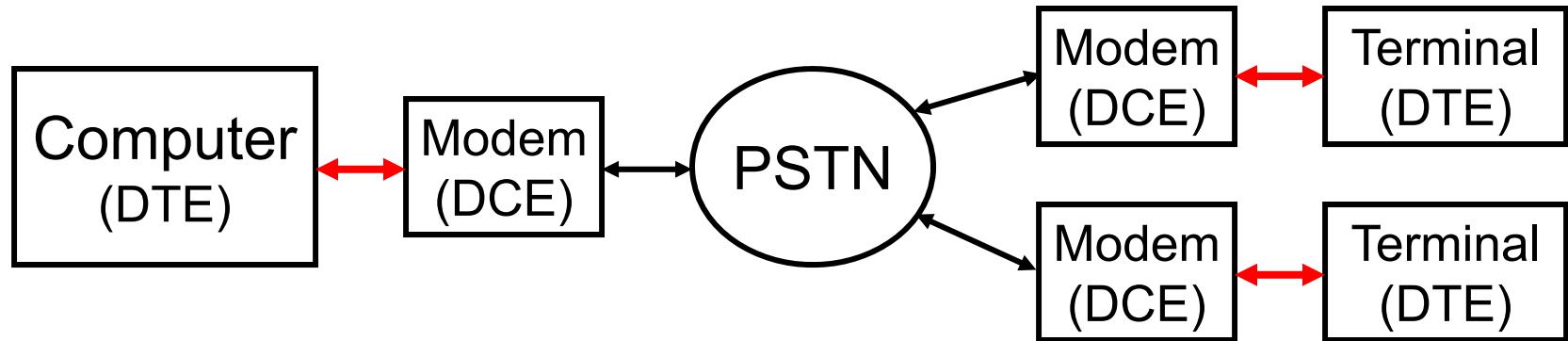
# Two Successor Standards to RS-232C

---

- The official RS-232C specification has serious limitations:
  - maximum transmission distance is only 15 m (50 ft.)
  - maximum bit/baud rate is 20,000 bits per second.  
(up to 200 Kbps is possible, but outside of the spec.)
- RS-422 is a balanced (differential mode) serial transmission standard with higher maximum bit rates.
  - transmits +/- 6 V down to +/- 2 V differential outputs
  - input signals can be attenuated down to +/- 0.2 V
  - differential signals reject common mode additive noise
  - transmits up to 90 Kbps over up to 1.2 Km (3,900 ft)
- RS-485 is a balanced (differential mode) serial transmission standard with higher maximum bit rates. E.g. 2 Mb/s @ 50 m
  - Supports multi-drop busses using tri-stateable drivers

# Intended RS-232C Connections

---



## ***Data Terminal Equipment (DTE)***

- Computers, terminals

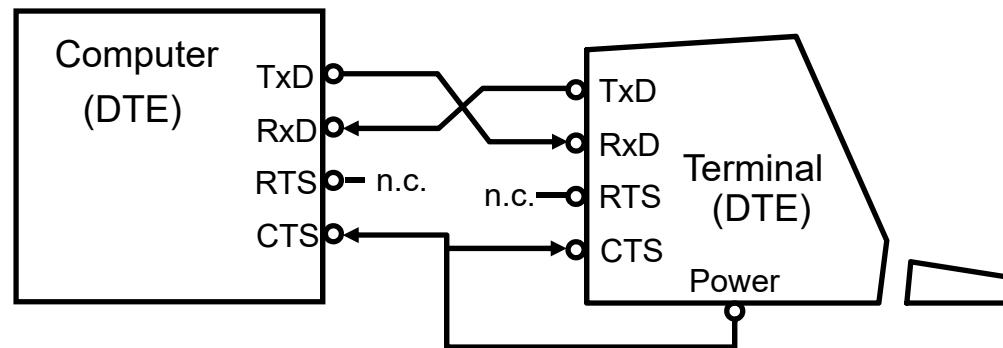
## ***Data Communications Equipment (DCE)***

- Modem for interfacing RS-232C binary signals to analog signals that lie within the standard 100 Hz to 3400 Hz telephone passband.

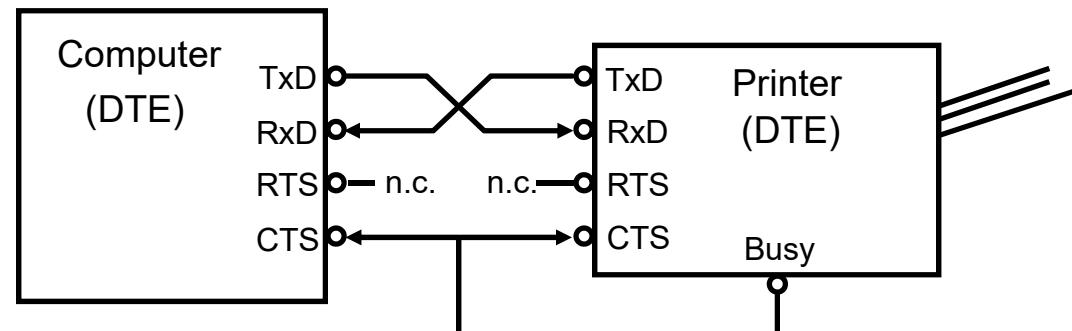
# Typical RS-232C Connections

---

Connection from a computer to a terminal



Connection from a computer to a printer



# The Scope of RS-232C

---

## ***Mechanical Aspects***

- 25-pin D-type connectors (a pre-existing standard)
- DB-25-P plug (male) for Data Terminal Equipment (DTE).
- DB-25-S socket (female) for Data Communications Equipment (DCE).

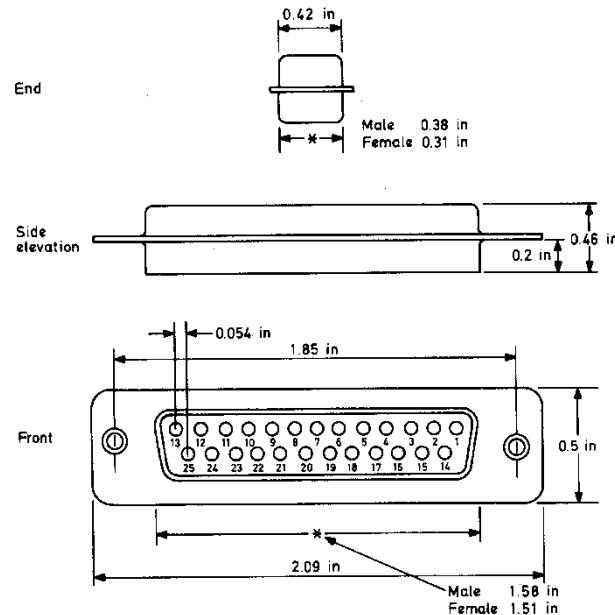
## ***Electrical Aspects***

- “Space” (“0”) signal is a voltage in the range +3 V to +15 V.
- “Mark” (“1”) signal is a voltage in the range -3 V to -15 V.
- An idle line is in the “mark” state.
- A sustained “space” voltage is called a “break” signal.

## ***Data Frame Format***

- Start bit & 7 or 8 data bits & optional parity bit & stop bit(s).
- No higher-level in-band protocol is included in the standard.

# Mechanical Aspects of RS-232C



- A 25-pin DB-25 connector (from Cannon Co., 1952) was specified in an appendix to the original RS-232C standard.
- Most modern serial standards use a much smaller connector. For a time, the smaller 9-pin DE-9 connector replaced the DB-25 for RS-232C connections.
- Ethernet, USB & WiFi have mostly replaced RS-232C.

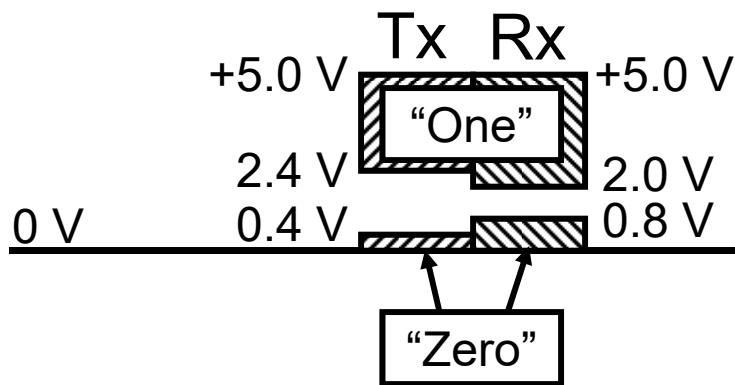
# RS-232C Signal Definitions

---

1. ***Protective ground (shield)***
2. ***Transmitted data (TxD)***
3. ***Received data (RxD)***
4. Request to send (RTS)
5. Clear to send (CTS)
6. Data set ready (DSR)
7. ***Signal ground (GND)***
8. Received signal detected
9. reserved for testing
10. reserved for testing
11. unassigned
12. Secondary signal detected
13. Secondary CTS
14. Secondary TxD
15. Tx Signal Timing for DCE
16. Secondary RxD
17. Rx Signal Element Timing
18. unassigned
19. Secondary RTS
20. Data terminal ready (DTR)
21. Signal quality detector
22. Ringing indicator (RI)
23. Data signal rate selector
24. Tx Signal Timing for DTE
25. unassigned

# Logic Levels for TTL (left) and RS-232C (right)

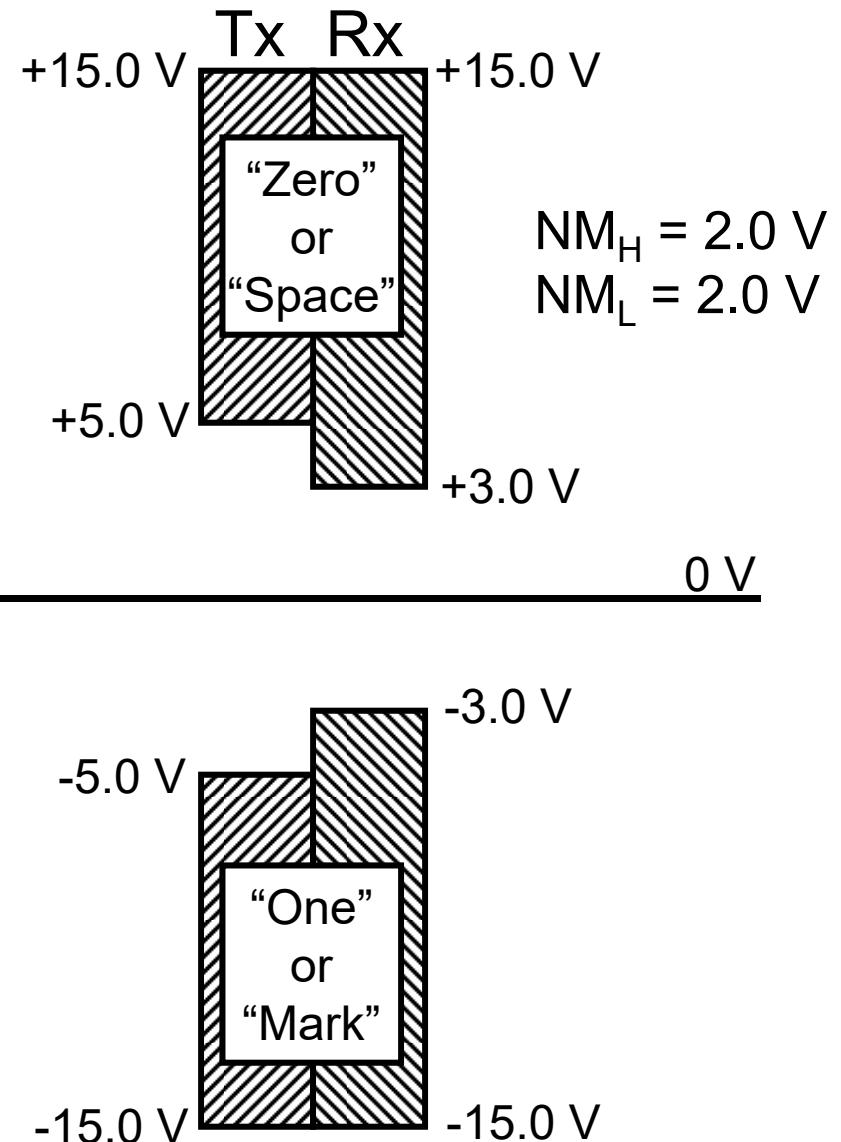
TTL = Transistor-Transistor Logic



**Noise Margins in TTL:**

$$NM_H = 2.4 - 2.0 = 0.4 \text{ V}$$

$$NM_L = 0.8 - 0.4 = 0.4 \text{ V}$$



# More on Logic Signal Ranges

---

- A digital transmitter/driver circuit must produce “high” and “low” voltage signals. Acceptable output “high” and “low” signals lie in two different analog ranges, which are separated by a range of undefined voltages. For example, in TTL a valid **output “high”** signal is a voltage  $V_{OH}$  in the range 2.4V to 5.0 V, and a valid **output “low”** signal is a voltage  $V_{OL}$  in the range 0V to 0.4V. Voltages between 2.4V and 0.4V occur when the signal is changing between high and low; these midrange voltages have no meaning.
- A digital receiver/input circuit must detect “high” and “low” signals from received analog voltages. For example, a valid **input “high”** signal is a voltage  $V_{IH}$  in the range 2.0V to 5.0V, and a valid **input “low”** signal is a voltage  $V_{IL}$  in the range 0V to 0.8V.

