

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

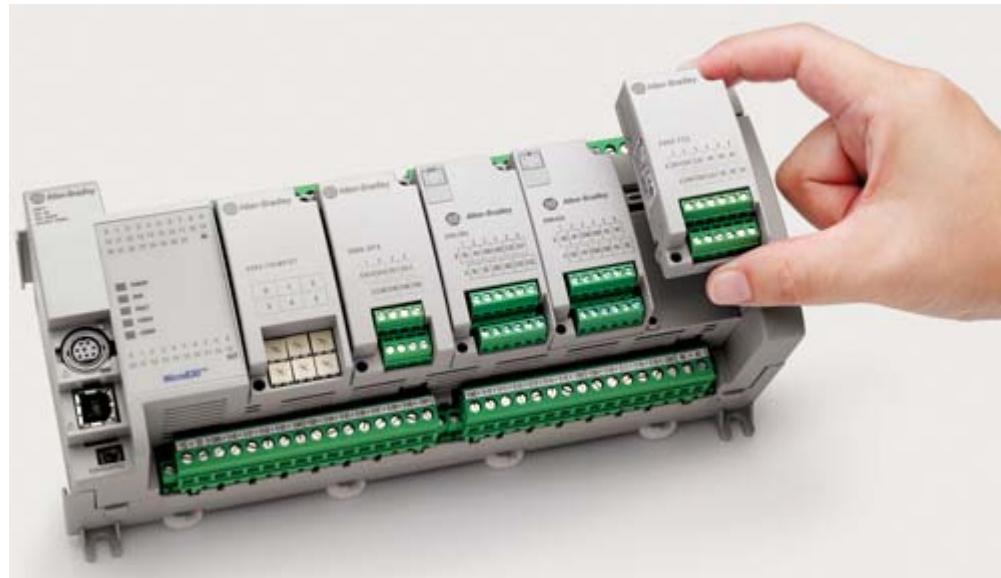


Photo courtesy of Rockwell Automation

- 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

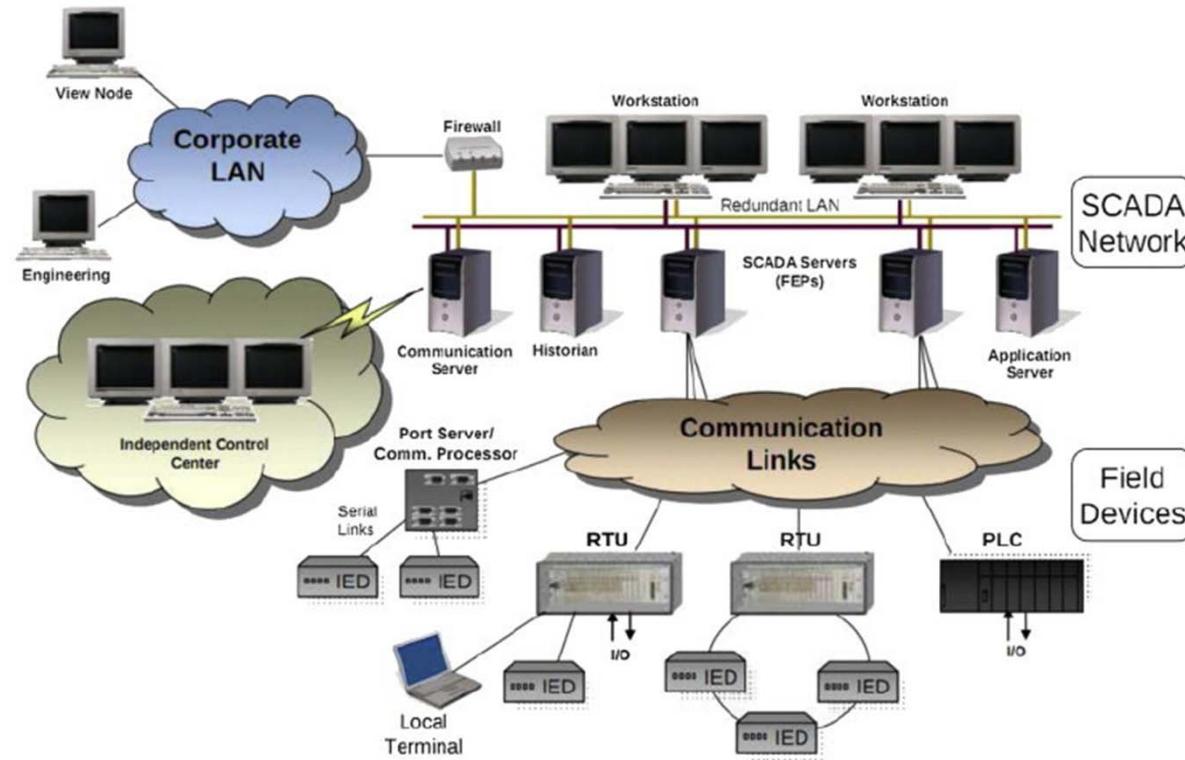


Photo courtesy of Dr. Helge Janicke, De Montfort Univ.

- 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 (μ 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*** (μ 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?

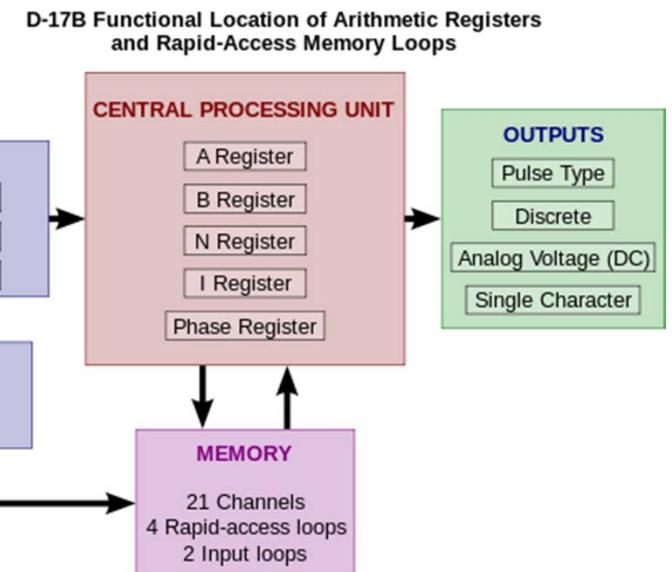
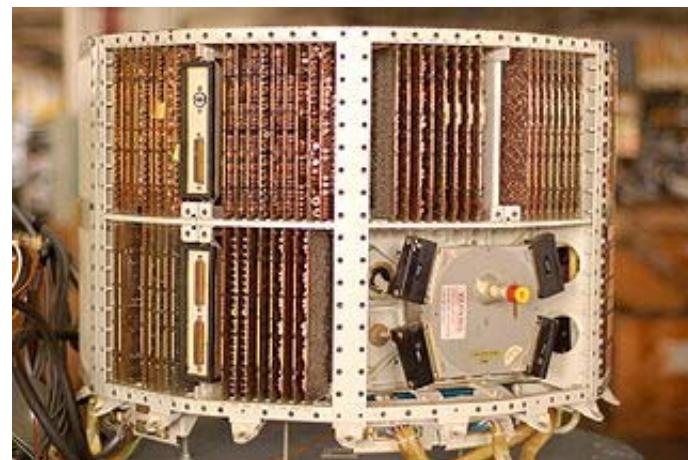
- In ***consumer electronics***: TVs, cell phones, electronic games, WiFi routers, entertainment systems, cameras, ...
- In ***residences***: microwave ovens, refrigerators, thermostats, high-efficiency furnaces, security systems, solar panels, ...
- In ***automobiles***: used to control the ignition 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

Copyright © 2020 by Bruce Cockburn

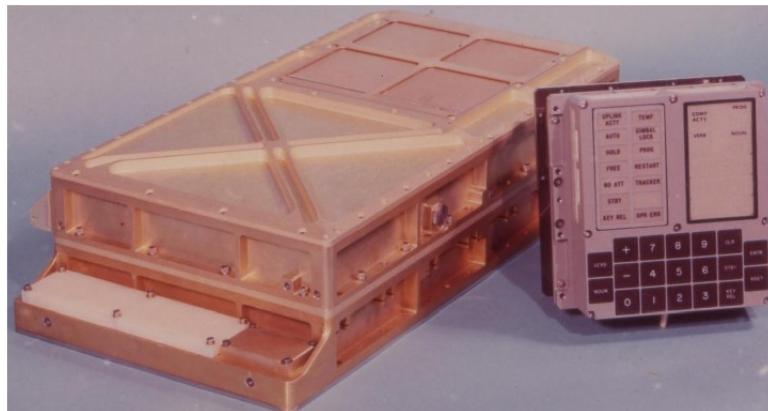
1-14

Early Embedded Systems (2)

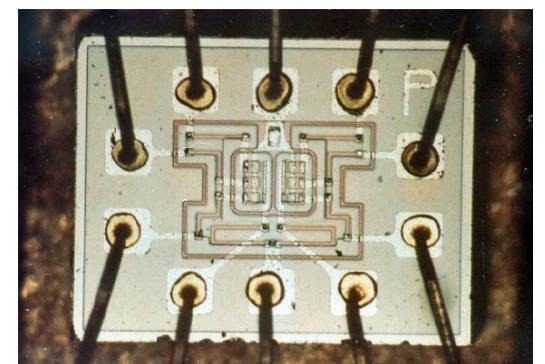
- 1966: MIT Instrumentation Laboratory / Raytheon Apollo Guidance Computer (AGC), used to control the Apollo spacecraft. Weighing 32 kg, the AGC contained 2800 integrated circuits (dual 3-input NOR gates) and dissipated 55 W with a 4-phase 1.024-MHz clock.



Apollo Command & Service Modules



AGC Computer and DSKY User Interface



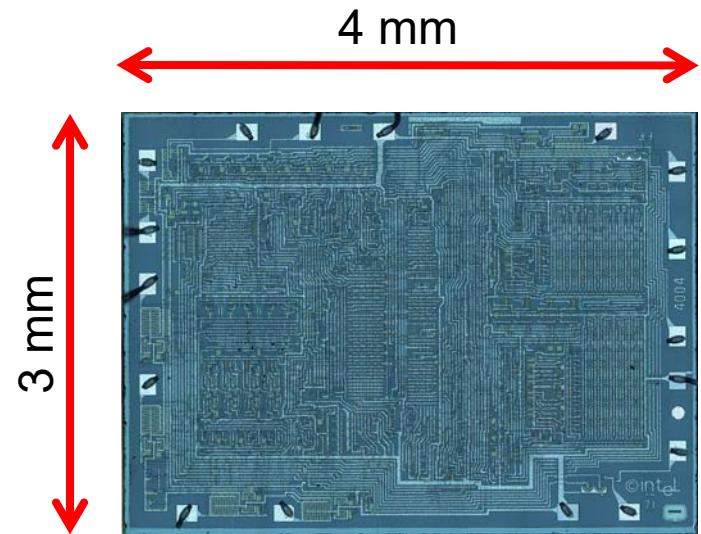
RTL Dual 3-input NOR IC

Data and images from www.wikipedia.org

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 μ P in 10- μ m PMOS

Data and images from 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

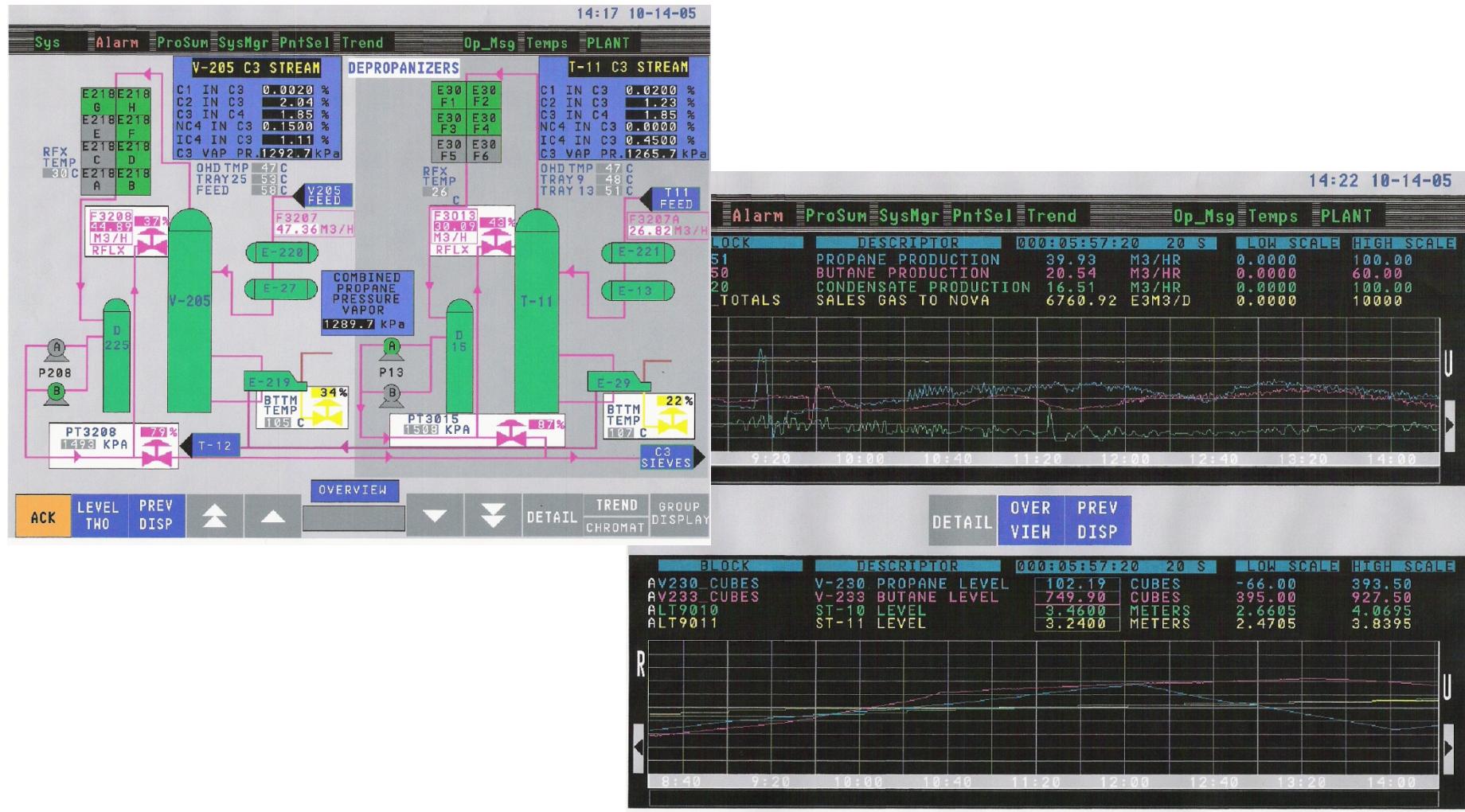


Photo courtesy of Keyera Corp.

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)

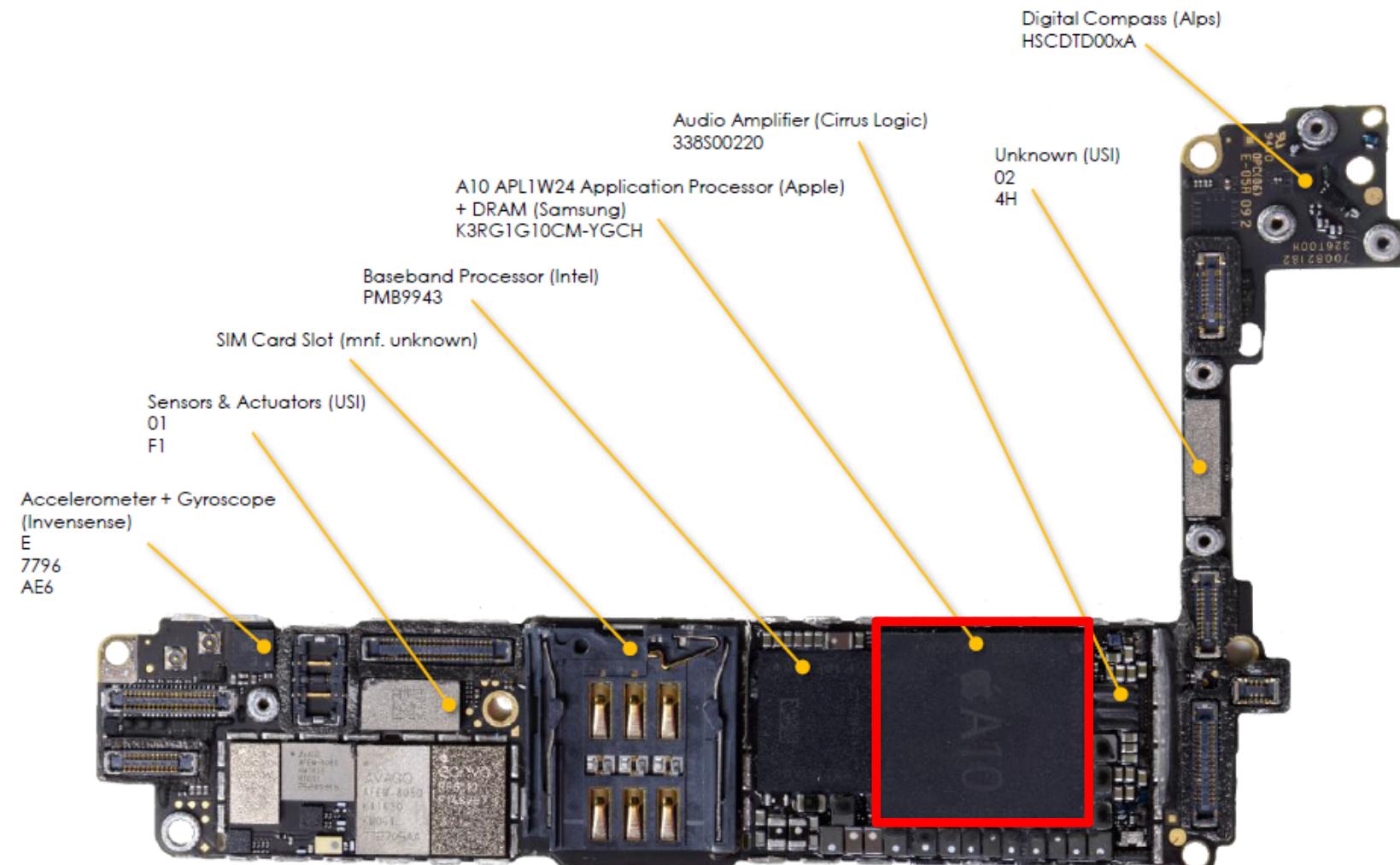


Photo from TechInsights-ChipWorks

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 mm² die area
- 3.3 billion transistors

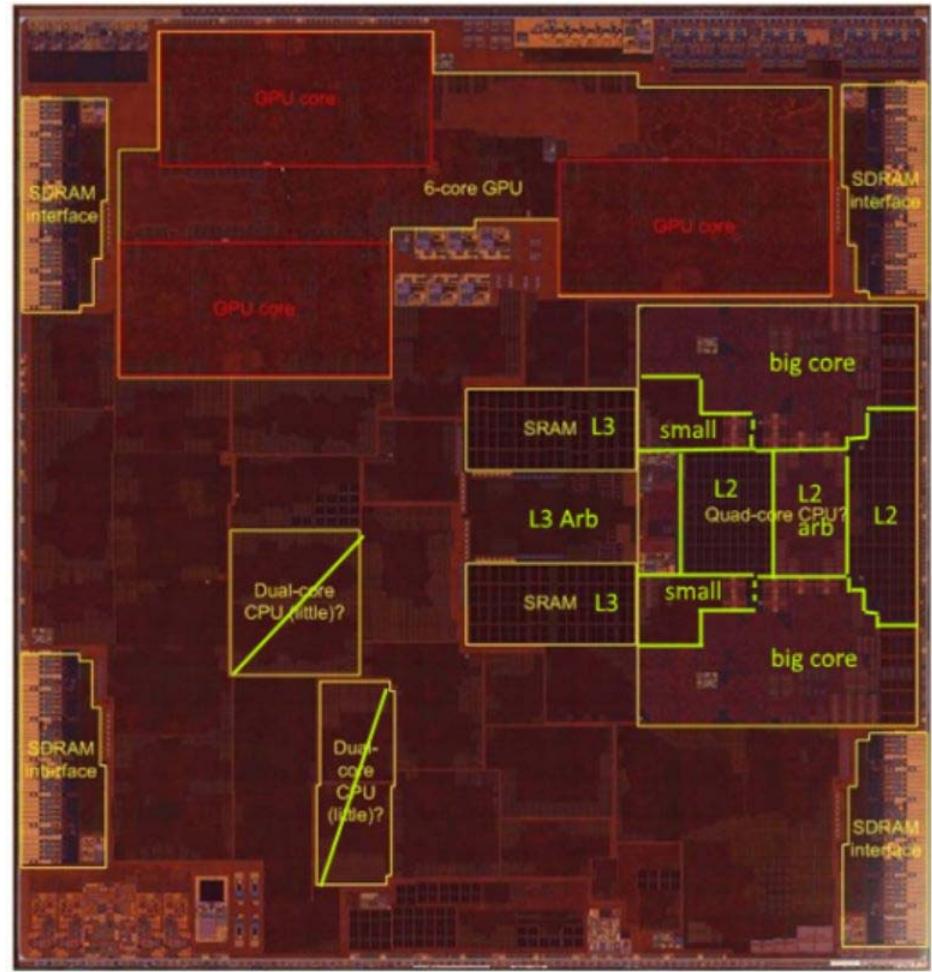
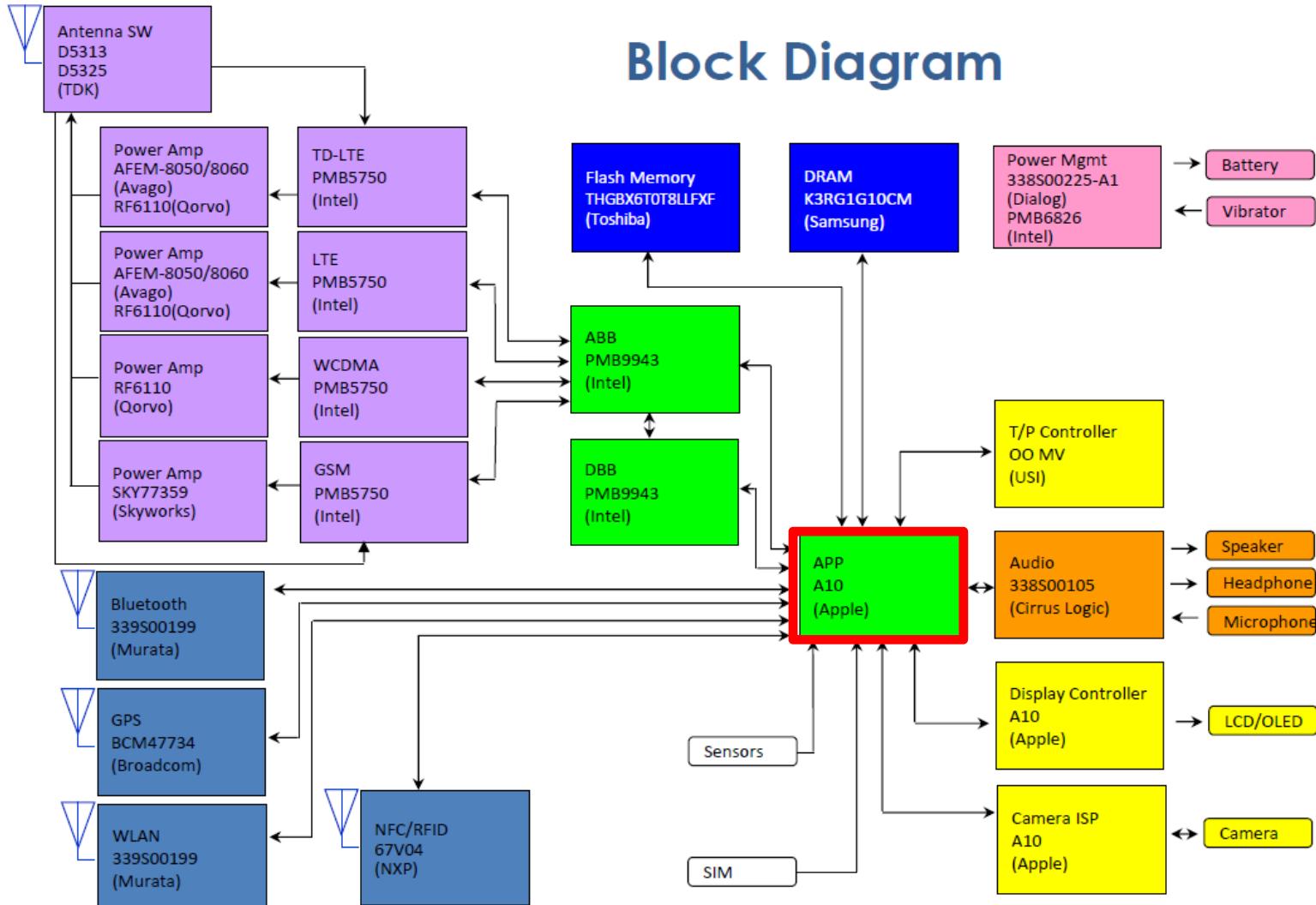


Photo from TechInsights-ChipWorks

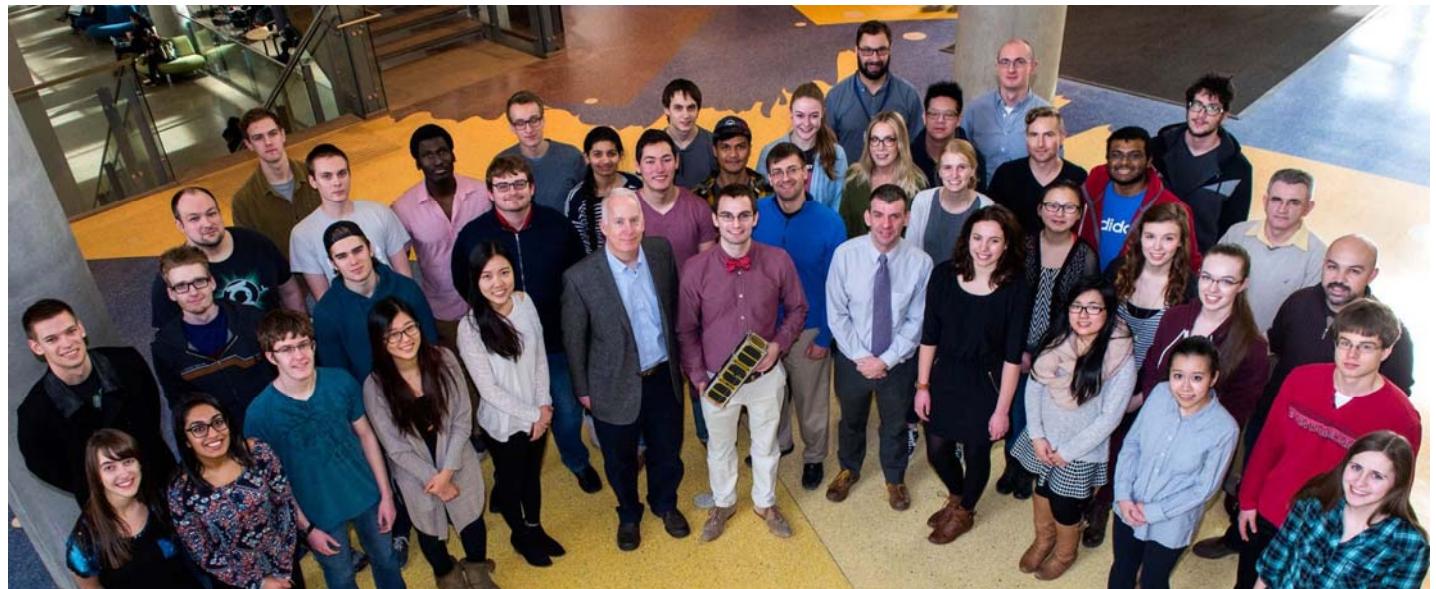
Apple iPhone 7 System Architecture



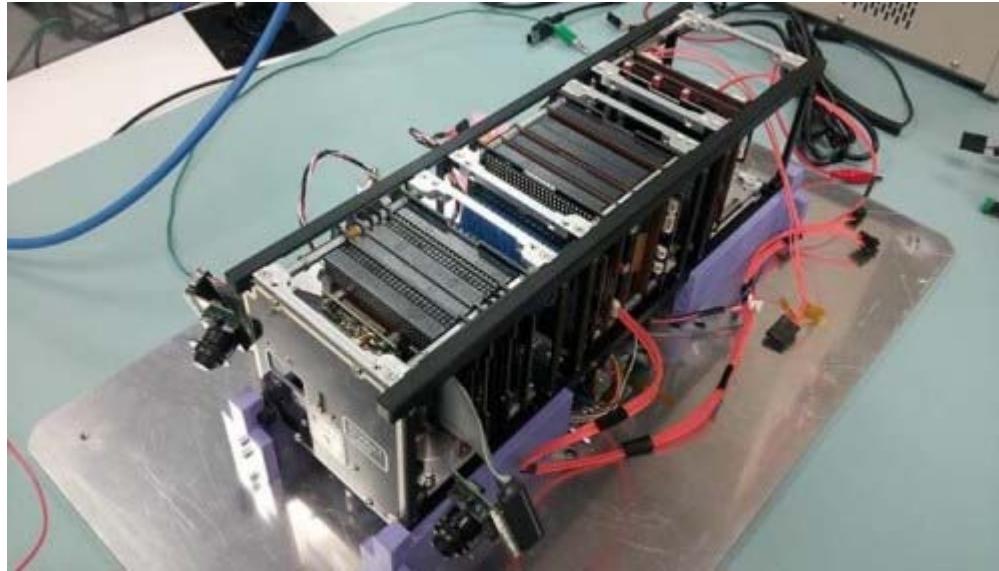
“Ex-Alta 1” CubeSat Satellite Designed at the U of A



- 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



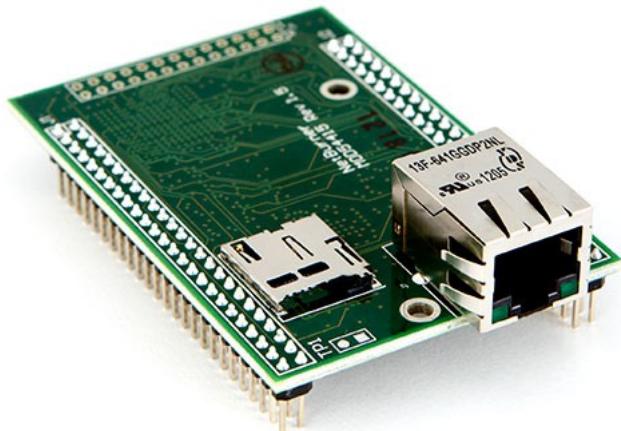
Ex-Alta 1 Embedded System Details



- 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 μ 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.



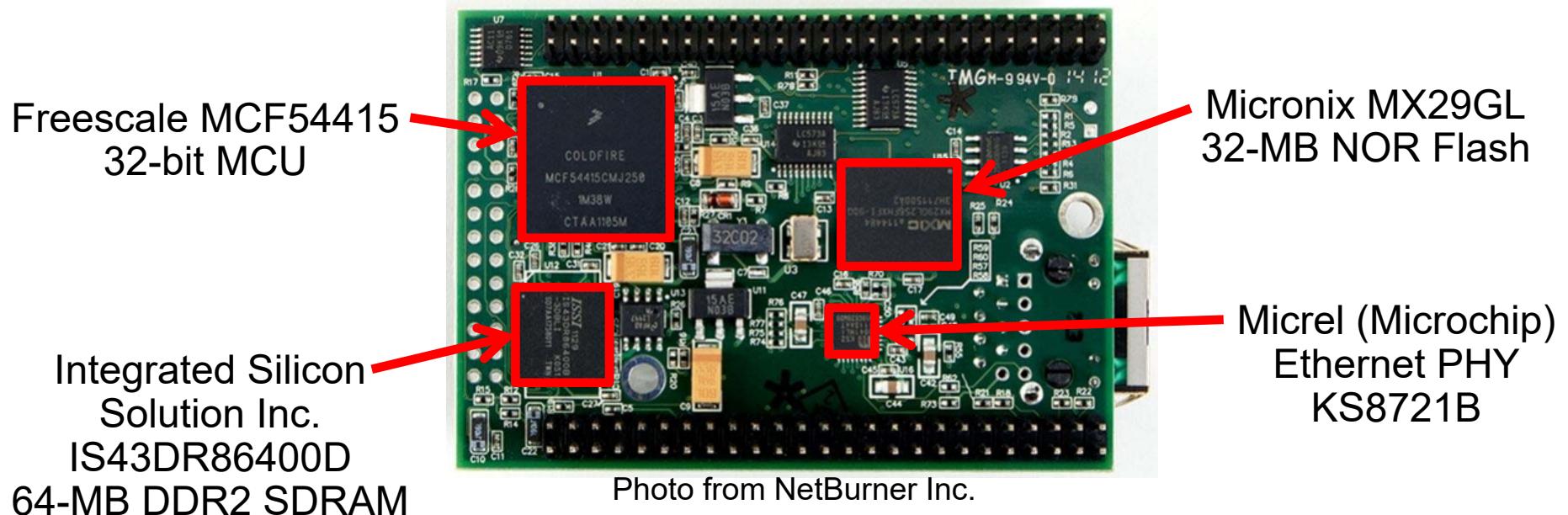
NetBurner MOD54415-100IR
Ethernet Core Module



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

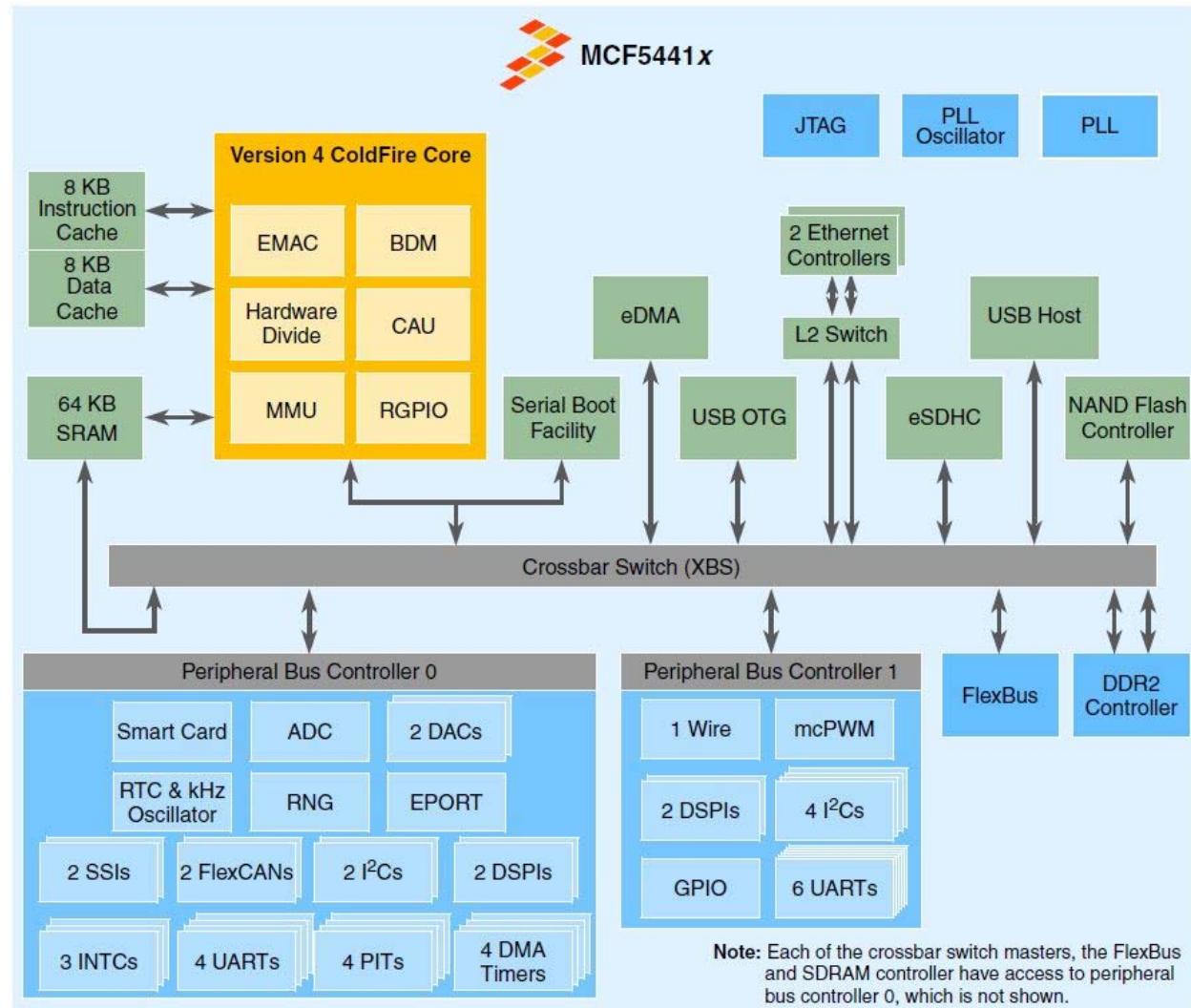


Diagram courtesy of Freescale Semiconductor Inc.

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

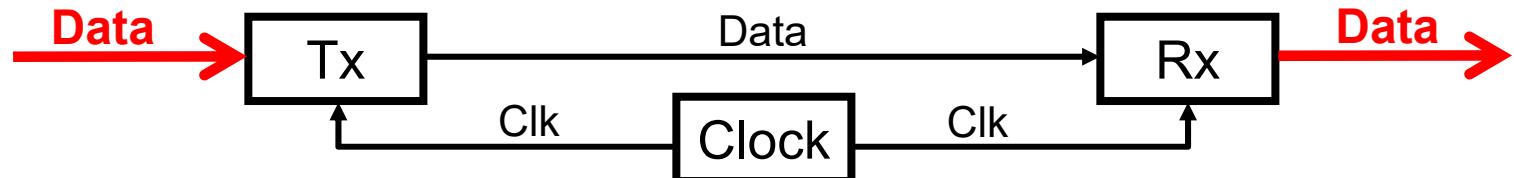
- 1) *Analog signal conditioning*
e.g. buffers/drivers, noise rejection filters, anti-aliasing filters
- 2) *Analog-to-digital* and *digital-to-analog* signal conversion
e.g. Analog inputs from sensors need to be converted into digital form to permit digital signal processing (DSP).
e.g. Digital outputs 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

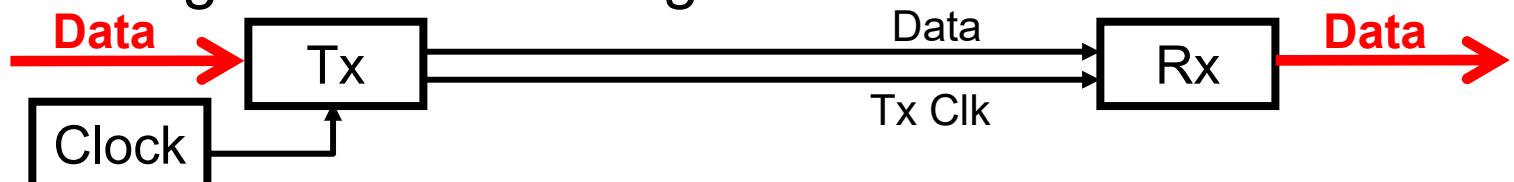
- 5) *Digital signal conversion, conditioning and isolation*
e.g. Voltage level conversion, voltage-current conversion
e.g. Optical isolation transceivers, isolation transformers
- 6) *Impedance matching (to avoid unwanted signal “ringing”)*
e.g. Parallel termination resistance at the far end of a line
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

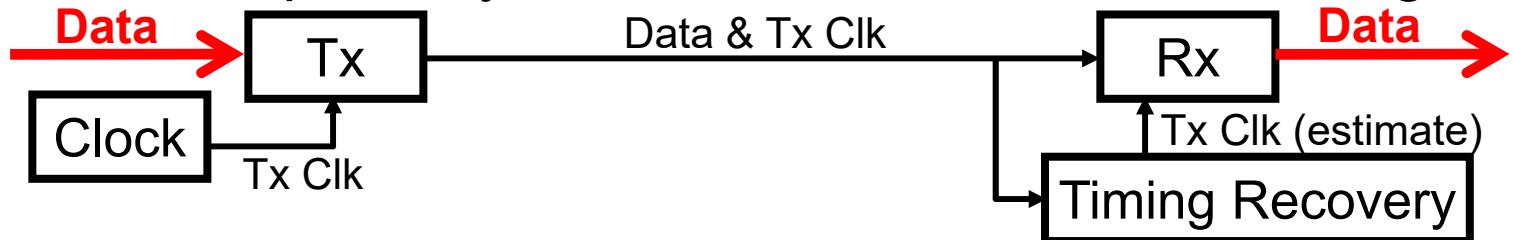
- 1) Transmitter (Tx) and receiver (Rx) clock signals are both directly derived from the same physical clock signal.



- 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.