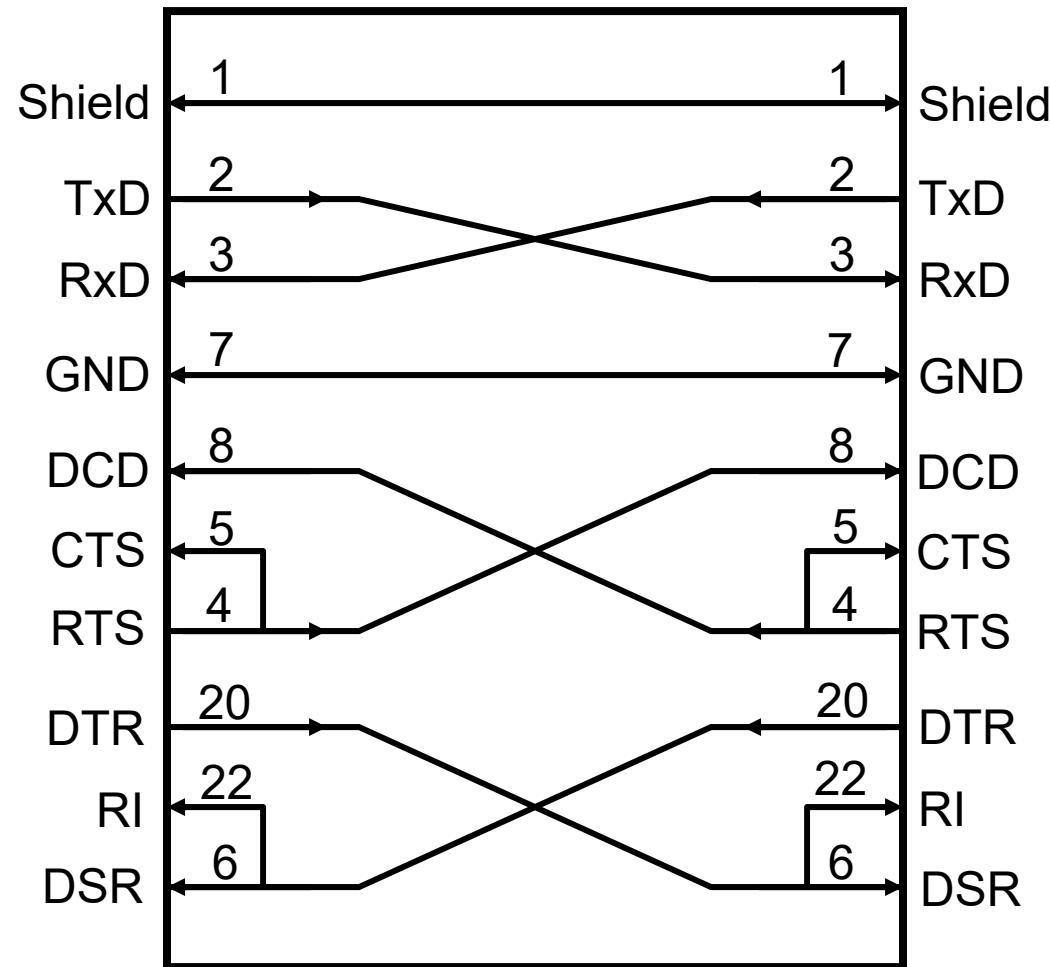
# Noise Margins

---

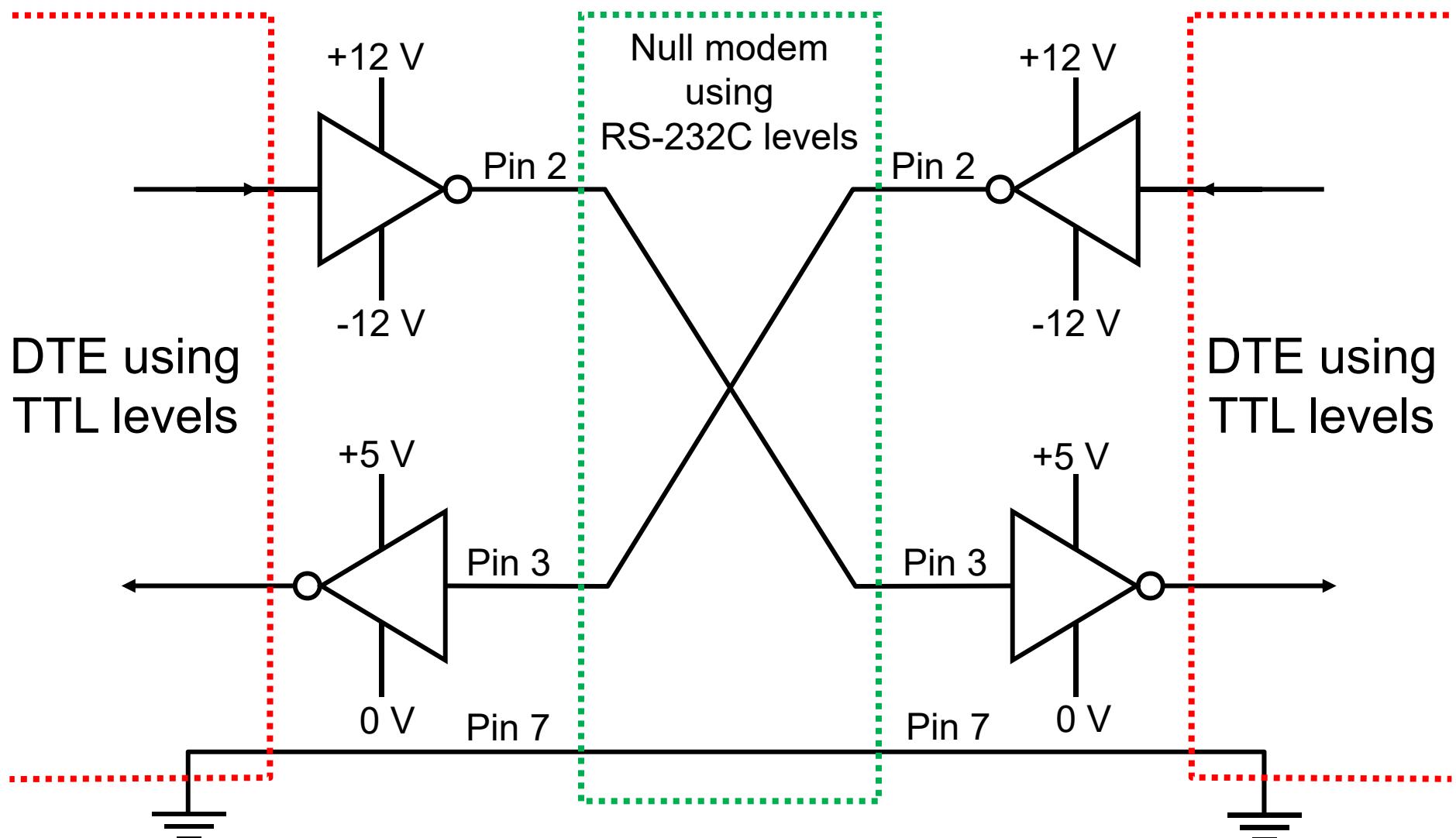
- During transmission between a digital transmitter/driver and a digital receiver/input, analog impairments can degrade the quality of the signal. Noise voltages (or currents) can be picked up by the signal by electromagnetic coupling. Offsets can also get added to the signal for various reasons, such as differences in the power supply voltages.
- The **high noise margin**  $NM_H$  is the difference between the lowest driven output high voltage  $V_{OH}$  and the lowest acceptable input high voltage  $V_{IH}$ . The **low noise margin**  $NM_L$  is the difference between the highest acceptable input low voltage  $V_{IL}$  and the highest driven output low voltage  $V_{OL}$ . The noise margins ensure that the digital signals are detected correctly at the receiver despite the presence of analog impairments (noise + offset) that are smaller in amplitude than the noise margins. Cheap noise immunity!

# “Null Modem” for DTE-DTE Connections

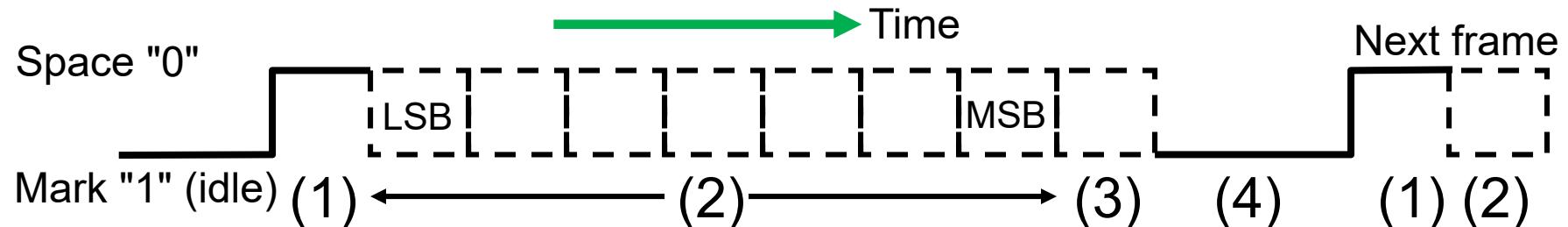
---



# TTL to RS-232C to TTL Duplex Connection



# RS-232C Frame Format



- (1) **Start bit** for indicating the starting time (mark level to space level transition) of a new character “frame”.
- (2) Seven or eight **data bits**. These are usually used to encode 7-bit ASCII or ISO characters, or 8-bit EBCDIC characters. Binary data can also be transmitted.
- (3) Optional even or odd **parity bit** for detecting bit errors.
- (4) **Stop bit(s)** spacer time after a character. Does not have to be a whole number of bit times long.

# Hardware Flow Control Signals in RS-232C

---

Several parallel control signals are provided in RS-232C to allow out-of-band (separate from the data communication path) signaling between the DTC and DCE. For example:

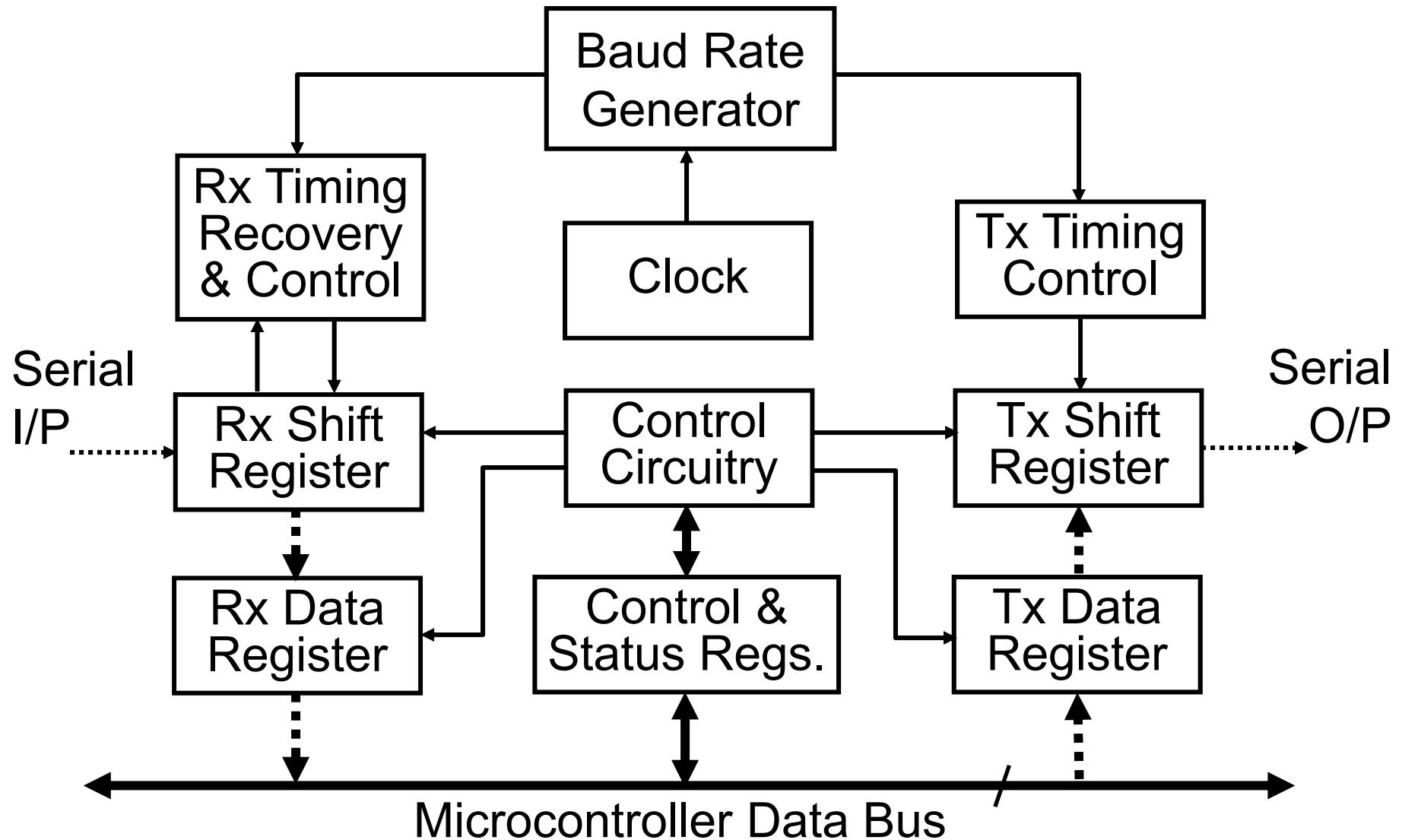
***Request to Send (RTS)*** is asserted to “mark” by the DTE to tell the DCE that the DTE wishes to transmit data.

***Clear to Send (CTS)*** is asserted to “mark” by the DCE to tell the DTE that the DCE is ready to receive data.

***Data Set Ready (DSR)*** is asserted to “mark” by the DCE to tell the DTE that it is switched on and is not in a test mode.