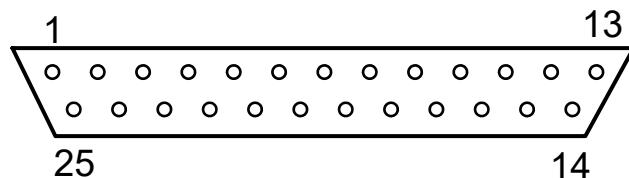
Semisynchronous Digital Interfaces

- 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



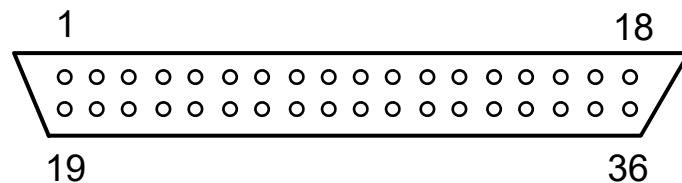
- Pin 1: ground (GND)
- Pin 2: *transmit data* (TxD)
- Pin 3: *receive data* (RxD)
- Pin 4: request to send (RTS)
- Pin 5: clear to send (CTS)
- Pin 6: data set ready (DSR)
- Pin 20: data terminal ready (DTR)

- 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

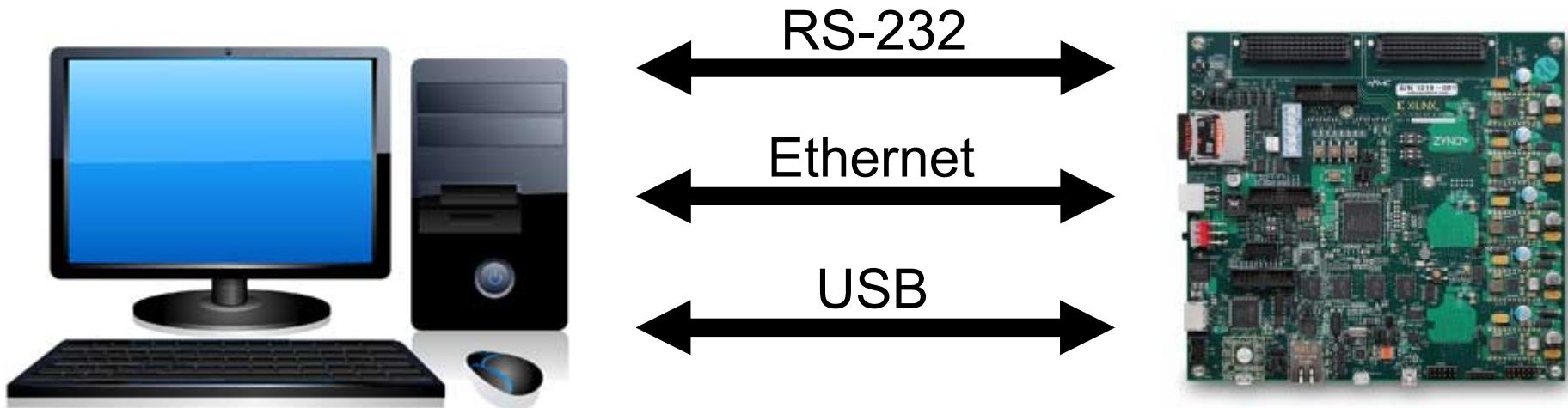


Pin 1: data strobe
Pins 2-9: 8 *data bits*
Pin 10: acknowledge
Pin 11: busy
Pin 12: paper end
Pin 13: select
Pin 17: ground
Pin 18: +5 V
Pin 32: fault detected

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 routine and 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.

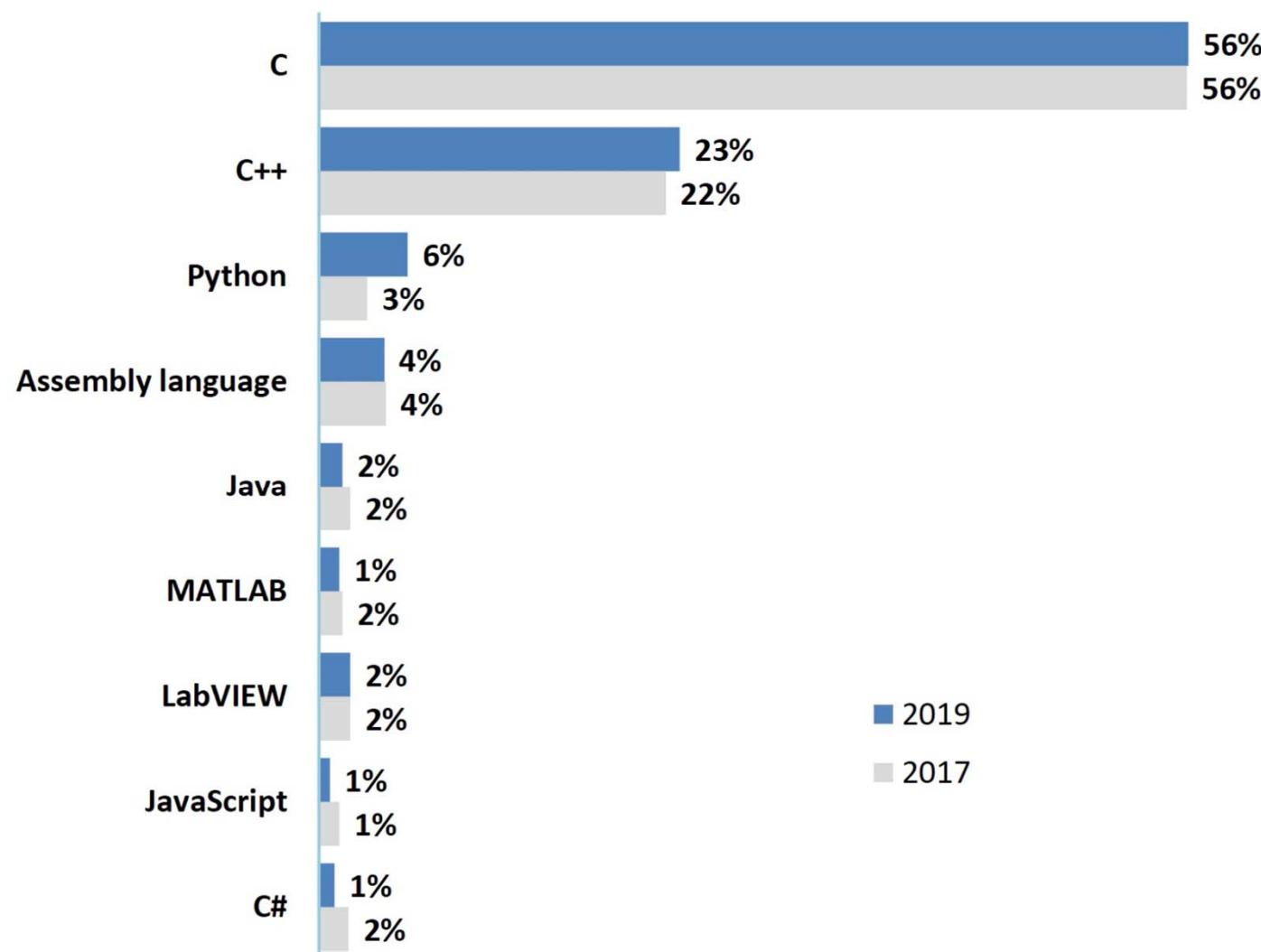
Software Engineering

- “*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

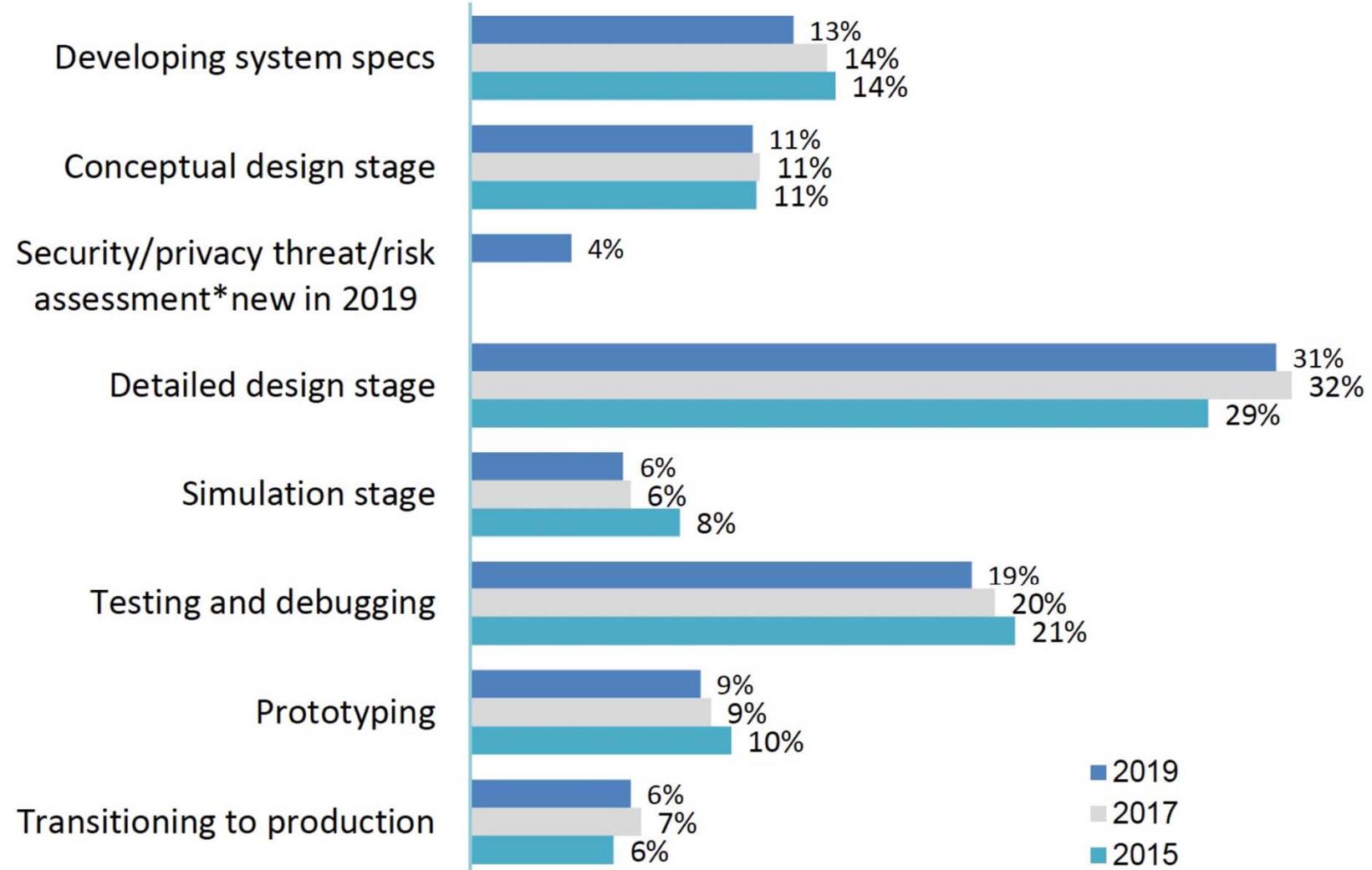
- The C programming language is one of the most widely used languages. Most computer architectures have C compilers.
- Developed by Dennis Ritchie at Bell Laboratories over 1969-1973, C was intended to provide both high-level language constructs (e.g., loops, pointers, I/O) as well as fairly direct access 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,

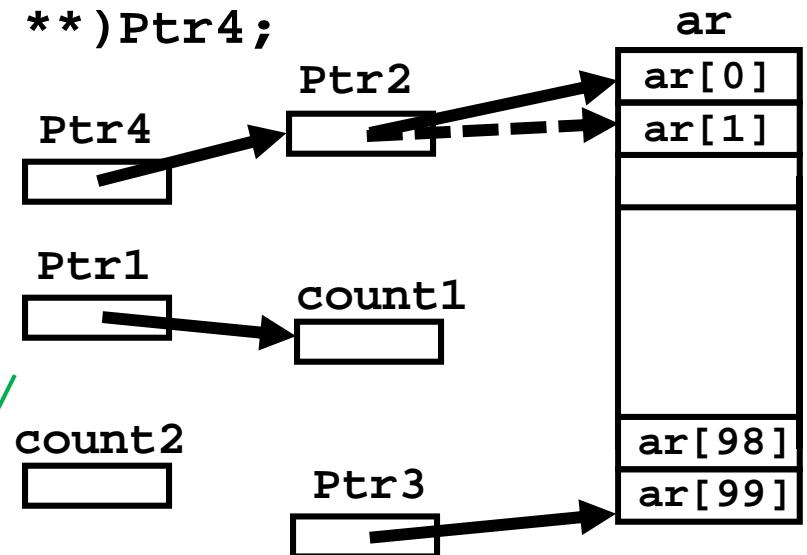
```
printf("hello, " );           /* string has 8 char's */  
printf("world\n");           /* string has 7 char's */
```

- 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 will 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.

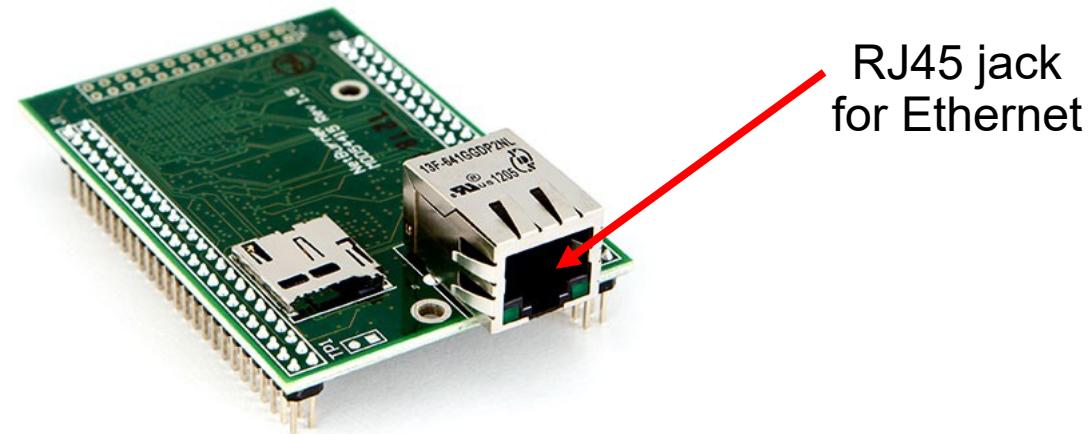
Freescale™ and ColdFire® are registered trademarks of Freescale Semiconductor, Inc.

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

Microprocessor-related Terminology

- A **microprocessor** (μ 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** (μ 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).

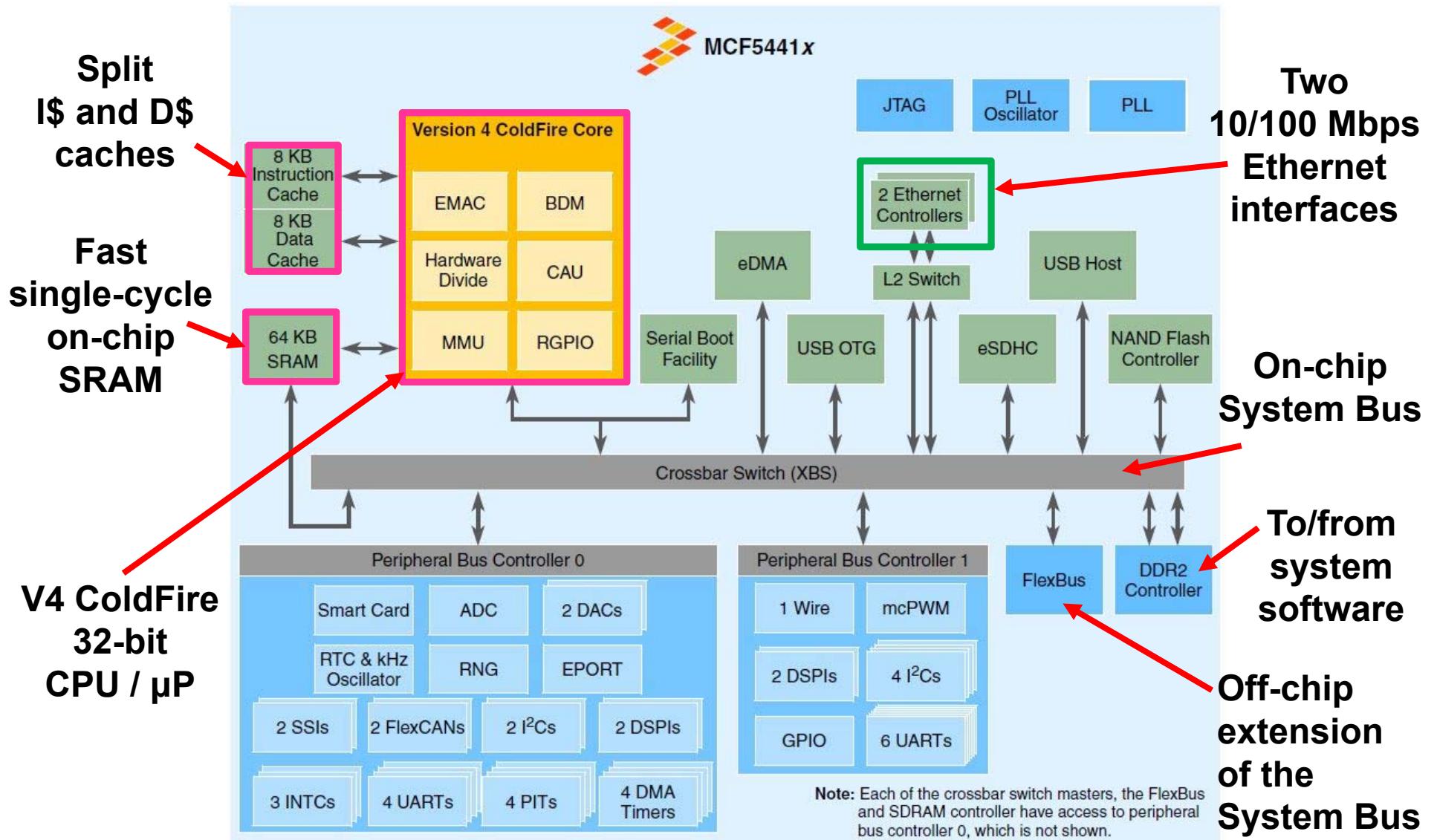
The ECE 315 Microcomputer: NetBurner MOD5441X



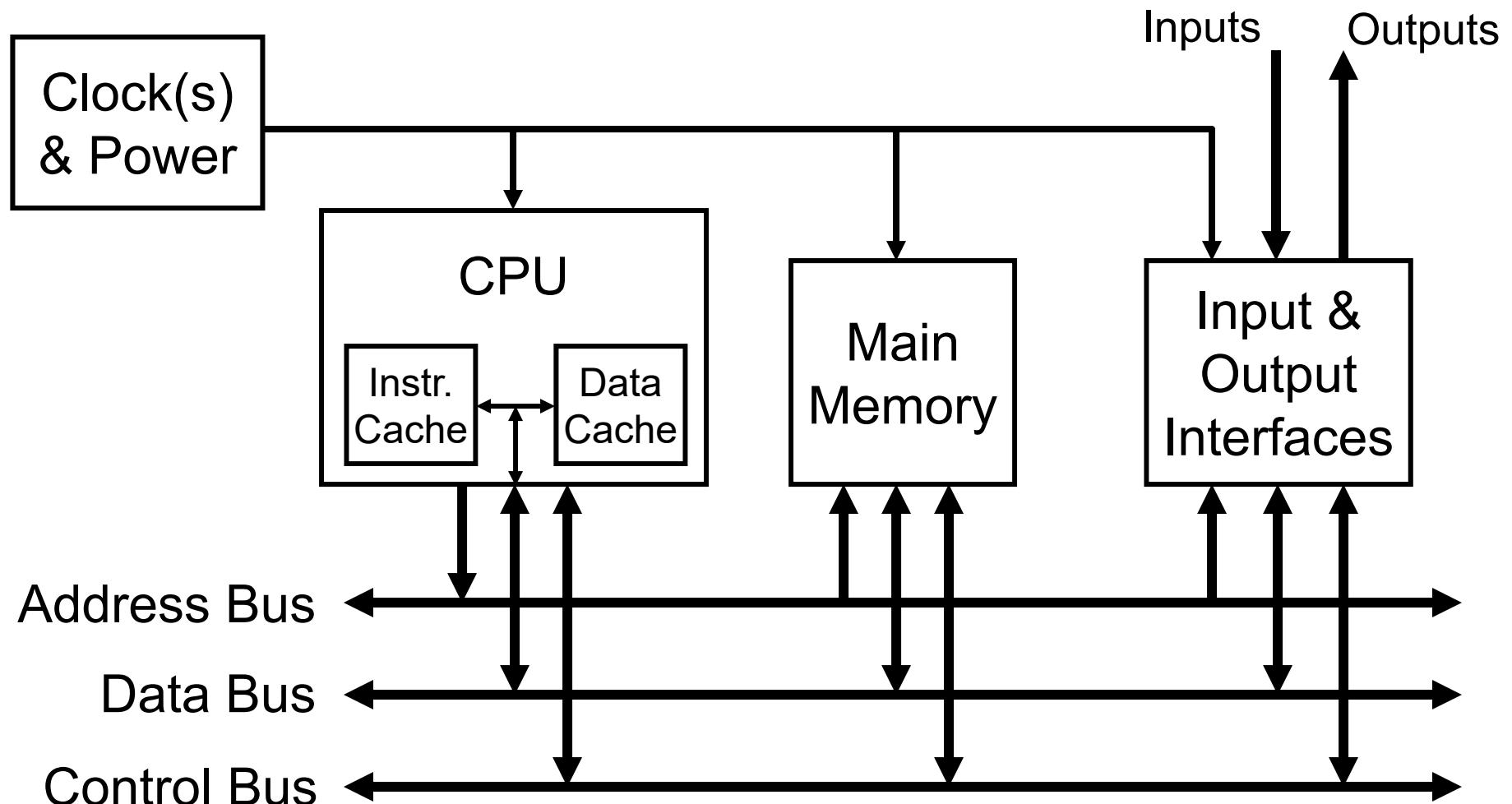
NetBurner MOD54415-100IR μ C
“Ethernet Core Module”

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

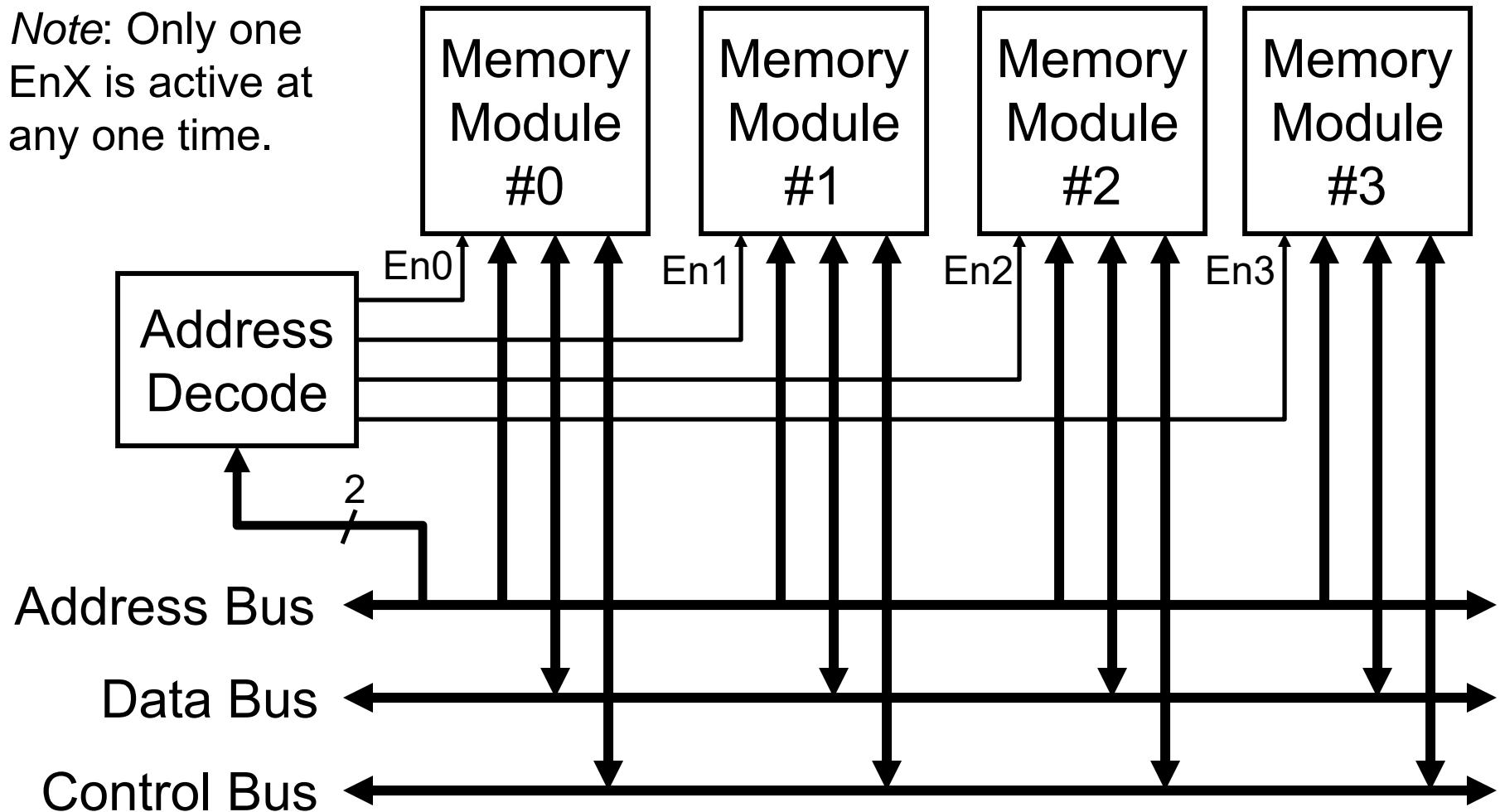
- A memory holds binary information, which consists of both *program instructions* and *data*. (In memory, all information is stored as 0's and/or 1's. Hexadecimal notation is often used to make binary values easier for humans to read and write.)
- Information “words” are identified in a memory using 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

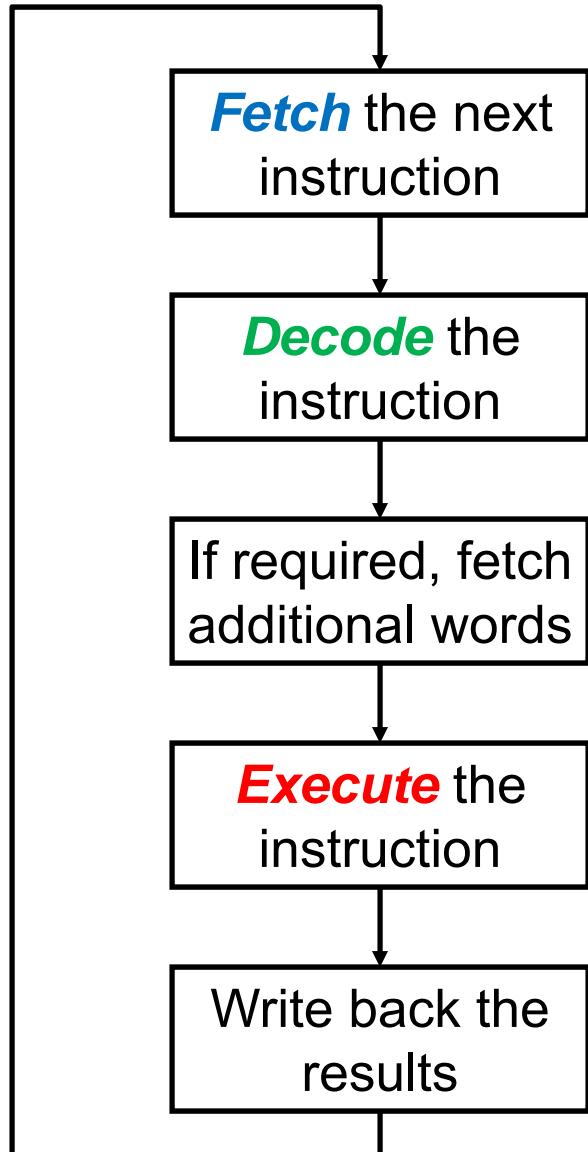
- Semiconductor memory is either *volatile* or *nonvolatile*.
- Volatile memory can retain stored data only as long as power is supplied.
 - *Static RAM* (SRAM) is fast but it is relatively power-hungry and expensive per bit. Smaller capacity than DRAM.
 - *Dynamic RAM* (DRAM) is cheap but relatively 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_x) 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.

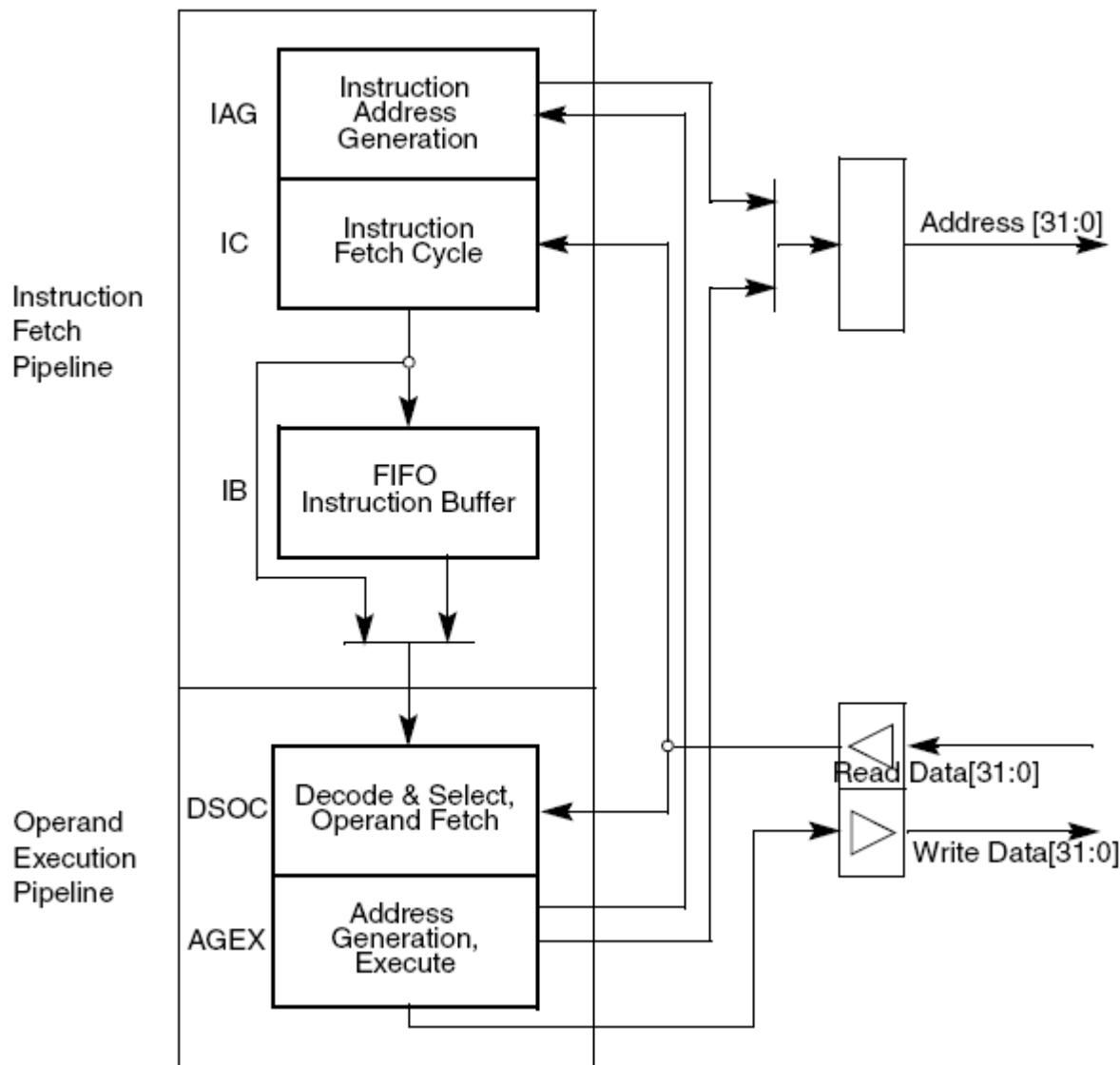


The Instruction Fetch-Decode-Execute Cycle



- 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



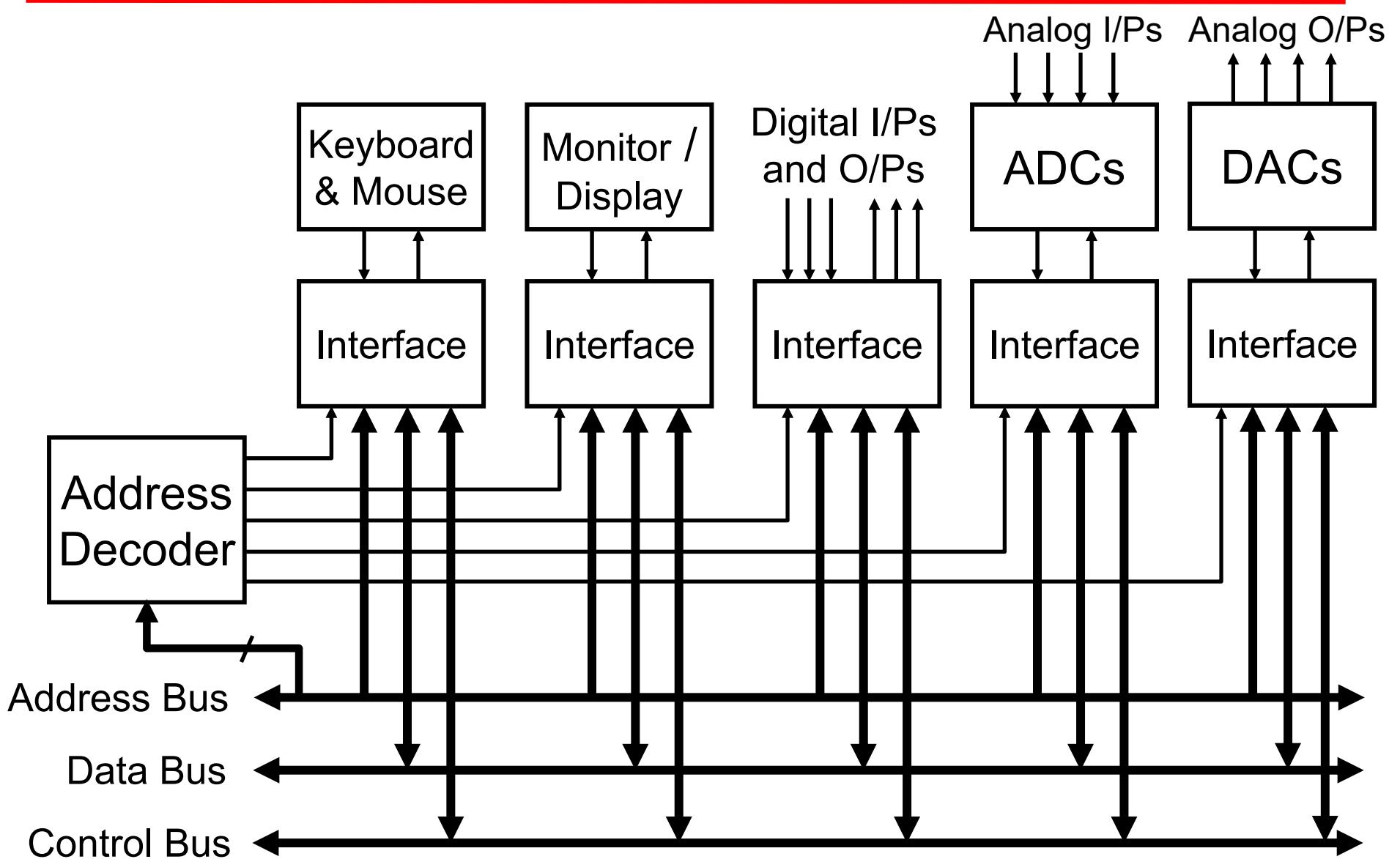
Many modern CPUs, such as the Freescale MCF54415, have the ability to “prefetch” instructions from slow DRAM and to store them in a fast *first-in first-out* (FIFO) instruction buffer.

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

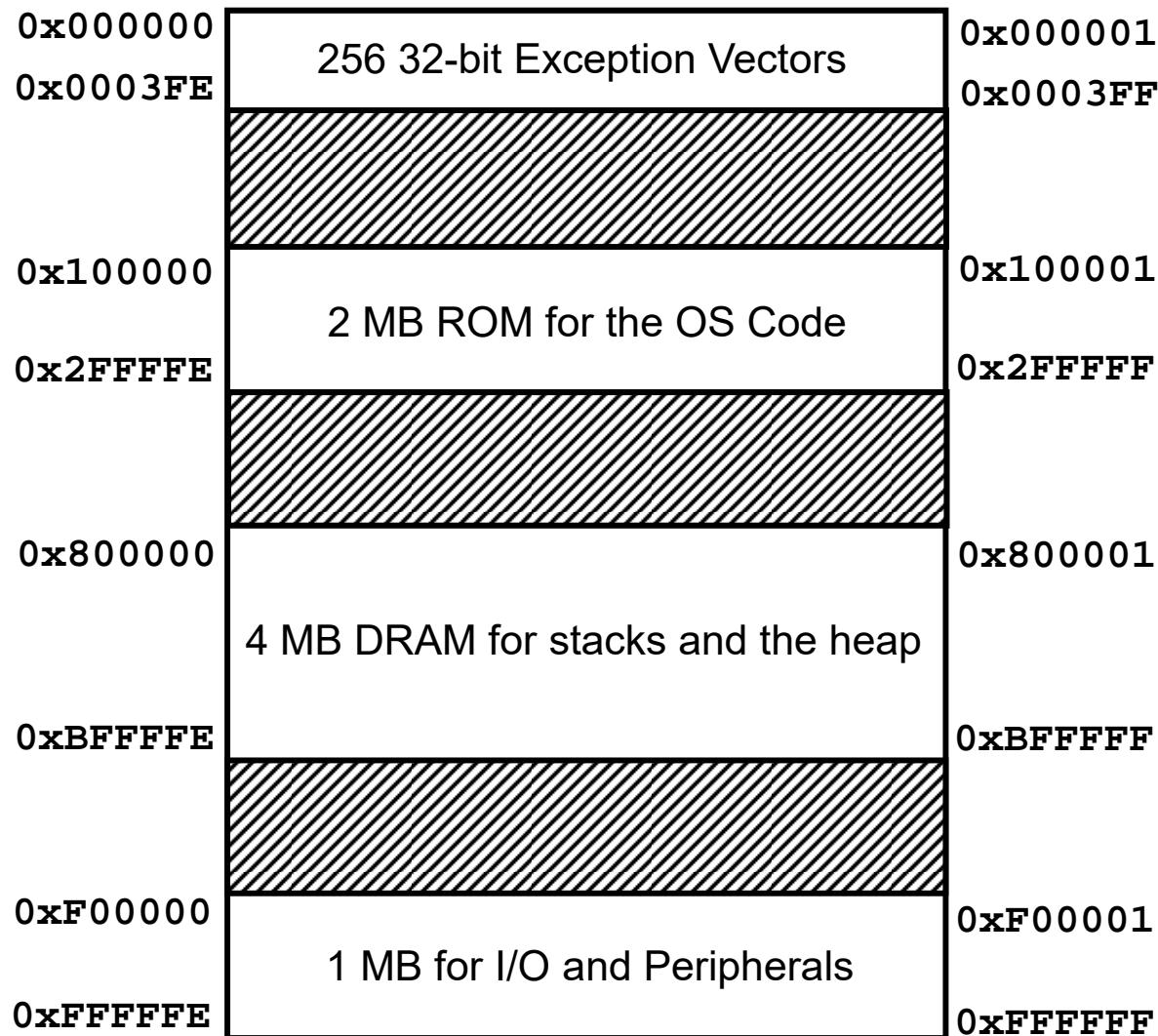
Input and Output Devices

- 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.