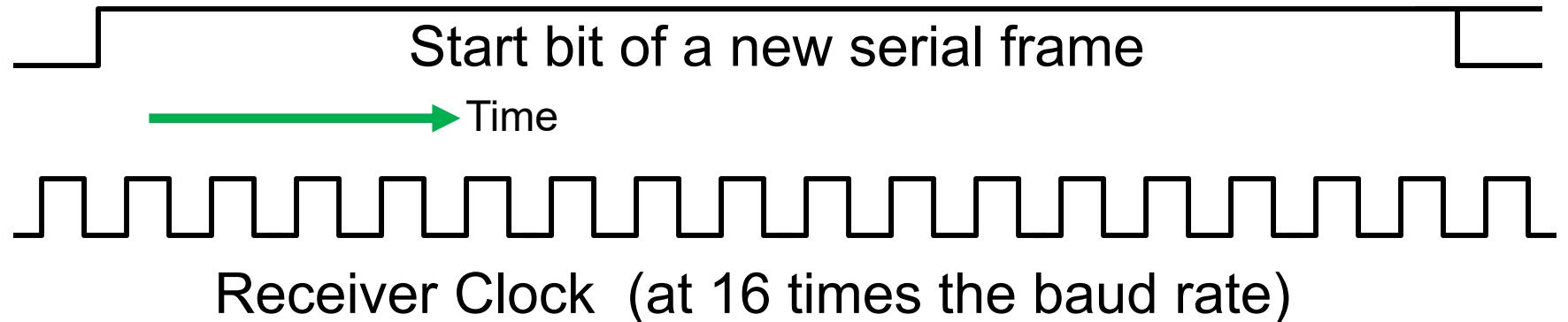
***Data Terminal Ready (DTR)*** is asserted to “mark” by the DTE as long as it wishes to keep the connection active.

# Architecture of a Generic Serial Interface



# Timing Recovery in an Asynchronous Serial Receiver

---



## Procedure for finding the bit/baud center times:

- 1) Look out for the leading edge of the start bit.
- 2) Count off 8 clock periods of the x16 receiver clock.
- 3) Now count 16 clock periods to get to the first data bit.
- 4) The remaining bits (data, parity, stop) should lie at further multiples of 16 receiver clock periods.

# Peripheral Interface Registers

---

- Computer interface hardware is accessed through registers, which are *write-only*, *read-only*, or *readable and writeable* locations in the memory address space.
- ***Control registers*** are writeable locations that allow the CPU to select interface operational modes and options.
- ***Status registers*** are readable locations that allow the CPU to determine interface status information, such as the presence of newly received data, active interrupt conditions, error conditions, etc.
- ***Data registers*** are used as ports to convey data between the CPU and the interface.

# Data Transfer Control Methods

---

- There are four basic methods in which input / output data is transferred within a microcomputer between locations in main memory and I/O interfaces:
  - 1) ***Unconditional data transfer*** by CPU writes to a data register, or CPU reads from a data register.
  - 2) ***Busy-waiting***, i.e., the CPU must wait for some status condition to go true before writing or reading (also called “conditional CPU-controlled data transfer”).
  - 3) ***Interrupt-driven data transfer***.
  - 4) ***Direct memory access (DMA)***.
- All four methods are ultimately under the control of the CPU. However, in the DMA method, the CPU temporarily assigns some control for the data transfer to a “DMA controller”.

# Motorola/Freescale Interfaces that Support Asynchronous Serial Communications

---

MC6850 - *Asynchronous Communications Interface Adaptor* (ACIA)

- Introduced in the mid 1970s to support the 6800 8-bit  $\mu$ P
- Provides one bi-directional serial port

MC68681 - *Dual Universal Receiver Transmitter* (DUART) chip

- Introduced in the mid 1980s to support the M68000  $\mu$ P family
- Provides two bi-directional serial ports

68HC05 - *Serial Communications Interface* (SCI) subsystem

- Introduced in the early 1980s as part of the 68HC05 8-bit  $\mu$ C

68HC11 - *Serial Communications Interface* (SCI) subsystem

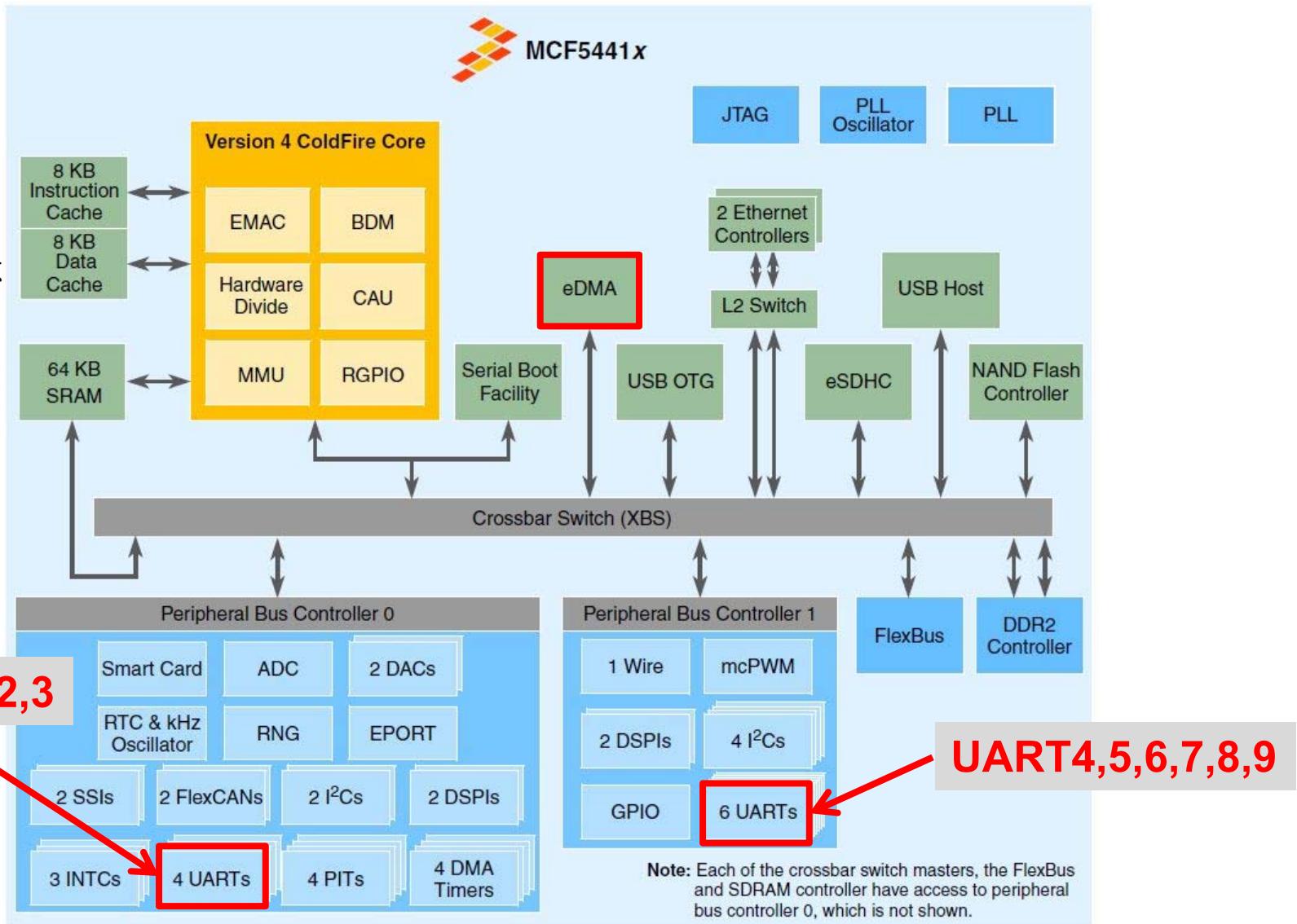
- Introduced in the mid 1980s as part of the 68HC11 8-bit  $\mu$ C

683xx - *Serial Communications Interface* (SCI) subsystem

- Introduced in the late 1980s as part of the 683xx 32-bit  $\mu$ Cs

# The Ten UART Interfaces in the MCF54415

Copyright of Freescale Semiconductor, Inc. 2012.  
Used by permission.

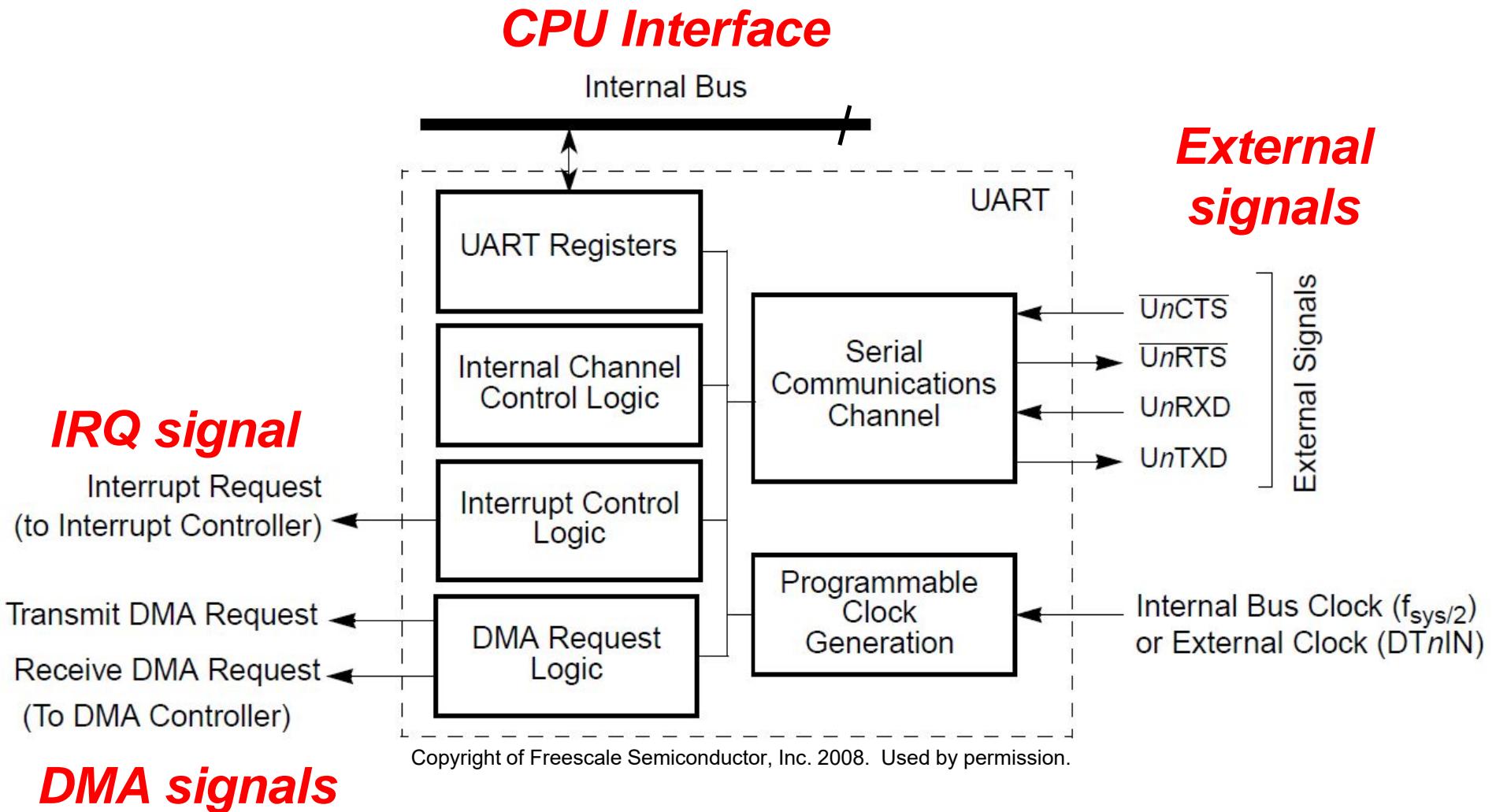


# Universal Asynchronous Receiver-Transmitter

---

- The MCF54415 contains ten ***Universal Asynchronous Receiver-Transmitter*** (UART) interfaces. Ten is a rather generous number!
- Each UART provides an independent, fully-programmable, ***full-duplex serial interface*** for the CPU.
- ***Clock generation*** can be programmed to derive different baud rates using either the MCF54415 clock or (only for UARTs #0-3) externally provided clocks.
- A wide ***variety of data formats*** is supported: 5, 6, 7 or 8-bit data; odd, even, no parity, or force constant parity; 1.0, 1.5 or 2.0 stop bits.
- ***Four interrupting conditions*** can be programmed for each UART.
- Serial communications can use ***polling, interrupts or DMA***.
- ***Three error conditions*** are detected by each receiver: (1) parity error; (2) framing error; and (3) overrun error.
- Receiver direction data is ***quadruple-buffered***, and the transmitter direction data is ***double-buffered***.

# Simplified UART Architecture Diagram, $n = 0, \dots, 9$



# External UART Signals

Signal	Description
Transmitter Serial Data Output (UnTXD)	UnTXD is held high (mark condition) when the transmitter is disabled, idle, or operating in the local loop-back mode. Data is shifted out on UnTXD on the falling edge of the clock source, with the least significant bit (lsb) sent first.
Receiver Serial Data Input (UnRXD)	Data received on UnRXD is sampled on the rising edge of the clock source, with the lsb received first.
Clear-to- Send (UnCTS)	This input can generate an interrupt on a change of state.
Request-to-Send (UnRTS)	This output can be programmed to be negated or asserted automatically by either the receiver or the transmitter. When connected to a transmitter's UnCTS, UnRTS can control serial data flow.

Copyright of Freescale Semiconductor, Inc. 2008. Used by permission.

Note: The four DT $n$ IN pins ( $n = 0, 1, 2, 3$ ) can also be used as an external clock input to drive the UART clock generation circuit in place of the  $f_{sys/2}$  signal, which is the MCF54415 internal system bus clock. DT0IN is shared by UARTs 0, 4 & 8. DT1IN is shared by UARTs 1, 5, 9. DT2IN is shared by UARTs 2 & 6. DT3IN is shared by UARTs 3 & 7.

# Different $\text{UART}_n$ Pin Connections, $n = 0, \dots, 9$

---

$n$	$\text{UART}_n\text{TXD}$	$\text{UART}_n\text{RXD}$	$\text{UART}_n\text{RTS}$	$\text{UART}_n\text{CTS}$	External Clk
0	D11	B10	B11	E13	H15 (DT0IN)
1	D9	C9	D10	C10	H13 (DT1IN)
2	N2	P1	M3	M4	H14 (DT2IN)
3	K1	L3	N/A	N/A	G13 (DT3IN)
4	E13	B11	N/A	N/A	H15 (DT0IN)
5	C10	D10	N/A	N/A	H13 (DT1IN)
6	M4	M3	N/A	N/A	H14 (DT2IN)
7	E15	A13	N/A	N/A	G13 (DT3IN)
8	G15	G14	N/A	N/A	H15 (DT0IN)
9	D14	D15	N/A	N/A	H13 (DT1IN)

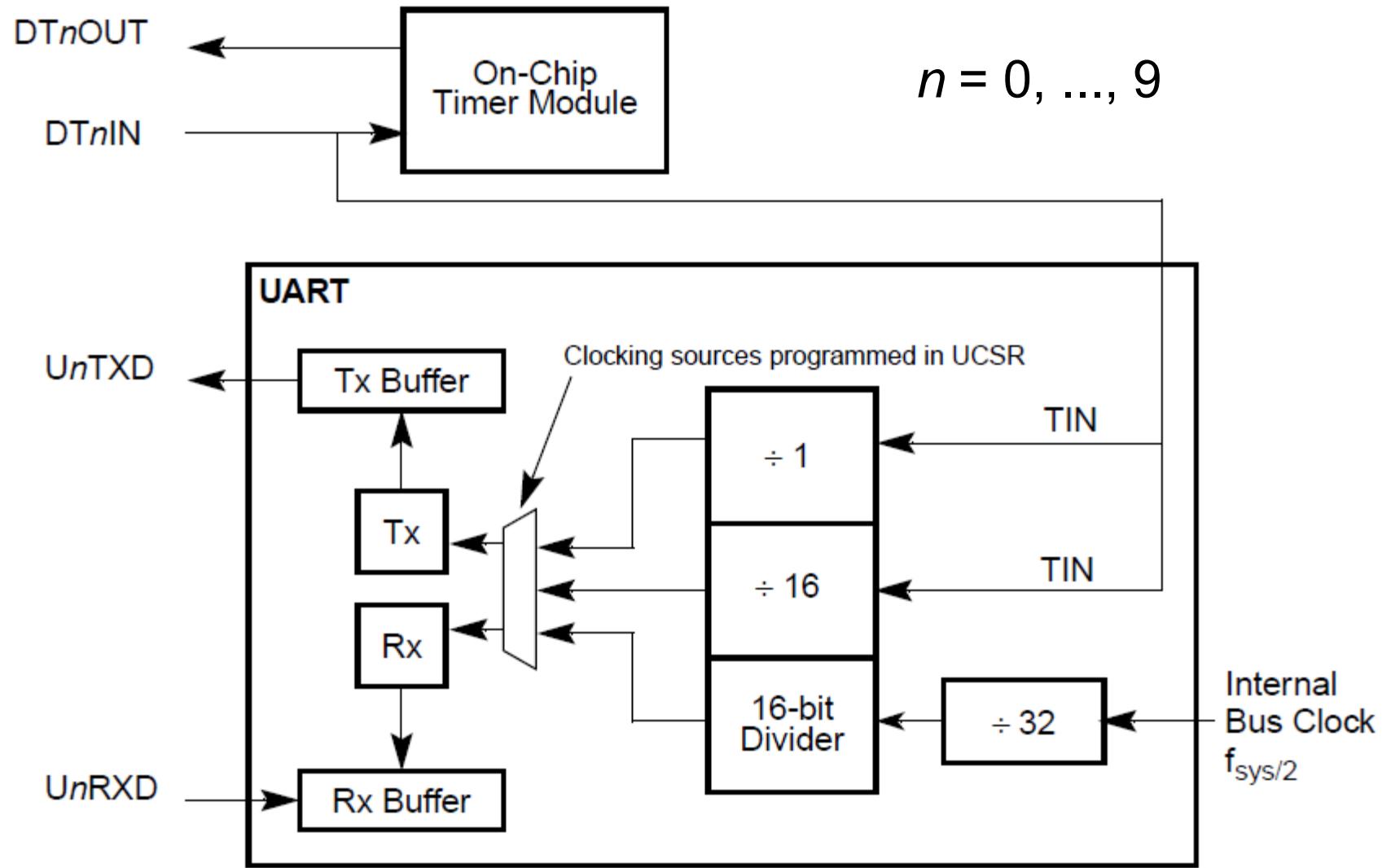
Note: The Request-To-Send (RTS) and Clear-To-Send (CTS) handshake lines are only provided for UART0, UART1 & UART2.

# MCF54415 Pinout for the 256-MAPBGA Package

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
A	VSS	FB <sub>_</sub> AD3	FB <sub>_</sub> AD13	FB <sub>_</sub> AD14	FB <sub>_</sub> AD16	FB <sub>_</sub> AD20	FB <sub>_</sub> AD22	FB <sub>_</sub> AD26	FB <sub>_</sub> AD29	SDHC <sub>_</sub> CLK	SIMO <sub>_</sub> CLK	SSI0 <sub>_</sub> MCLK	SSI0 <sub>_</sub> BCLK	USBO <sub>_</sub> DM	USBH <sub>_</sub> DM	VSS	
B	FB <sub>_</sub> CS4	FB <sub>_</sub> AD2	FB <sub>_</sub> AD8	FB <sub>_</sub> AD11	FB <sub>_</sub> AD15	FB <sub>_</sub> AD19	FB <sub>_</sub> AD24	FB <sub>_</sub> AD28	FB <sub>_</sub> AD31	UART0 <sub>_</sub> RXD	UART0 <sub>_</sub> RTS	SDHC <sub>_</sub> DAT0	SDHC <sub>_</sub> DAT3	USBO <sub>_</sub> DP	USBH <sub>_</sub> DP	RTC <sub>_</sub> EXTAL	
C	FB <sub>_</sub> BE/ BWE3	FB <sub>_</sub> AD1	FB <sub>_</sub> AD7	FB <sub>_</sub> AD9	FB <sub>_</sub> AD10	FB <sub>_</sub> AD17	FB <sub>_</sub> AD23	FB <sub>_</sub> AD30	UART1 <sub>_</sub> RXD	UART1 <sub>_</sub> CTS	SDHC <sub>_</sub> CMD	SSI0 <sub>_</sub> RXD	SSI0 <sub>_</sub> TXD	SIMO <sub>_</sub> PD	SIMO <sub>_</sub> RST	RTC <sub>_</sub> XTAL	
D	FB <sub>_</sub> BE/ BWE1	FB <sub>_</sub> ALE	FB <sub>_</sub> AD5	FB <sub>_</sub> AD12	FB <sub>_</sub> AD18	FB <sub>_</sub> AD21	FB <sub>_</sub> AD25	FB <sub>_</sub> AD27	UART1 <sub>_</sub> TXD	UART1 <sub>_</sub> RTS	UART0 <sub>_</sub> TXD	SDHC <sub>_</sub> DAT1	SIMO <sub>_</sub> VEN	CAN1 <sub>_</sub> TX	CAN1 <sub>_</sub> RX	VSS	
E	FB <sub>_</sub> CS1	FB <sub>_</sub> BE/BW E2	FB <sub>_</sub> AD4	FB <sub>_</sub> AD6	FB <sub>_</sub> VDD	FB <sub>_</sub> VDD	FB <sub>_</sub> VDD	VSS	IVDD	IVDD	IVDD	SIMO <sub>_</sub> XMT	UART0 <sub>_</sub> CTS	SDHC <sub>_</sub> DAT2	SSI0 <sub>_</sub> FS	VSTBY <sub>_</sub> RTC	
F	FB <sub>_</sub> OE	FB <sub>_</sub> CS5	FB <sub>_</sub> AD0	FB <sub>_</sub> BE/ BWE0	FB <sub>_</sub> VDD	FB <sub>_</sub> VDD	VSS	VSS	IVDD	IVDD	IVDD	IRQ7	IRQ1	IRQ4	VDD <sub>_</sub> OSC_A <sub>_</sub> PLL	VSS <sub>_</sub> OSC_A <sub>_</sub> PLL	
G	FB <sub>_</sub> CLK	FB <sub>_</sub> R/W	FB <sub>_</sub> CS0	ADC <sub>_</sub> IN4	FB <sub>_</sub> VDD	VSS	VSS	VSS	VSS	VSS	VDD <sub>_</sub> USBO	T3IN	I2CO <sub>_</sub> SDA	I2CO <sub>_</sub> SCL	EXTAL	G	
H	ADC <sub>_</sub> INO	ADC <sub>_</sub> IN6	FB <sub>_</sub> TA	AVDD <sub>_</sub> ADC	AVSS <sub>_</sub> ADC	VSS	VSS	EVDD	VSS	VSS	VSS	VDD <sub>_</sub> USBH	T1IN	T2IN	TOIN	XTAL	
J	ADC <sub>_</sub> IN1	ADC <sub>_</sub> IN2	ADC <sub>_</sub> IN5	VDDA <sub>_</sub> DAC <sub>_</sub> ADC	VSSA <sub>_</sub> DAC <sub>_</sub> ADC	VSS	EVDD	EVDD	EVDD	EVDD	VSS	PST3	PST0	PST1	PST2	J	
K	DSP10 <sub>_</sub> SOUT	DSP10 <sub>_</sub> PCS0	ADC <sub>_</sub> IN7	ADC <sub>_</sub> IN3	BOOT <sub>_</sub> MOD1	EVDD	EVDD	EVDD	EVDD	EVDD	VSS	TRST	TDO	RESET	TMS	K	
L	DSP10 <sub>_</sub> PCS1	DSP10 <sub>_</sub> SCK	DSP10 <sub>_</sub> SIN	VSS	BOOT <sub>_</sub> MOD0	EVDD	VSS	VSS	VSS	VSS	VSS	TDI	DDATA0	DDATA3	RST <sub>_</sub> OUT	L	
M	IRQ3	IRQ2	UART2 <sub>_</sub> RTS	UART2 <sub>_</sub> CTS	VSS	VSS	SD <sub>_</sub> VDD	SD <sub>_</sub> VDD	SD <sub>_</sub> VDD	SD <sub>_</sub> VDD	SD <sub>_</sub> VDD	DDATA2	MII0 <sub>_</sub> RXCLK	DDATA1	TCLK	M	
N	IRQ6	UART2 <sub>_</sub> TXD	SD <sub>_</sub> A5	SD <sub>_</sub> A10	SD <sub>_</sub> A2	SD <sub>_</sub> BA1	SD <sub>_</sub> CS	SD <sub>_</sub> CAS	SD <sub>_</sub> D3	SD <sub>_</sub> VTT	OW <sub>_</sub> IO	MII0 <sub>_</sub> TXD2	MII0 <sub>_</sub> RXER	JTAG <sub>_</sub> EN	MII0 <sub>_</sub> MDIO	N	
P	UART2 <sub>_</sub> RXD	SD <sub>_</sub> A1	SD <sub>_</sub> A9	SD <sub>_</sub> A3	SD <sub>_</sub> A4	SD <sub>_</sub> A14	SD <sub>_</sub> BA2	SD <sub>_</sub> ODT	SD <sub>_</sub> D1	SD <sub>_</sub> VREF	MII0 <sub>_</sub> CRS	MII0 <sub>_</sub> TXEN	MII0 <sub>_</sub> TXD0	MII0 <sub>_</sub> RXDV	MII0 <sub>_</sub> RXD3	MII0 <sub>_</sub> MDC	P
R	SD <sub>_</sub> A12	SD <sub>_</sub> A7	SD <sub>_</sub> A11	SD <sub>_</sub> A13	SD <sub>_</sub> BA0	SD <sub>_</sub> RAS	SD <sub>_</sub> CKE	SD <sub>_</sub> WE	SD <sub>_</sub> D0	SD <sub>_</sub> D4	SD <sub>_</sub> D6	MII0 <sub>_</sub> COL	MII0 <sub>_</sub> TXD1	MII0 <sub>_</sub> TXER	MII0 <sub>_</sub> RXD1	TEST	R
T	VSS	SD <sub>_</sub> A6	SD <sub>_</sub> A0	SD <sub>_</sub> A8	SD <sub>_</sub> CLK	SD <sub>_</sub> CLK	SD <sub>_</sub> DM	SD <sub>_</sub> DQS	SD <sub>_</sub> DQ0	SD <sub>_</sub> D2	SD <sub>_</sub> D5	SD <sub>_</sub> D7	MII0 <sub>_</sub> TXD3	MII0 <sub>_</sub> TXCLK	MII0 <sub>_</sub> RXD0	VSS	T

Figure 8. MCF54415, MCF54416, MCF54417, and MCF54418 Pinout (256 MAPBGA)