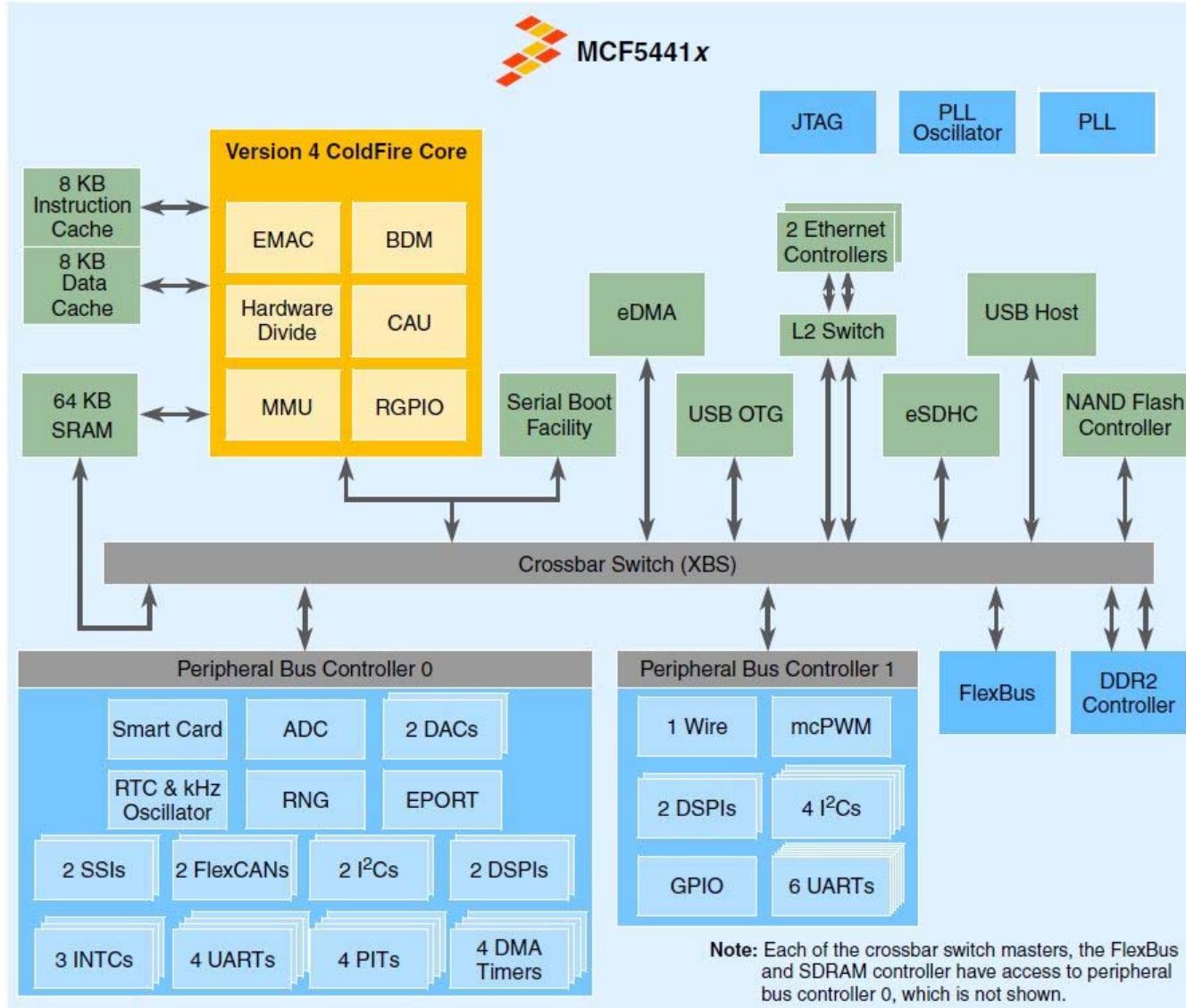
The ColdFire Family of 32-bit CPUs

- 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 + I²C 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)

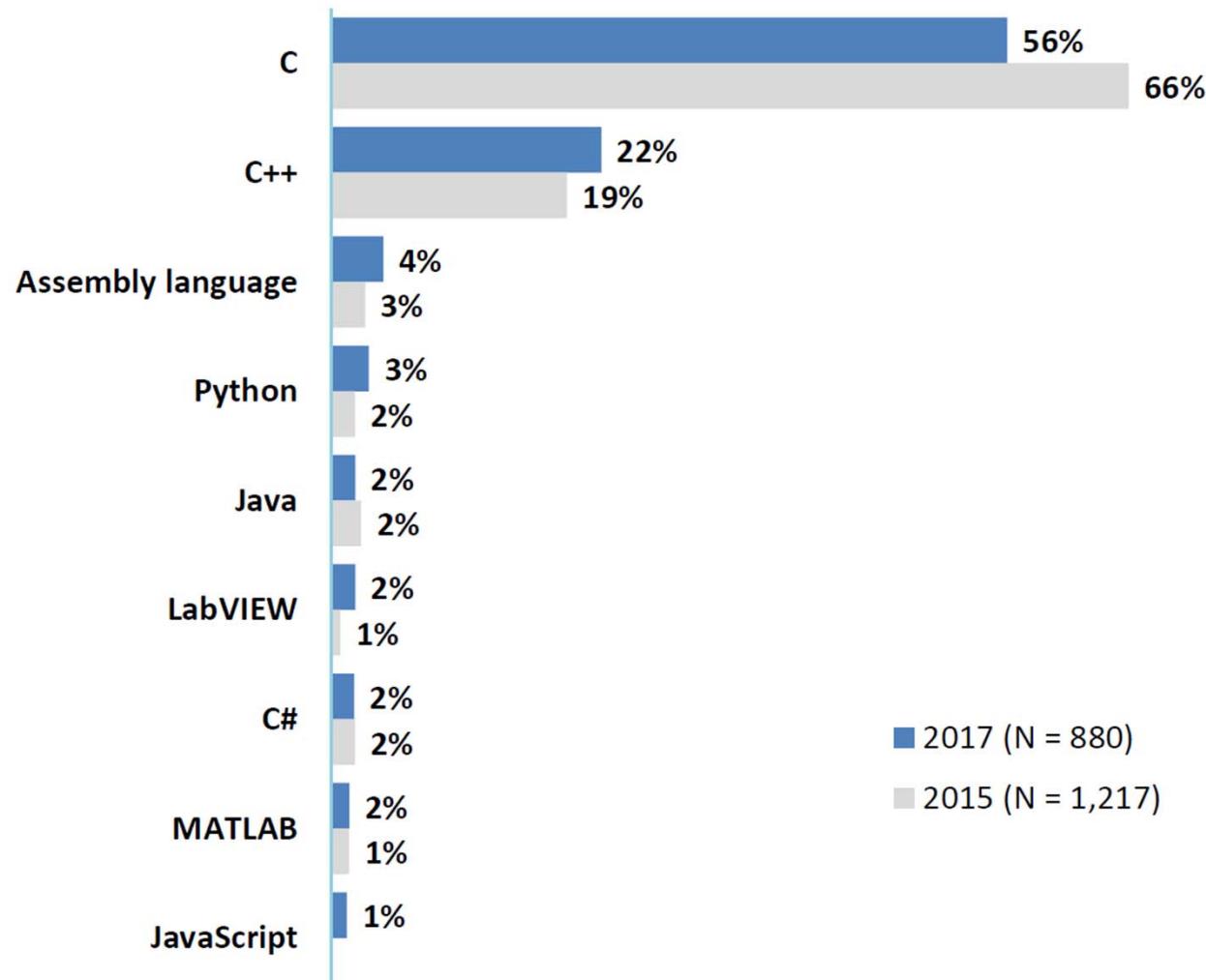
Architecture of the MCF5441x MCUs



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



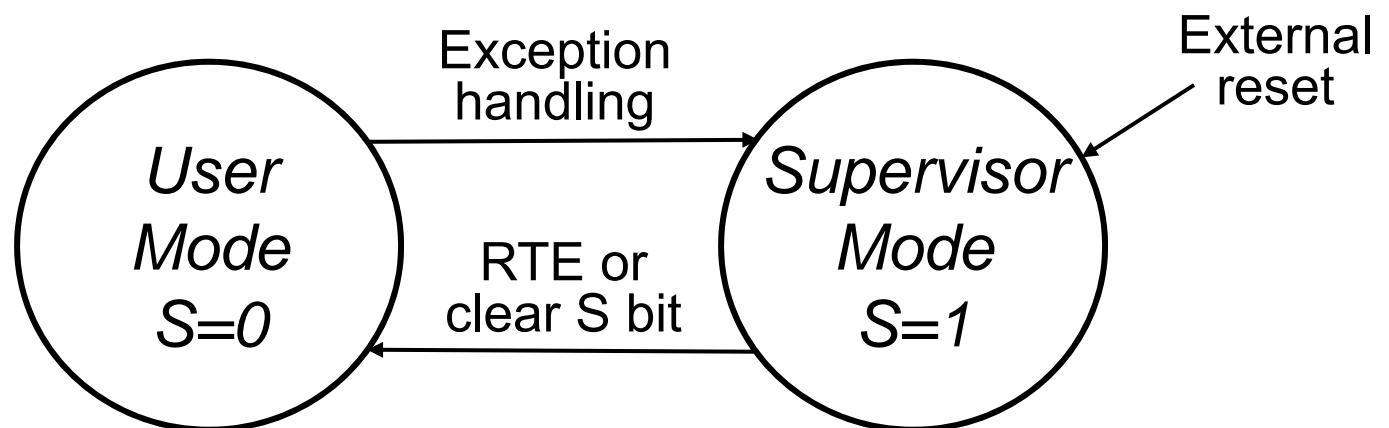
Courtesy of AspenCore's 2017 Embedded Markets Study

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

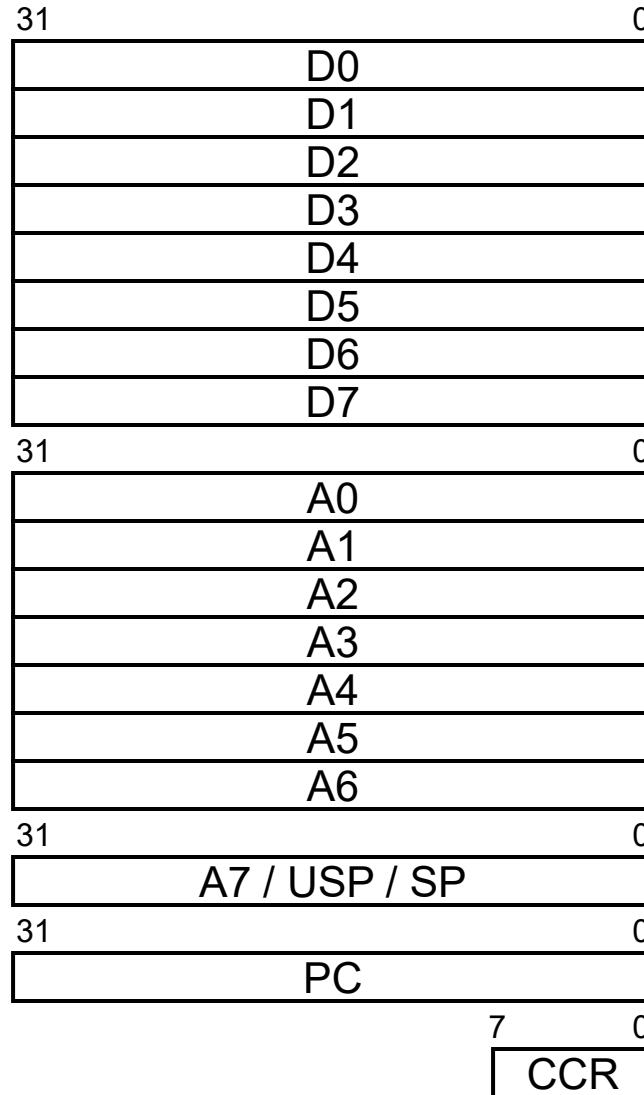
- When a CPU is reset by asserting the external hardware reset, it will start executing instructions in supervisor mode, with the S bit in the status register (SR) set to 1.
- All of the software could be executed in supervisor mode, but it is often desirable to restrict the privileges of at least some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

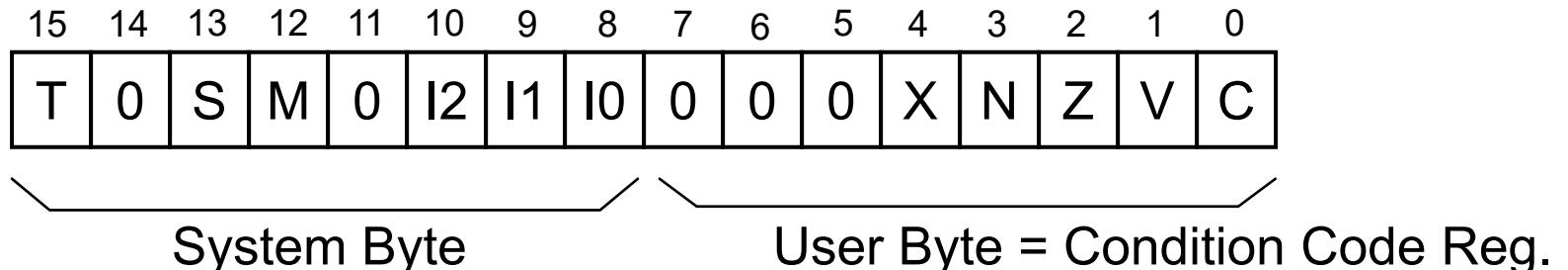
Copyright © 2008 by Freescale Semiconductor, Inc.

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)

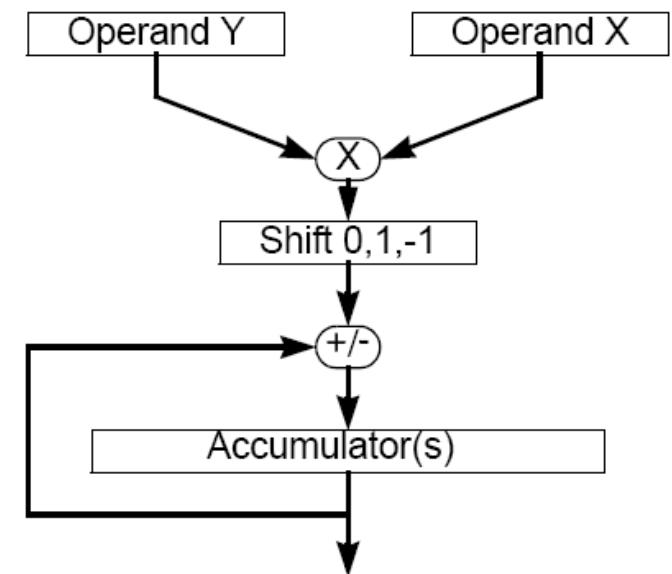
Common Multiply-Accumulate (MAC) Calculations

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

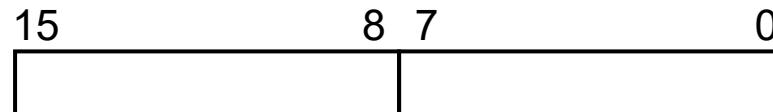
31:24	23:16	15:8	7:0	Mnemonic
MAC Status Register				MACSR
MAC Accumulator 0				ACC0
MAC Accumulator 1				ACC1
MAC Accumulator 2				ACC2
MAC Accumulator 3				ACC3
Extensions for ACC0 and ACC1				ACCext01
Extensions for ACC2 and ACC3				ACCext23
MAC Mask Register				MASK

Copyright © 2008 by Freescale Semiconductor, Inc.

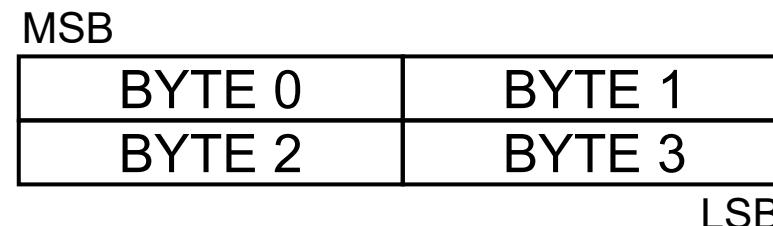
The MASK register is used to optionally constrain the address in MAC instructions to simplify accesses to data stored in circular queues (discussed later in the course).

Formats of the Native Data Types in Memory

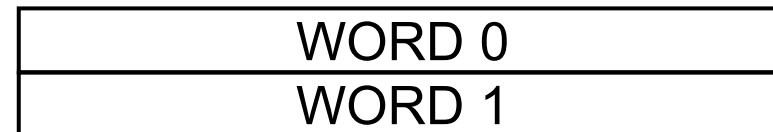
1) Bits



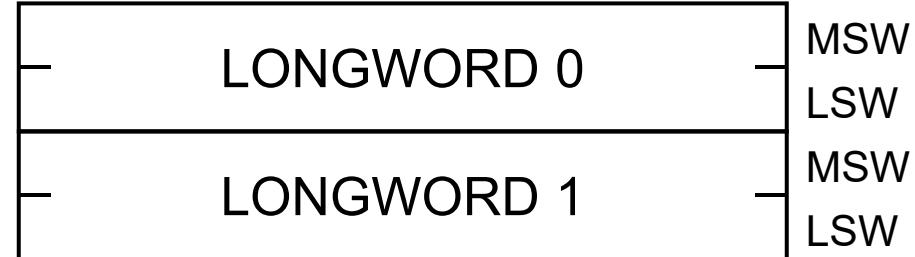
2) Bytes (8 bits)



3) Words (16 bits)



4) Longwords (32 bits)



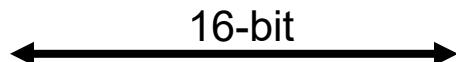
Note: The Binary-Coded Decimal (BCD) native data type that was supported in the M68000 and CPU32 is not present in the ColdFire.

Aside: Byte Address Order Conventions

Big Endian

(IBM, Motorola/Freescale, HP, Sun)

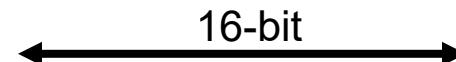
MSB	LSB
BYTE 0	BYTE 1
BYTE 2	BYTE 3
BYTE 4	BYTE 5
BYTE 6	BYTE 7



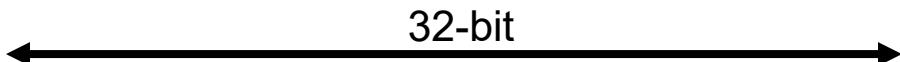
Little Endian

(Intel, AMD, early ARM CPUs)

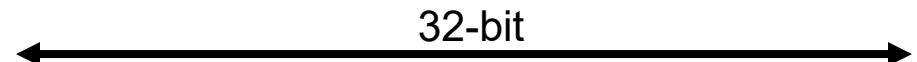
MSB	LSB
BYTE 1	BYTE 0
BYTE 3	BYTE 2
BYTE 5	BYTE 4
BYTE 7	BYTE 6



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



MSB	BYTE 3	BYTE 2	BYTE 1	BYTE 0	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 ₁₆ ,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 ₈ ,An,Xi*SF)	110	reg. no.	X	X	X	X
Program Counter Indirect with Displacement	(d ₁₆ ,PC)	111	010	X	X	X	—
Program Counter Indirect with Scaled Index and 8-Bit Displacement	(d ₈ ,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

($<\text{d16}>$, 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 An to give the address of the operand in memory.

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

($<\text{d8}>$, 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 ($\text{Xn.s} = \text{An.W}$, An.L , Dn.W or Dn.L) are added to the address from the given register An to give the address of the operand in memory.

Compound Address Modes (cont'd)

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

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

($<\text{d8}>,\text{PC},\text{Xn.s} * \text{scale}$) 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 ($\text{Xn.s} = \text{An.L}, \text{An.W}, \text{Dn.L}$ 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

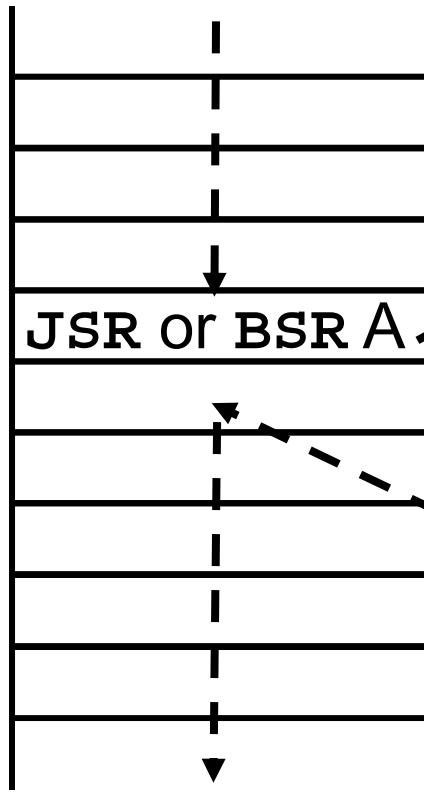
CPUSHL is a way of invalidating a single cached instruction. It is faster than starting a full cache invalidation sequence. CPUSHL is a privileged instruction.

Subroutines

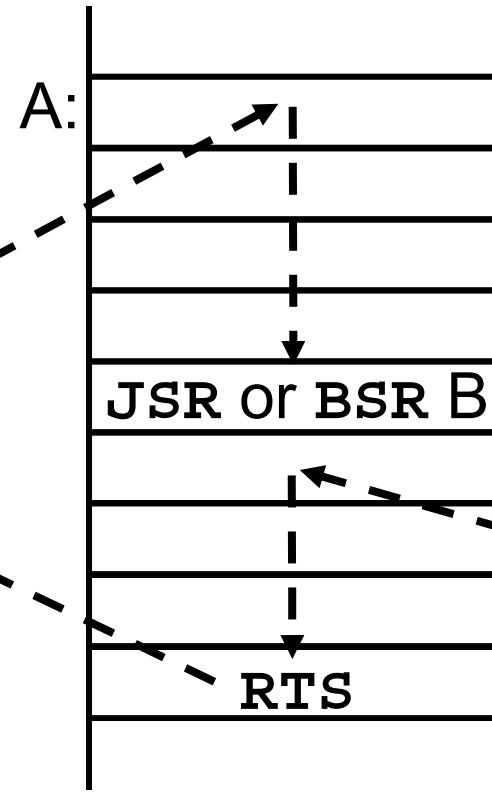
- A subroutine is a short program segment that can be called by any other program, including itself.
- Subroutines are typically used to encapsulate frequently used code.
- Subroutines provide more efficient use of memory since the enclosed code does not have to be repeated in several places in other 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

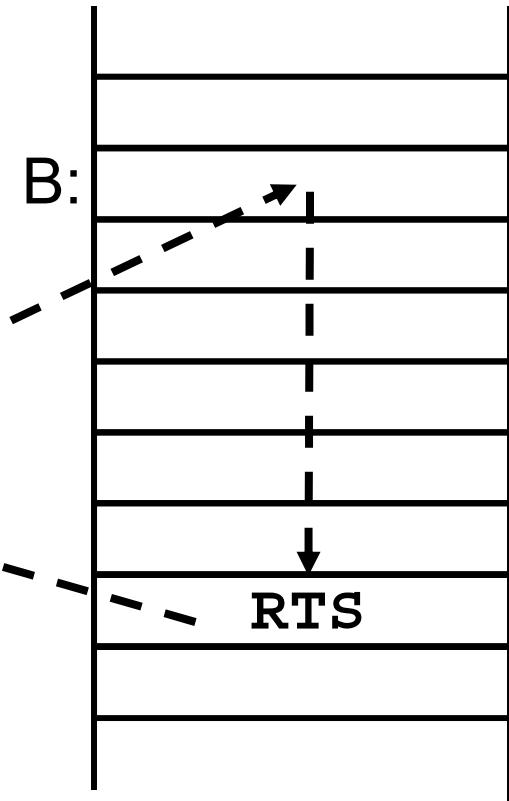
Main Program



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.

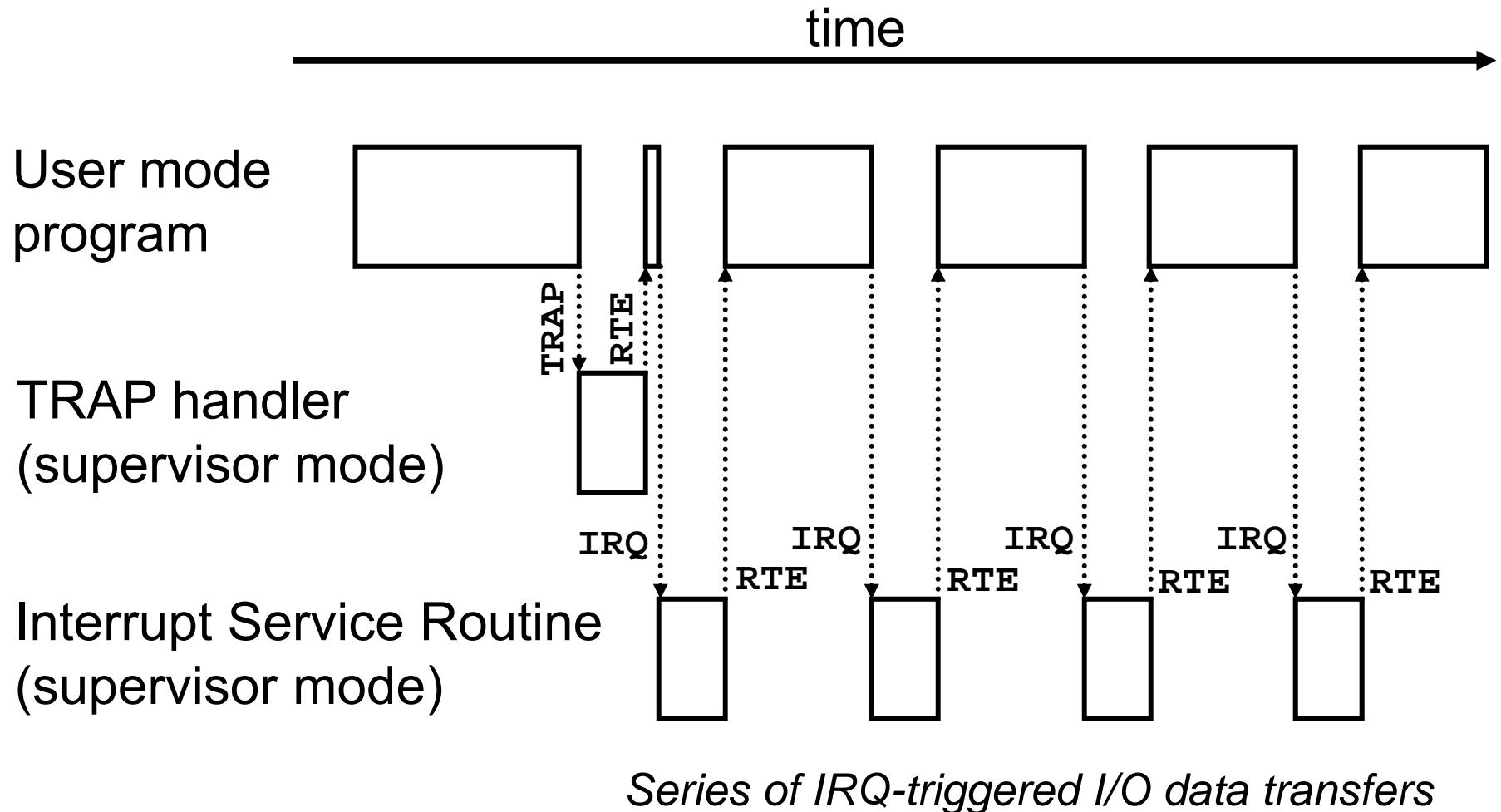
Exceptions, Traps, and Interrupts

- 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

Interrupt Mask Value	Levels Disabled (Masked)
I₂ I₁ I₀	
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

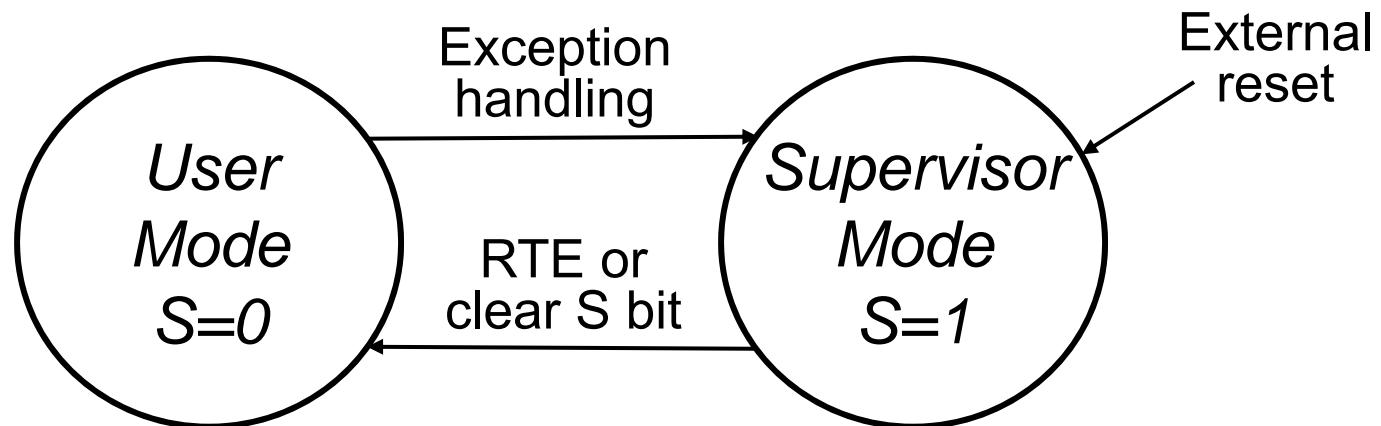
3-70

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

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 3rd party)
 - the file system (often from a 3rd 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 */

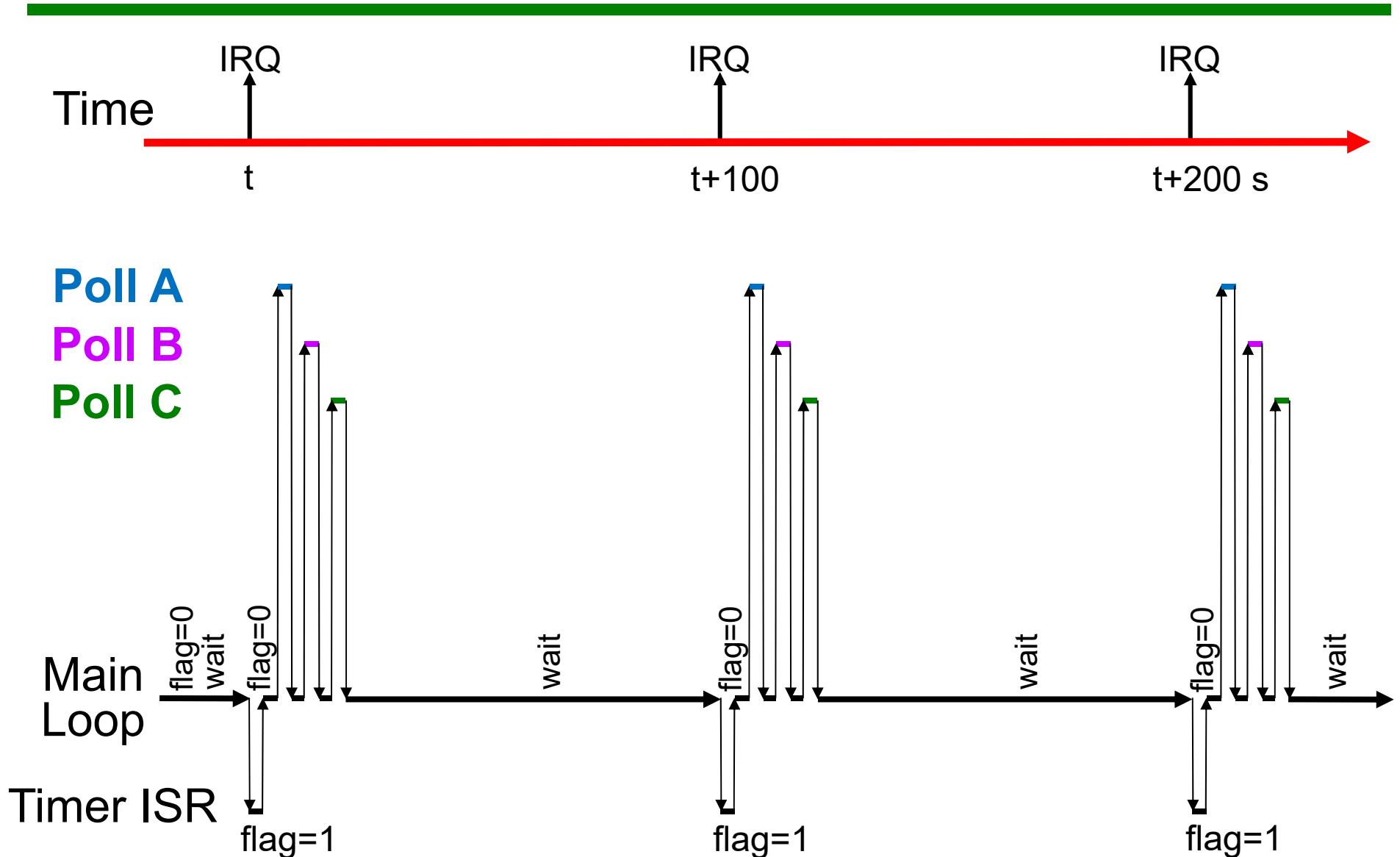
    poll_A();

    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) {

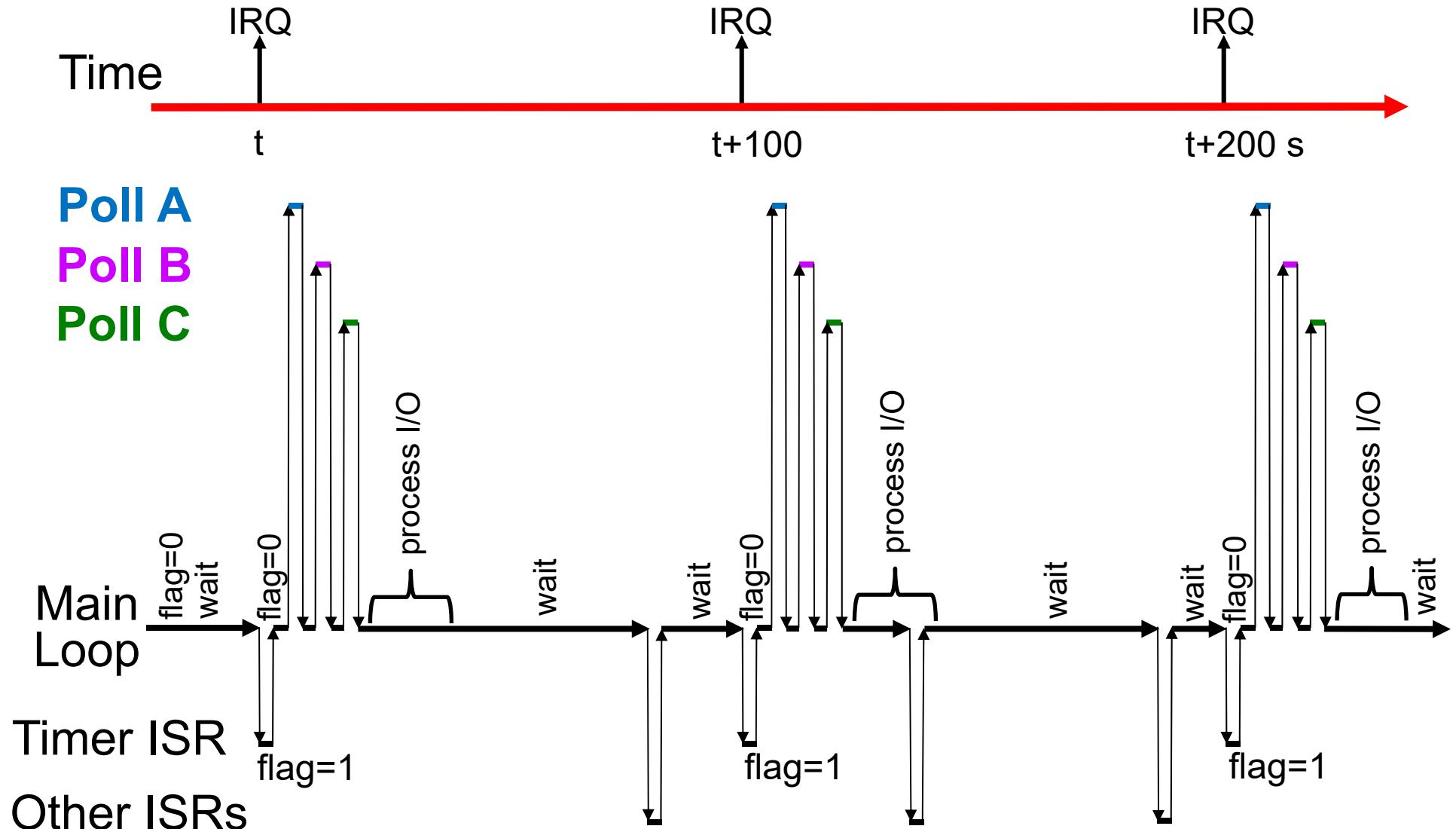
    /* wait for the timer ISR to set flag to 1 */
    while ( !flag ) {}

    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)