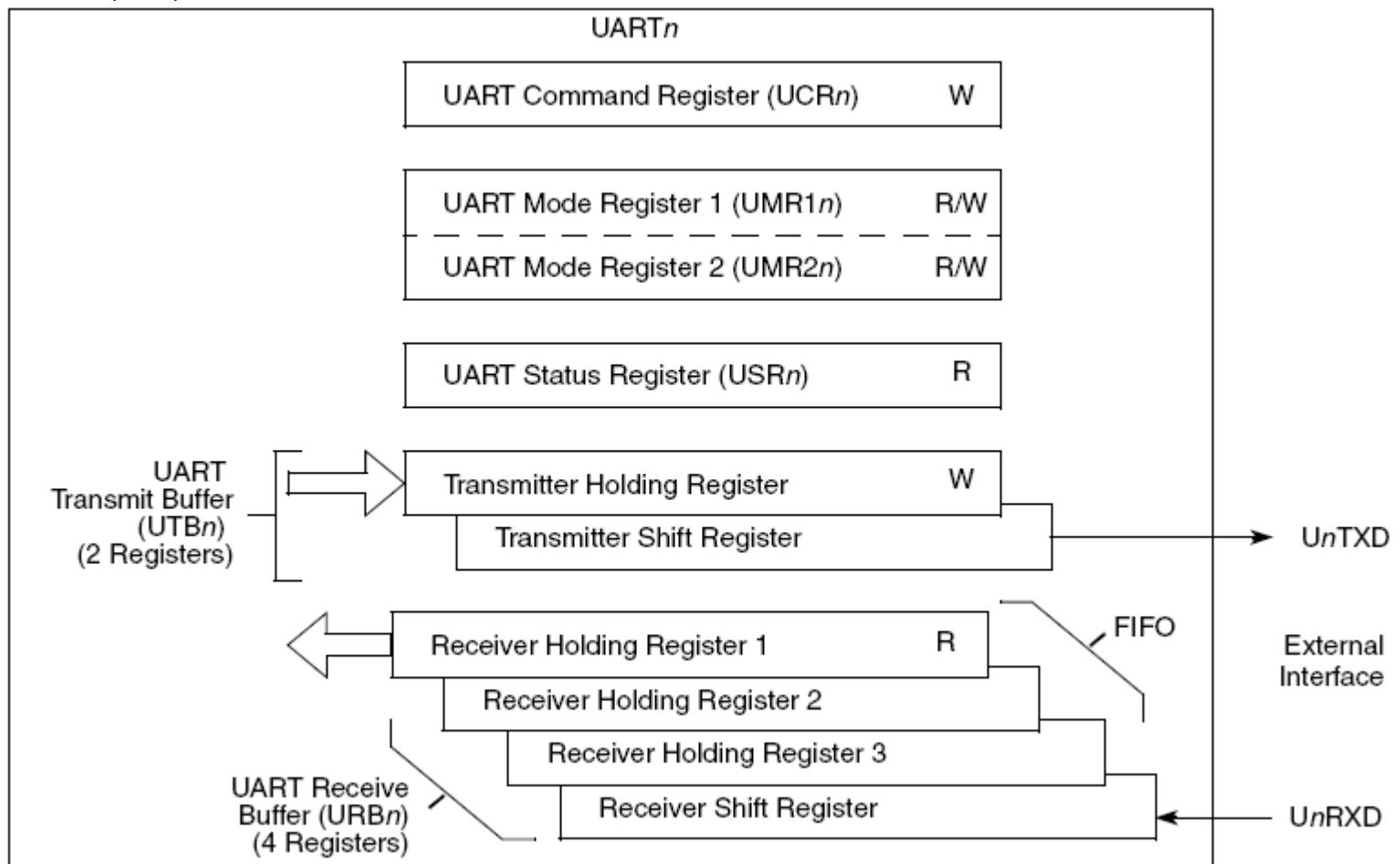
# UART $n$ Clock Generation Sub-Block



Copyright of Freescale Semiconductor, Inc. 2012. Used by permission.

# UART Transmitter/Receiver Registers

$n = 0, \dots, 9$



Copyright of Freescale Semiconductor, Inc. 2008.  
Used by permission.

# **UART $n$ Status Register, $n = 0, \dots, 9$**

---

The CPU-accessible registers of the ten UARTs are mapped into 16-Kbyte regions in the CPU's memory as follows:

	<b>Base Address</b>
<b>UART0</b>	0xFC06_0000
<b>UART1</b>	0xFC06_4000
<b>UART2</b>	0xFC06_8000
<b>UART3</b>	0xFC06_C000
<b>UART4</b>	0xEC06_0000
<b>UART5</b>	0xEC06_4000
<b>UART6</b>	0xEC06_8000
<b>UART7</b>	0xEC06_C000
<b>UART8</b>	0xEC07_0000
<b>UART9</b>	0xEC07_4000

# UART $n$ Status Register, $n = 0, \dots, 9$

	7	6	5	4	3	2	1	0
R	RB	FE	PE	OE	TXEMP	TXRDY	FFULL	RXRDY
W								
Reset:	0	0	0	0	0	0	0	0

Copyright of Freescale Semiconductor, Inc. 2012. Used by permission.

**RB** = 1 (0) : a ***break*** signal has been (has not been) received.

**FE** = 1 (0) : a ***framing error***: a stop bit was not received (was received) at the expected time for the last received character.

**PE** = 1 (0) : a ***parity error*** occurred (did not occur) for the last received character. Note: the parity can be disabled, even, odd, or fixed.

**OE** = 1 (0) : an ***overrun error*** has occurred (not occurred). In an overrun error, the CPU or DMA controller was too slow to read data, and so  $\geq 1$  received characters were lost.

# UART $n$ Status Register, $n = 0, \dots, 9$

	7	6	5	4	3	2	1	0
R	RB	FE	PE	OE	TXEMP	TXRDY	FFULL	RXRDY
W								
Reset:	0	0	0	0	0	0	0	0

Copyright of Freescale Semiconductor, Inc. 2012. Used by permission.

**TXEMP** = 1 (0) : the *transmitter data buffers* are all (are not all) **empty**. The TXEMP =1 condition is also called “underrun”.

**TXRDY** = 1 (0) : the *transmitter holding register* has room (does not have room) and is **ready** (is not ready) for another character to be loaded by the CPU or DMA controller.

**FFULL** = 1 (0) : the *receiver FIFO* buffer is (is not) **full** and cannot accept (can accept) a new character. When FFULL = 1, any more received characters will be lost.

**RXRDY** = 1 (0) : the *receiver FIFO* contains at least one new character (contains no new characters) **ready** to be read.

# UART Interrupt Status and Mask Registers

	7	6	5	4	3	2	1	0
R (UISRn)	COS	0	0	0	0	DB	FFULL/ RXRDY	TXRDY
W (UIMRn)	COS	0	0	0	0	DB	FFULL/ RXRDY	TXRDY
Reset	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x0214 (UISR0); IPSBAR + 0x0254 (UISR1); IPSBAR + 0x0294 (UISR2)							

COS = “Change of State”

DB = “Delta Break”

FFULL = “FIFO Full”

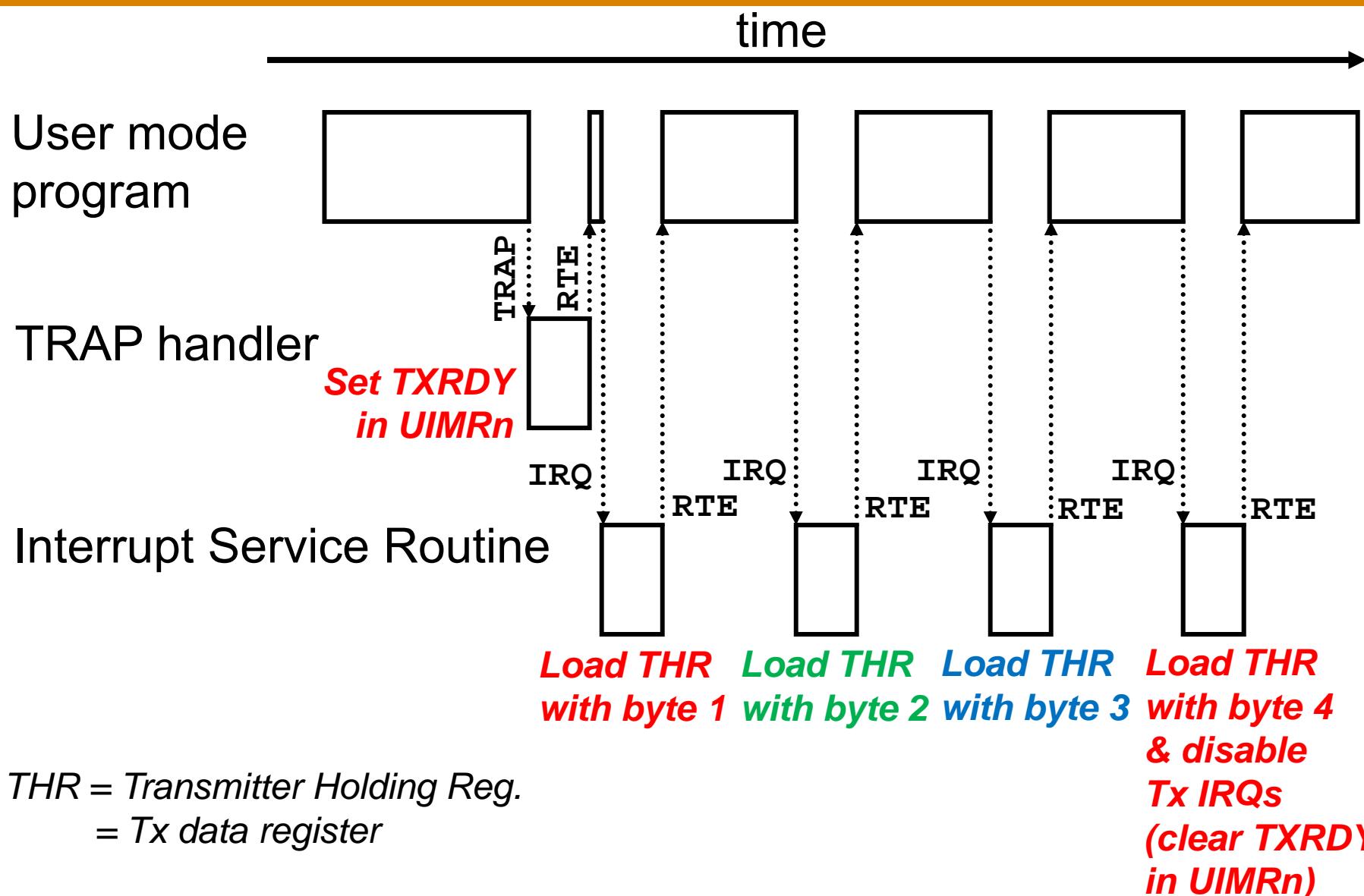
RXRDY = “Receiver Ready”

TXRDY = “Transmitter Ready”

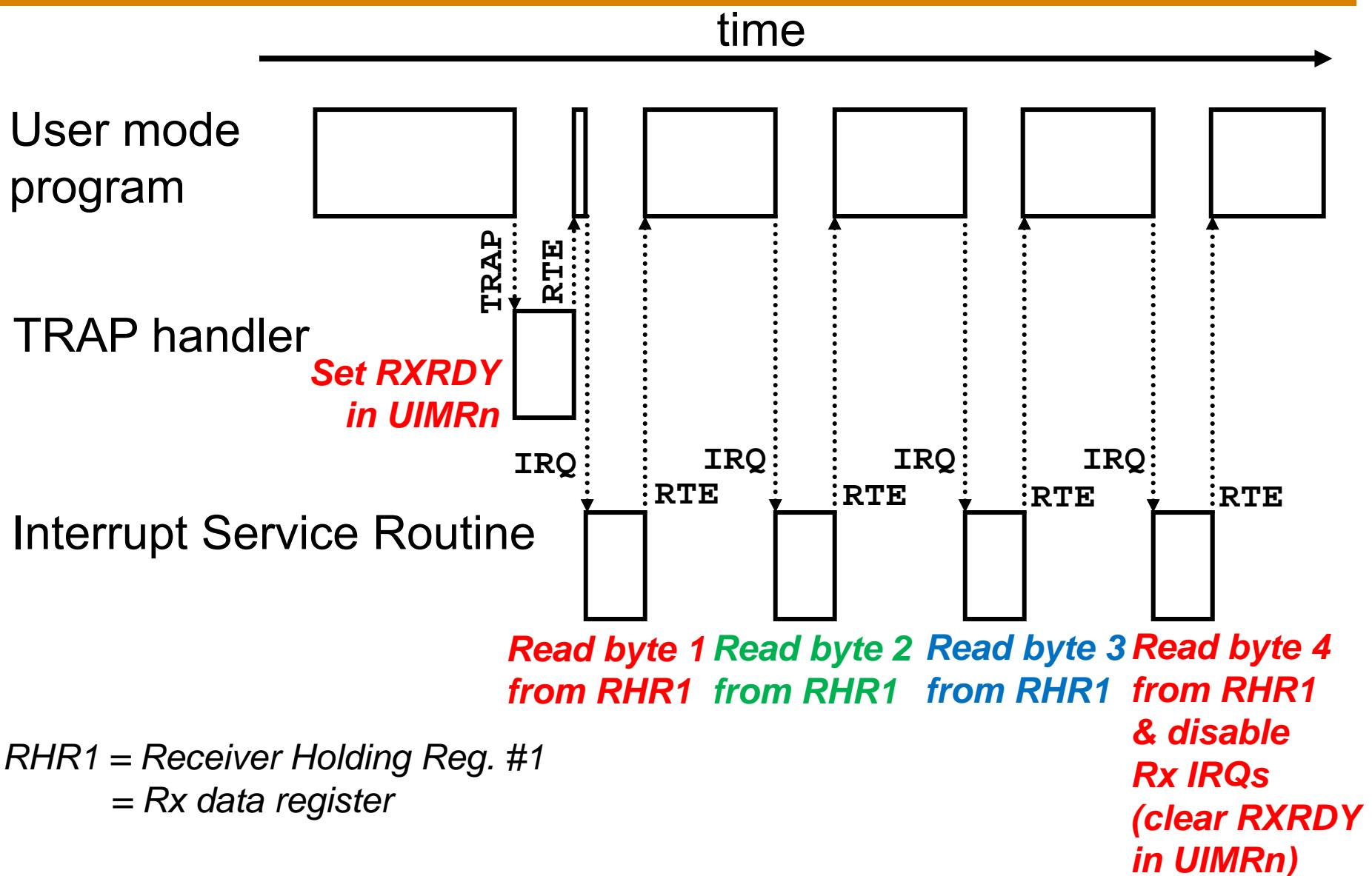
Copyright of Freescale Semiconductor, Inc. 2008. Used by permission.