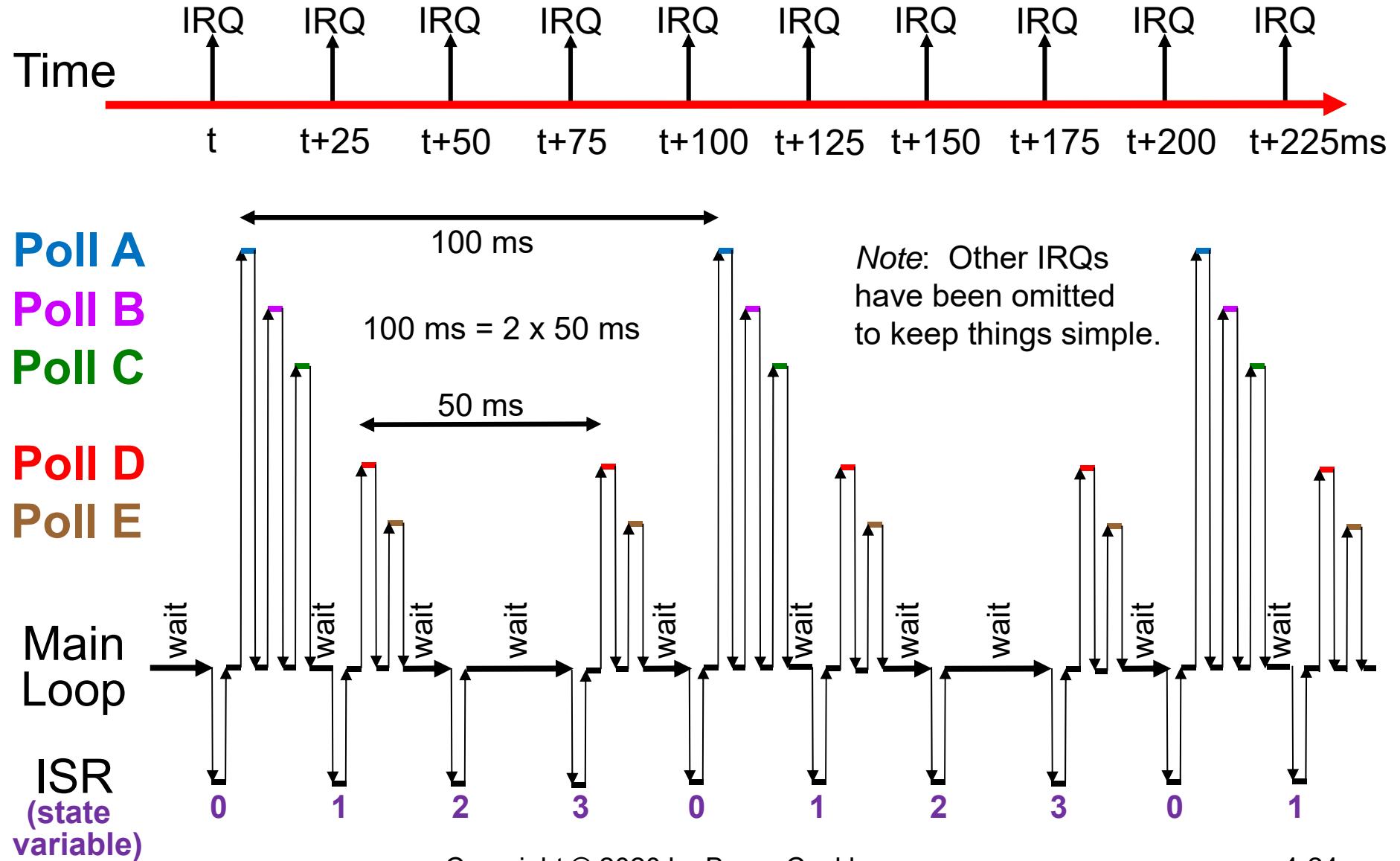
```
void function do_100_ms_loop() { poll_A(); poll_B(); poll_C(); }
void function do_50_ms_loop() { poll_D(); poll_E(); }

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.

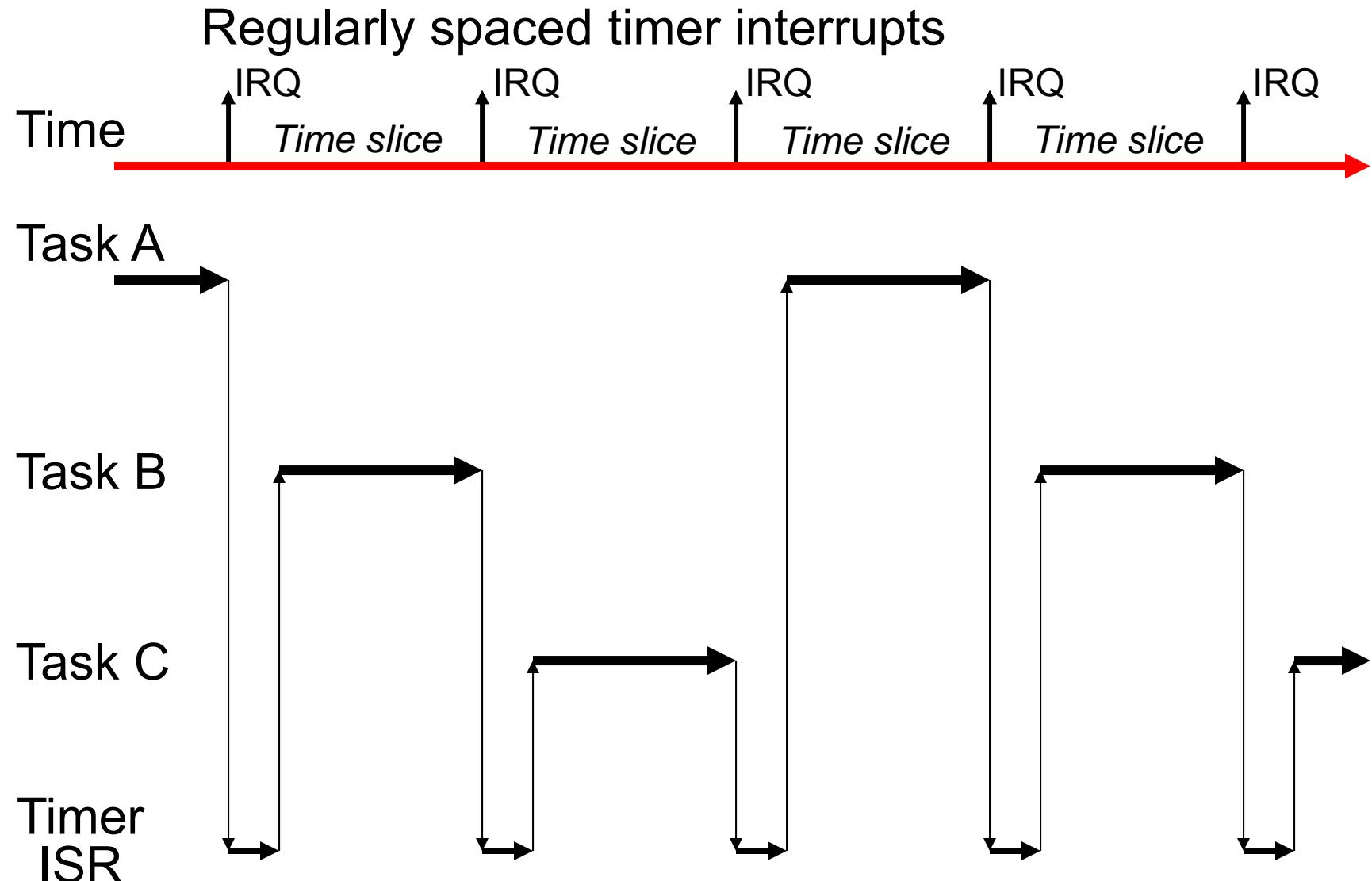
Tasks

- 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?

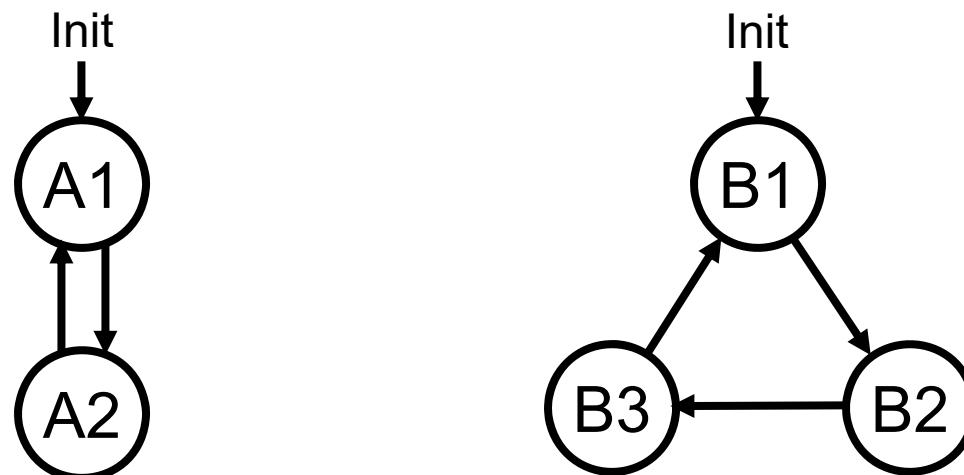
- The time slicing multitasking architecture effectively creates multiple virtual CPUs, with one such CPU for each task.
- Time slicing is a simple way of *sharing one CPU* for the execution of *multiple non-hard-real-time software tasks*.
- The different tasks can be allocated different proportions 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

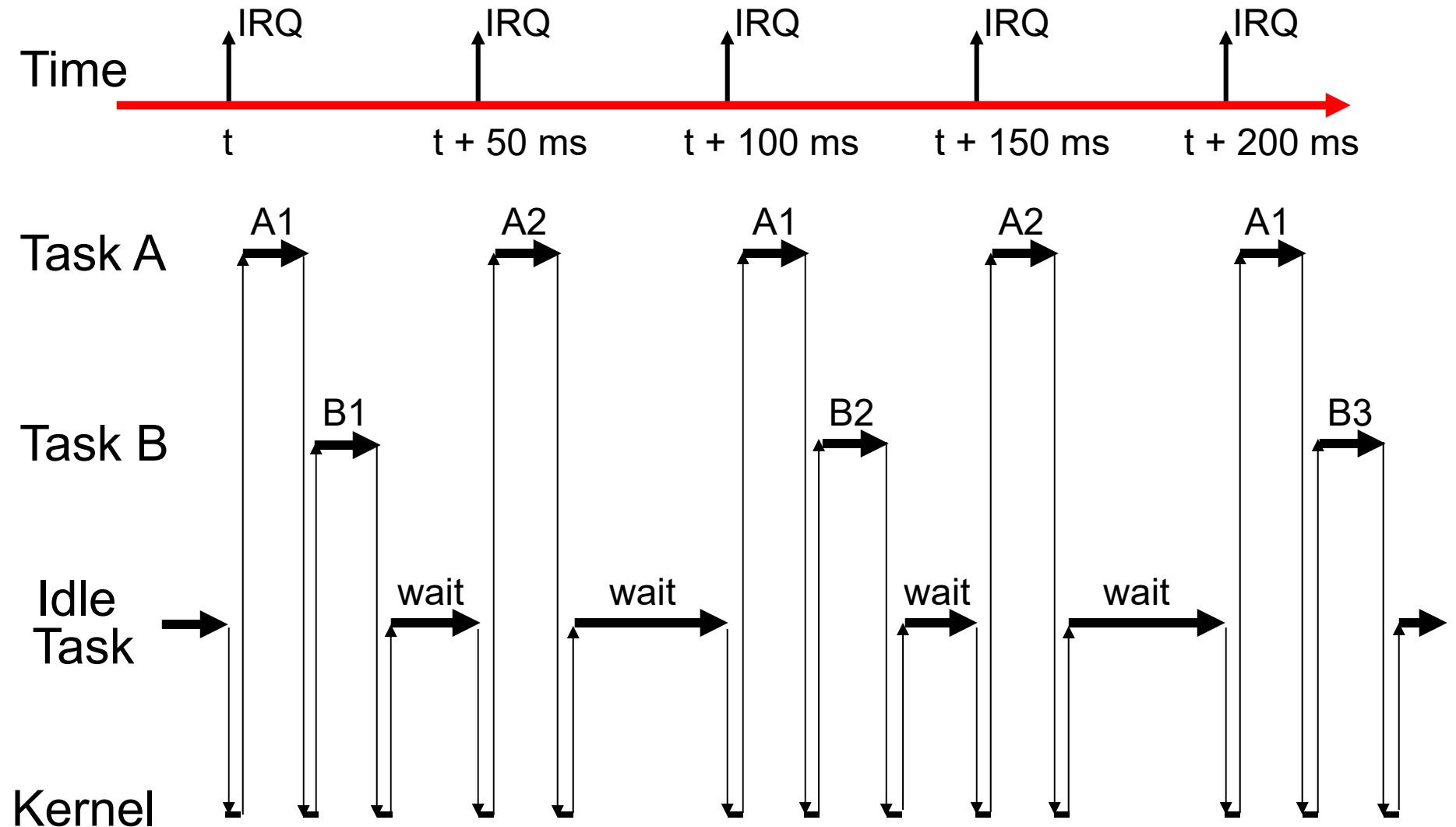
- In some applications, it is natural and/or convenient to partition each task into a collection of interconnected *states*.
- Each state corresponds to a code segment that needs to be executed in that state (e.g., control and input/output actions).
- After a state code segment has been executed, the 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.

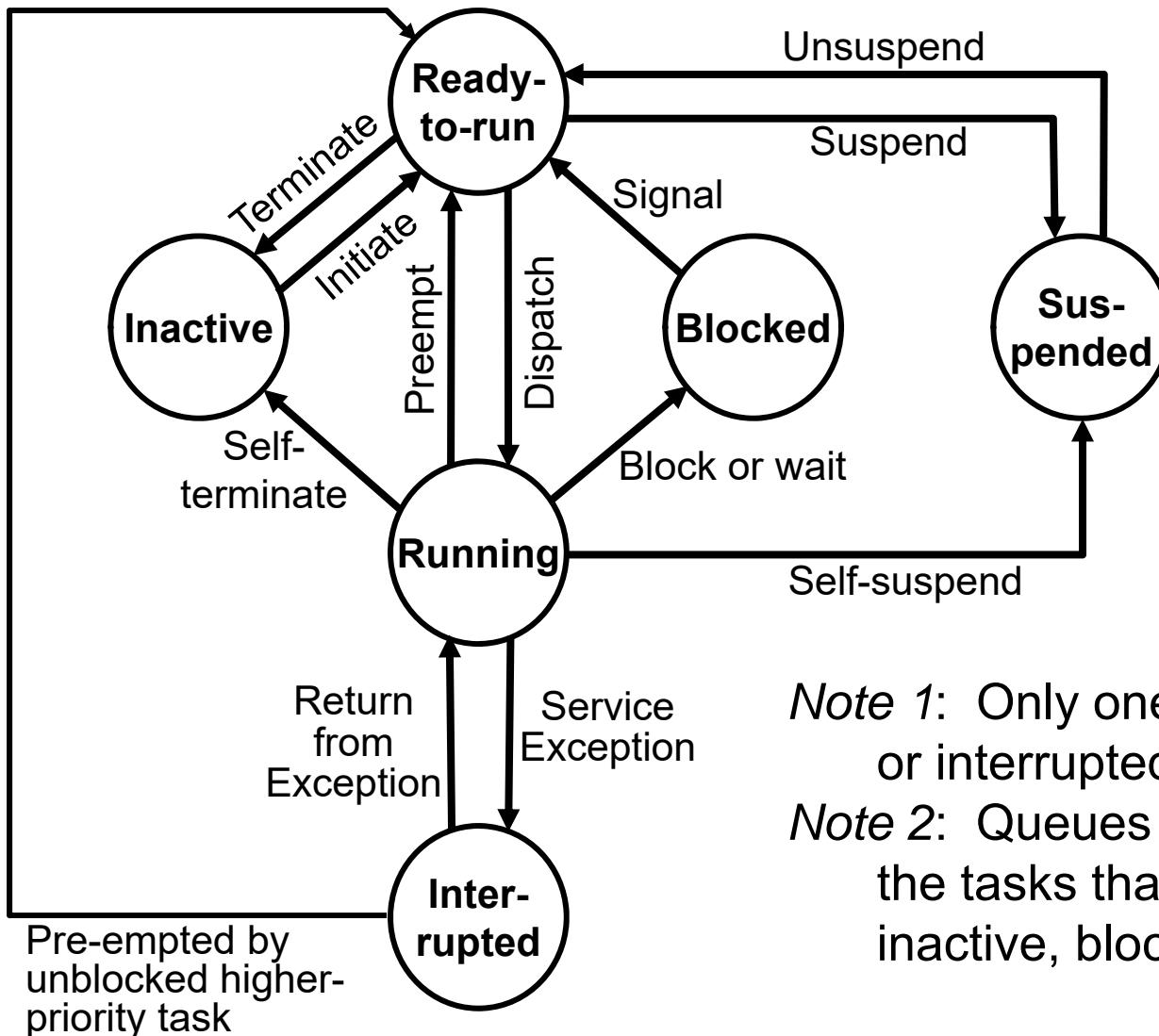
Disadvantages:

- 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.

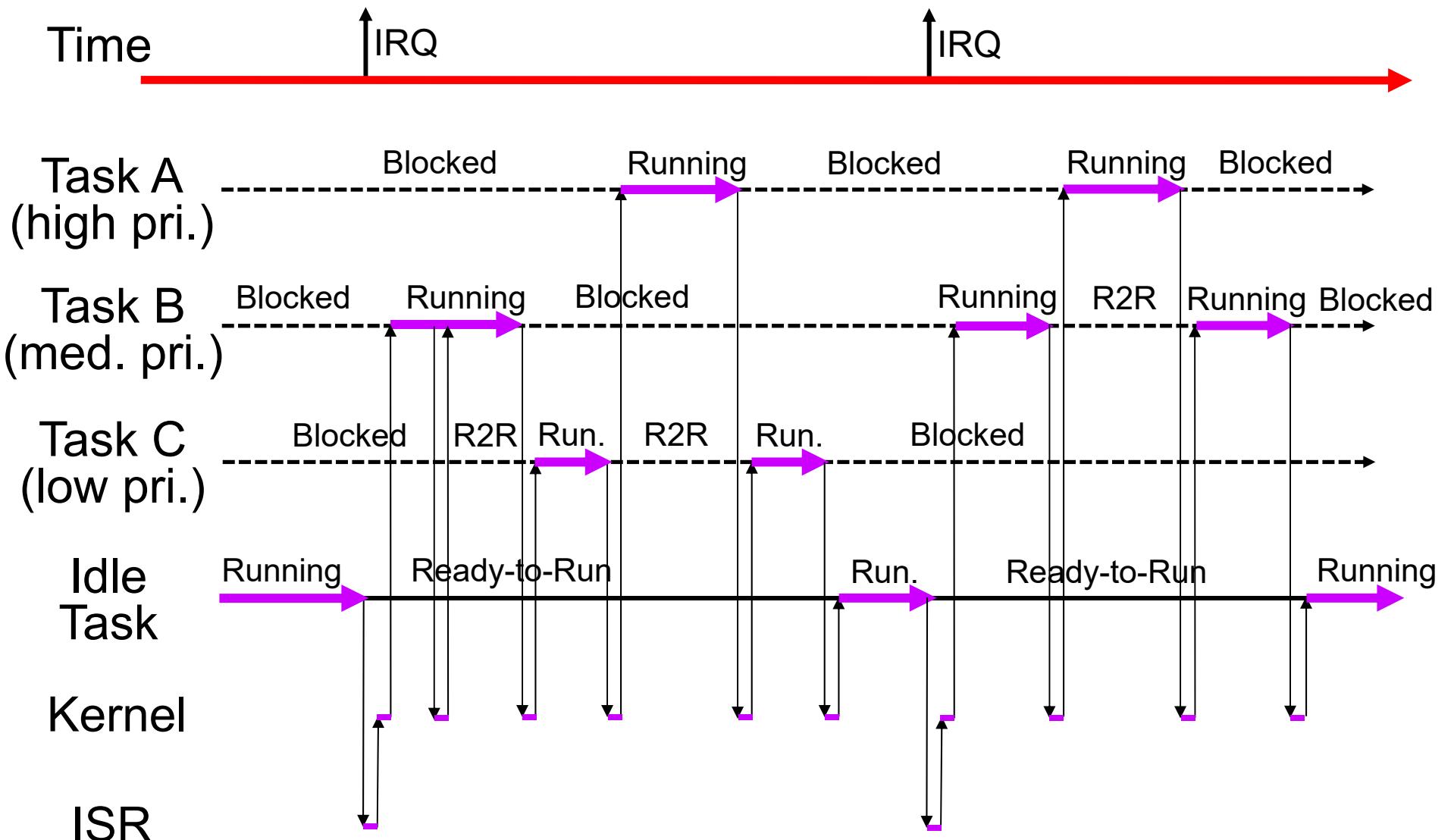
Ready-to-run: The process is available to start executing instructions as soon as the CPU becomes available.

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.

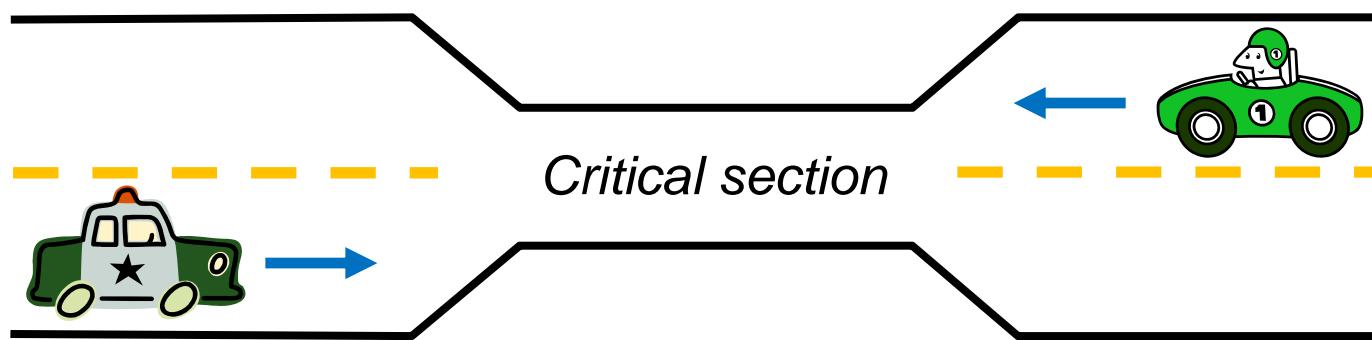
Real-Time Kernels

- 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.

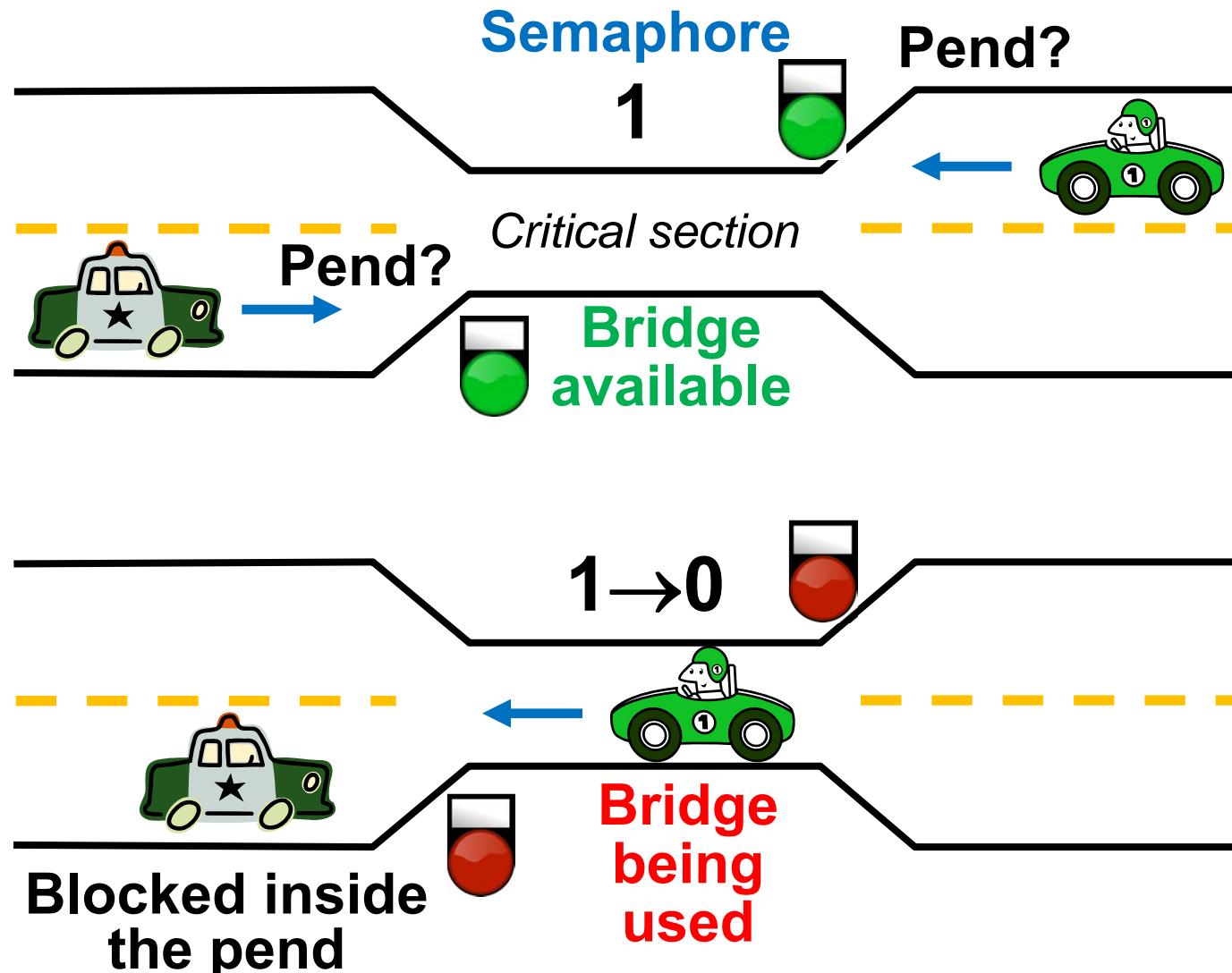
Updating Counting (and Binary) Semaphores

- **Pend** operation: A task needs exclusive use of a resource.
 - The count value is decremented by 1.
 - If the new count ≥ 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!

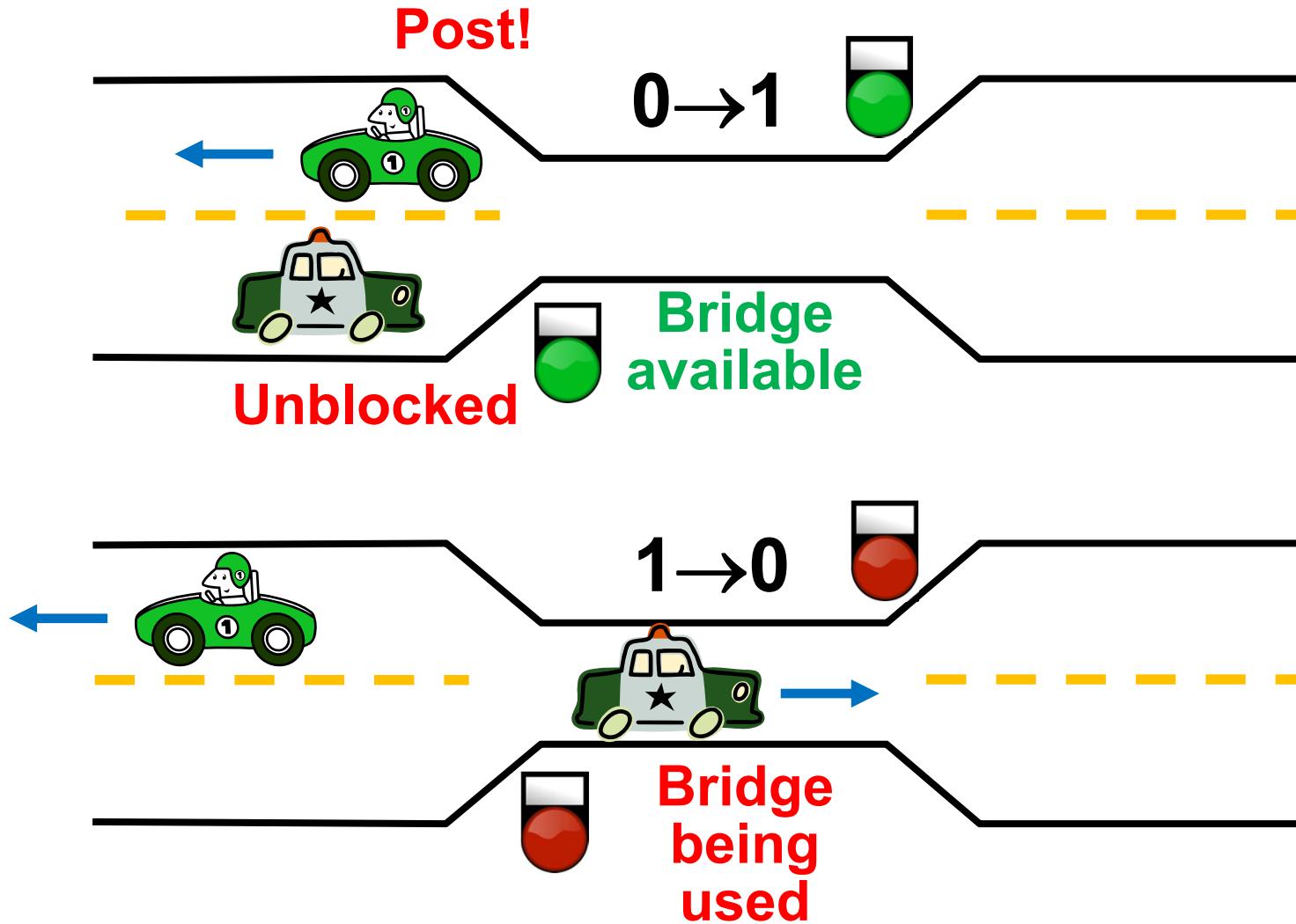
Binary Semaphores

- 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.

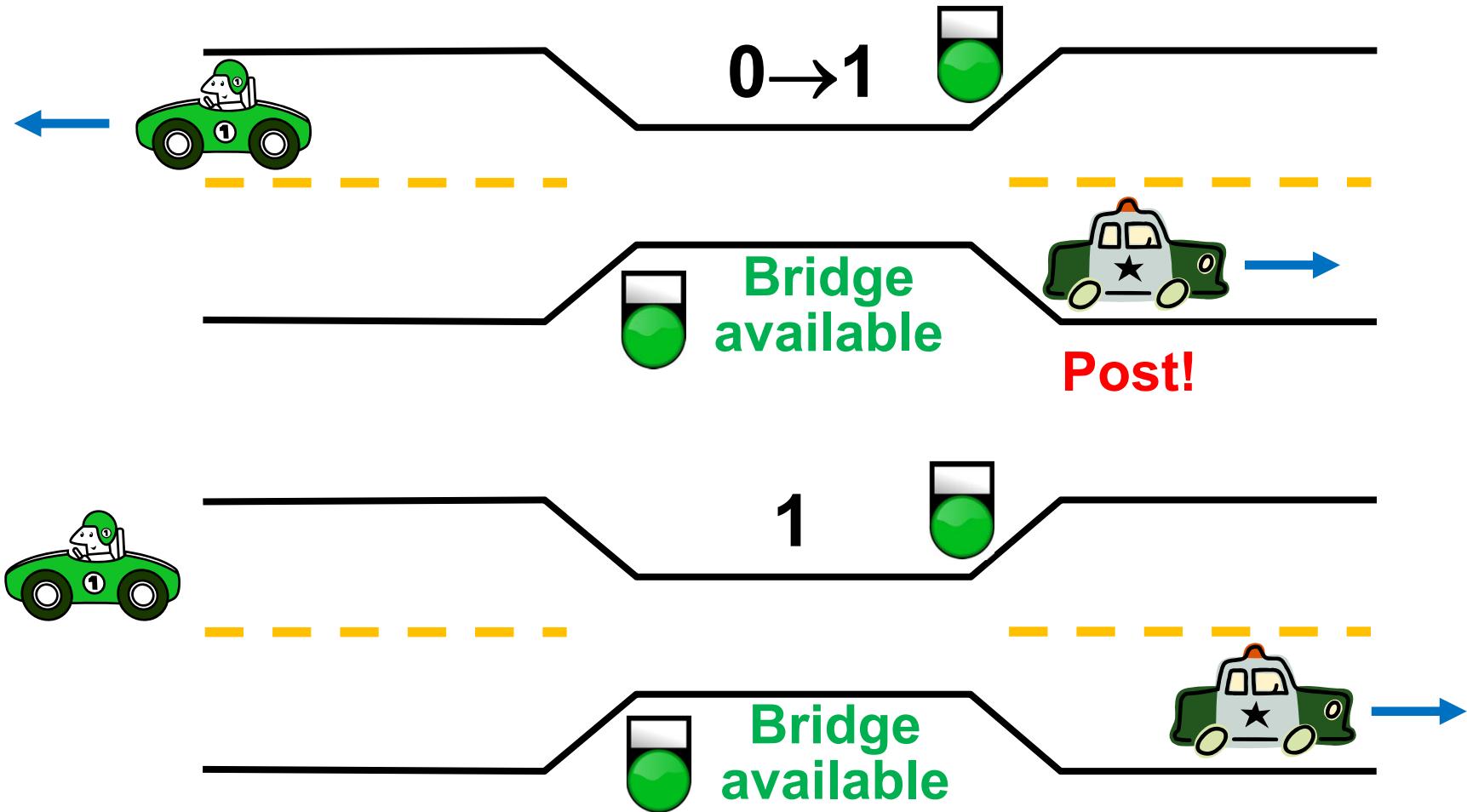
Back to the Single-Lane Bridge (1)



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)

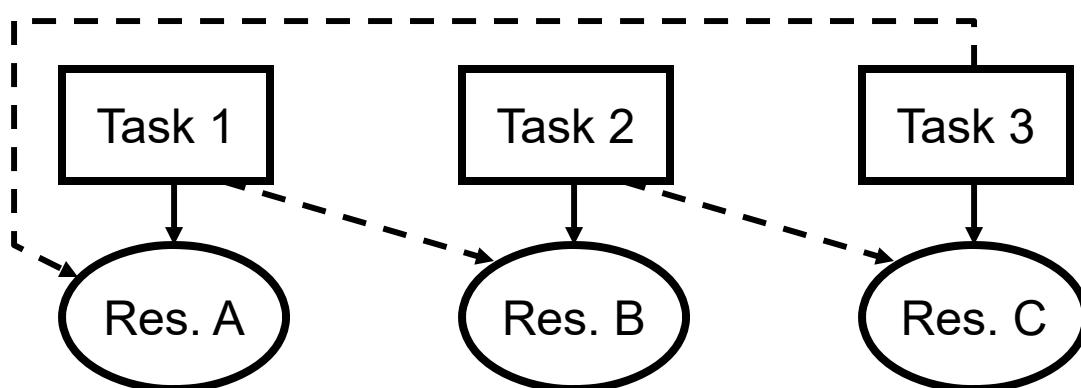
- A **counting semaphore** is typically used to keep track of the number of available resources in a finite pool of resources.
 - The semaphore count is initialized to the size of the pool. All resources are initially available.
 - Each time that a resource is assigned and becomes busy, the count is 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.



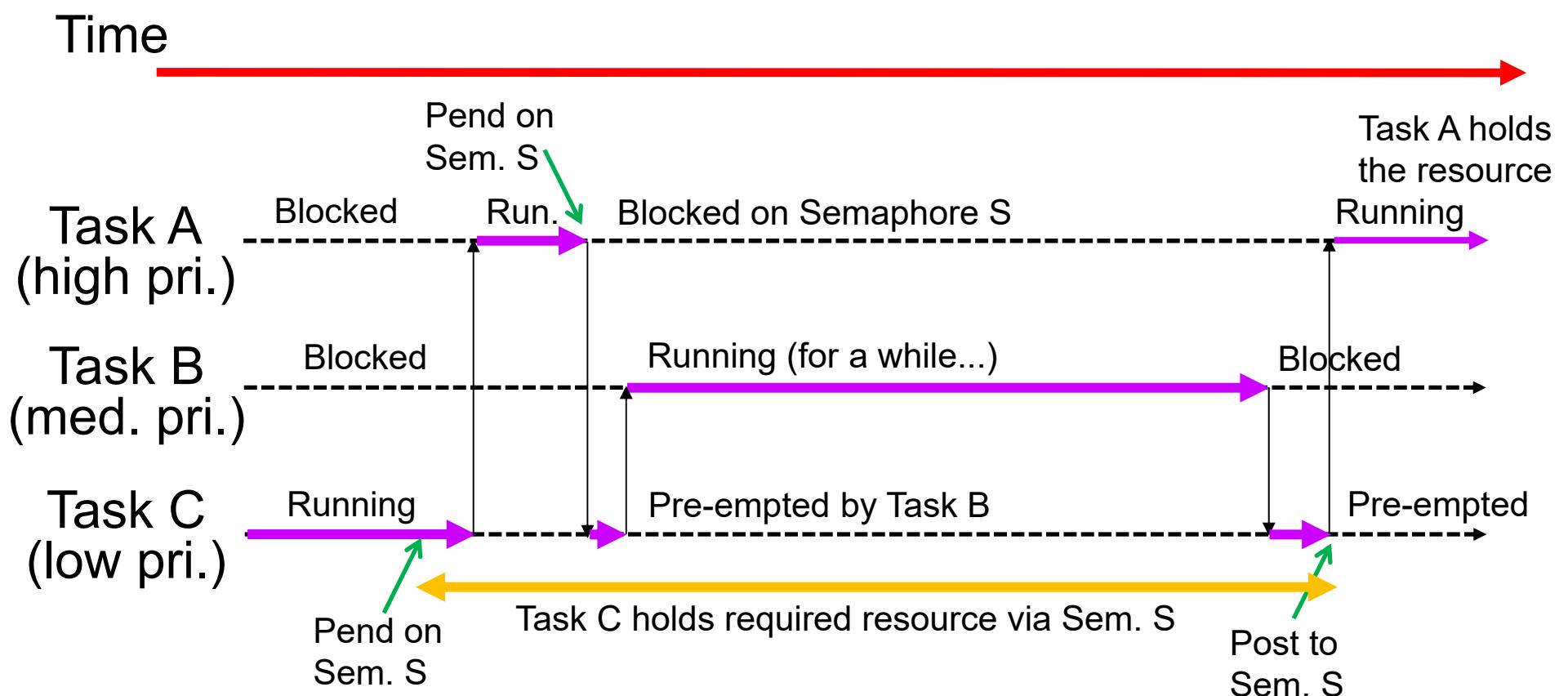
Watchdog Timers

- 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 controls allocation of the contested resource. This allows the low-priority task to finish using its resources quickly so that the high-priority task can proceed sooner. 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

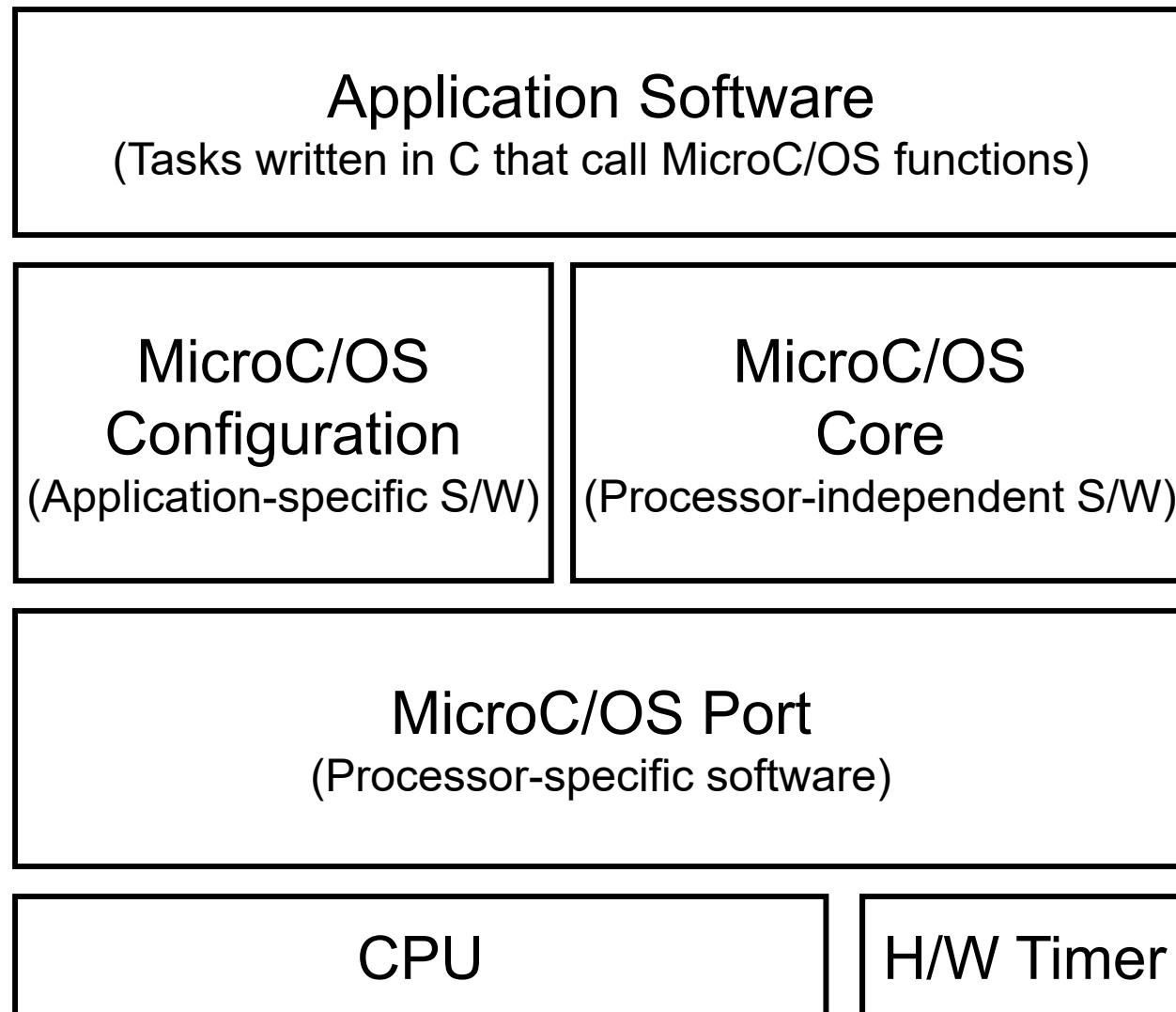
A Brief History of MicroC/OS

- Designed by Jean J. Labrosse in the early 1990s.
- The first version, called μ 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 *μ 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";
```