- Each of the ten UARTs has one read-only **Interrupt Status Register** (UISRn) and one write-only **Interrupt Mask Register** (UIMRn), which share the same address.
- Only 4 out of the 8 bits are used in each register.
- The 4 bits correspond to potentially interrupt-causing conditions.
- A UART will assert an active interrupt signal only if a status bit and the corresponding mask bit are both 1.
- An interrupt can be stopped by writing the corresponding mask bit to 0.

# Interrupt-Driven UART $n$ Output for Four Bytes



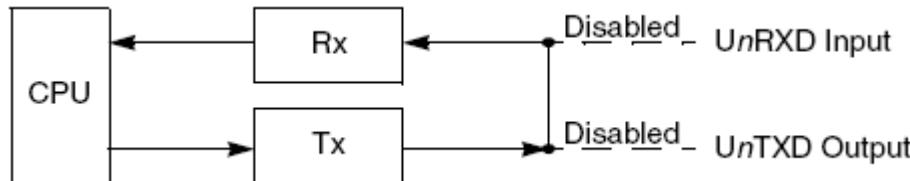
# Interrupt-Driven UARTn Input for Four Bytes



# UART Loopback Test Modes

## 1) Local Loopback Mode

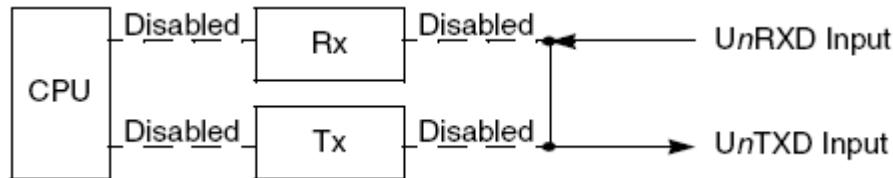
- Allows local UART to verify its Tx and Rx circuits
- External Tx and Rx connections are not tested



Copyright of Freescale Semiconductor, Inc. 2008. Used by permission.

## 2) Remote Loopback Mode

- Allows remote UART to verify its Tx and Rx circuits
- External Tx and Rx connections are tested



Copyright of Freescale Semiconductor, Inc. 2008. Used by permission.

# Simplified UART Initialization Sequence

---

Register	Setting
UCR <i>n</i>	Reset the receiver and transmitter. Reset the mode pointer (MISC[2–0] = 0b001).
UIMR <i>n</i>	Enable the preferred interrupt sources.
UACR <i>n</i>	Initialize the input enable control (IEC bit).
UCSR <i>n</i>	Select the receiver and transmitter clock. Use timer as source if required.
UMR1 <i>n</i>	If preferred, program operation of receiver ready-to-send (RxRTS bit). Select receiver-ready or FIFO-full notification (RxRDY/FFULL bit). Select character or block error mode (ERR bit). Select parity mode and type (PM and PT bits). Select number of bits per character (B/Cx bits).
UMR2 <i>n</i>	Select the mode of operation (CMx bits). If preferred, program operation of transmitter ready-to-send (TxRTS). If preferred, program operation of clear-to-send (TxCTS bit). Select stop-bit length (SBx bits).
UCR <i>n</i>	Enable transmitter and/or receiver.

Copyright of Freescale Semiconductor, Inc. 2008.  
Used by permission.

# Background: Network Topology (1)

---

- **Network topology** is the arrangement of communicating **nodes** and the communication **links** among those nodes. Typical nodes are computers, servers, switches, and I/O devices (e.g. printers, scanners, cameras).
- A network topology can be modelled as a mathematical **graph**  $G = \{ V, E \}$ , with a finite set  $V$  of **vertices** (i.e., nodes) and a finite set  $E$  of **edges** (i.e., links) such that  $E \subseteq V \times V$ .
- A **physical network topology** accurately shows the relative physical positions of the nodes and links. The distances on the topology may be scaled differently for convenience in diagrams, but the relative positions of the nodes should correspond to their physical/geographical positions.
- A **logical network topology** shows the data flows through a network. The logical network topology may differ from the underlying physical network topology. For example, multiple parallel physical links in a physical network topology, or a linear chain of connected nodes, may be merged into a single logical link between the two end nodes.

# Background: Network Topology (2)

---

- The communication wire(s) or cable(s) form either (1) a point-to-point link or (2) a shared multidrop bus (often just called a bus).
- For a ***point-to-point link*** there are two communicating nodes, one at either end of the link. If the link only allows communication in one direction at a time it is called a ***simplex*** link. If the link allows simultaneous communication in both directions it is called a ***duplex*** link (e.g., RS-232C).
- A ***shared multidrop bus*** provides connections among two or more nodes that share the bus (e.g. RS-485). Only one node can broadcast on the bus at any one time, creating either a point-to-point link (one receiving node) or a ***point-to-multipoint broadcast*** (two or more receiving nodes).
- When a shared bus is used, a method needs to be provided to deal with the possibility that two nodes may wish to transmit at the same time. E.g.
  - Allow ***collisions*** to occur, but detect them and recover from them.
  - Circulate one “token” on the bus that allows only one node to transmit.
  - Provide a central “arbiter” that selects the one active transmitting node.

# Background: Network Topology (3)

---

- Many different network topologies can be created using only point-to-point links. Here are some of the common resulting topologies:

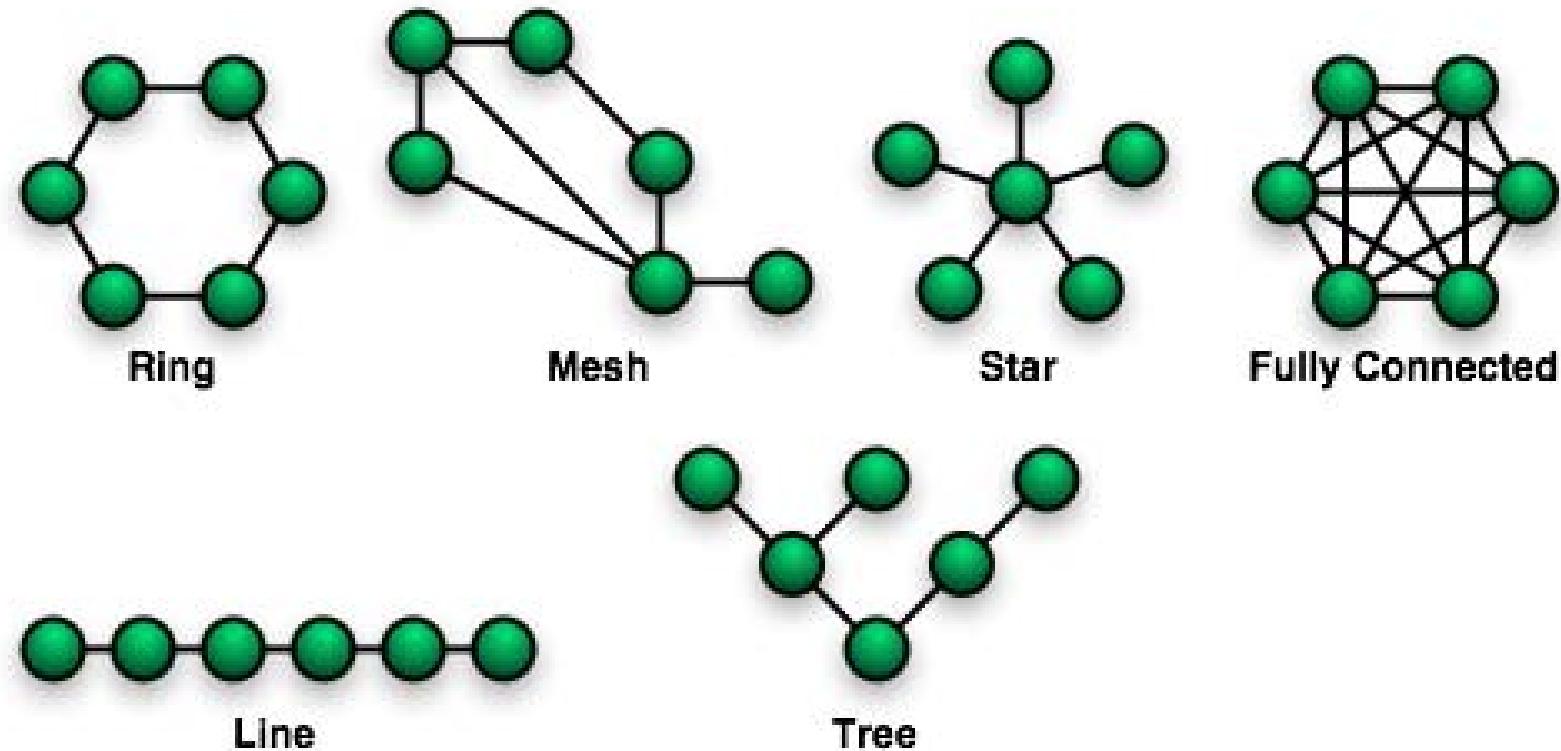


Figure courtesy of Wikimedia Commons

# The Ethernet Local Area Network (LAN)

---

- **Local Area Network** standards were developed to allow computers and other office equipment to communicate within limited areas (e.g., a department) at much higher speeds than the telephone network.
- In 1973, Xerox Corp. released the **Ethernet** LAN standard. This standard was then adopted as IEEE Std 802.3 and is now widely used.
- The original Ethernet standard uses a **shared coaxial cable** as the medium, and has a peak data rate of 10 Mbps. Typical data rates were a few Mbps because of the need for idle time on the cable.
- Only one transmitter station on the cable can transmit at a time.
- A transmitter with data to send will wait until the cable is available before attempting to transmit data onto the cable.
- If two (or more) transmitters happen to start transmitting at the same time, then the resulting “**collision**” is detected by those transmitters.
- After a collision, the active transmitters each wait for a **random back-off time** before trying to transmit again (to hopefully avoid a 2<sup>nd</sup> collision).

# Ethernet Frame Format

---

The bits in an Ethernet signal are organized into “frames” as follows:

- **Preamble**: 7 bytes (56 bits) of alternating 0s and 1s for synchronization
- **Start Frame Delimiter** (SFD): 1 byte = 0b10101011
- **Destination Address** (DA): 6 bytes specifying the destination station
  - An example address, given in hex, is: 07-01-02-01-2C-4B
- **Source Address** (SA): 6 bytes specifying the source station
- **Length / Type**: 2 bytes
  - If <1518, then this field specifies the length of the data field
  - If >1536, then this field specifies the upper layer LAN protocol
- **Data**: Anywhere from 46 to 1500 bytes of data
- **Cyclic Redundancy Check** (CRC): 4 bytes computed using CRC-32

The CRC is computed at the source station as the remainder obtained by dividing the data field (padded with an all-0 field) by a standard 33-bit binary polynomial divisor. At the destination station, the data (padded with the received CRC) is divided by the same divisor. The remainder will be zero if there were no errors in transmission; if nonzero, cause a retransmission.

# The CRC-32 Calculation

- **Polynomial division** over the finite field GF(2) is easy to implement in hardware. The hardware just requires shift operations and binary XORs.
- For an  $n$ -bit binary CRC, the **divisor polynomial** contains  $n+1$  bits. The most significant bit (MSB) of the divisor is always a 1.
- The divisor is shifted past the input bit stream, starting out left justified with the most significant bit (e.g., the first arriving bit).

$10110110110110111101 \leftarrow$  *Input bitstream*  
 $1101 \longleftarrow$  *Divisor for 3-bit CRC*

- If the divisor MSB is alongside a 1 in the input stream, then the two vectors are XOR-ed together bitwise. Shift the divisor right, and repeat.

$01100110110110110101$   
 $1101$

- Once the process is finished, all of the input bits will have been zeroed, except maybe the remainder at the right end. This is the computed CRC.

$00000000000000000000 \longleftarrow$  *CRC (zero, so no errors detected)*  
 $1101$

# Example CRC-3 Calculation (no error)

10110110110110111101  
1101  
01100110110110111101  
1101  
00001110110110111101  
1101  
00001110110110111101  
1101  
00001110110110111101  
1101  
00000011110110111101  
1101  
00000011110110111101  
1101  
00000000100110111101  
1101  
00000000100110111101  
1101

0000000010010111101  
1101  
0000000001000111101  
1101  
0000000000101111101  
1101  
0000000000011011101  
1101  
0000000000001101101  
1101  
00000000000000001101  
1101  
000000000000000000001101  
1101  
0000000000000000000000001101  
1101  
00000000000000000000000000001101  
1101  
000000000000000000000000000000001101  
1101  
0000000000000000000000000000000000001101  
1101  
000000000000000000000000000000000000001101  
1101  
00  
No errors detected!