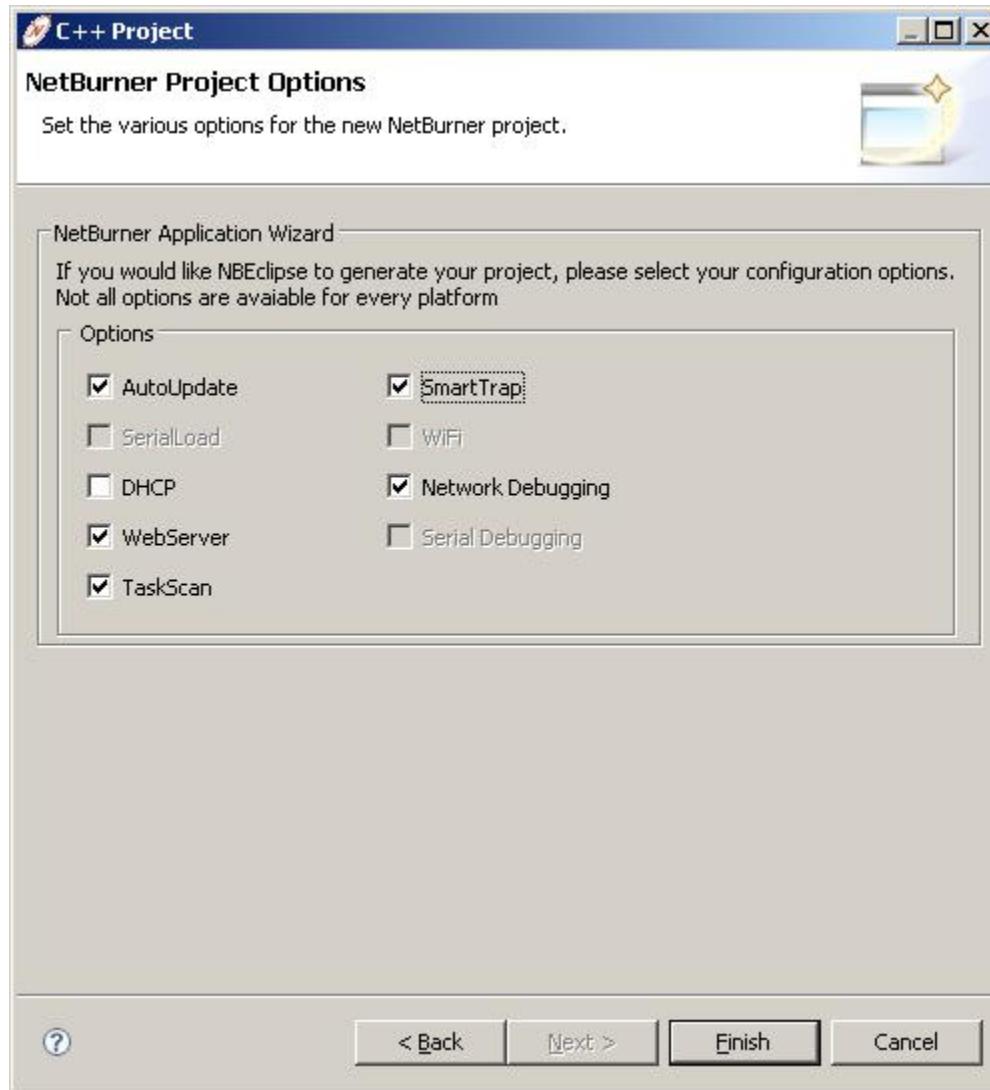
“Hello World!” Example, Part 2

```
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;

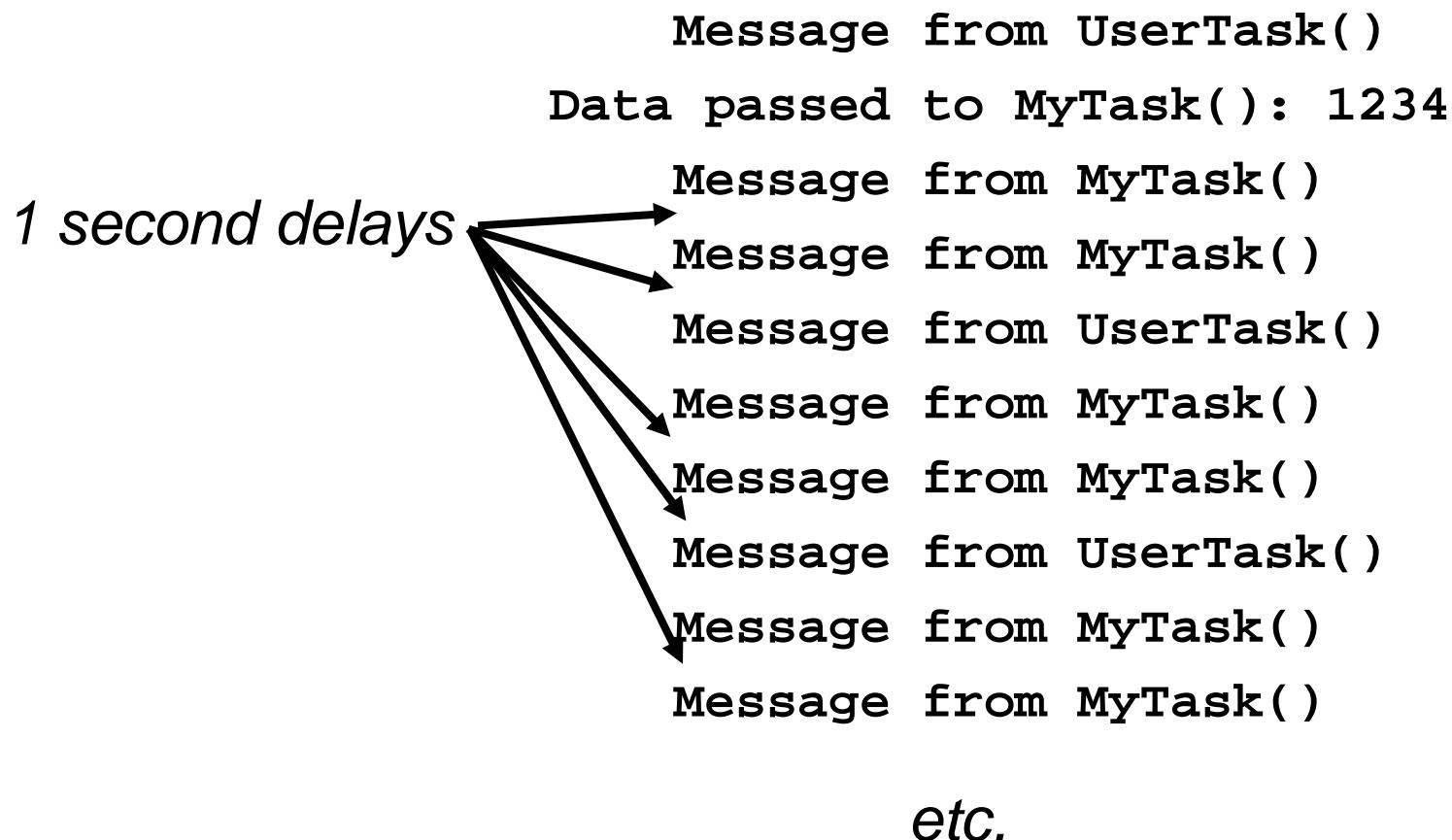
    if ( OSTaskCreate ( MyTask, (void *) &MyTaskData,
                        (void *) &MyTaskStack[USER_TASK_STK_SIZE],
                        (void *) MyTaskStack, MAIN_PRIO+1 ) != OS_NO_ERR )
    {
        iprintf( "**** Error creating MyTask\r\n" );
    }

    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)

.OSEventType							
.OSEventCnt							
OSEventPtr							
.OSEventGrp							
7	6	5	4	3	2	1	0
63	62	61	60	59	58	57	56

- 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

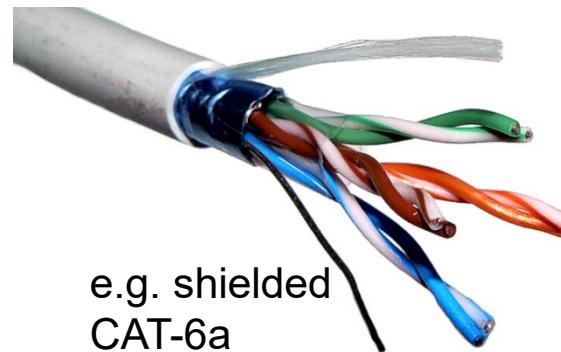
- ***Unbalanced or Single-ended Mode***
 - One wire carries the signal voltage that is defined with respect to a ground (0 volt) reference potential on a second wire or the shield.
 - *Advantages:* One ground wire can be shared for multiple signals.
 - *Disadvantages:* Noise that is added to either the 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

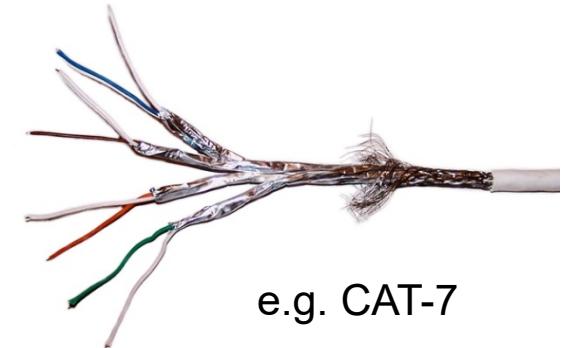
Unshielded
Twisted
Pair (UTP)



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**)

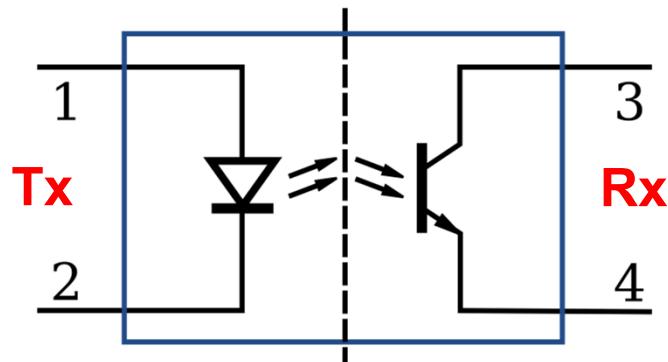
- **Cable is shielded.**
- **Foil for each TP.**

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

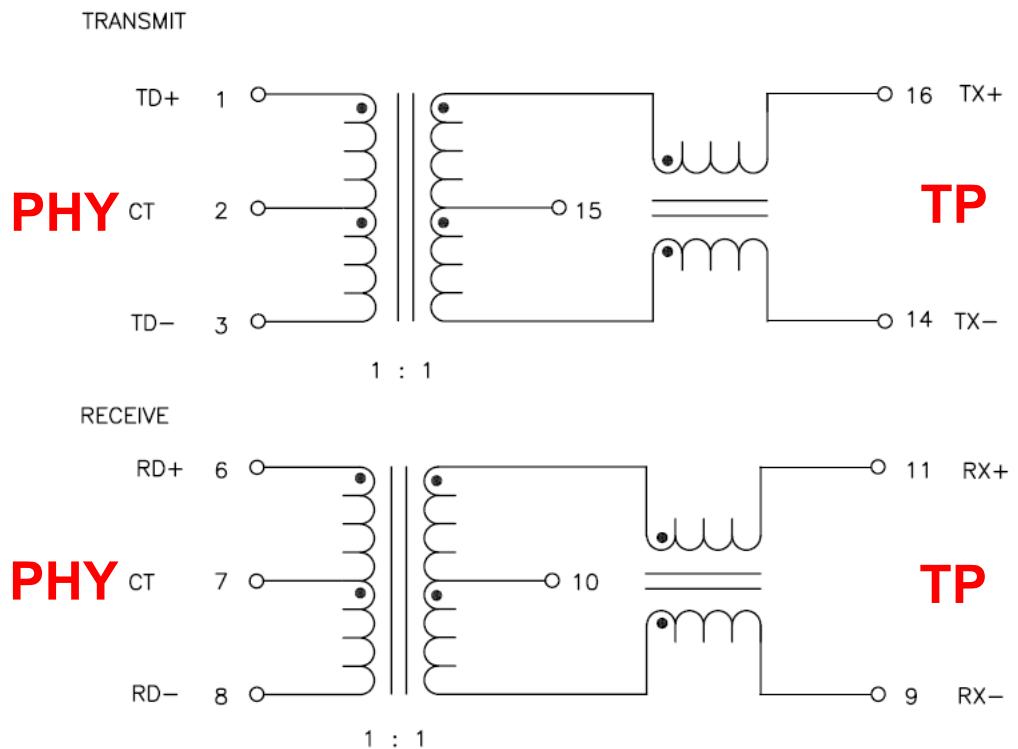
- Protects receiver from high voltage signals and noise
- Ground potentials can differ
- Relatively slow
- Audio applications
- Biomedical sensors



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***.

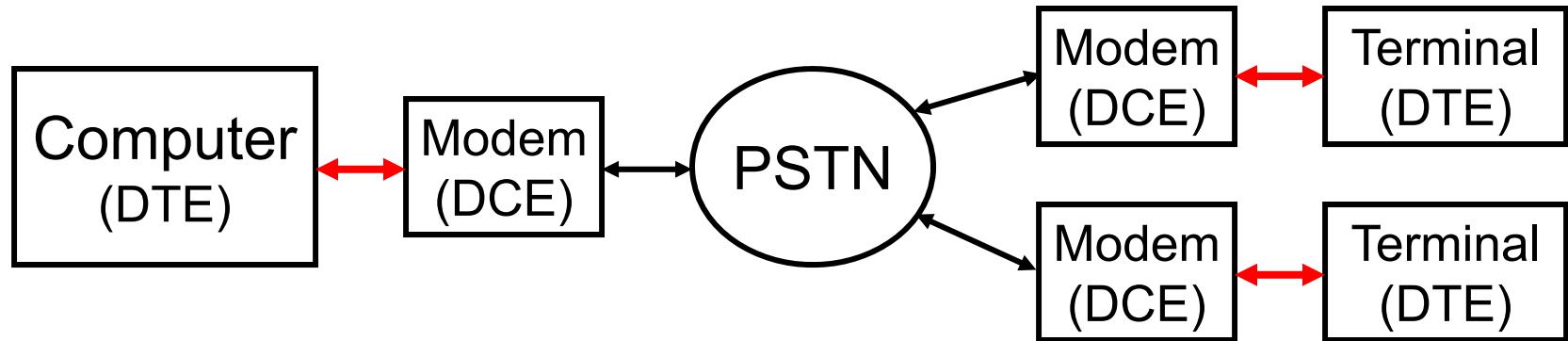
A Brief History of RS-232C

- 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)

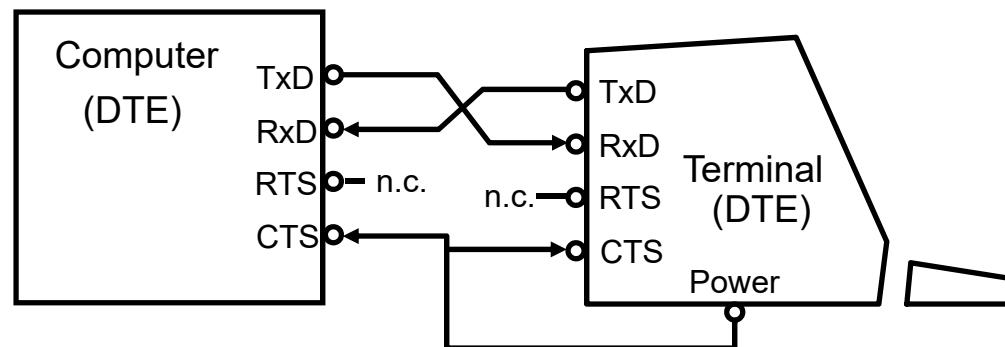
- Computers, terminals

Data Communications Equipment (DCE)

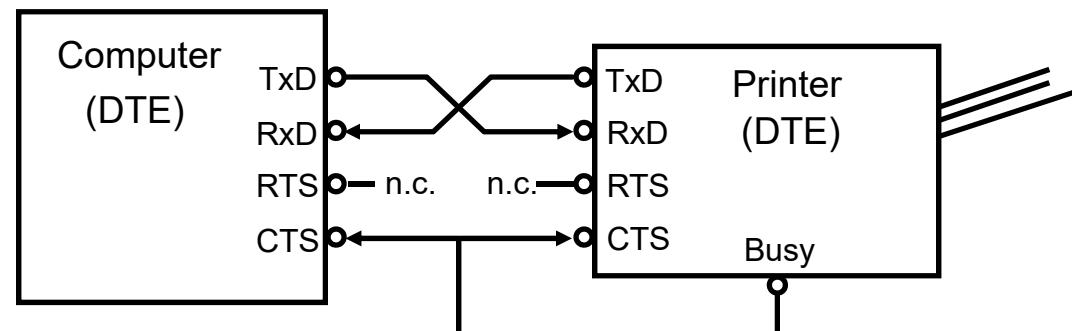
- Modem for interfacing RS-232C binary signals to analog signals that lie within the standard 100 Hz to 3400 Hz telephone passband.

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).

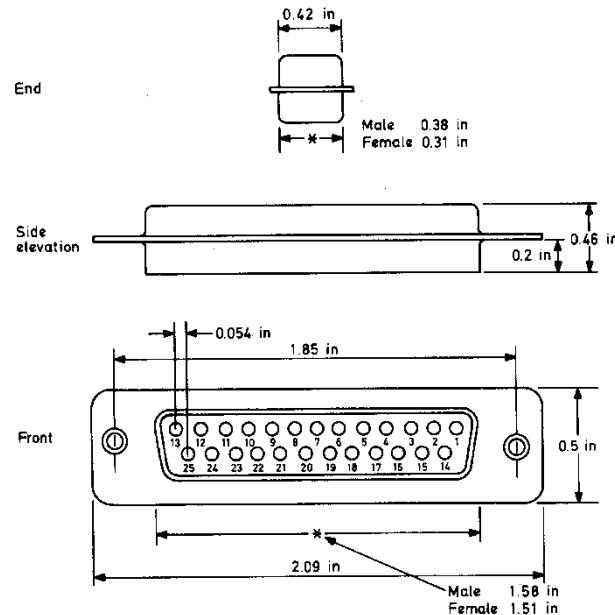
Electrical Aspects

- “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



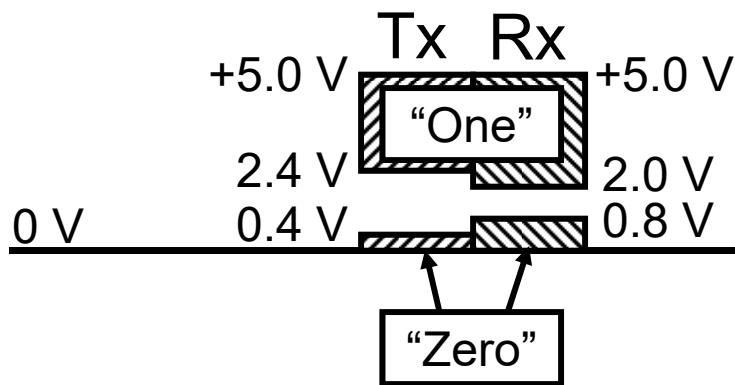
- A 25-pin DB-25 connector (from Cannon Co., 1952) was specified in an appendix to the original RS-232C standard.
- Most modern serial standards use a much smaller connector. For a time, the smaller 9-pin DE-9 connector replaced the DB-25 for RS-232C connections.
- Ethernet, USB & WiFi have mostly replaced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

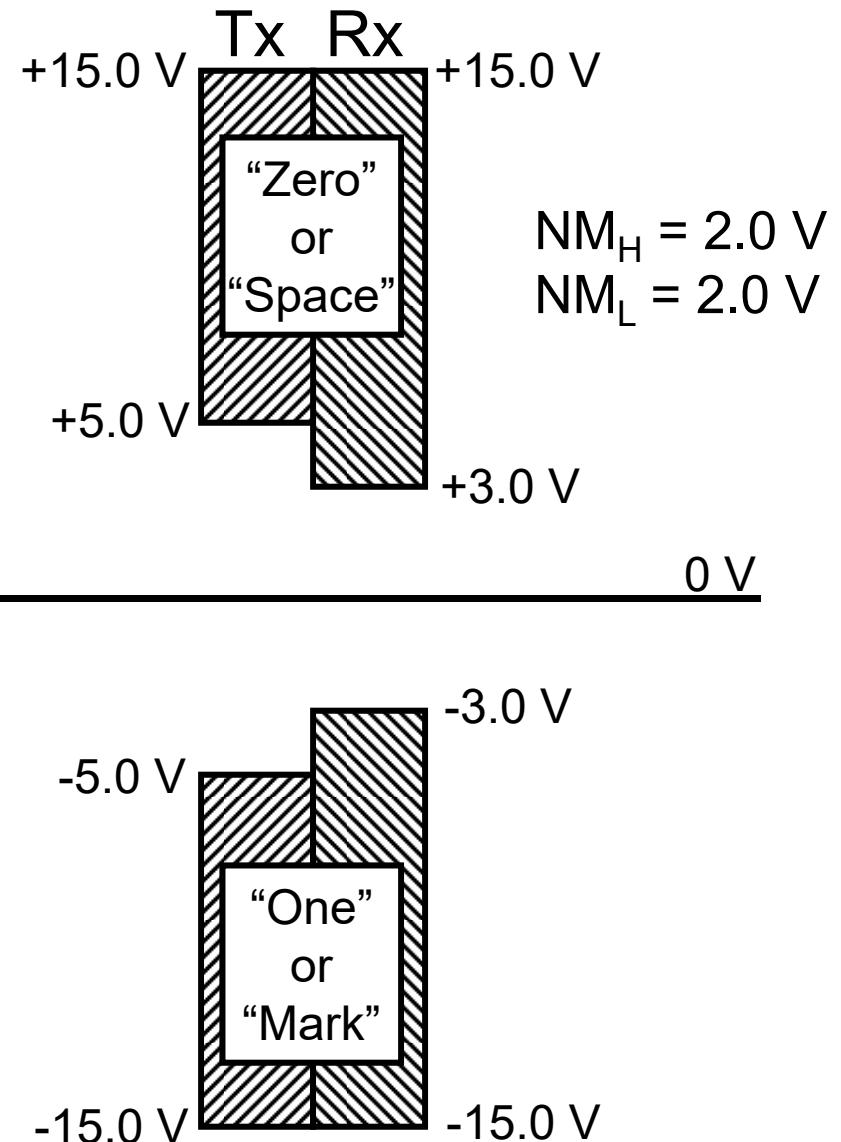
TTL = Transistor-Transistor Logic



Noise Margins in TTL:

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

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



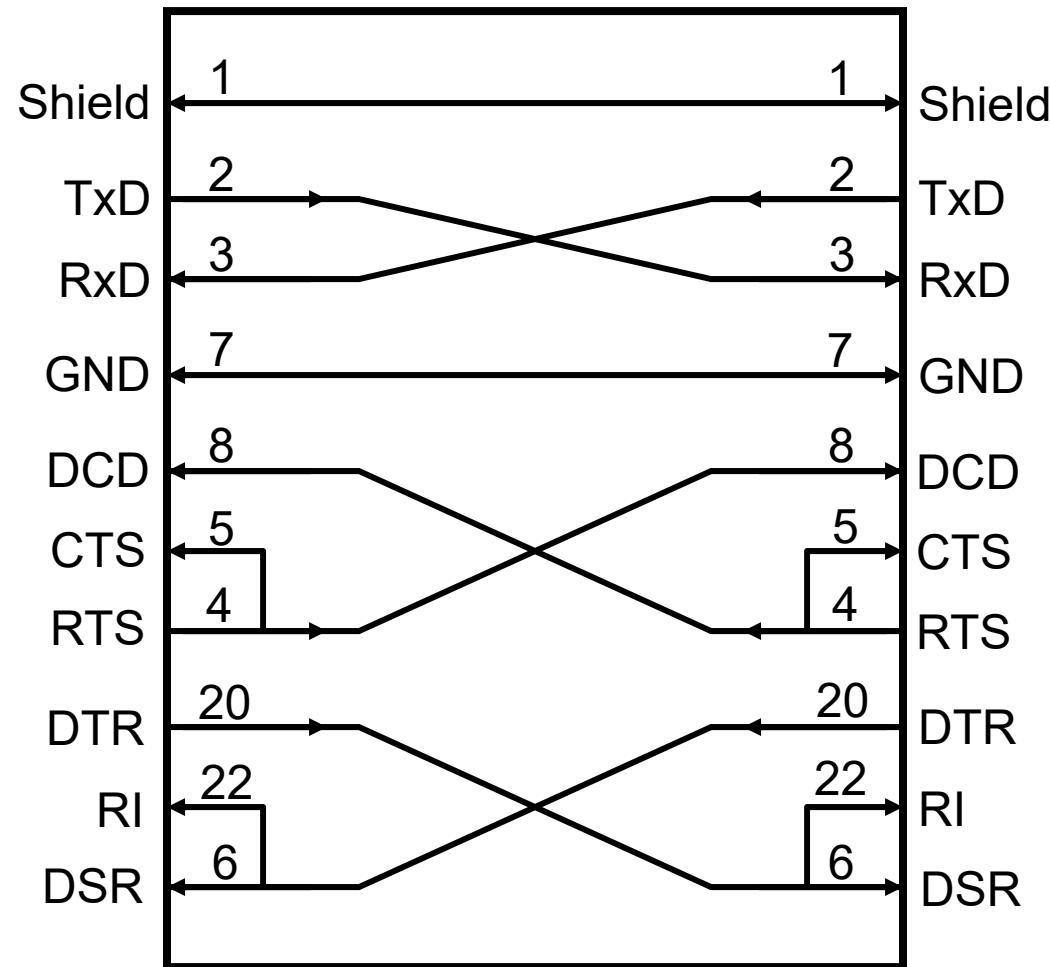
More on Logic Signal Ranges

- 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.

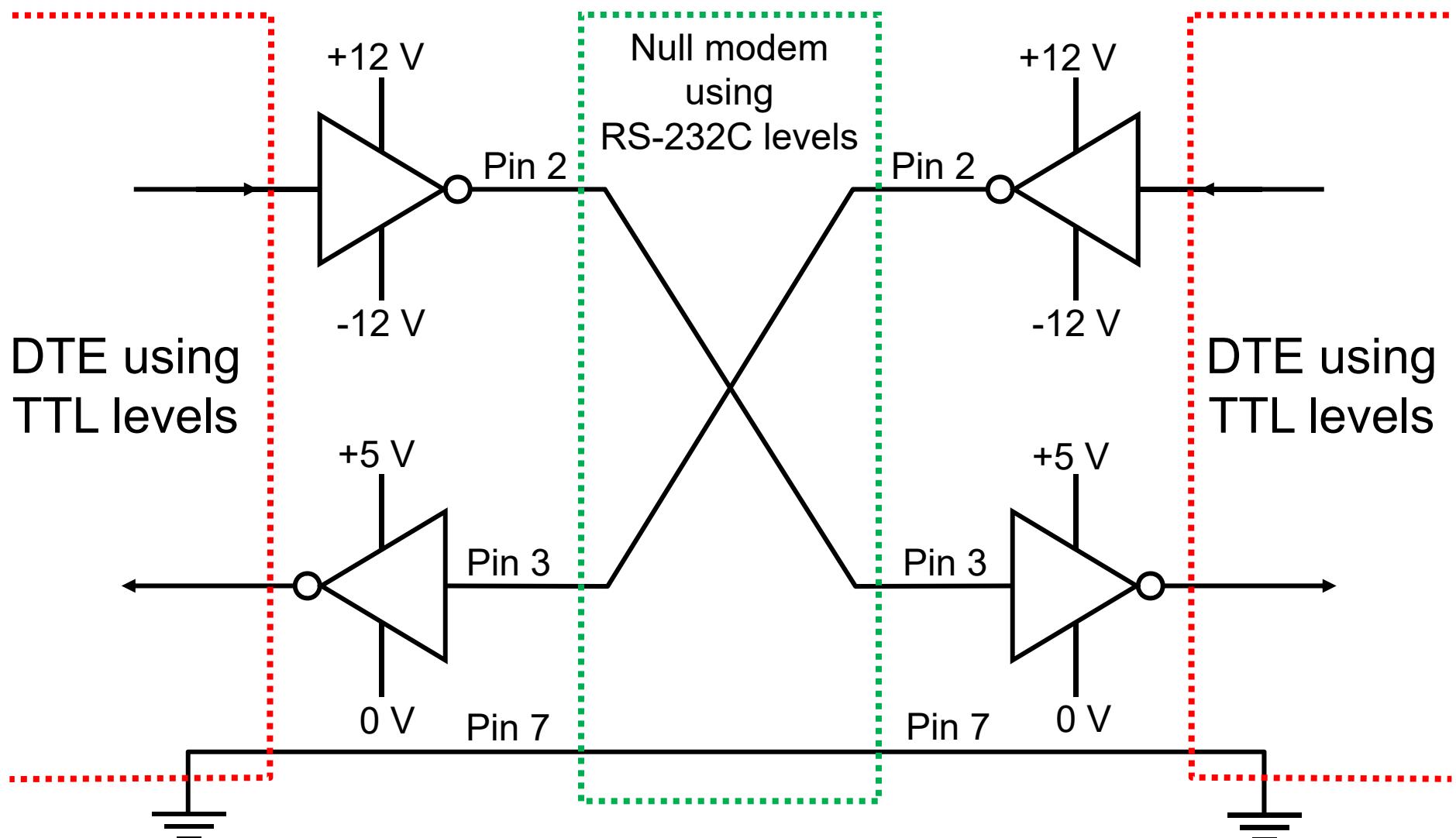
Noise Margins

- 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!

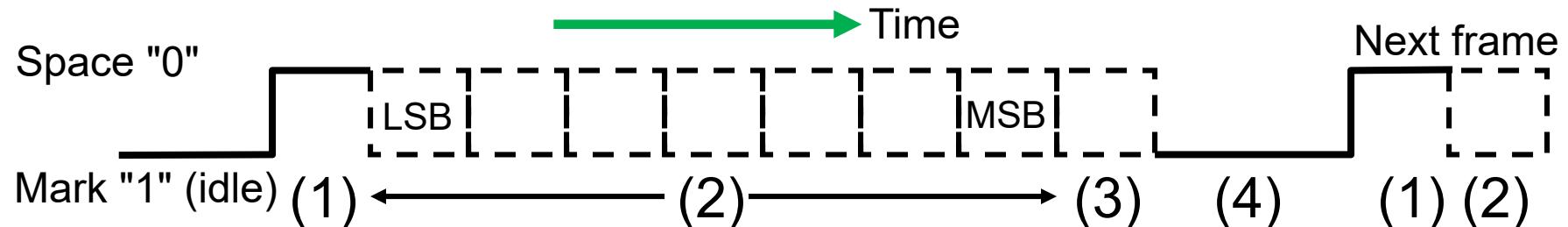
“Null Modem” for DTE-DTE Connections



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:

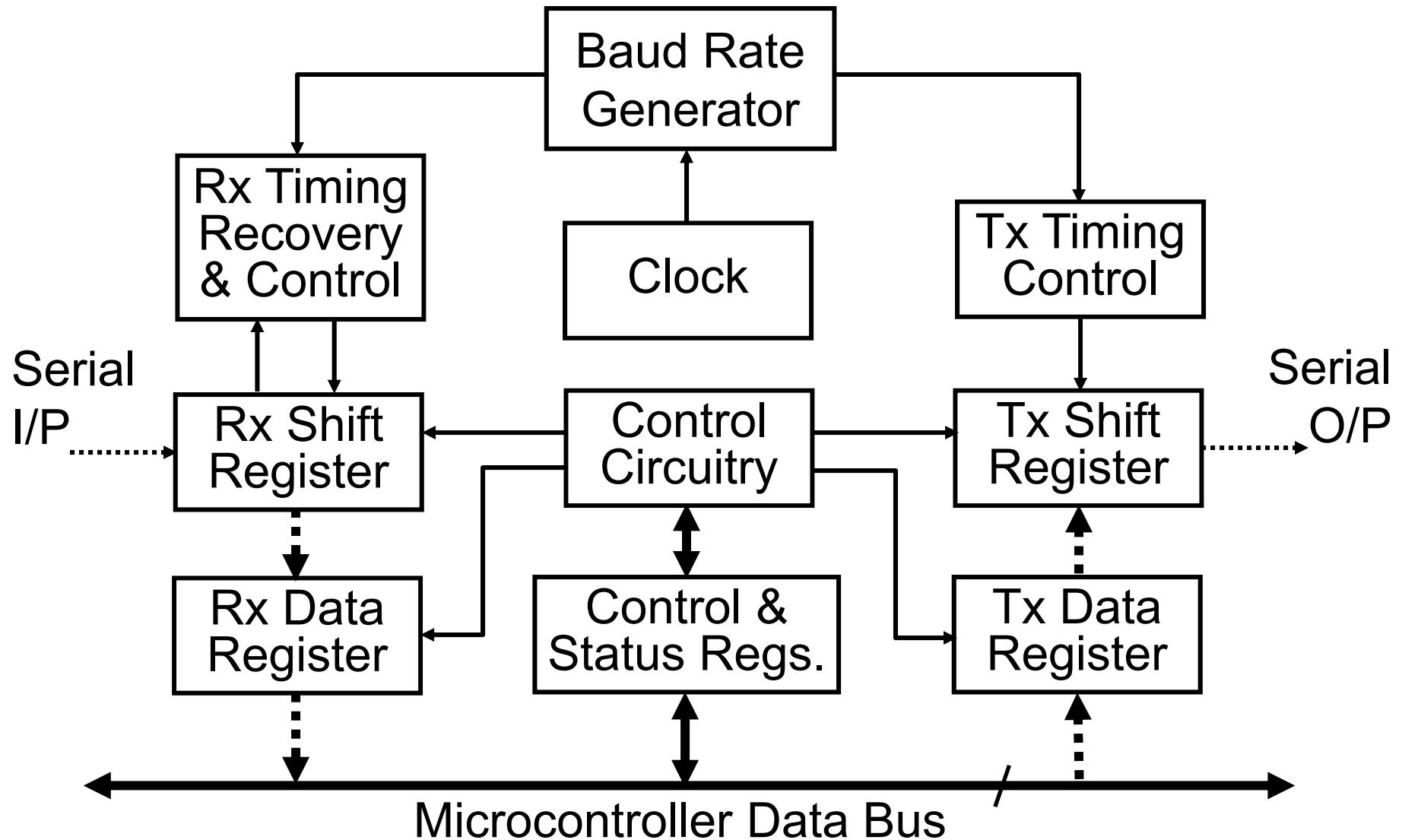
Request to Send (RTS) is asserted to “mark” by the DTE to tell the DCE that the DTE wishes to transmit data.

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

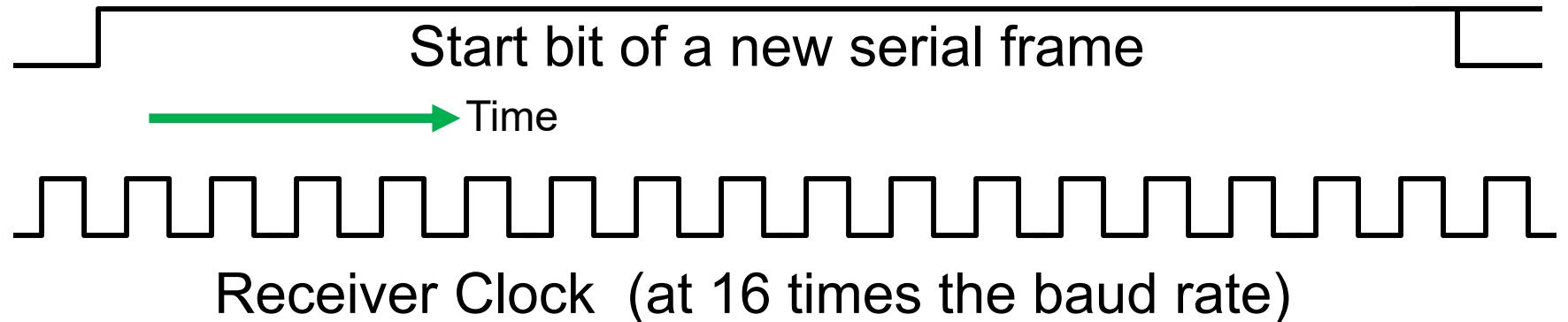
Data Set Ready (DSR) is asserted to “mark” by the DCE to tell the DTE that it is switched on and is not in a test mode.

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 μ P
- Provides one bi-directional serial port

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

- Introduced in the mid 1980s to support the M68000 μ 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 μ C

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

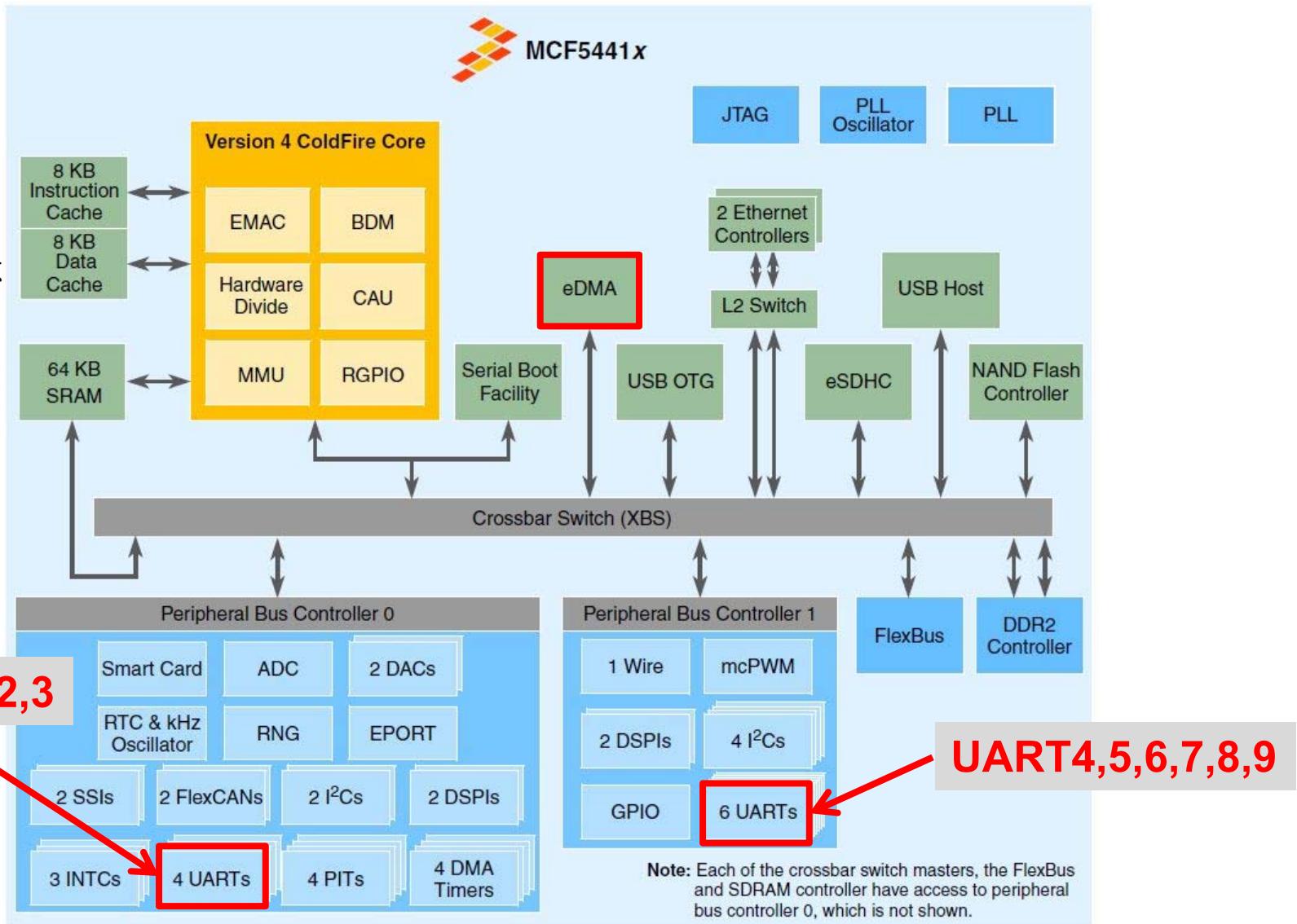
- Introduced in the mid 1980s as part of the 68HC11 8-bit μ C

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

- Introduced in the late 1980s as part of the 683xx 32-bit μ Cs

The Ten UART Interfaces in the MCF54415

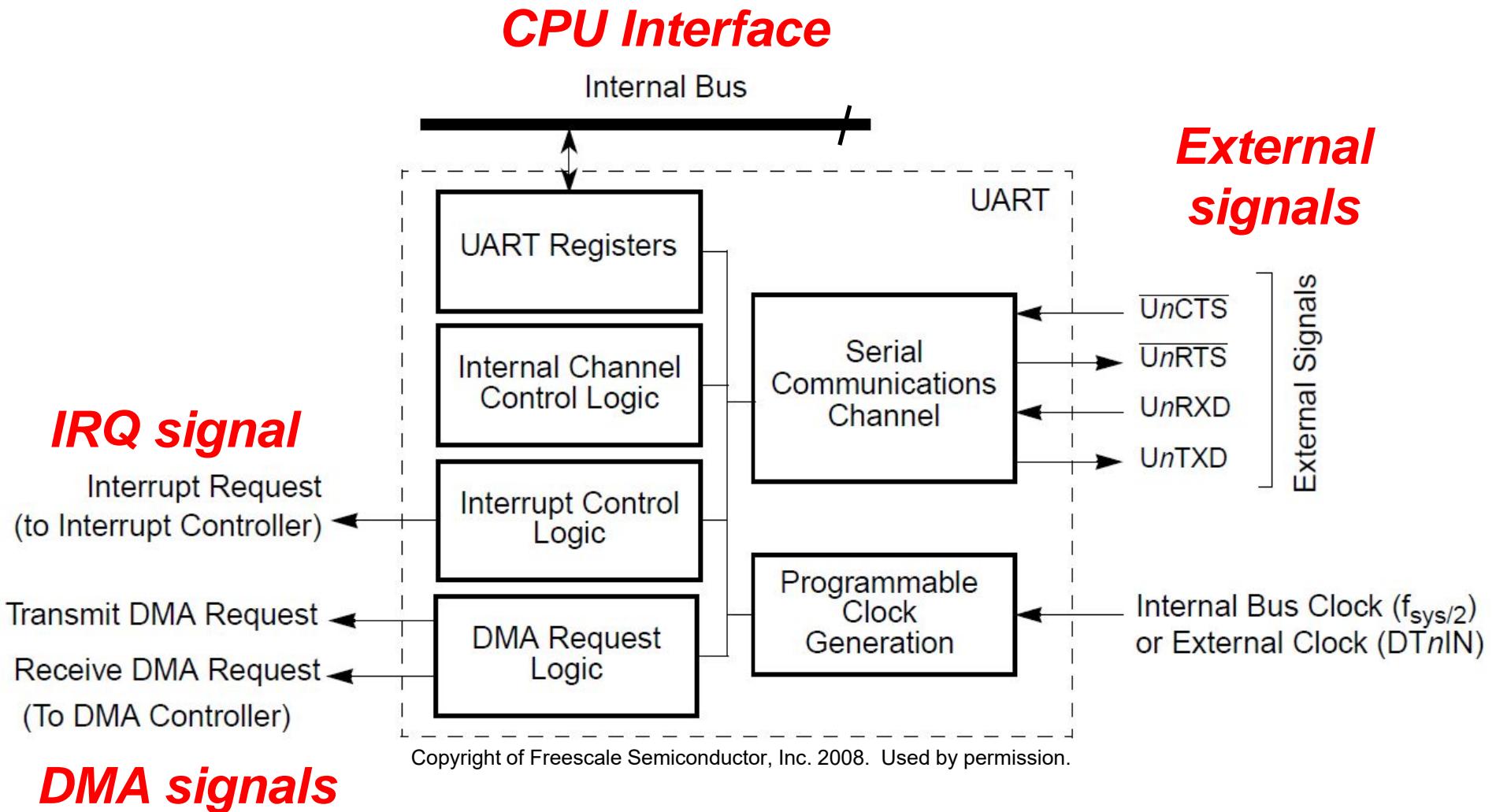
Copyright of Freescale Semiconductor, Inc. 2012.
Used by permission.



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 UART_n Pin Connections, $n = 0, \dots, 9$

n	UART_nTXD	UART_nRXD	UART_nRTS	UART_nCTS	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)

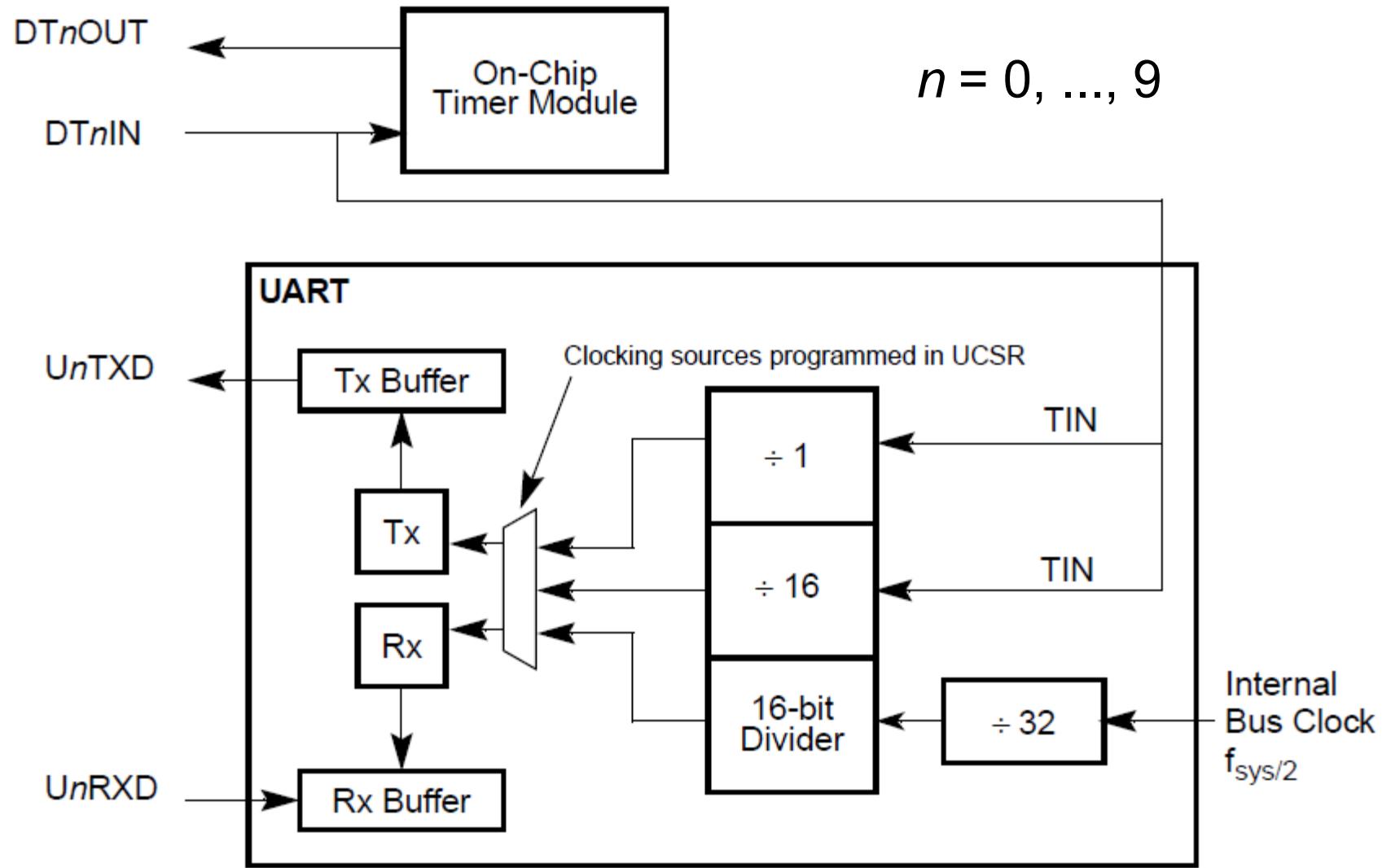
Note: The Request-To-Send (RTS) and Clear-To-Send (CTS) handshake lines are only provided for UART0, UART1 & UART2.

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

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

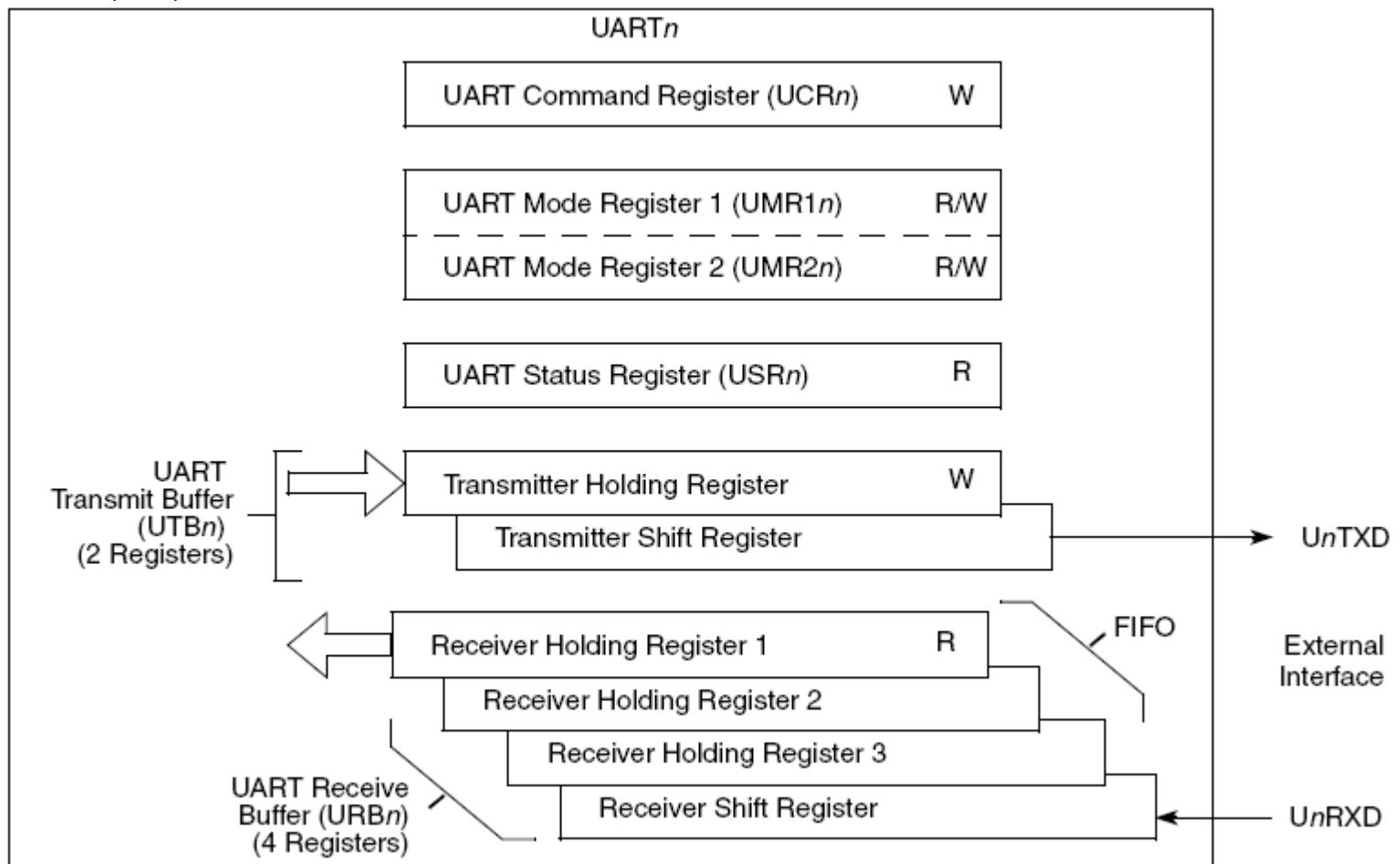
UART n Clock Generation Sub-Block



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:

	Base Address
UART0	0xFC06_0000
UART1	0xFC06_4000
UART2	0xFC06_8000
UART3	0xFC06_C000
UART4	0xEC06_0000
UART5	0xEC06_4000
UART6	0xEC06_8000
UART7	0xEC06_C000
UART8	0xEC07_0000
UART9	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 ≥ 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”

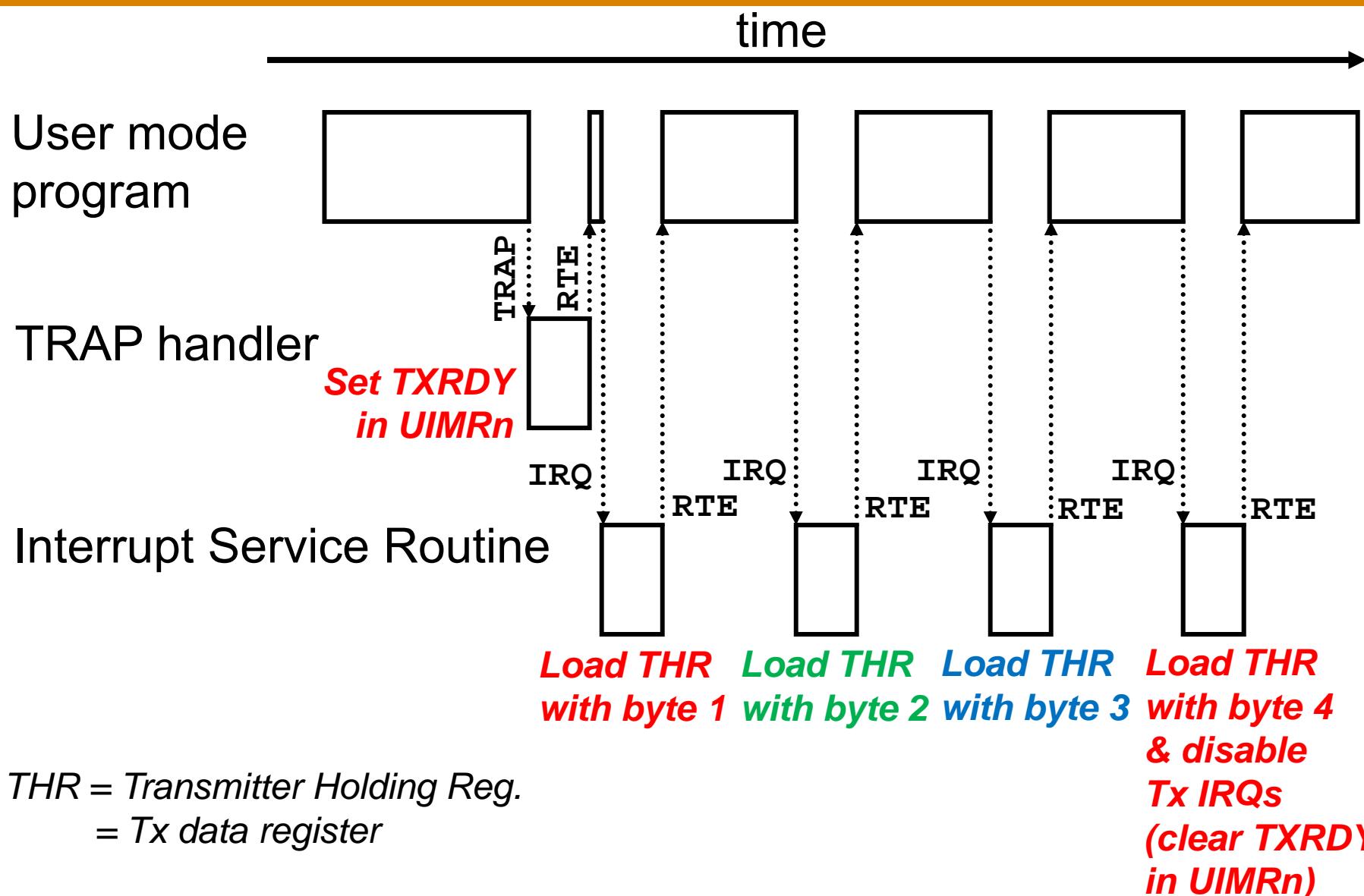
RXRDY = “Receiver Ready”

TXRDY = “Transmitter Ready”

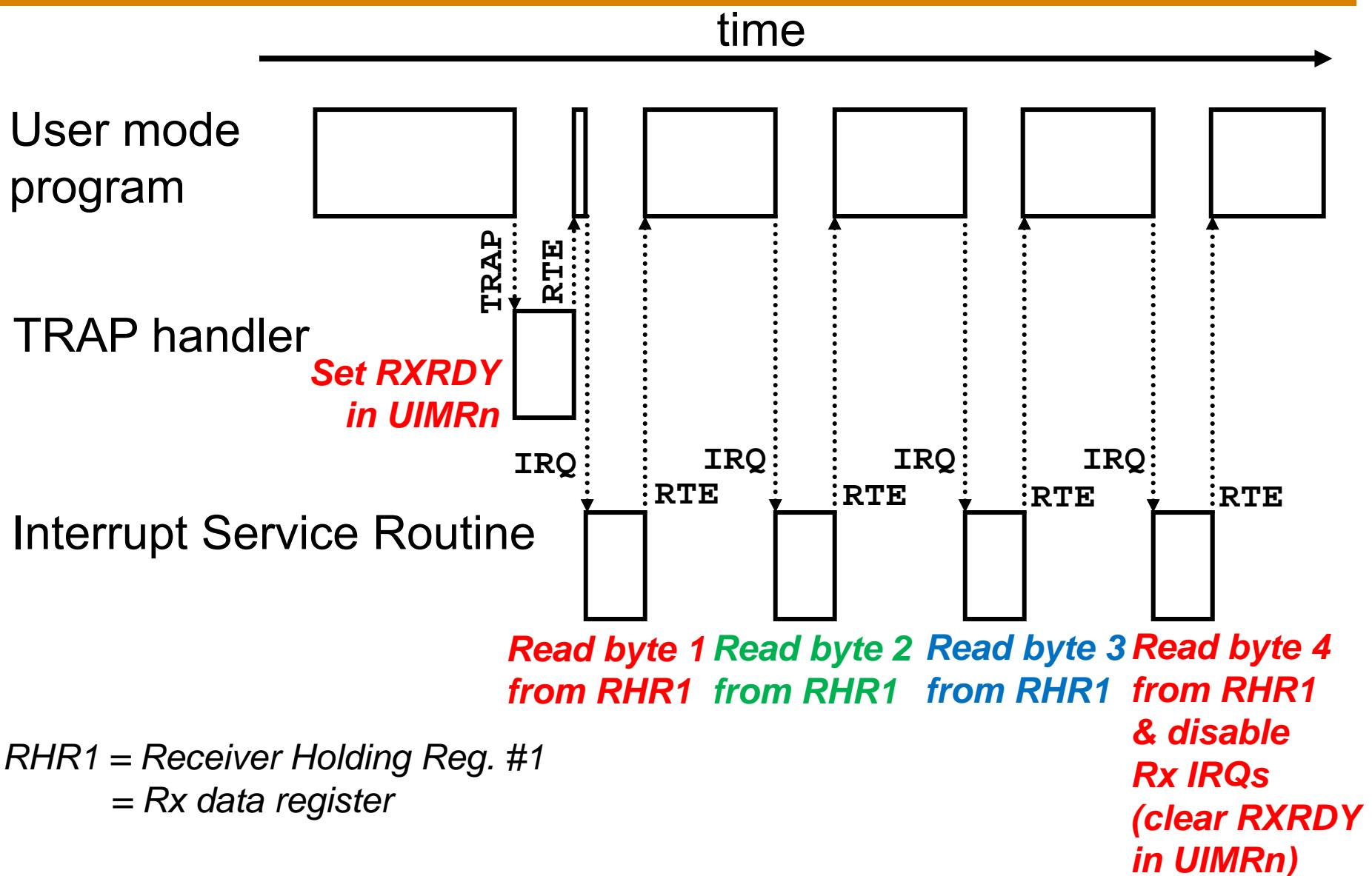
Copyright of Freescale Semiconductor, Inc. 2008. Used by permission.

- 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



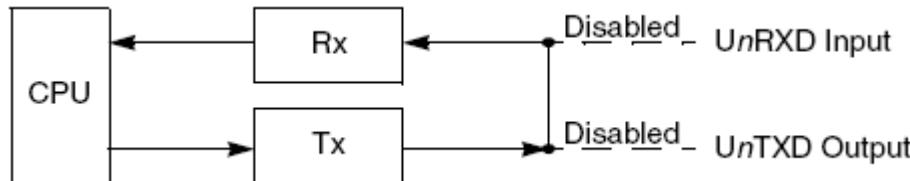
Interrupt-Driven UARTn Input for Four Bytes



UART Loopback Test Modes

1) Local Loopback Mode

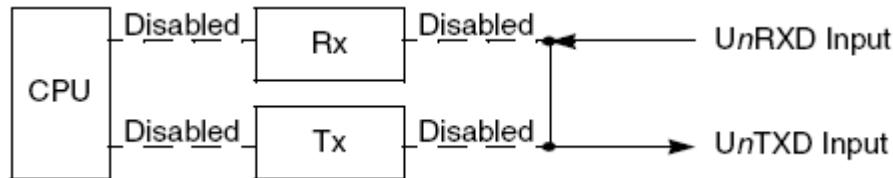
- Allows local UART to verify its Tx and Rx circuits
- External Tx and Rx connections are not tested



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:

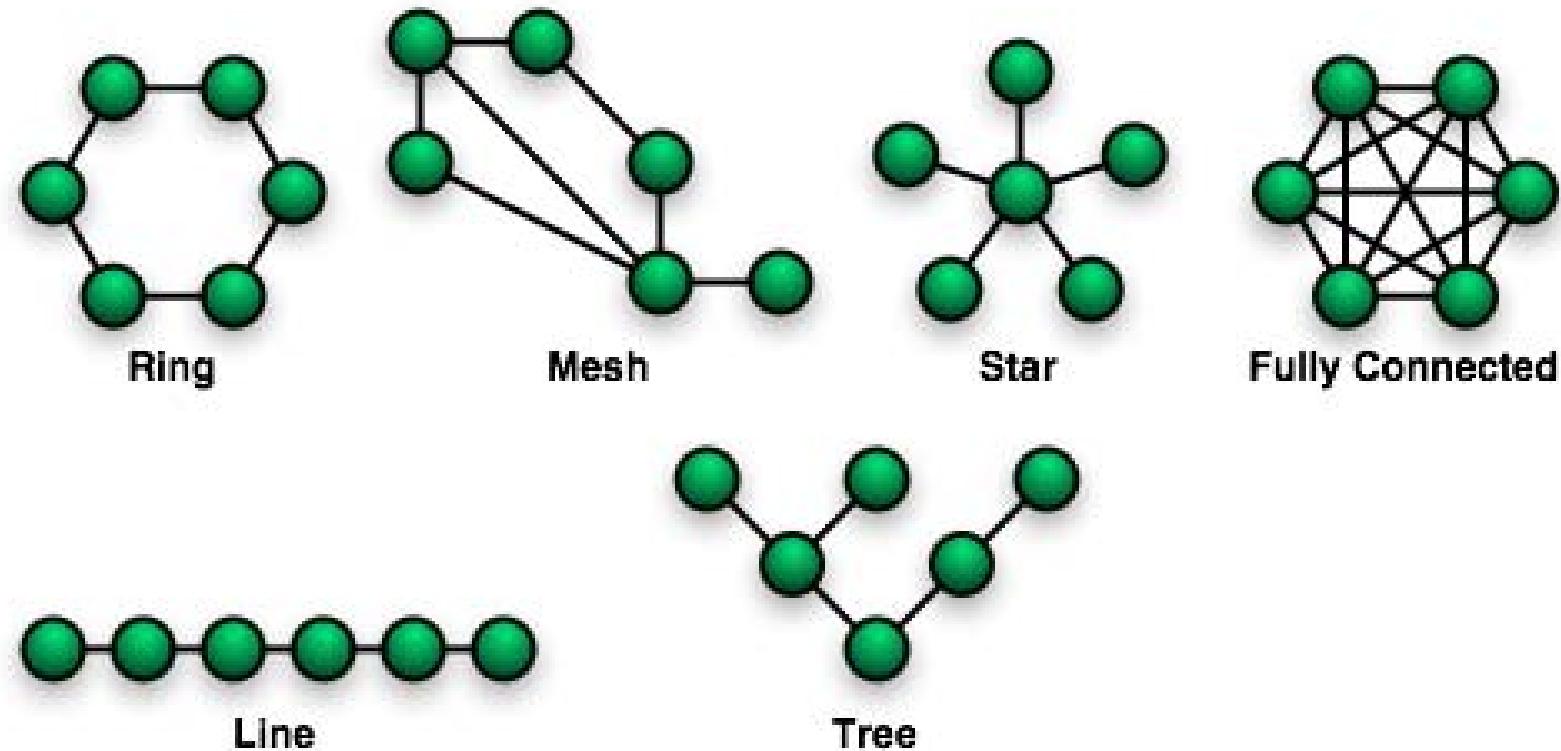


Figure courtesy of Wikimedia Commons

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 2nd 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
00000000000110111101
1101
000000000000110111101
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

00000000110110111101
1101

00000000110110111101
1101

00000000000010111101
1101

00000000000010111101
1101

00000000000010111101
1101

00000000000010111101
1101

00000000000010111101
1101

00000000000010111101
1101

00000000000011011101
1101

00000000000011011101
1101

00000000000000000000101
1101

00000000000000000000101
1101

00000000000000000000101
1101

00000000000000000000101
1101

00000000000000000000101
1101

00000000000000000000101
1101

Error(s) detected!

Ethernet Data Rates and Transmission Media

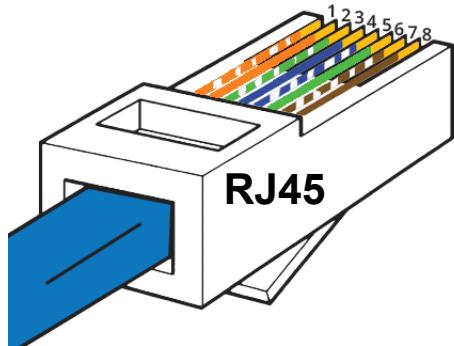
- Ethernet is available for several different data rates (10, 100, 1000, 10000 Mbps, etc.) and a variety of media:
 - **RG-8** thick 50- Ω coaxial cable, for 10-Mbps (10BASE5, the original Ethernet standard). Now obsolete.
 - **RG-58** “thin” 50- Ω coaxial cable, for 10-Mbps (10BASE2).
- Coax cables 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.

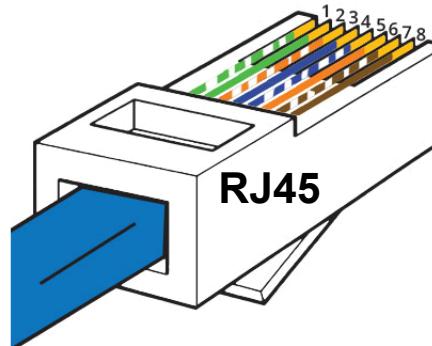
Type A and B Pin-outs at the RJ45 Plug

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

- 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

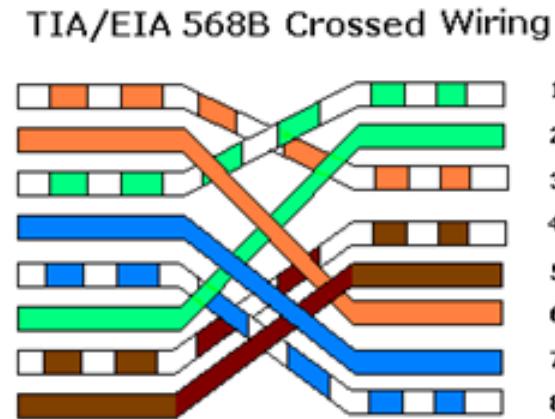
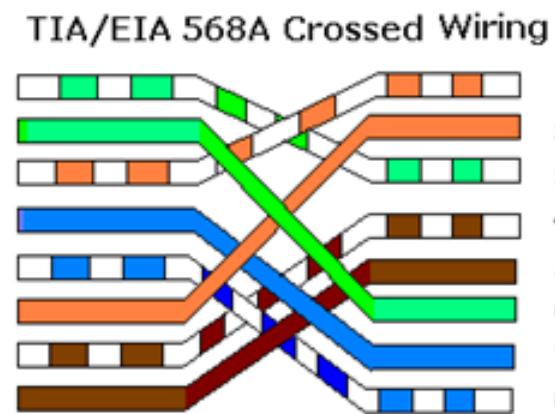


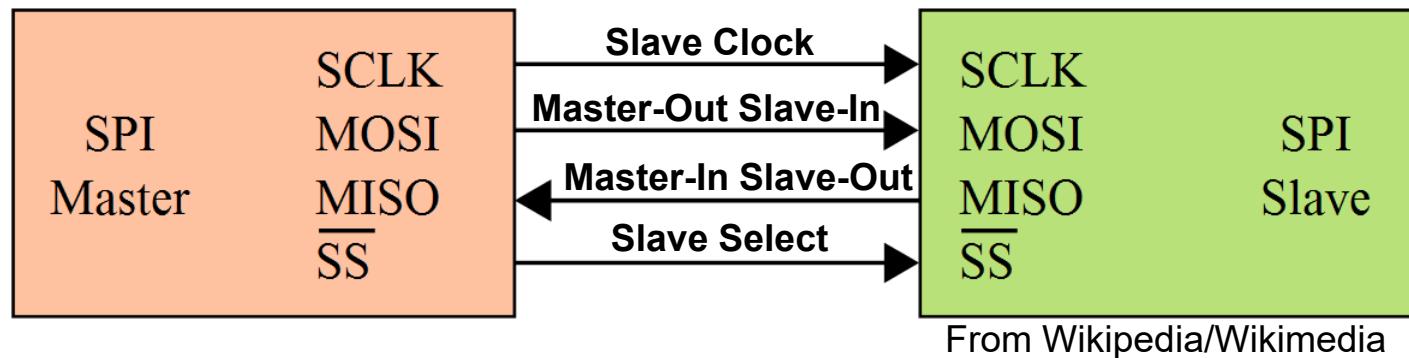
Figure B

Shows the Pin Out of Crossover Cables

Figure courtesy of electronicsforu.com

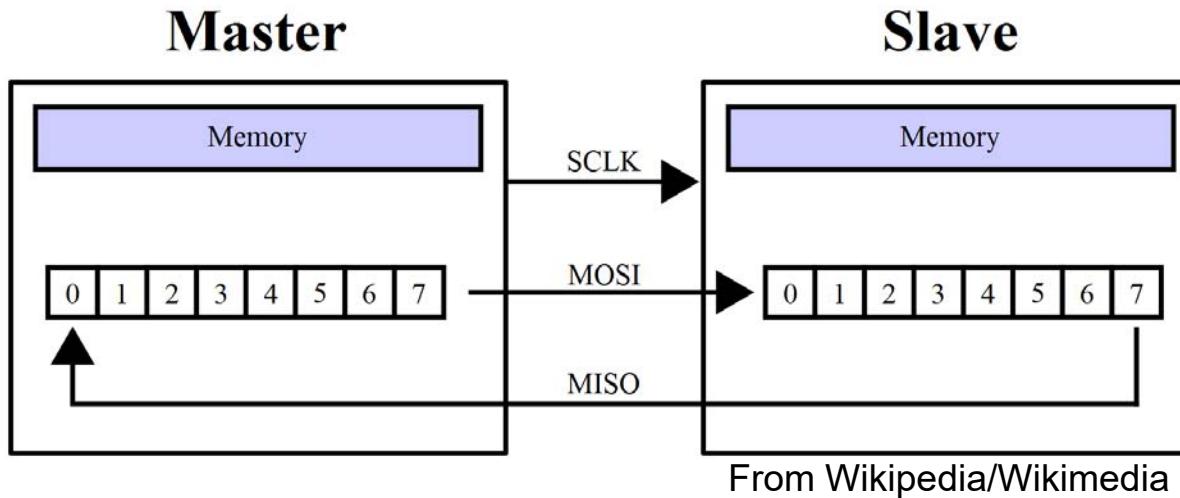
The Serial Peripheral Interface (SPI) bus

- 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.



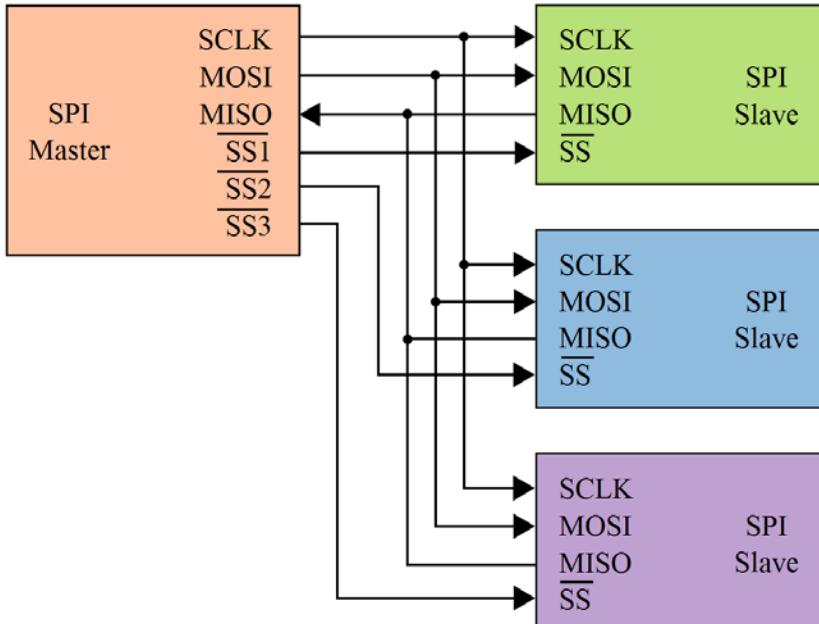
- *Caution:* There is some variation in the spelling of the signal abbreviations. The standard is not well enforced.

SPI Data Transfer Shift Register

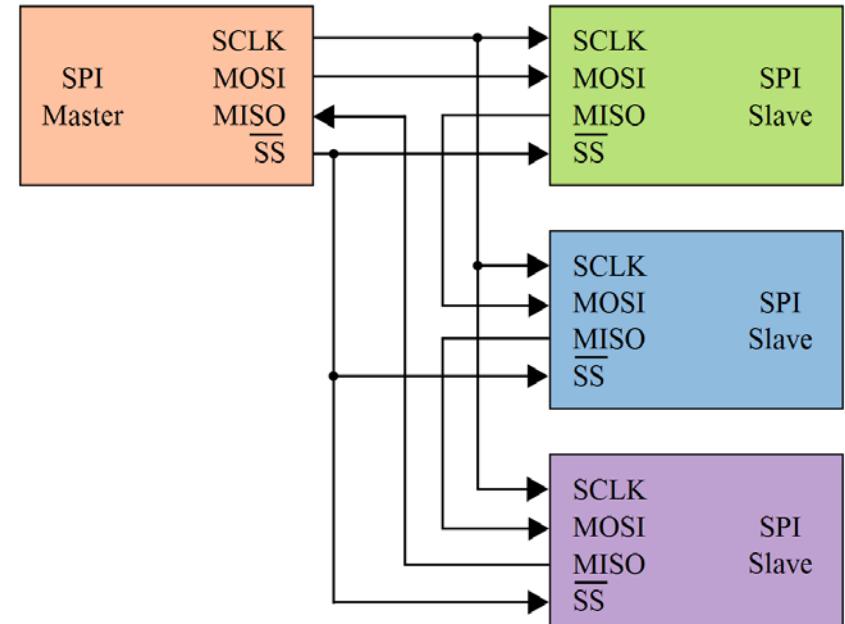


- The standard architecture for transferring data between the master and one slave is to create a large shift register that is split equally between the master and slave.
- New master data shifts over MOSI into the slave while the previous slave data shifts out over MISO to the master. The master effectively 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.

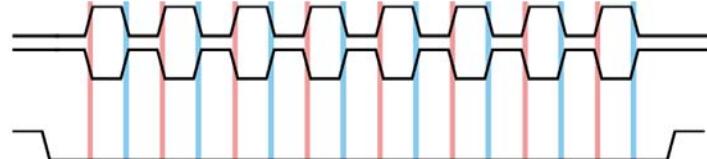
CPHA=0

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

Cycle #

MISO

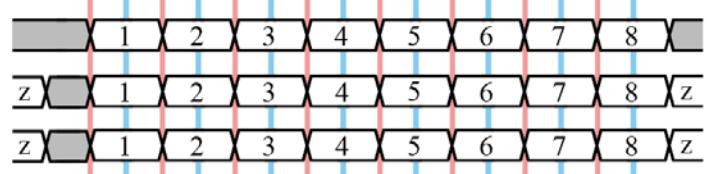
MOSI



Cycle #

MISO

MOSI

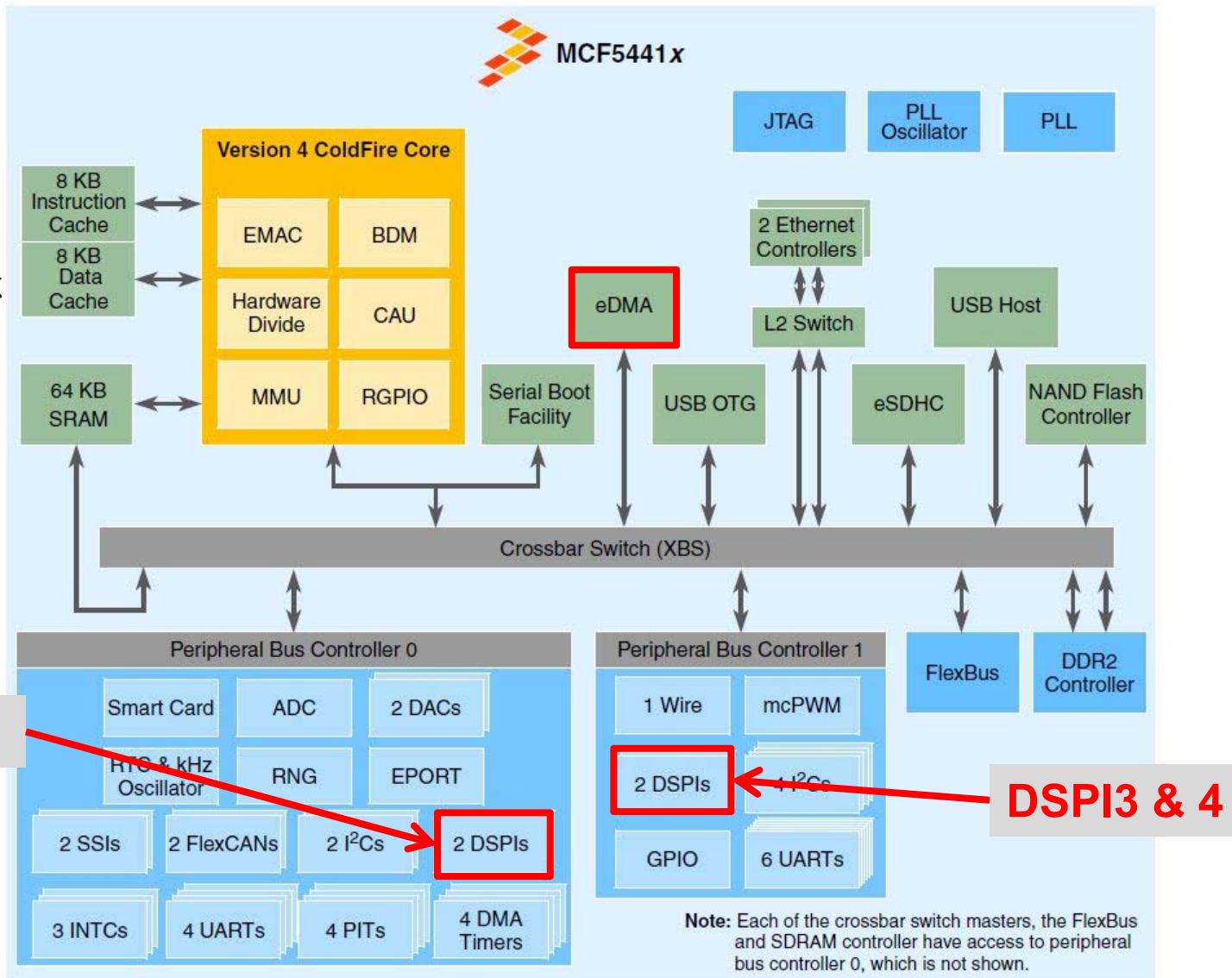


From Wikipedia/Wikimedia

- 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

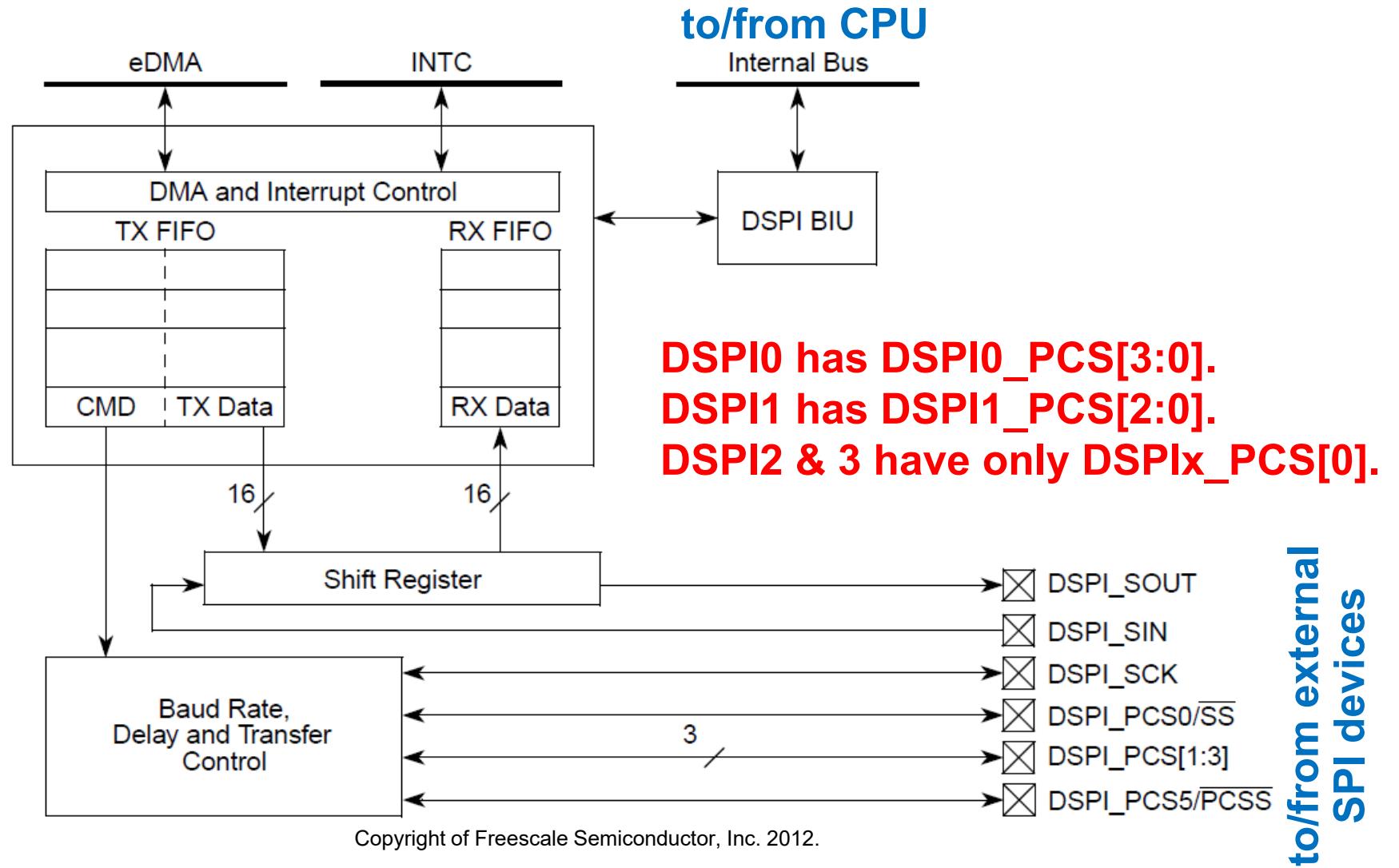
Copyright of Freescale Semiconductor, Inc. 2012.
Used by permission.



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



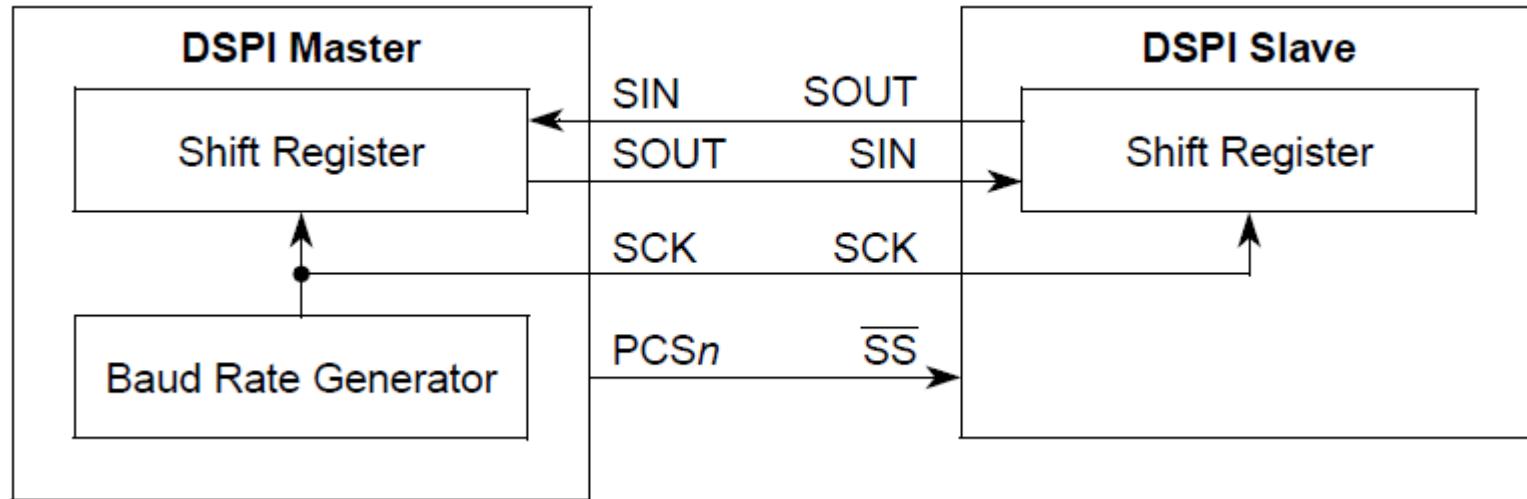
DSPI Interface Architecture on the MCF5441x

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

Offset	Register Name and Abbreviation	Size	Mode
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			

The Ethernet Interface of the NetBurner MOD54415

References:

- Micrel Inc., KS8721B.BT Datasheet, Rev. 2.3, March 2006.
- 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, January 2012.

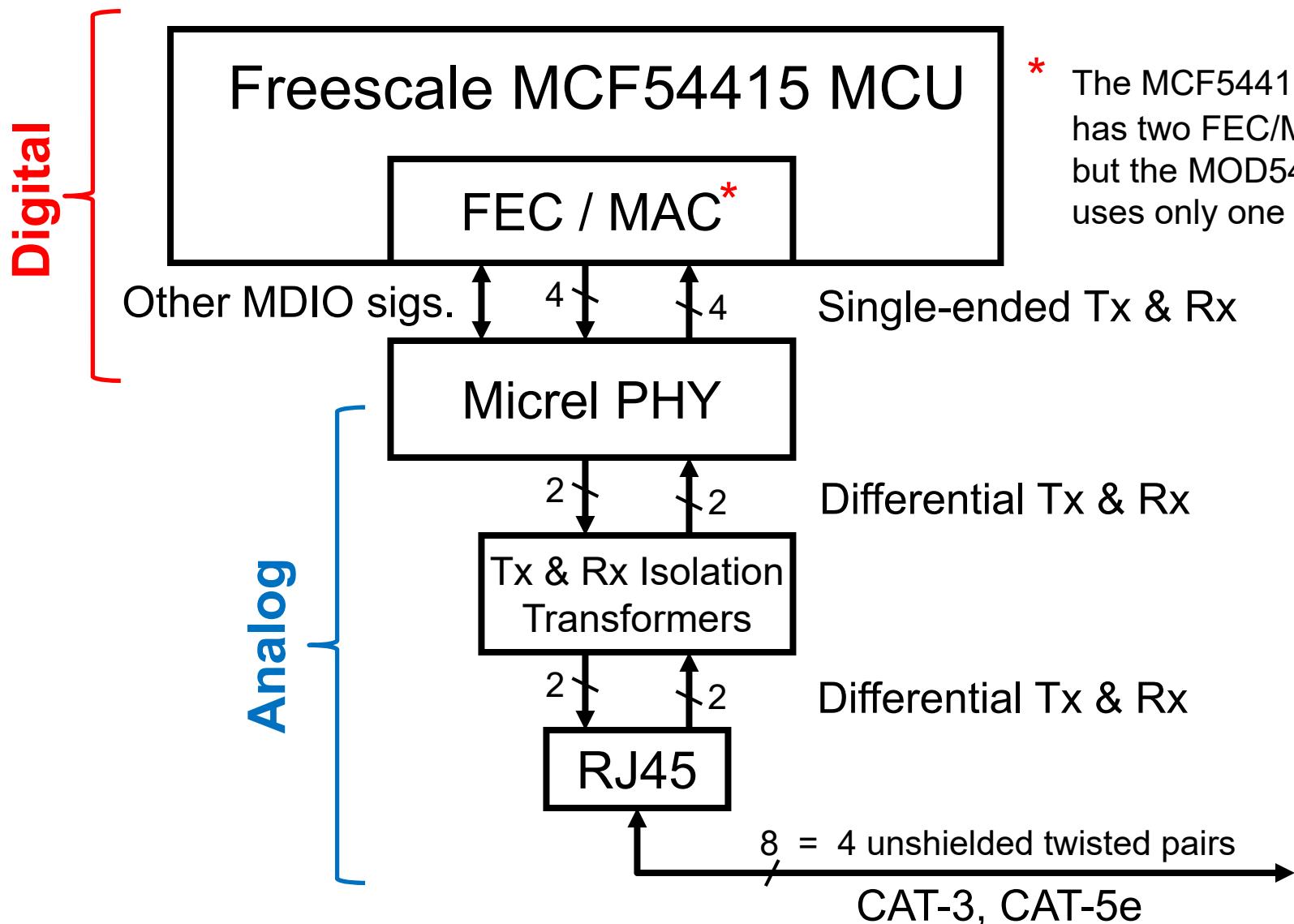
Figures and tables from the above documents have been included in these course notes for educational purposes in ECE 315 only. The original documentation should be consulted to ensure accuracy.

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

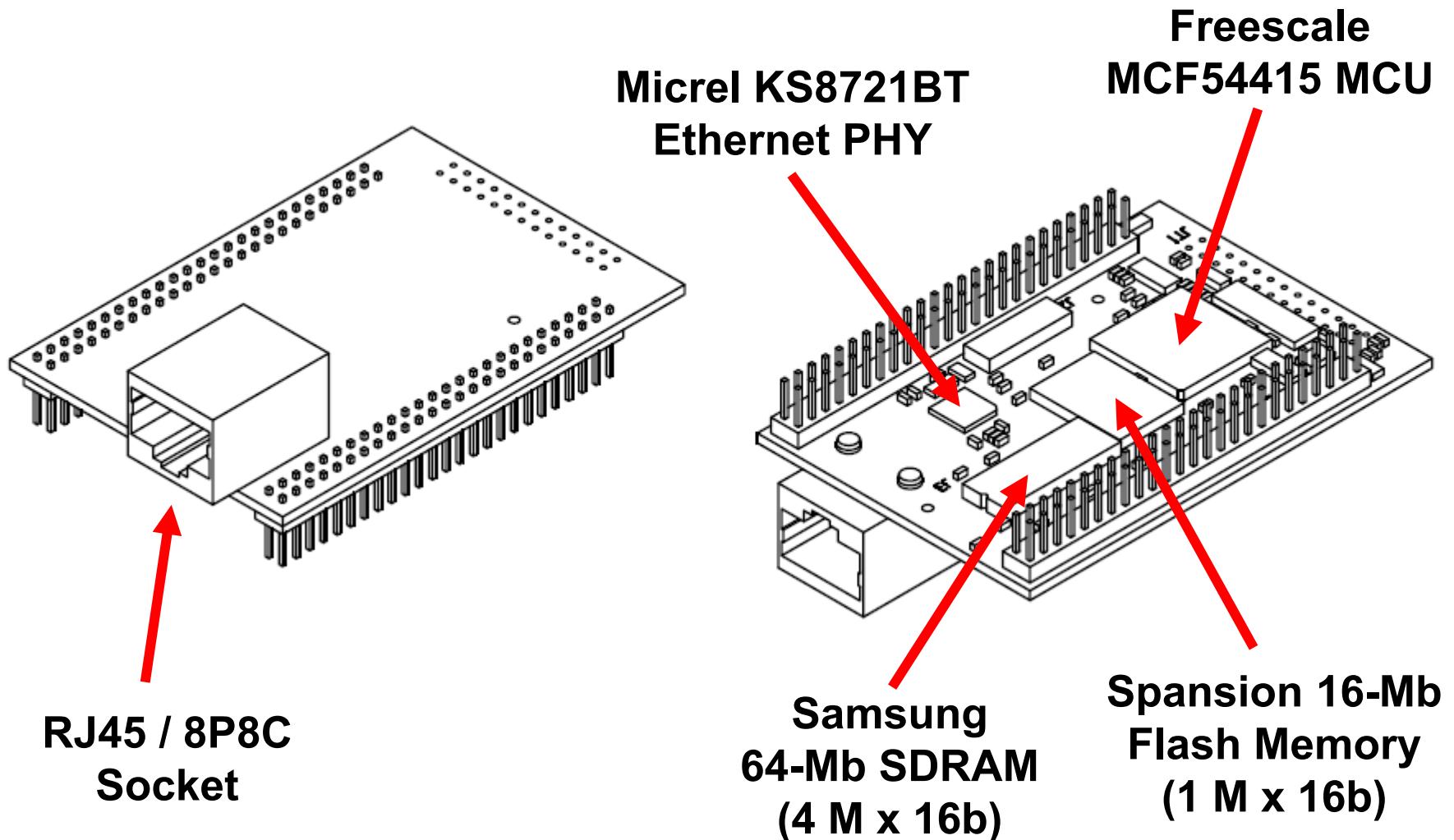
Ethernet Data Rates and Transmission Media

- To simplify the design and reduce the cost, Ethernet interface hardware is usually partitioned into two layers:
 - ***Media Access Control*** (MAC) layer, which implements the data link protocol. This is a purely digital function.
 - ***Physical Layer Transceiver*** (PHY), which provides a connection from the digital MAC to the analog communication medium (electrical, optical, or wireless).
- The MAC-PHY interface has been standardized under ***IEEE Std 802.3 MII Management Interface***, also called the Management Data Input / Output (MDIO) Interface.
- The MAC and PHY blocks can be designed by different teams or companies, and can be implemented in different integrated circuits (ICs) made with different processes.

MAC-PHY Architecture in the MOD54415

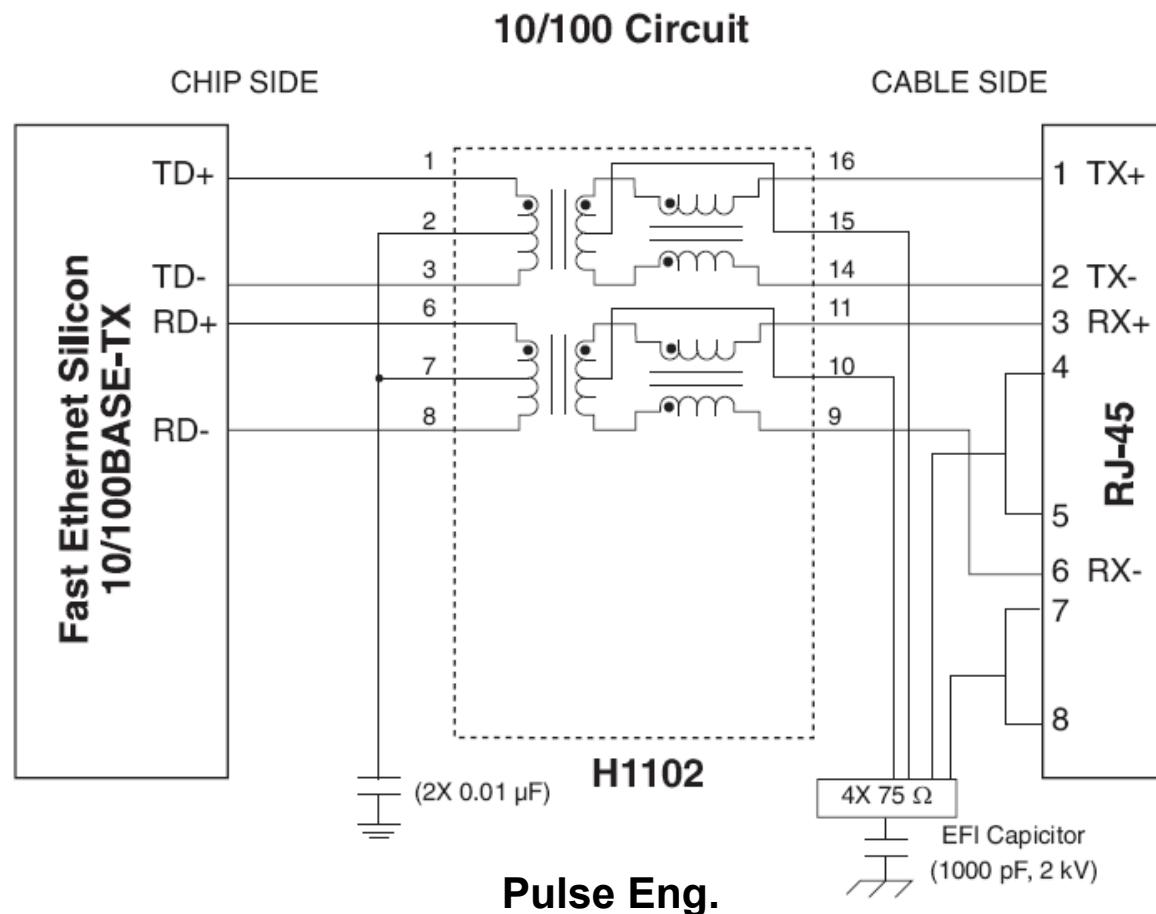


Physical Appearance of the NetBurner MOD54415

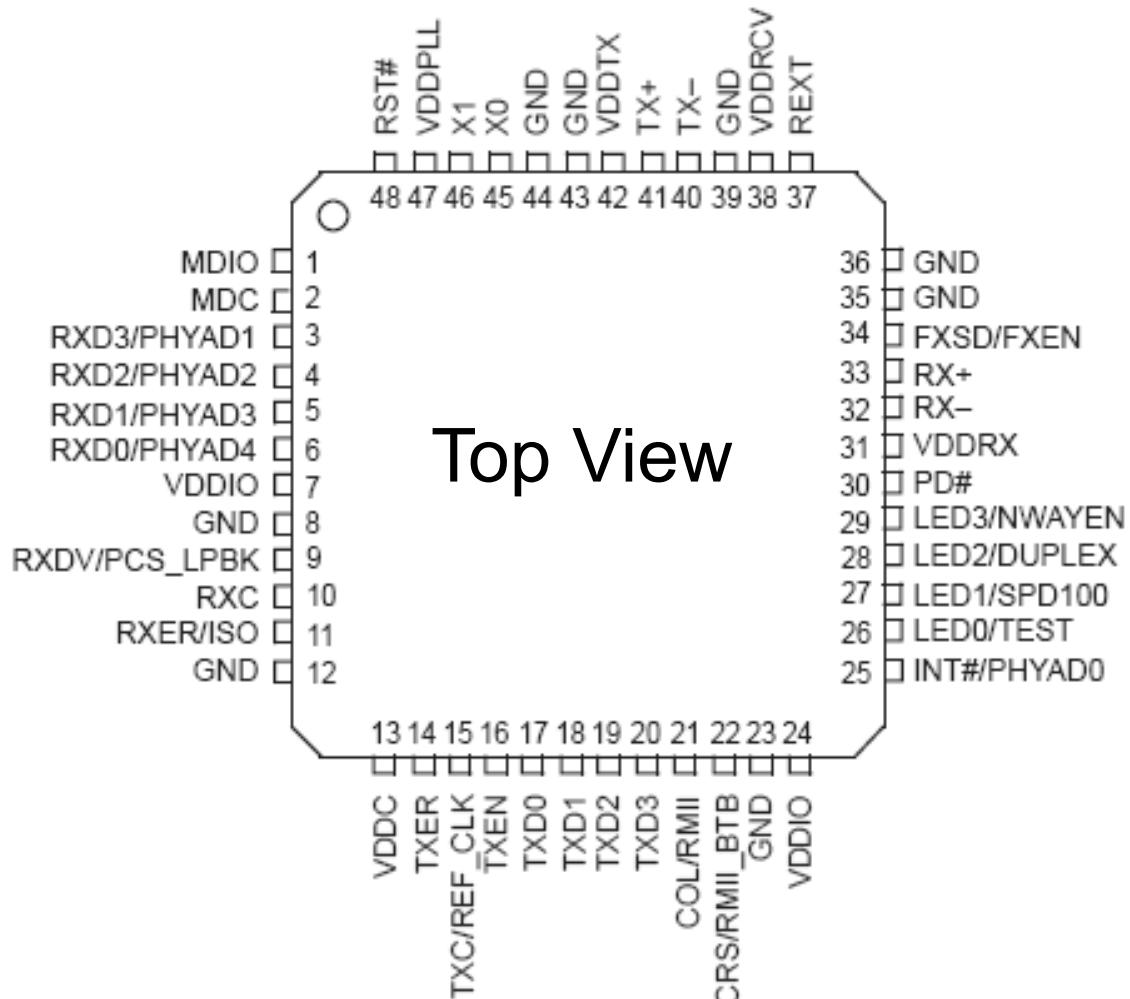


Isolation Transformer Connection Details

An isolation transformer is required by the Ethernet standard so that any offset between the ground potentials at the ends of the connection will have no effect.



Package Outline of the Micrel KS8721BT PHY



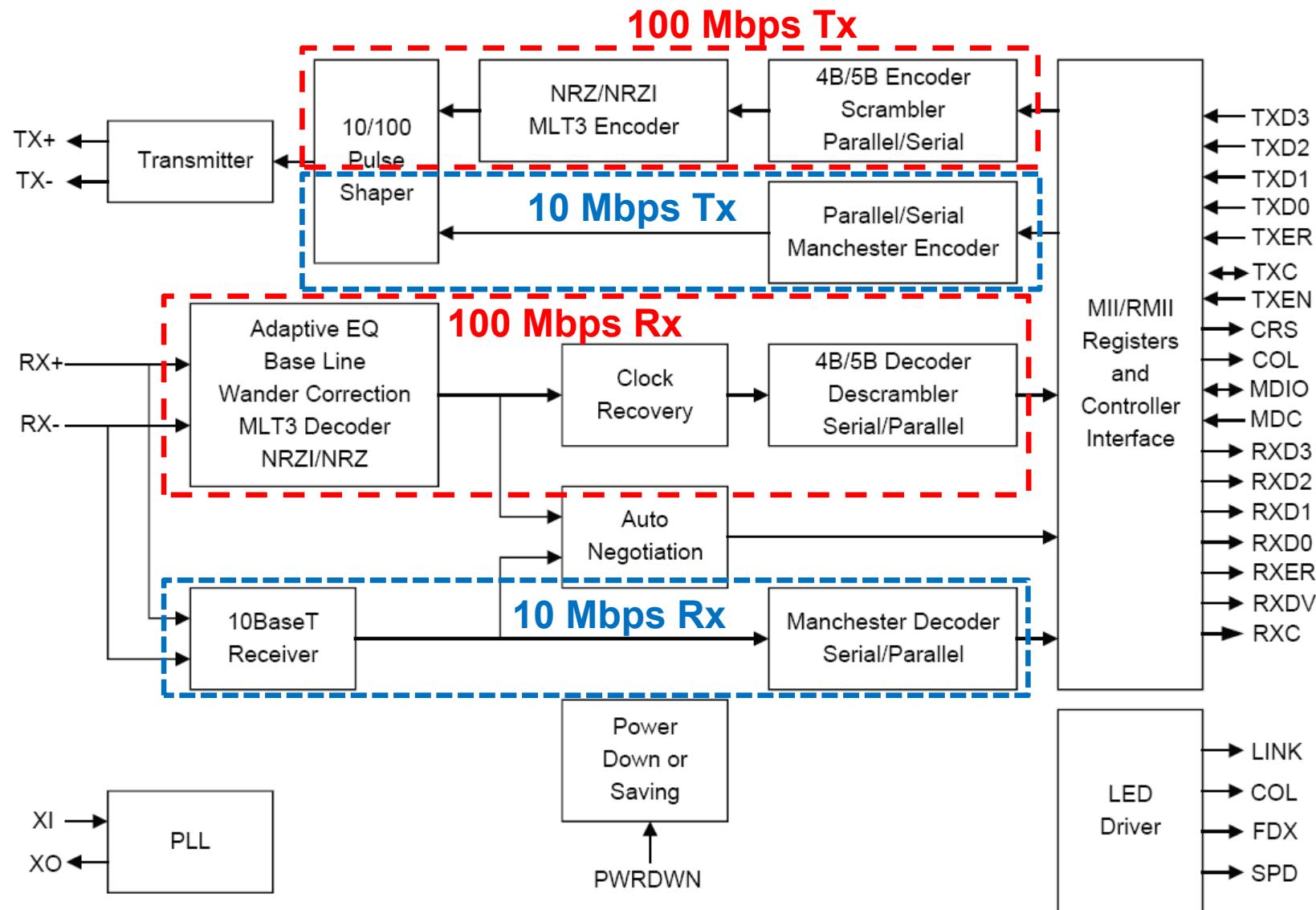
48-Pin TQFP (TQ) = Thin Quad Flat Package

Functional Overview of the Micrel KS8721BT PHY

- Provides a single-chip PHY for both **10** and **100-Mbps Ethernet**:
 - 10BaseT (10 Mbps over CAT-3 cable)
 - 100BaseTX (100 Mbps over CAT-5e cable)
 - 100BaseFX (100 Mbps over optical fibre, with added transceiver)
- Compliant with the IEEE Std 802.3u MAC-PHY interface standard.
- Operation is configurable through a standard MII / MDIO serial management port as well as through external control pins. Twelve 16-bit registers are serially accessible through the MII interface.
- Provides automatic speed negotiation between the 10BaseT and 100BaseTX modes.
- Automatically negotiates full and half-duplex modes.
- Supports a power down mode and a power saving mode.
- Provides on-chip analog filtering to avoid the need for external filters. The connection to the cable medium can be made through a simple isolation transformer.

Functional Blocks inside the Micrel Ethernet PHY

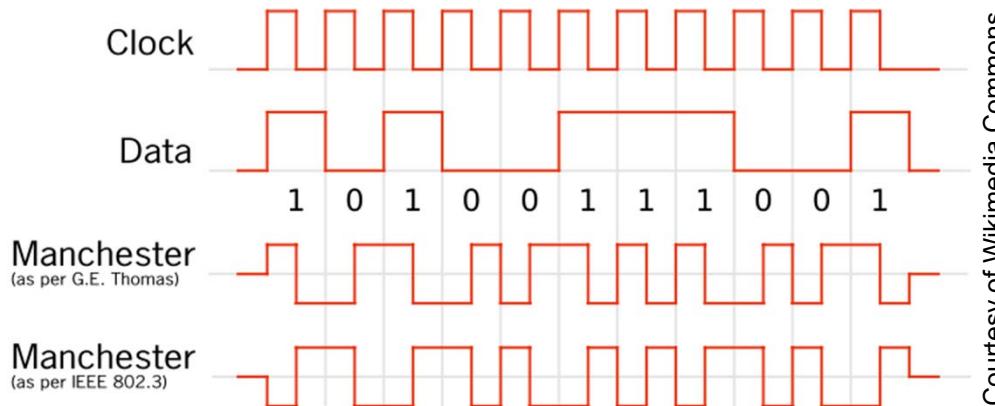
Analog signals to / from line isolation



Digital signals to / from FEC/MAC

The Manchester Code

- The Manchester Code provides two analog waveforms for encoding bits when they are stored or communicated on a physical medium. The code was developed in the late 1940s at the University of Manchester as part of the influential Mark 1 electronic computer project.
- A '0' is represented using a rising edge, while a '1' is represented using a falling edge. IEEE Std 802.3 reversed the waveform mapping to bits.



- Assuming that the bit sequence is reasonably random, the waveform will include a tone at the same frequency as the transmitter bit clock. This makes it easier for the receiver circuit to recover the bit clock.
- Also, the waveform for a bit sequence will not have a DC component, which allows the signal to be passed through capacitors & transformers.

The 4B5B Code

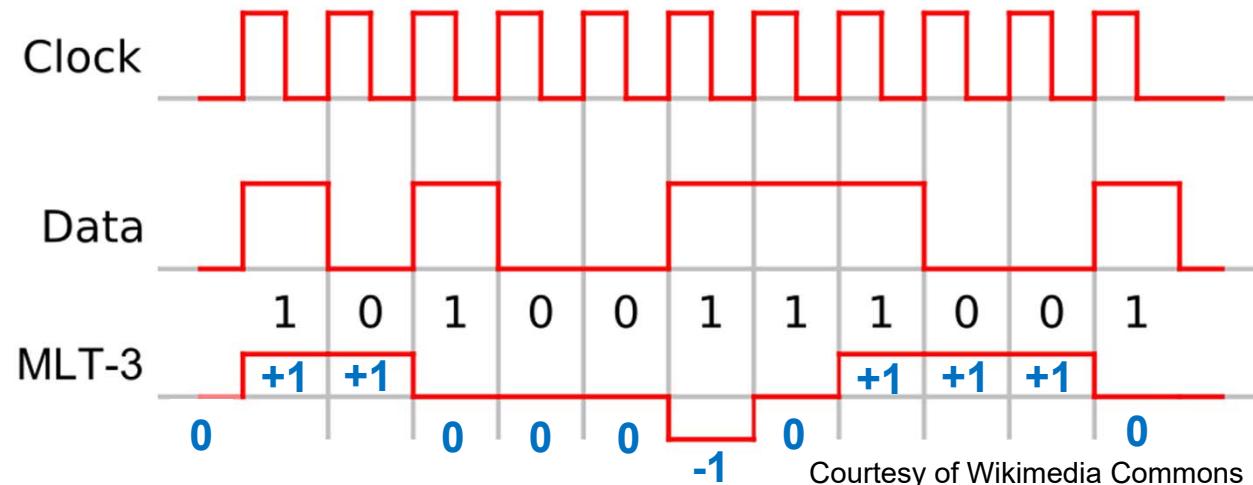
Data		4B5B code
(Hex)	(Binary)	
0	0000	11110
1	0001	01001
2	0010	10100
3	0011	10101
4	0100	01010
5	0101	01011
6	0110	01110
7	0111	01111
8	1000	10010
9	1001	10011
A	1010	10110
B	1011	10111
C	1100	11010
D	1101	11011
E	1110	11100
F	1111	11101

Courtesy of Wikimedia Commons

- The 4B5B line code partitions a given bit sequence into 4-bit groups and then replaces each group with a corresponding 5-bit codeword.
- The codewords in a 4B5B code are chosen to ensure that there is at least one 0-to-1 or 1-to-0 transition in every codeword. Thus, at the cost of increasing the bit rate by 25%, the resulting signal is guaranteed to contain a tone that is related to the original bit clock.
- The 4B5B code shown to the left was adopted in the 100-Mbit/s 100Base-TX Ethernet standard, as defined in IEEE Std 802.3u in 1995.
- Only $2^4 = 16$ of the $2^5 = 32$ possible 5-bit codewords are used to encode data. The 16 remaining codewords can be used to encode control signals.

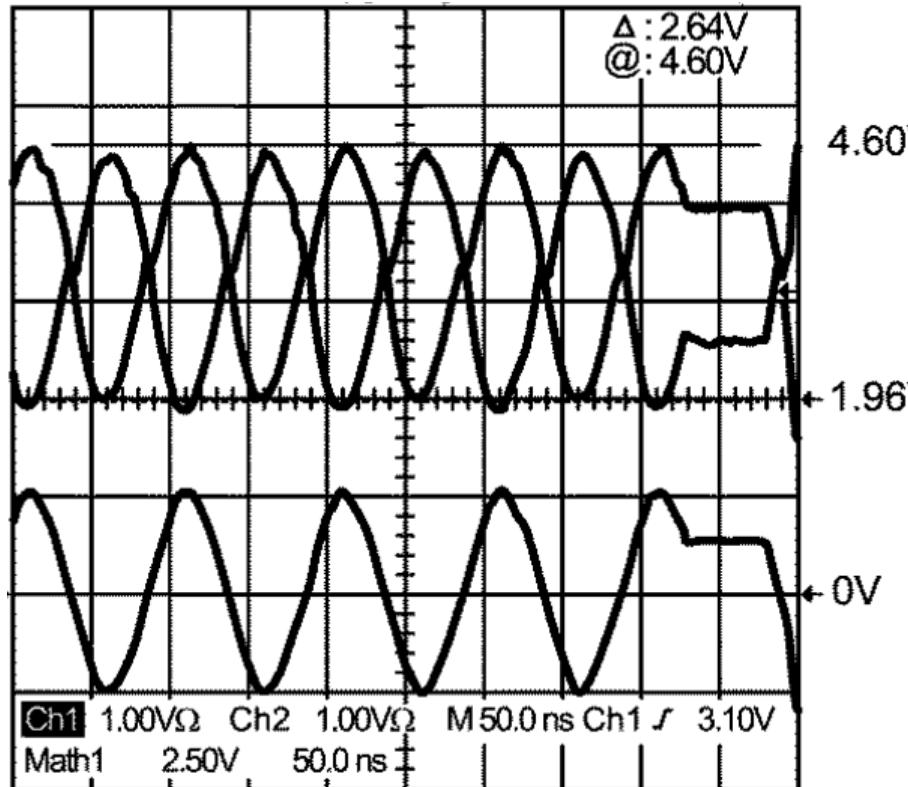
The MLT-3 Line Code

- The MLT-3 line code is a three-level code for signalling binary data on a communications medium. The three levels, which could be voltages, are represented symbolically as -1, 0 and +1.
- When the input bit is a '0', the output signal level does not change.
- When the input bit is a '1', output signal level changes to the next level in the following 4-step staircase sequence: 0, +1, 0, -1, 0, +1, 0, etc.
- An MLT-3 encoder should be preceded by a binary encoder or scrambler to ensure that long input strings containing only 0s do not occur.

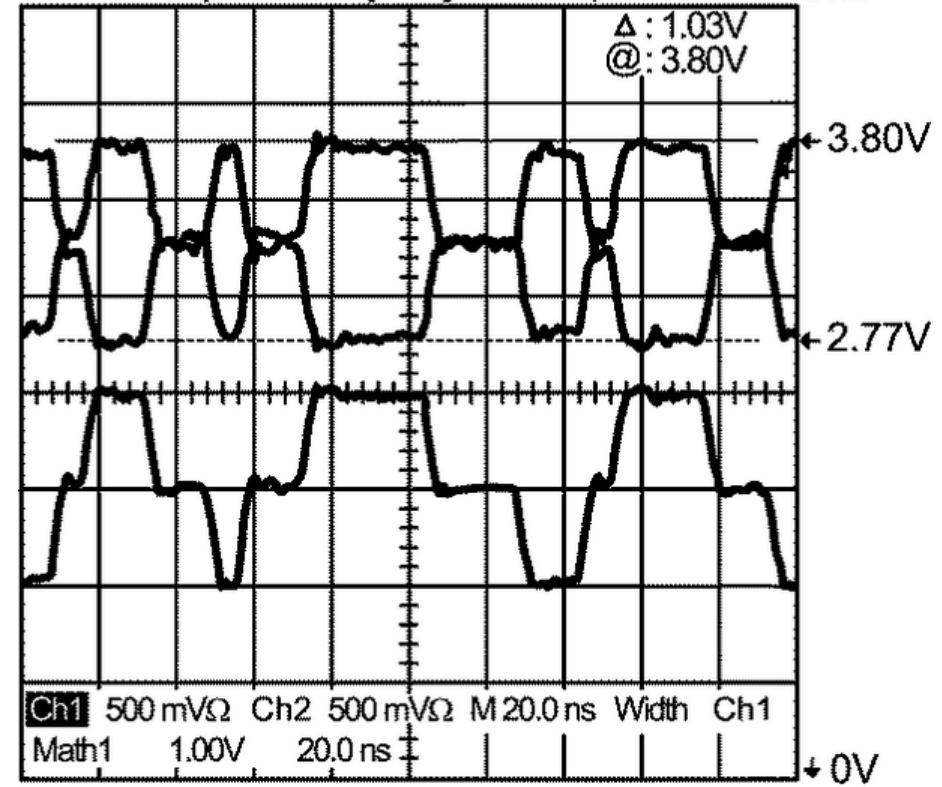


Courtesy of Wikimedia Commons

10BaseT and 100BaseTX Waveforms



10BaseT (10 Mbps)
Manchester-coded, 2-level



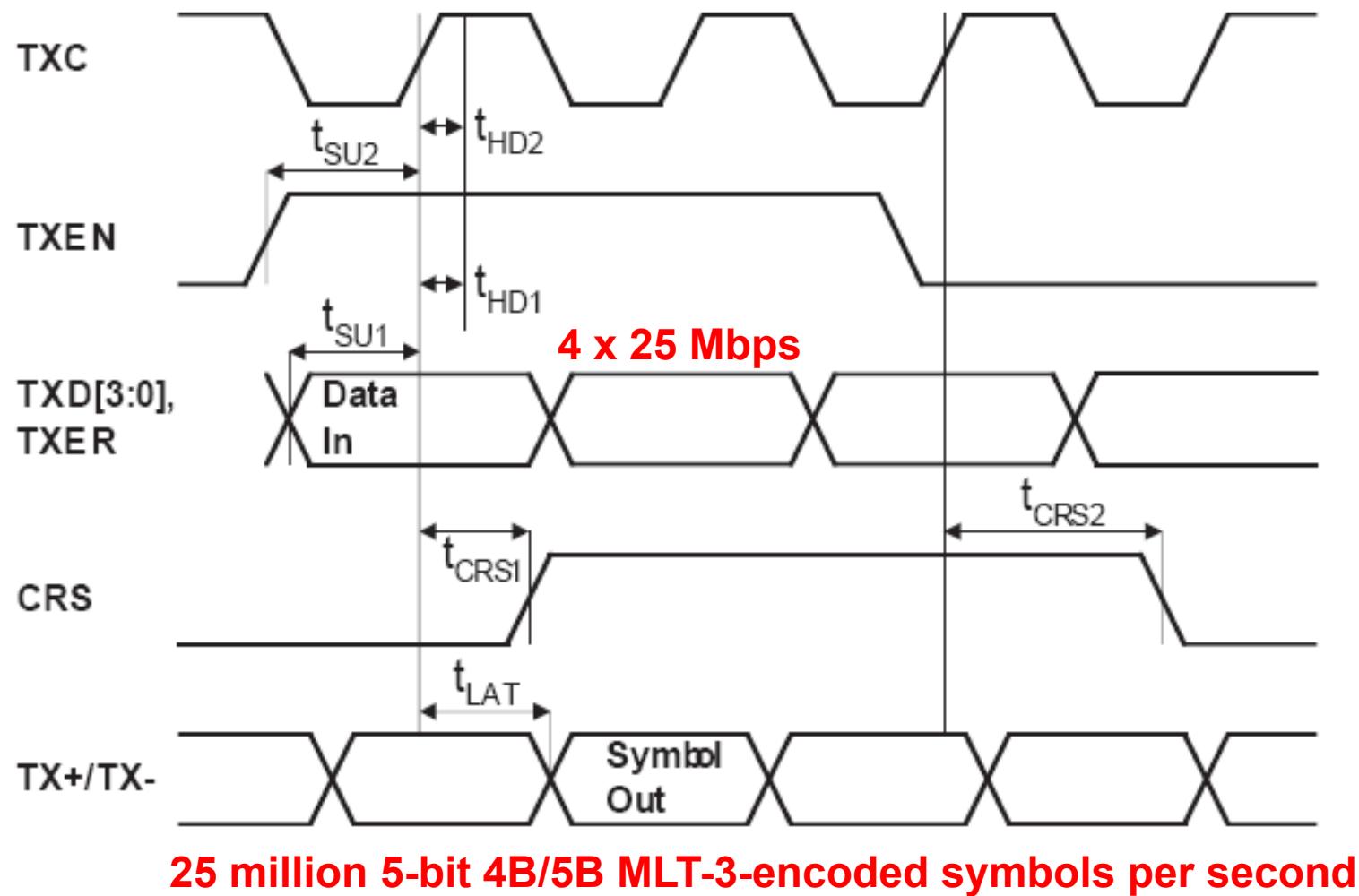
100BaseTX (100 Mbps)
4B/5B-coded, 3-level (MLT-3)

From the Texas Instruments datasheet for DP83848C/I/VYB/YB PHYTER™ QFP Single Port 10/100 Mb/s Ethernet Physical Layer Transceiver, SNLS266E, May 2007.

PHY 100-Mbps Transmit Signals

TXC	MII Transmit Clock (input to PHY from MAC) Derived from 25-MHz clock input at X1 pin.
TXEN	MII Transmit Enable (input)
TXD[3:0]	MII Transmit Data (parallel inputs) at 25 MHz. Raw serial data rate is thus 100 Mbps.
TXER	MII Transmit Error (input)
CRS	MII Carrier Sense (output from PHY to MAC)
TX+/TX-	Differential shaped analog output signals (output from PHY to isolation transformer). Analog bit signals (called “symbols”) transmitted at 125 MHz because of the 4B/5B line code. A 3-level MLT3 waveform is used.

PHY 100-Mbps Transmit Signal Waveforms

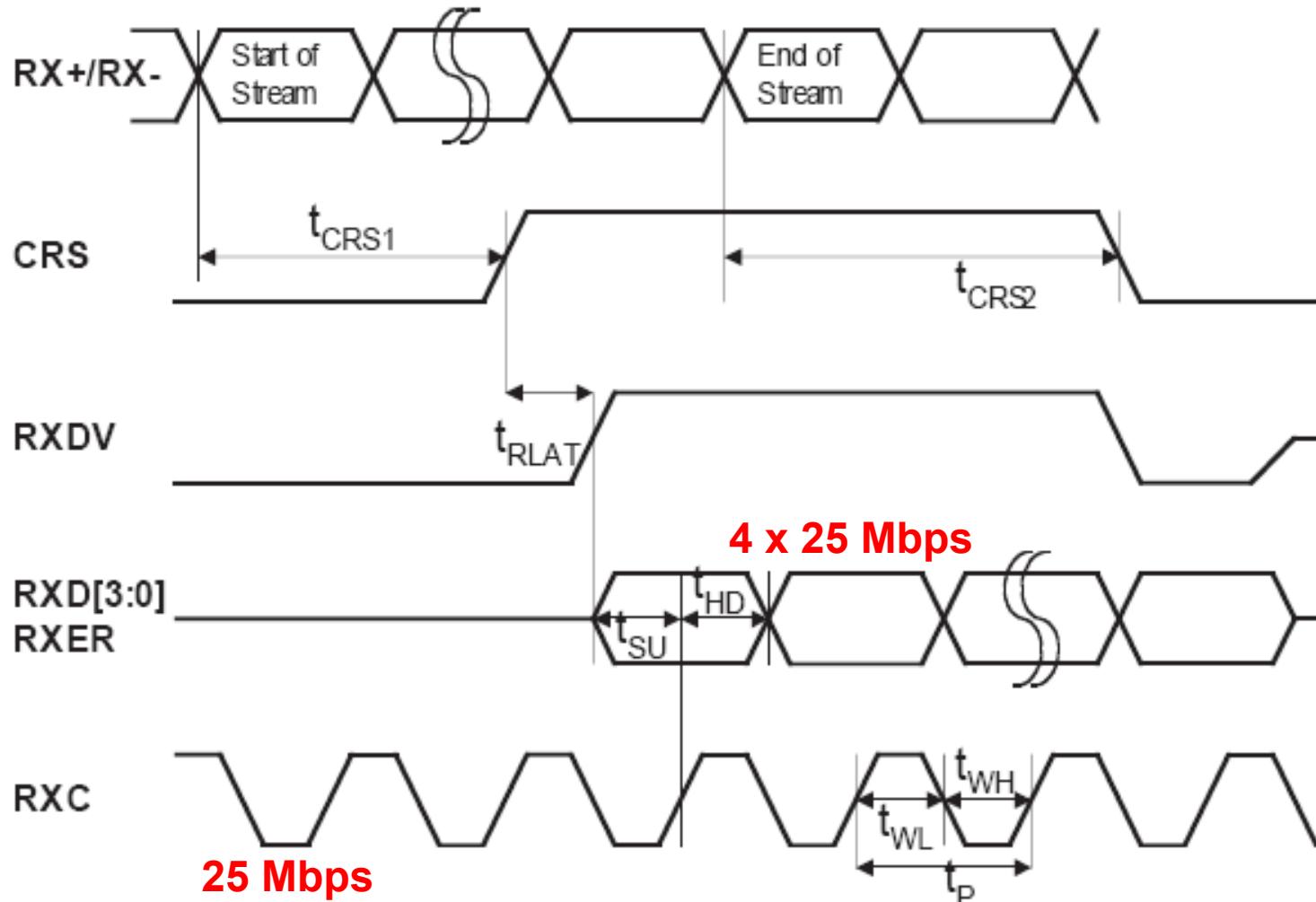


PHY 100-Mbps Receive Signals

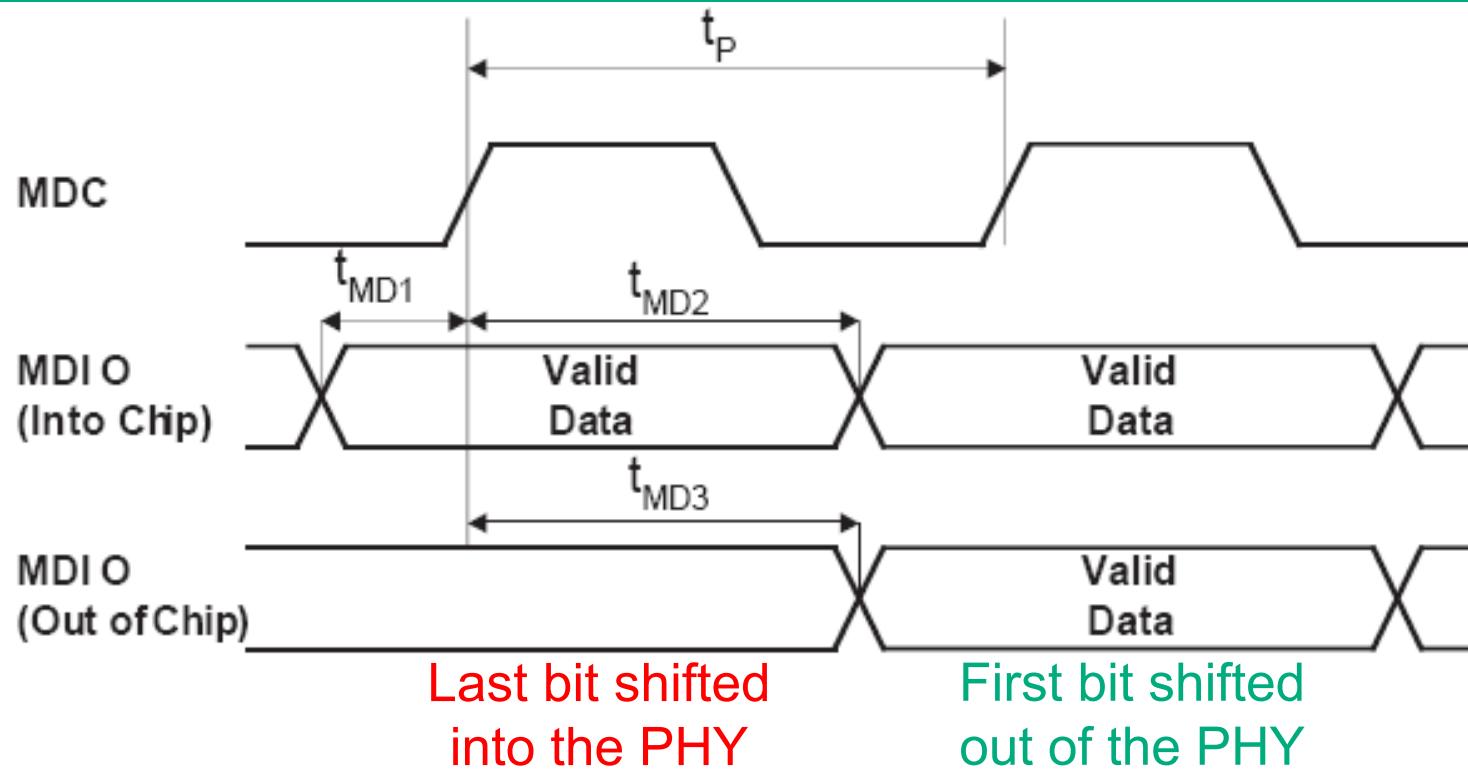
RX+/RX-	Distorted received differential analog signals (signals from isolation transformer to PHY)
CRS	MII Carrier Sense (output from PHY to MAC)
RXDV	MII Receive Data Valid (output to MAC)
RXD[3:0]	MII Receive Data (output to MAC)
RXER	MII Receiver Error (output to MAC)
RXC	MII Receive Clock (output to MAC) 100-Mbps => 25-MHz recovered clock 10-Mbps => 2.5-MHz recovered clock

PHY 100-Mbps Receive Signal Waveforms

25 million 5-bit 4B/5B MLT-3-encoded symbols per second



PHY MII Serial Management Waveforms



Shift-in phase: The MDC clock is used to shift in one 32-bit control frame, via the MDIO pin used as an input. 16 bits are loaded into one of twelve 16-bit PHY registers. An addressing scheme selects the desired register. **Shift-out phase:** Contents of addressed register are shifted out through the MDIO pin, used as an output from the PHY.

Serially-Accessible 16-bit MDIO Registers

Reg. No.	Register Name
0x00	Basic Control Register
0x01	Basic Status Register
0x02	PHY Identifier I
0x03	PHY Identifier II
0x04	Auto-Negotiation Advertisement Register
0x05	Auto-Negotiation Link Partner Ability Register
0x06	Auto-Negotiation Expansion Register
0x07	Auto-Negotiation Next Page Register
0x08	Link Partner Next Page Ability
0x15	RXER Counter Register
0x1b	Interrupt Control/Status Register
0x1f	100BaseTX PHY Control Register

Ex: “Basic Control Register” in the PHY

Bit 15: 1 => reset the PHY (self-clearing bit)

Bit 14: 1 => enable loopback mode

Bit 13: 1 => 100 Mbps mode; 0 => 10 Mbps mode

Bit 12: 1 => enable auto-negotiation mode

Bit 11: 1 => enable power down mode

Bit 10: 1 => isolate the PHY from the MII data and TX+/TX-

Bit 9: 1 => restart auto-negotiation process (self-clearing bit)

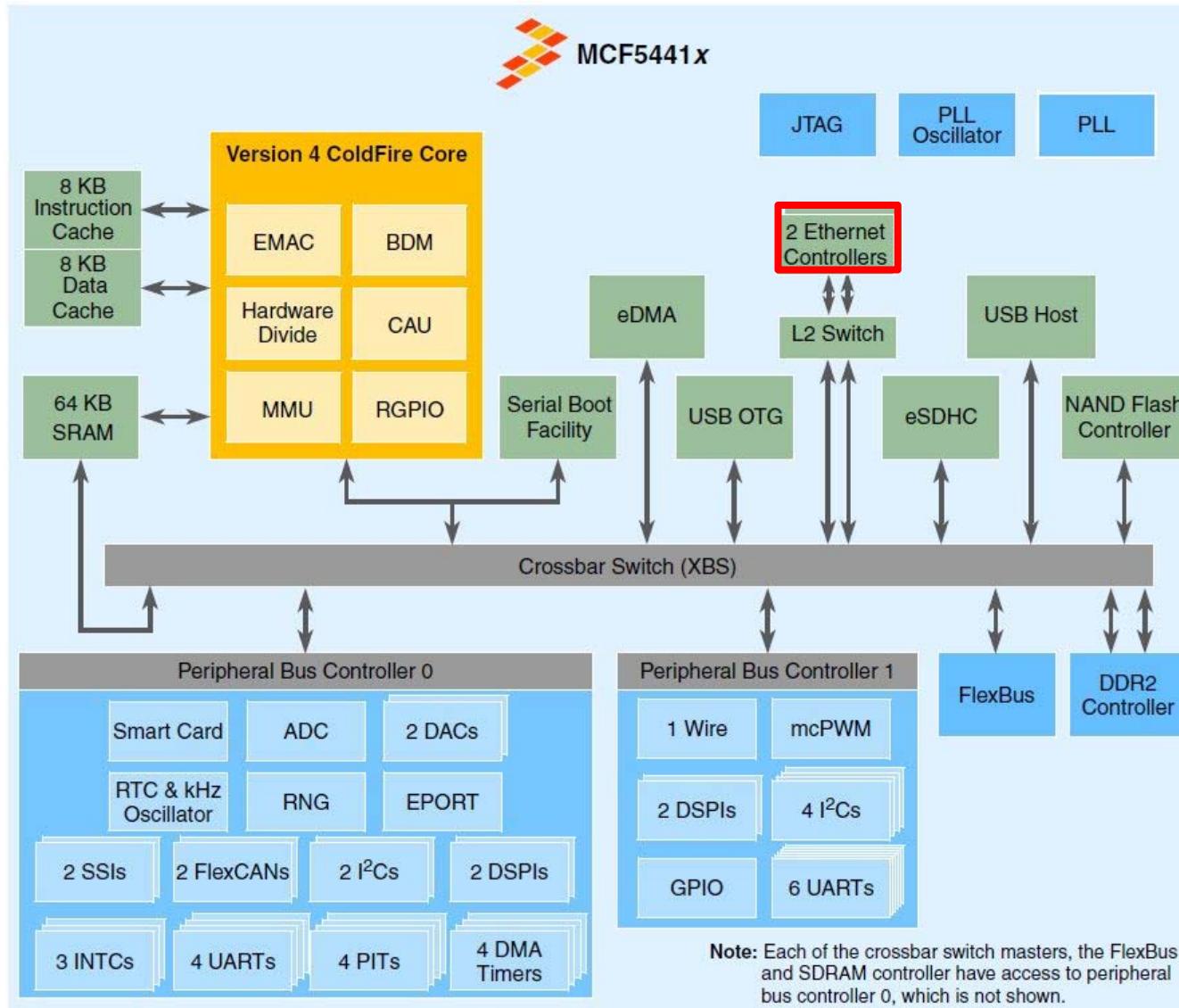
Bit 8: 1 => full duplex mode; 0 => half duplex mode

Bit 7: 1 => enable collision test

Bits 7-1: Reserved bits (for proprietary test modes?)

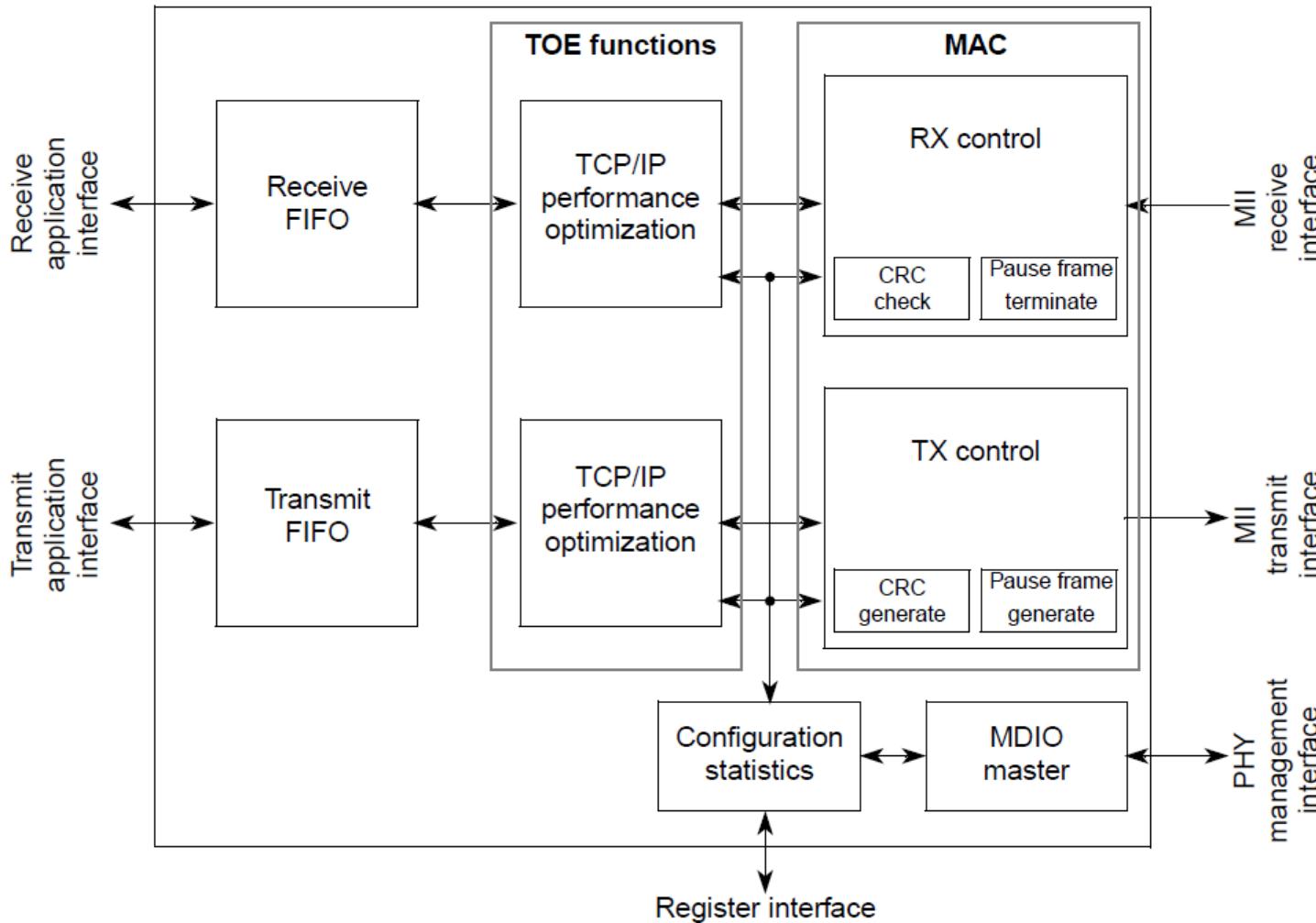
Bit 0: 1 => disable transmitter; 0 => enable transmitter

Architecture of the Freescale MCF54415 µC



MAC-NET Ethernet Interface in the MCF54415 MCU

Digital signals to / from Crossbar Switch



Digital signals to / from PHY

Copyright of Freescale Semiconductor, Inc. 2012.

Copyright © 2020 by Bruce Cockburn

7-21

MII and RMII Signals at the MAC-PHY Interface

Signal		Description
MII	RMII	
MII_COL	—	Asserted upon detection of a collision and remains asserted while the collision persists. This signal is not defined for full-duplex mode.
MII_CRS	—	Carrier sense. When asserted, indicates transmit or receive medium is not idle. In RMII mode, this signal is present on the RMII_CRS_DV pin.
MII_MDC	RMII_MDC	Output clock provides a timing reference to the PHY for data transfers on the MDIO signal.
MII_MDIO	RMII_MDIO	Transfers control information between the external PHY and the media-access controller. Data is synchronous to MDC. This signal is an input after reset.