## Example CRC-3 Calculation (with detected error)

10110110100110111101  
1101                              ↑ Bit error  
01100110100110111101  
1101  
00001110100110111101  
1101  
00001110100110111101  
1101  
00001110100110111101  
1101  
00000011100110111101  
1101  
00000011100110111101  
1101  
0000000110110111101  
1101  
0000000110110111101  
1101

# Ethernet Data Rates and Transmission Media

---

- Ethernet is available for several different data rates (10, 100, 1000, 10000 Mbps, etc.) and a variety of media:
    - **RG-8** thick 50- $\Omega$  coaxial cable, for 10-Mbps (10BASE5, the original Ethernet standard). Now obsolete.
    - **RG-58** “thin” 50- $\Omega$  coaxial cable, for 10-Mbps (10BASE2).
- Coax cables have been replaced by point-to-point twisted pair links.
- **CAT-3** twisted pair, for 10-Mbps (10BASE-T)
  - **CAT-5e** twisted pair, for both 100-Mbps (100BASE-TX, a.k.a. Fast Ethernet) and 1-Gbps (1000BASE-T)
  - **CAT-6a**, for 10-Gbps (10GBASE-T)
  - **CAT-7**, intended for 10-Gbps, but CAT-6a is now used instead
  - **CAT-8**, for short range (5 to 30m) at 25 or 40 Gbps
  - **Optical fibre**, for 100-Mbps (100BASE-FX), 1-Gbps (1000BASE-SX), 10-Gbps (10GBASE-LX4), 40-Gbps

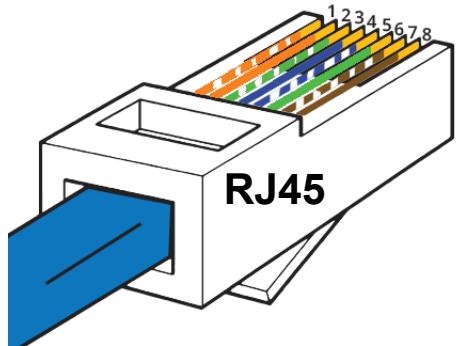
# Twisted Pairs within an Ethernet Cable

---

- Just as in RS-232C, the 10BASE-T and 100BASE-TX standards are asymmetrical interfaces, where each cable has a computer/peripheral end and a communications equipment (e.g., router, hub) end.
- CAT-5e (and better) cables contain four twisted pairs.
- One twisted pair (terminating on pins 1 & 2) is used for the transmit (Tx) direction, and a second twisted pair (terminating on pins 3 & 6) is used for the receive direction.
- The two other twisted pairs (terminating on pins 4 & 5, and on 7 & 8) can be used to carry a second bi-directional 10BASE-T or 100BASE-TX connection.
- Faster Ethernet standards (e.g., 1000BASE-T, 10GBASE-T) use all four twisted pairs, with signals travelling simultaneously in both directions.
- A *crossover cable* was originally required to connect two devices (e.g. computer-to-computer) of the same type. However, most modern Ethernet ports have a circuit that automatically creates a crossover if necessary, using a mechanism defined in the Auto MDI-X specification.

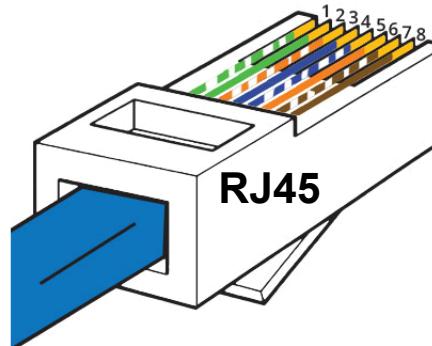
# Type A and B Pin-outs at the RJ45 Plug

EIA/TIA 568B



1. White Orange	5. White Blue
2. Orange	6. Green
3. White Green	7. White Brown
4. Blue	8. Brown

EIA/TIA 568A



1. White Green	5. White Blue
2. Green	6. Orange
3. White Orange	7. White Brown
4. Blue	8. Brown

Figure courtesy of [www.cat6wiringdiagram.com](http://www.cat6wiringdiagram.com)

- Normal (straight-through) patch cables are A↔A or B↔B.
- Ethernet crossover cables are A↔B or B↔A.

# Straight-through and Crossover Ethernet Cables



Figure A

*Shows the Pin Out of Straightthrough Cables*

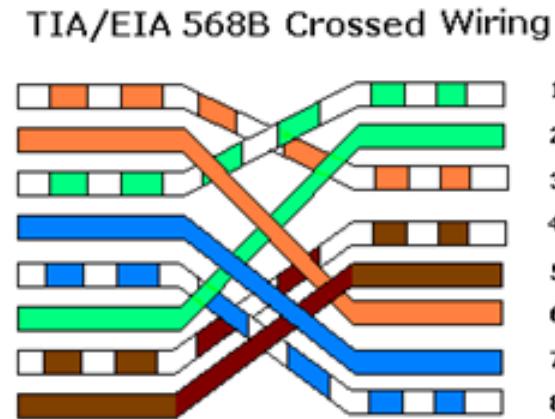
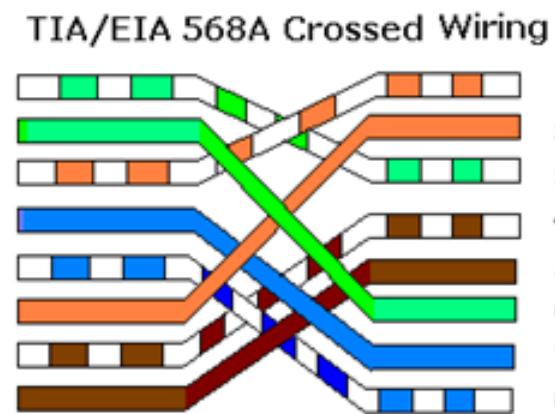


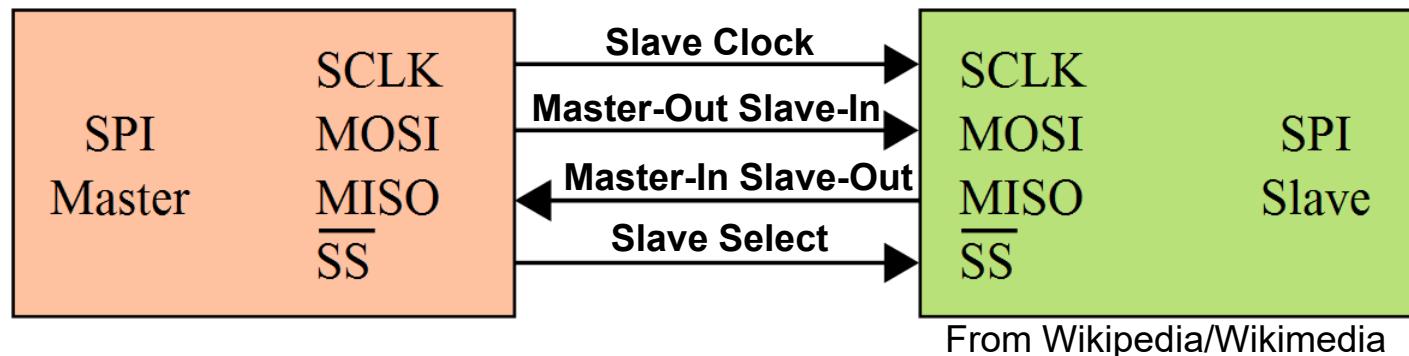
Figure B

*Shows the Pin Out of Crossover Cables*

Figure courtesy of electronicsforu.com

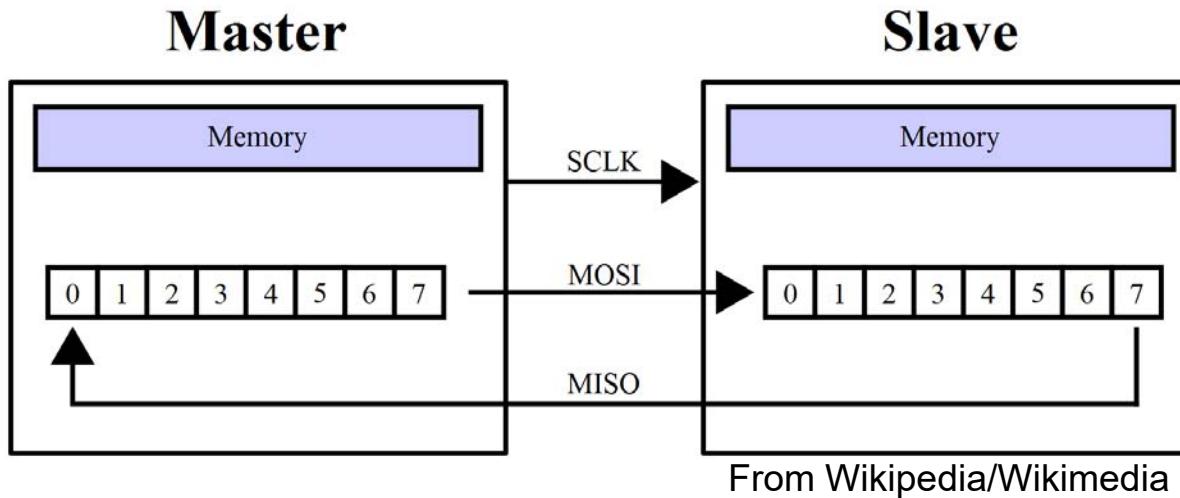
# The Serial Peripheral Interface (SPI) bus

- An *ad hoc* standard that was first proposed by Motorola in the late 1980s. (Motorola spun off its semiconductor division as Freescale Semiconductor in 2004.)
- SPI became, and remains, a popular low-cost interface used by embedded systems to communicate with each other and with peripherals (e.g., LCD displays).
- SPI has a simple master-slave architecture where the master and slave communicate over four wires.



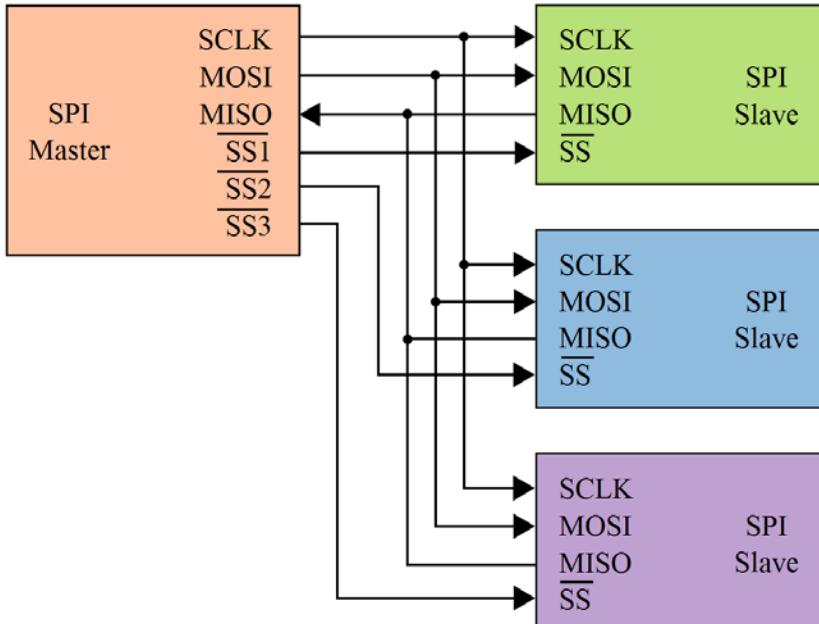
- *Caution:* There is some variation in the spelling of the signal abbreviations. The standard is not well enforced.

# SPI Data Transfer Shift Register

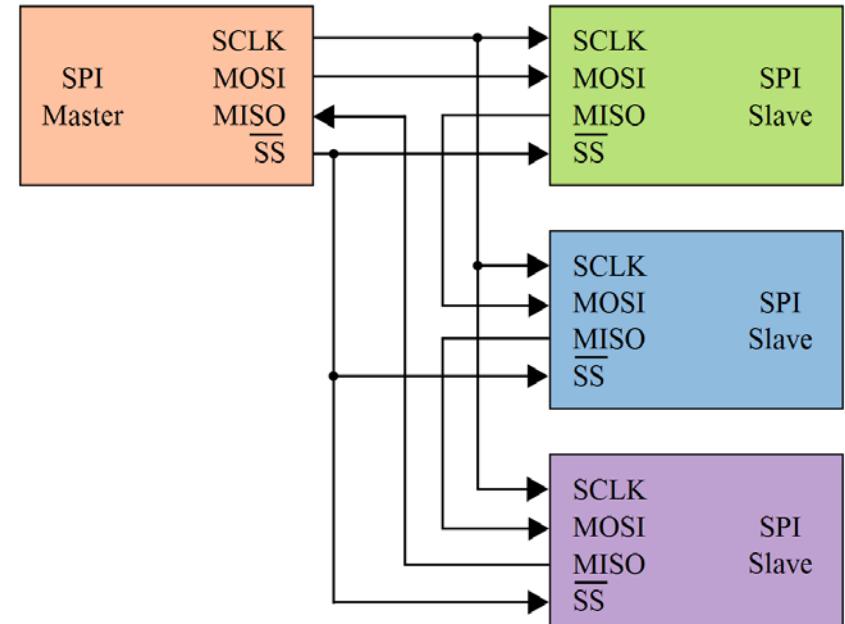


- The standard architecture for transferring data between the master and one slave is to create a large shift register that is split equally between the master and slave.
- New master data shifts over MOSI into the slave while the previous slave data shifts out over MISO to the master. The master effectively reads old data from the slave while writing new data into the slave.
- Note: The MOSI signal is always driven by the master. MISO is actively driven only when the slave is selected; otherwise MISO is tri-stated.

# SPI Configurations for Multiple Slaves



From Wikipedia/Wikimedia



From Wikipedia/Wikimedia

- Separately selected slaves.
- Master must provide one slave select signal (SS\*) for each slave, and only one of these can be active at a time (to avoid conflicting signals on MISO).
- **Daisy-chained** slaves.
- The slaves effectively become one large slave with a shared slave select signal.
- The shift register is the concatenation of all serial registers.

# SPI Clock Polarity and Phase Relationships

1st edge is rising, 2nd edge is falling.

SCK

CPOL=0  
CPOL=1

1st edge is falling, 2nd edge is rising.

SS

Sample input data on first edge; update output on second clock edge.

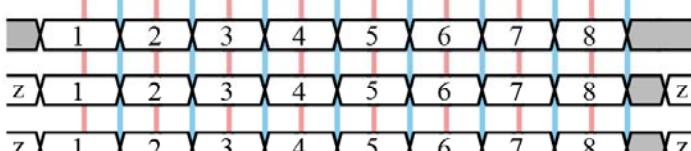
CPHA=0

Sample data on second clock edge; update output on first clock edge.