MII_RXCLK	—	Provides a timing reference for RXDV, RXD[3:0], and RXER.
MII_RXDV	RMII_CRS_DV	Asserting this input indicates the PHY has valid nibbles present on the MII. RXDV must remain asserted from the first recovered nibble of the frame through to the last nibble. Asserting RXDV must start no later than the SFD and exclude any EOF. In RMII mode, this pin also generates the CRS signal.
MII_RXD0	RMII_RXD0	Contains the Ethernet input data transferred from PHY to the media-access controller when RXDV is asserted.
MII_RXD1	RMII_RXD1	
MII_RXD[3:2]	—	
MII_RXER	RMII_RXER	When asserted with RXDV, indicates the PHY detects an error in the current frame. When RXDV is negated, RXER has no effect.
MII_TXCLK	—	Input clock which provides a timing reference for TXEN, TXD[3:0], and TXER.
MII_TXD0	RMII_TXD0	The serial output Ethernet data and only valid during the assertion of TXEN.
MII_TXD1	RMII_TXD1	
MII_TXD[3:2]	—	
MII_TXEN	RMII_TXEN	Indicates when valid nibbles are present on the MII. This signal is asserted with the first nibble of a preamble and is negated before the first TXCLK following the final nibble of the frame.
MII_TXER	—	When asserted for one or more clock cycles while TXEN is also asserted, PHY sends one or more illegal symbols. TXER has no effect at 10 Mbps or when TXEN is negated.
—	RMII_REF_CLK	In RMII mode, this signal is the reference clock for receive, transmit, and the control interface.

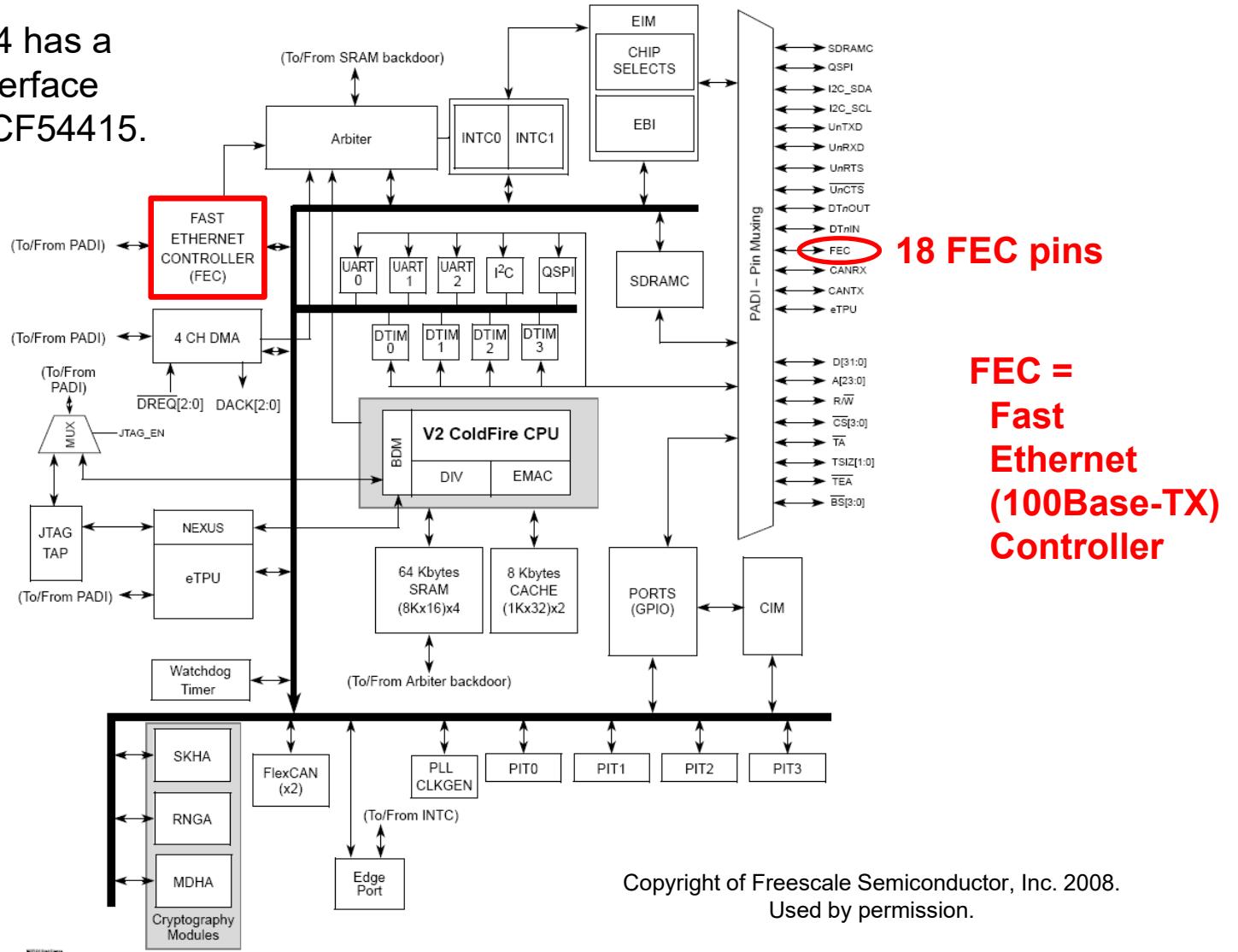
Copyright of Freescale Semiconductor, Inc. 2012.

Copyright © 2020 by Bruce Cockburn

7-22

Architecture of the Freescale MCF5234 MCU

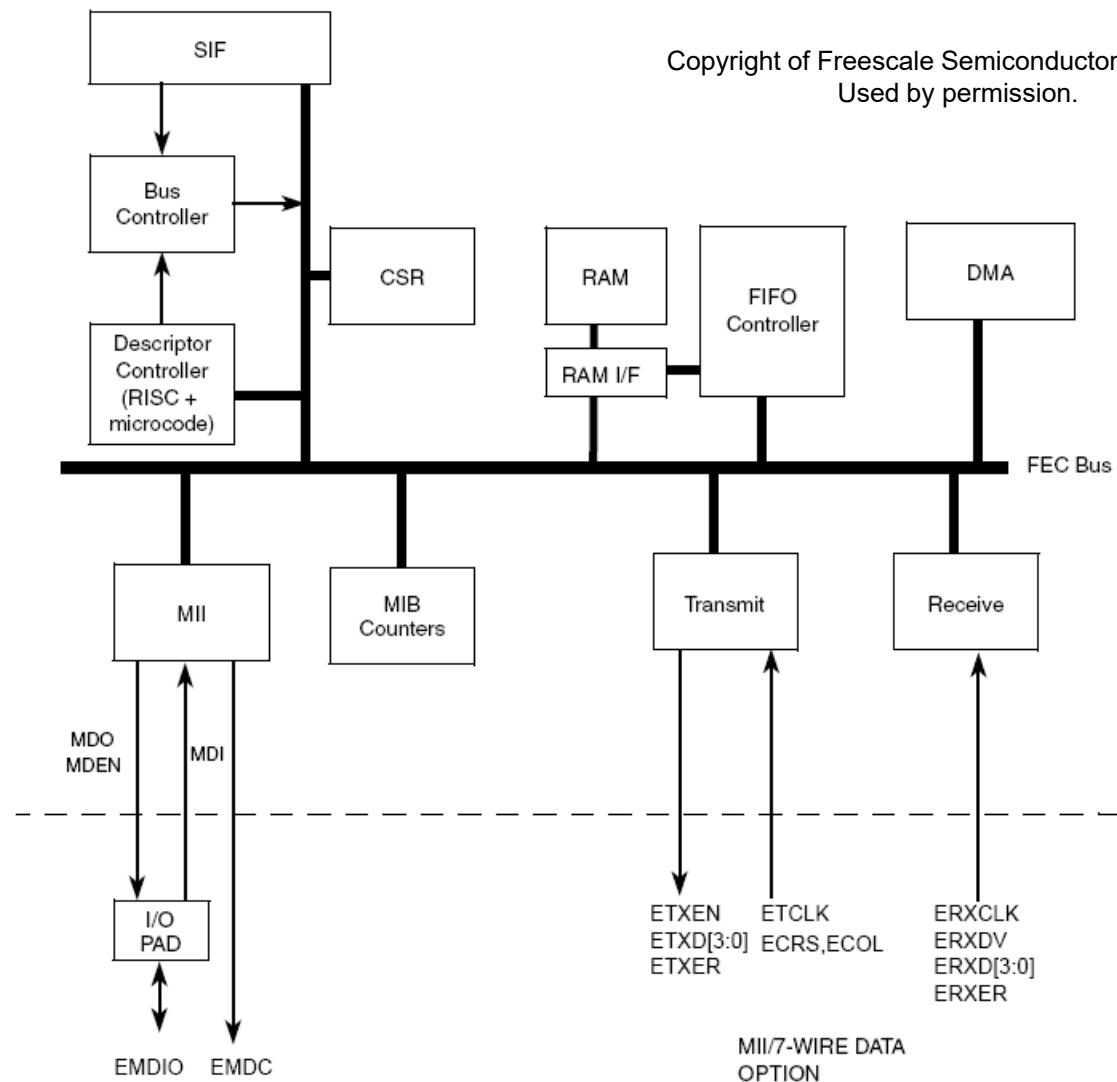
Note: The MCF5234 has a simpler Ethernet interface compared to the MCF54415.



Functional Overview of the FEC

- Provides most ***Media Access Control*** (MAC) functionality
- Supports three kinds of MAC-PHY interfaces:
 - 10Base-T under the full MII standard
 - 10Base-T under a simplified 7-wire RM standard
 - 100Base-TX under the full MII standard
- Data transmission can be full-duplex or half-duplex
- Off-loads most packet transmission chores from the CPU
- Memory-mapped registers in the CPU's memory map
- On-chip transmit and receiver first-in, first-out (FIFO) buffers
- Supports flexible memory-based data structures

Fast Ethernet Controller (FEC) Architecture



Functional Overview of the FEC (cont'd)

- FEC sub-systems communicate with each other over a local **FEC bus**.
- **Descriptor Controller** is a special-purpose FEC controller:
 - Microcode program is fixed during chip manufacture.
 - Initializes private registers (independent of the CPU).
 - Interprets Rx and Tx **data buffer descriptors**.
 - Processes addresses in received data frames.
 - Generates random numbers for **collision back-off timer**.
 - Starts autonomous **Direct Memory Access** (DMA) operations to transfer Rx and Tx data at high speed to and from other locations in the CPU's memory map.

FEC Memory Map (viewed from the ColdFire CPU)

Address	Function
IPSBAR + 0x1000–11FF	Control/Status Registers
IPSBAR + 0x1200–13FF	MIB Block Counters

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

- FEC registers are mapped to 1024 contiguous bytes of memory, starting at the address loaded in the ***Internal Peripheral System Base Address Register*** (IPSBAR) in the System Control Module (SCM) + 0x1000. The default IPSBAR contents are 0x4000_0000.
- Twenty-three 32-bit ***Control/Status Registers*** are used by the CPU to configure the FEC and to monitor FEC status.
- Data is exchanged via data structures, called ***buffer descriptors***, in a buffer region in the CPU's data RAM space.
- ***MIB Block Counters*** automatically count the occurrences of events of likely interest.

FEC Control & Status Registers

IPSBAR Offset	Name	Description	Size (bits)
0x1004	EIR	Interrupt Event Register	32
0x1008	EIMR	Interrupt Mask Register	32
0x1010	RDAR	Receive Descriptor Active Register	32
0x1014	TDAR	Transmit Descriptor Active Register	32
0x1024	ECR	Ethernet Control Register	32
0x1040	MDATA	MII Data Register	32
0x1044	MSCR	MII Speed Control Register	32
0x1064	MIBC	MIB Control/Status Register	32
0x1084	RCR	Receive Control Register	32
0x10C4	TCR	Transmit Control Register	32
0x10E4	PALR	Physical Address Low Register	32
0x10E8	PAUR	Physical Address High+ Type Field	32
0x10EC	OPD	Opcode + Pause Duration	32
0x1118	IAUR	Upper 32 bits of Individual Hash Table	32
0x111C	IALR	Lower 32 Bits of Individual Hash Table	32
0x1120	GAUR	Upper 32 bits of Group Hash Table	32
0x1124	GALR	Lower 32 bits of Group Hash Table	32
0x1144	TFWR	Transmit FIFO Watermark	32
0x114C	FRBR	FIFO Receive Bound Register	32
0x1150	FRSR	FIFO Receive FIFO Start Registers	32
0x1180	ERDSR	Pointer to Receive Descriptor Ring	32
0x1184	ETDSR	Pointer to Transmit Descriptor Ring	32
0x1188	EMRBR	Maximum Receive Buffer Size	32

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

Copyright © 2020 by Bruce Cockburn

7-28

FEC Ethernet Control Register (ECR)

Only the two least significant bits are used:

- Bit #0 controls an ***FEC-level reset***
- Bit #1 controls an ***FEC-level enable***

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	ETHER _EN	RESET
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x1024															

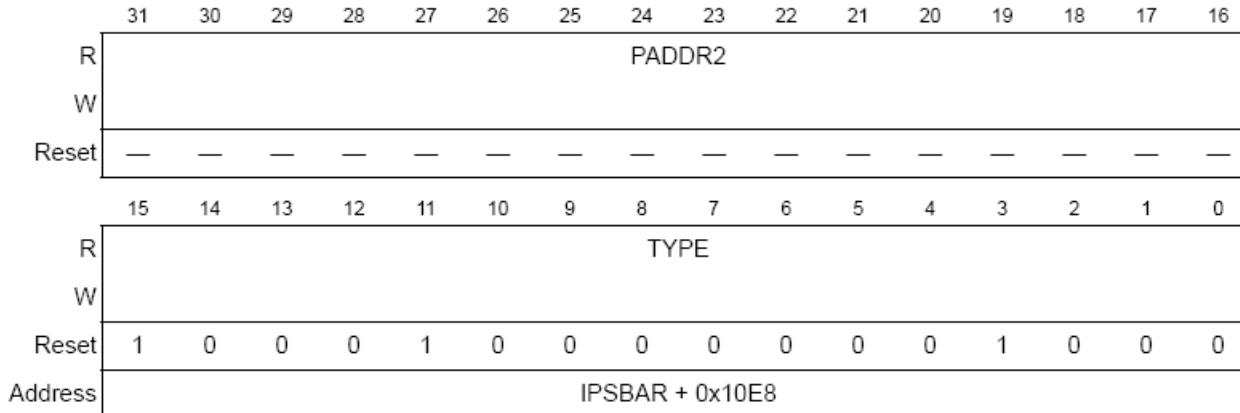
Copyright of Freescale Semiconductor, Inc. 2008. Used by permission.

FEC ECR Bit Definitions

Bits	Name	Description
1	ETHER_EN	<p>When this bit is set, the FEC is enabled, and reception and transmission are possible. When this bit is cleared, reception is immediately stopped and transmission is stopped after a bad CRC is appended to any currently transmitted frame. The buffer descriptor(s) for an aborted transmit frame are not updated after clearing this bit. When ETHER_EN is deasserted, the DMA, buffer descriptor, and FIFO control logic are reset, including the buffer descriptor and FIFO pointers. The ETHER_EN bit is altered by hardware under the following conditions:</p> <ul style="list-style-type: none">• ECR[RESET] is set by software, in which case ETHER_EN will be cleared• An error condition causes the EIR[EBERR] bit to set, in which case ETHER_EN will be cleared
0	RESET	<p>When this bit is set, the equivalent of a hardware reset is performed but it is local to the FEC. ETHER_EN is cleared and all other FEC registers take their reset values. Also, any transmission/reception currently in progress is abruptly aborted. This bit is automatically cleared by hardware during the reset sequence. The reset sequence takes approximately 8 system clock cycles after RESET is written with a 1.</p>

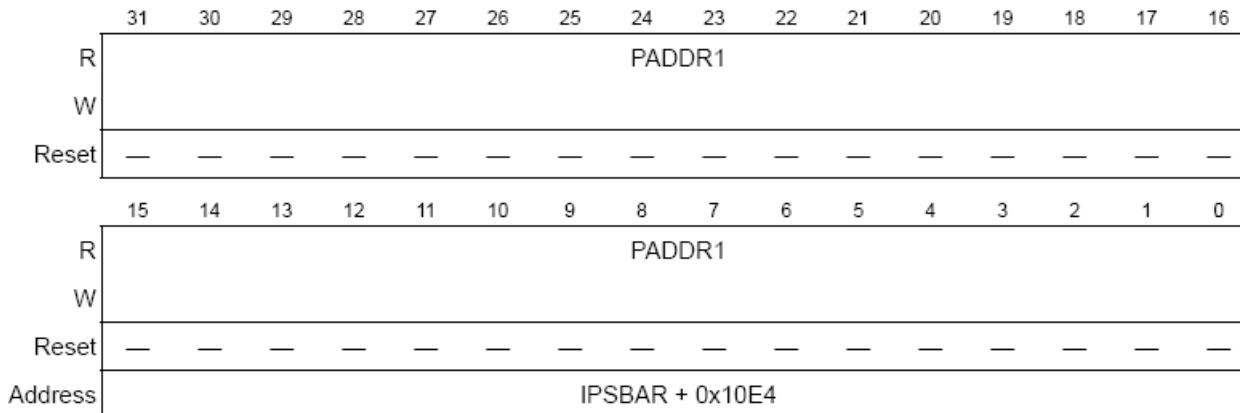
Copyright of Freescale Semiconductor, Inc. 2008. Used by permission.

FEC Physical Address Upper Register (PAUR)



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

FEC Physical Address Lower Register (PALR)



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

FEC Transmit Control Register (TCR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	0	0	0	0	0	0	RFC_PAUSE	TFC_PAUSE	FEDN	HBC	GTS
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x10C4															

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

FEC TCR Bit Definitions

Bits	Name	Description
31–5	—	Reserved, should be cleared.
4	RFC_PAUSE	Receive frame control pause. This read-only status bit will be asserted when a full duplex flow control pause frame has been received and the transmitter is paused for the duration defined in this pause frame. This bit will automatically clear when the pause duration is complete.
3	TFC_PAUSE	Transmit frame control pause. Transmits a PAUSE frame when asserted. When this bit is set, the MAC will stop transmission of data frames after the current transmission is complete. At this time, the GRA interrupt in the EIR register will be asserted. With transmission of data frames stopped, the MAC will transmit a MAC Control PAUSE frame. Next, the MAC will clear the TFC_PAUSE bit and resume transmitting data frames. Note that if the transmitter is paused due to user assertion of GTS or reception of a PAUSE frame, the MAC may still transmit a MAC Control PAUSE frame.
2	FDEN	Full duplex enable. If set, frames are transmitted independent of carrier sense and collision inputs. This bit should only be modified when ETHER_EN is deasserted.
1	HBC	Heartbeat control. If set, the heartbeat check is performed following end of transmission and the HB bit in the status register will be set if the collision input does not assert within the heartbeat window. This bit should only be modified when ETHER_EN is deasserted.
0	GTS	Graceful transmit stop. When this bit is set, the MAC will stop transmission after any frame that is currently being transmitted is complete and the GRA interrupt in the EIR register will be asserted. If frame transmission is not currently underway, the GRA interrupt will be asserted immediately. Once transmission has completed, a “restart” can be accomplished by clearing the GTS bit. The next frame in the transmit FIFO will then be transmitted. If an early collision occurs during transmission when GTS = 1, transmission will stop after the collision. The frame will be transmitted again once GTS is cleared. Note that there may be old frames in the transmit FIFO that will be transmitted when GTS is reasserted. To avoid this deassert ECR[ETHER_EN] following the GRA interrupt.