Cycle #

MISO

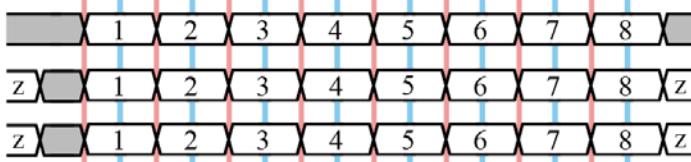
MOSI



Cycle #

MISO

MOSI

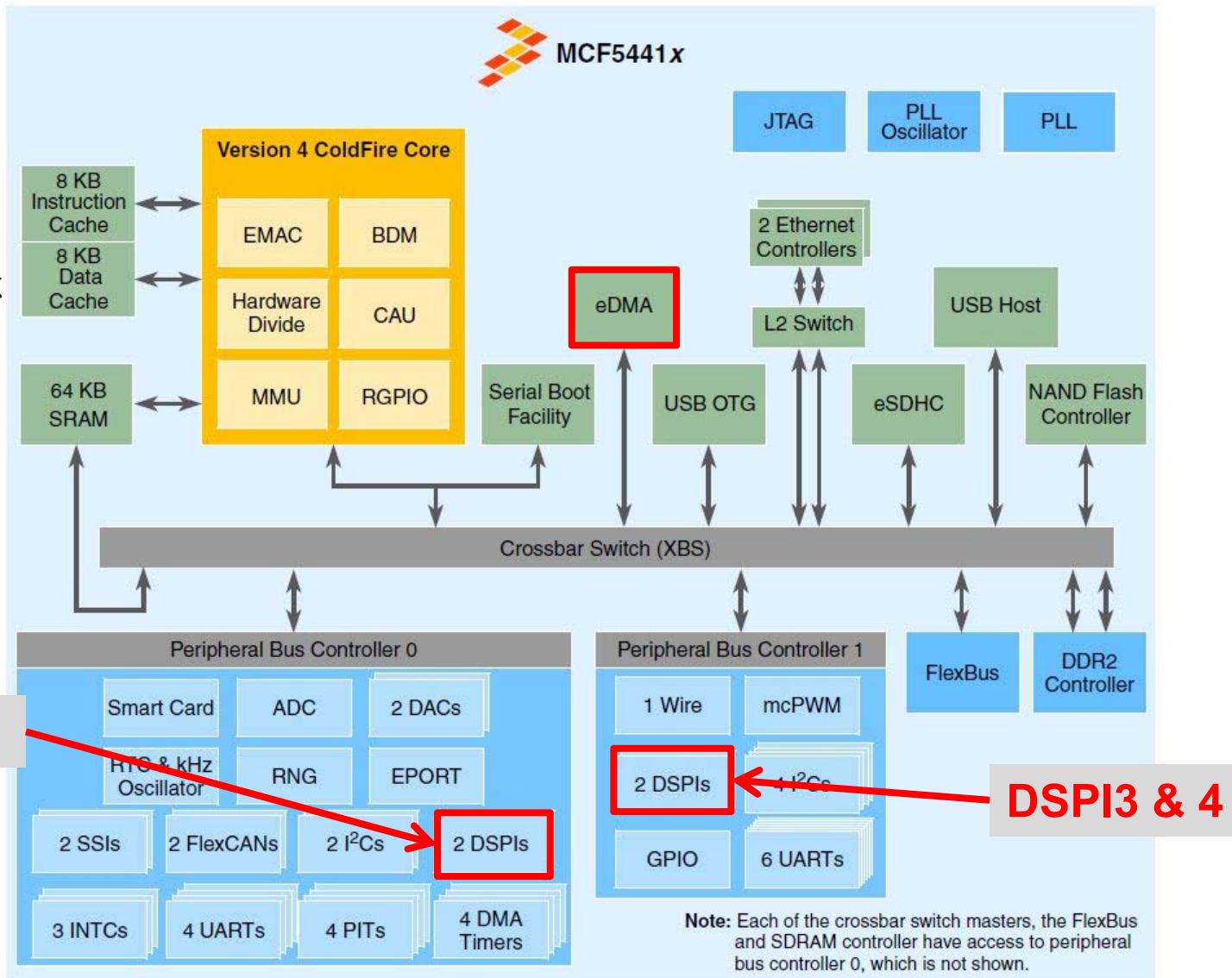


From Wikipedia/Wikimedia

- Unfortunately, because SPI is an *ad hoc* standard that is not policed by any organization, variations have appeared in the waveforms in “SPI compatible” devices.
- CPOL = 0 (1) means SCK starts off at low (high) voltage.
- CPHA = 0 (1) sample data on first (second) SCK edge.

# The Four DMA SPI Interfaces in the MCF54415

Copyright of Freescale Semiconductor, Inc. 2012.  
Used by permission.

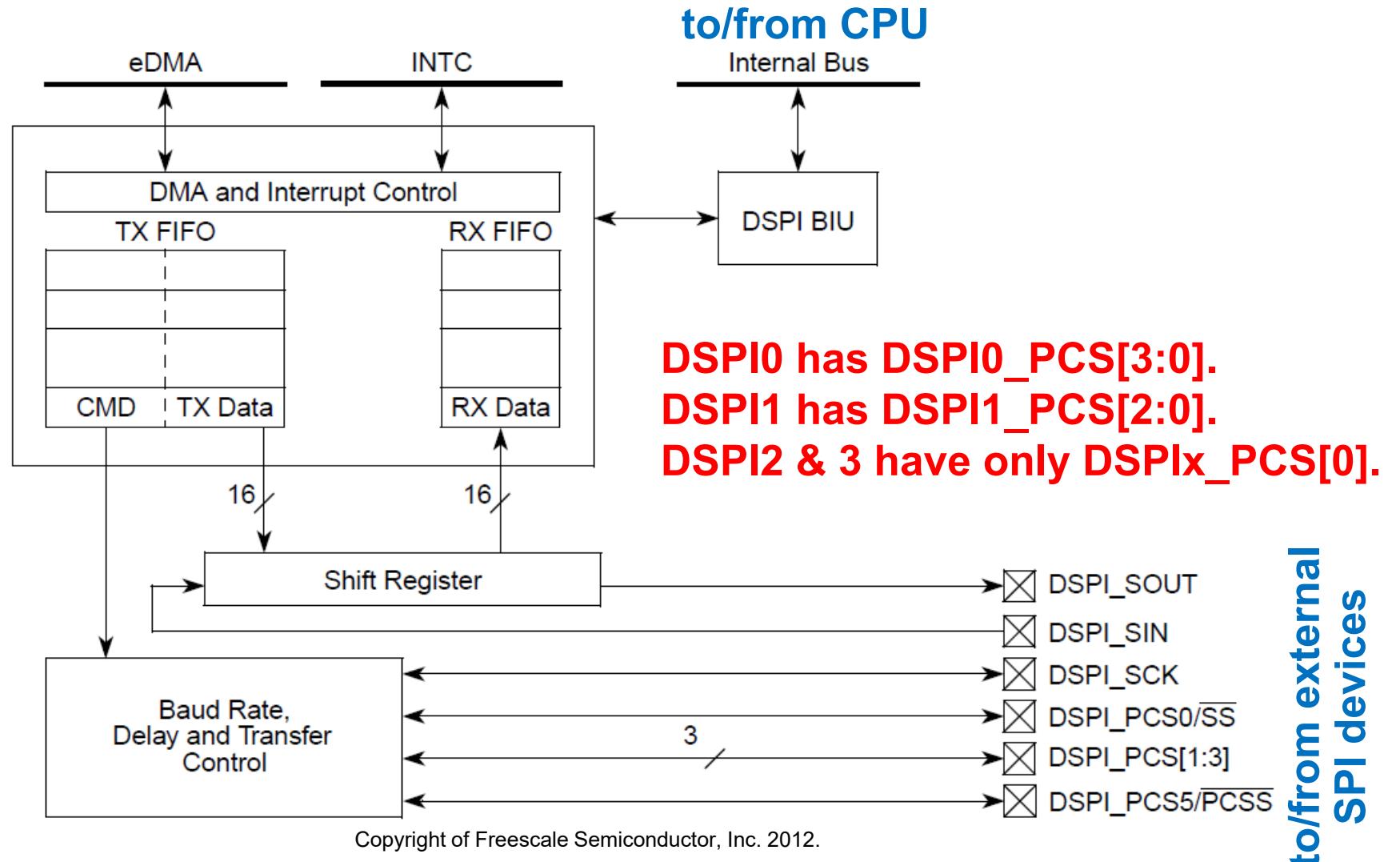


# The Four DSPI Interfaces

---

- The MCF54415 has four **D**irect Memory Access **S**erial **P**eripheral **I**nterfaces (DSPIs), labelled DSPI0, DSPI1, DSPI2 and DSPI3.
- Each DSPI can connect to multiple external devices (e.g., LCD displays, microcontrollers) that have SPI interfaces.
- Each DSPI block can be in one of four modes: (1) master mode; (2) slave mode; (3) disabled (low power); & (4) debug.
- All four DSP interfaces can handle 16 queued output transfers and 16 queued input transfers. The transfers are controlled, without requiring detailed CPU involvement, by an ***enhanced direct memory access*** (eDMA) controller that is also provided in the MCF54415.

# DSPI Interface Architecture on the MCF54415



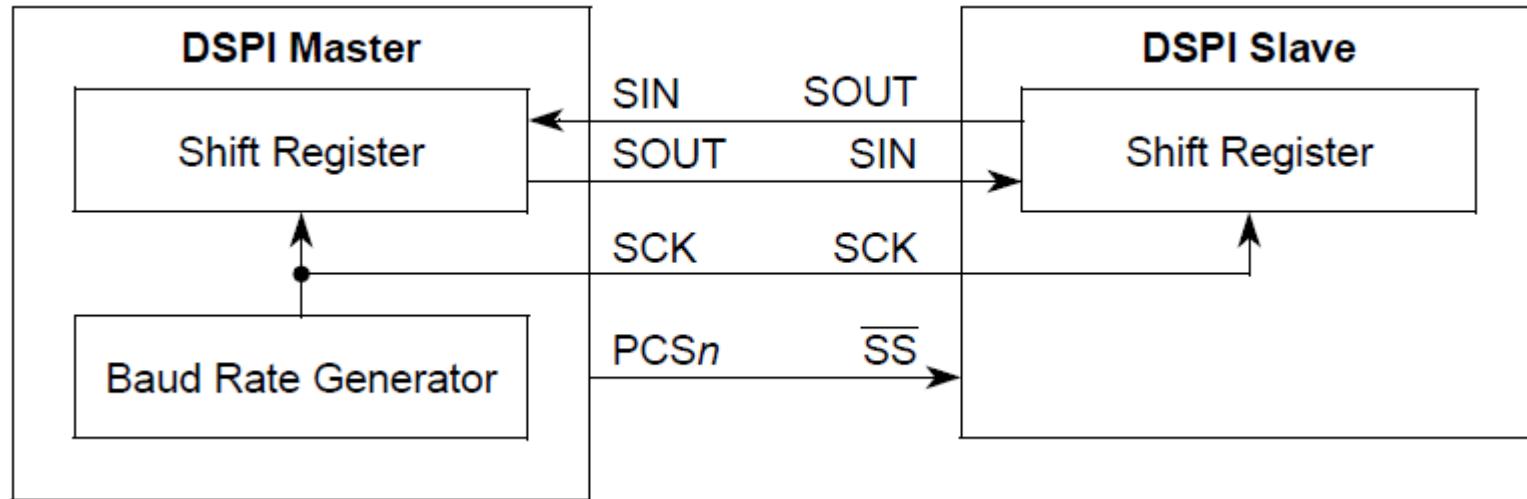
# DSPI Interface Architecture on the MCF5441x

Name	Function			
	Master Mode	I/O	Slave Mode	I/O
DSPI_PCS0/SS	Peripheral chip select 0	Output	Slave select	Input
DSPI_PCS[1:3]	Peripheral chip select 1–3	Output	Unused	—
DSPI_PCS5/PCSS	Peripheral chip select 5/ Peripheral chip select strobe	Output	Unused	—
DSPI_SIN	Serial data in	Input	Serial data in	Input
DSPI_SOUT	Serial data out	Output	Serial data out	Output
DSPI_SCK	Serial clock	Output	Serial clock	Input

Copyright of Freescale Semiconductor, Inc. 2012.

- In **master mode**, a DSPI block drives the \_SCK clock and the peripheral chip selects \_PCSn. It also determines the SPI transactions using its Tx FIFO.
- In **slave mode**, an external SPI master drives the \_SCK clock. The DSPI block is enabled when \_SS is asserted.

# DSPI Interface Architecture on the MCF5441x



Copyright of Freescale Semiconductor, Inc. 2012.

- In **master mode**, a DSPI block drives the \_SCK clock and the peripheral chip selects \_PCSS. It also determines the SPI transactions using its Tx FIFO.
- In **slave mode**, an external SPI master drives the \_SCK clock. The DSPI block is enabled when \_SS is asserted.

# Memory-mapped DSPI Registers

---

- The DPSI blocks are controlled by the CPU through memory-mapped registers. Each DSPI is assigned a 16 K-(16384-) byte large region in the CPU's memory map.

Base Address	Module
0xFC05_C000	DSPI 0
0xFC03_C000	DSPI 1
0xEC03_8000	DSPI 2
0xEC03_C000	DSPI 3

Copyright of Freescale Semiconductor, Inc. 2012.

- Each DSPI block has 46 32-bit memory-mapped registers that the CPU can access. 33 of these registers are read-only; the other 13 registers are readable and writeable.

# Memory-mapped DSPI Registers

---

<b>Offset</b>	<b>Register Name and Abbreviation</b>	<b>Size</b>	<b>Mode</b>
0x00	DSPI module configuration reg. (DSPI_MCR)	32-bit	R/W
0x08	DSPI transfer count register (DSPI_TCR)	32-bit	R/W
0x0C	DSPI clock and transfer attributes regs. (DSPI_CTAR0..7)	32-bit	R/W
...			
0x28			
0x2C	DSPI status register (DSPI_SR)	32-bit	R/W
0x30	DSPI DMA/interrupt request & enable reg.	32-bit	R/W
0x34	DSPI push Tx FIFO register (DSPI_PUSHR)	32-bit	R/W
0x38	DSPI pop Rx FIFO register (DSPI_POPR)	32-bit	R
0x3C	DSPI transmit FIFO registers (DSPI_TXFR0..15)	32-bit	R
...			
0x78			
0x7C	DSPI receive FIFO registers (DSPI_RXFR0..15)	32-bit	R
...			
0xB8			