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

Copyright © 2020 by Bruce Cockburn

7-33

FEC Receive Control Register (RCR)

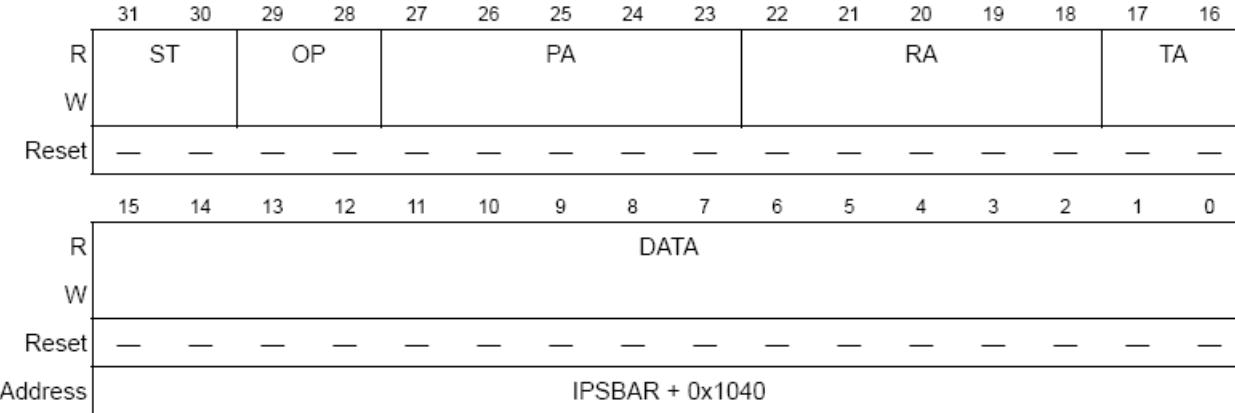
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	0	0											
W																
Reset	0	0	0	0	0	1	0	1	1	1	1	0	1	1	1	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	0	0	0	0	0	FCE	BC_ REJ	PROM	MII_ MODE	DRT	LOOP
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Address	IPSBAR + 0x1084															

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

FEC RCR Bit Definitions

Bits	Name	Description
31–27	—	Reserved, should be cleared.
26–16	MAX_FL	Maximum frame length. Resets to decimal 1518. Length is measured starting at DA and includes the CRC at the end of the frame. Transmit frames longer than MAX_FL will cause the BABT interrupt to occur. Receive Frames longer than MAX_FL will cause the BABR interrupt to occur and will set the LG bit in the end of frame receive buffer descriptor. The recommended default value to be programmed by the user is 1518 or 1522 (if VLAN Tags are supported).
15–6	—	Reserved, should be cleared.
5	FCE	Flow control enable. If asserted, the receiver will detect PAUSE frames. Upon PAUSE frame detection, the transmitter will stop transmitting data frames for a given duration.
4	BC_REJ	Broadcast frame reject. If asserted, frames with DA (destination address) = FF_FF_FF_FF_FF_FF will be rejected unless the PROM bit is set. If both BC_REJ and PROM = 1, then frames with broadcast DA will be accepted and the M (MISS) bit will be set in the receive buffer descriptor.
3	PROM	Promiscuous mode. All frames are accepted regardless of address matching.
2	MII_MODE	Media independent interface mode. Selects external interface mode. Setting this bit to one selects MII mode, setting this bit equal to zero selects 7-wire mode (used only for serial 10 Mbps). This bit controls the interface mode for both transmit and receive blocks.
1	DRT	Disable receive on transmit. 0 Receive path operates independently of transmit (use for full duplex or to monitor transmit activity in half duplex mode). 1 Disable reception of frames while transmitting (normally used for half duplex mode).
0	LOOP	Internal loopback. If set, transmitted frames are looped back internal to the device and the transmit output signals are not asserted. The system clock is substituted for the ETXCLK when LOOP is asserted. DRT must be set to zero when asserting LOOP.

MII Management Frame Register (MMFR)



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

Bit	Name	Description
31–30	ST	Start of frame delimiter. These bits must be programmed to 01 for a valid MII management frame.
29–28	OP	Operation code. This field must be programmed to 10 (read) or 01 (write) to generate a valid MII management frame. A value of 11 will produce “read” frame operation while a value of 00 will produce “write” frame operation, but these frames will not be MII compliant.
27–23	PA	PHY address. This field specifies one of up to 32 attached PHY devices.
22–18	RA	Register address. This field specifies one of up to 32 registers within the specified PHY device.
17–16	TA	Turn around. This field must be programmed to 10 to generate a valid MII management frame.
15–0	DATA	Management frame data. This is the field for data to be written to or read from the PHY register.

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

Interrupts Produced by the FEC

- The FEC hardware can be configured to produce hardware interrupts when certain conditions become true.
- Thirteen different possible interrupting conditions have been defined.
- When a condition becomes true, the corresponding bit in the ***Ethernet Interrupt Event Register*** (EIR) is latched to 1 at the next rising clock edge.
- The single hardware interrupt signal from the FEC will be asserted if both a bit in the EIR and the corresponding bit in the ***Ethernet Interrupt Mask Register*** (EIMR) are 1.
- A bit in the EIR is cleared upon hardware reset, and by being written by CPU software to a 1. *Note:* Writing an EIR bit to 0 has no effect.

FEC Ethernet Interrupt Event Register (EIR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	HB ERR	BABR	BABT	GRA	TXF	TXB	RXF	RXB	MII	EB ERR	LC	RL	UN	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x1004															

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

FEC Ethernet Interrupt Mask Register (EIMR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	HB ERR	BABR	BABT	GRA	TXF	TXB	RXF	RXB	MII	EB ERR	LC	RL	UN	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x1008															

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

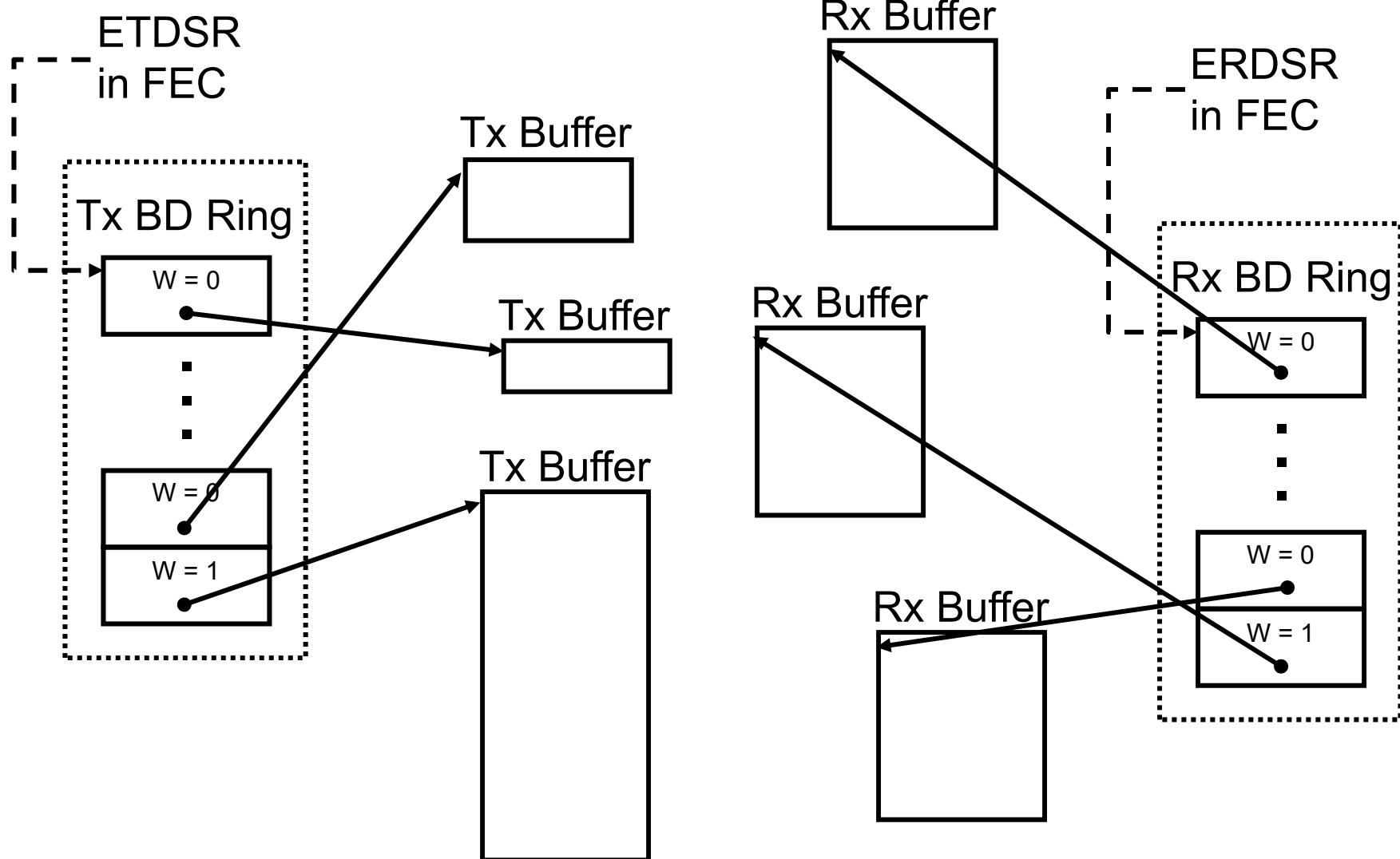
Frequently Used Bits in the EIR and EIMR

Bit No.	Name	Description
27	TXF	Transmit Frame Interrupt: The last Tx buffer in the Tx frame has been transmitted.
26	TXB	Transmit Buffer Interrupt: A Tx Buffer (not the last in the frame) has been transmitted.
25	RXF	Receive Frame Interrupt: The last Rx buffer for the Rx frame has been received.
24	RXB	Receive Buffer Interrupt: An Rx buffer (not the last) has been received.
23	MII	MII Interrupt: MII data transfer completed.
19	UN	Transmit FIFO Underrun: The Tx FIFO emptied out before the frame was fully transmitted.

FEC Transmit and Receive Buffer Overview

- Data is processed by the application(s) in blocks called ***payloads***.
- In addition to the application data, address information must be associated with the payload (e.g., TCP, IP and Ethernet ***headers*** and an Ethernet Cyclic Redundancy Check (CRC) ***trailer***).
- The entire data sequence (headers + payload) is called a ***frame***.
- The frame is typically partitioned and stored in multiple data structures called ***buffers***. Partitioning into buffers makes it easier for different software layers to access relevant data in the frame.
- The locations of the buffers in memory are controlled using data structures called ***buffer descriptors*** (BDs).
- Each buffer descriptor contains a ***pointer*** to one buffer.
- The transmit and receive directions have ***independent buffer systems***, with separate pools of buffer descriptors and buffers.

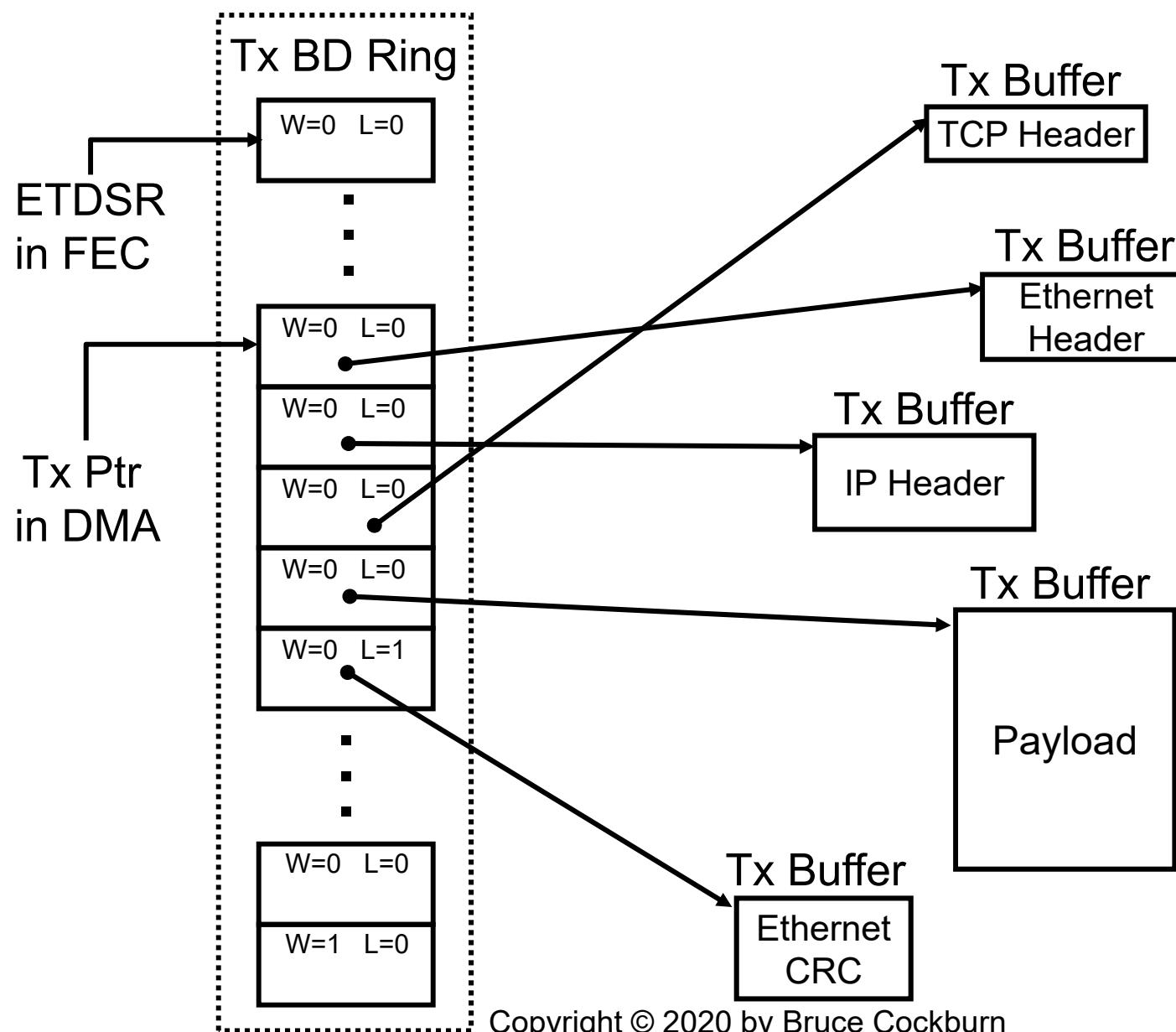
Data Buffers in External Memory



Transmit and Receive Buffer Overview (cont'd)

- The Tx and Rx buffer descriptors are 8-byte data structures that are packed into two arrays called the **Tx** and **Rx BD Rings**.
- The Tx and Rx BDs have multiple fields (described later).
- The W field in the last BD in a ring is set to 1; otherwise, it is 0.
- The Tx buffers can have variable length, as specified in the data length field in the corresponding Tx BDs.
- The Rx buffers have the same length, which is specified in the **Ethernet Maximum Receiver Buffer Size Register** (EMRBR). This length is negotiated in the TCP layer during connection set-up.
- The Tx and Rx buffers are automatically concatenated together to form a frame. The field L = 1 in the BD corresponding to the last buffer in a frame; the field L = 0 in the BDs for all other buffers in the frame.
- A buffer management system keeps track of the pool of buffers and recycles used empty buffers into a list of idle buffers.

Example of a Tx Frame



Transmit Buffer Descriptor (TxBD) 1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Offset + 0	R	TO1	W	TO2	L	TC	ABC	—	—	—	—	—	—	—	—	—
Data Length																
Tx Data Buffer Pointer - A[31:16]																
Tx Data Buffer Pointer - A[15:0]																

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

Word	Bits	Field Name	Description
Offset + 0	15	R	<p>Ready. Written by the FEC and the user.</p> <p>0 The data buffer associated with this BD is not ready for transmission. The user is free to manipulate this BD or its associated data buffer. The FEC clears this bit after the buffer has been transmitted or after an error condition is encountered.</p> <p>1 The data buffer, which has been prepared for transmission by the user, has not been transmitted or is currently being transmitted. No fields of this BD may be written by the user once this bit is set.</p>
Offset + 0	14	TO1	Transmit software ownership. This field is reserved for software use. This read/write bit will not be modified by hardware, nor will its value affect hardware.
Offset + 0	13	W	<p>Wrap. Written by user.</p> <p>0 The next buffer descriptor is found in the consecutive location</p> <p>1 The next buffer descriptor is found at the location defined in ETDSR.</p>

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

Transmit Buffer Descriptor (TxBD) 2

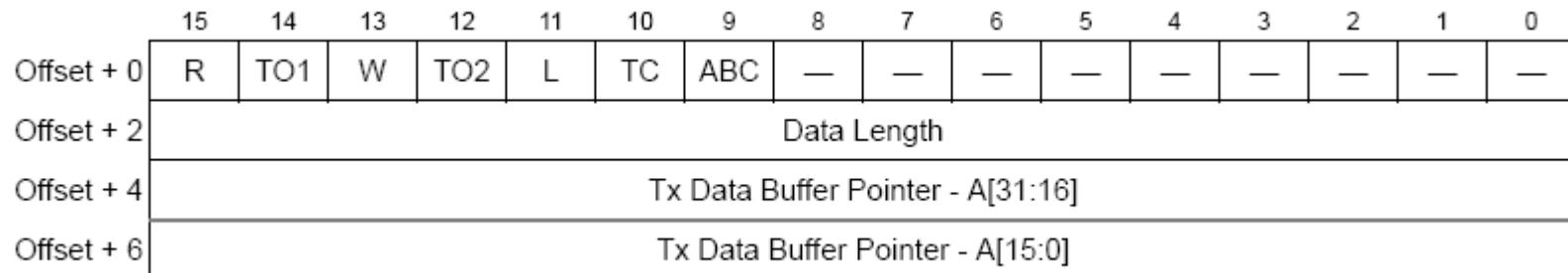
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Offset + 0	R	TO1	W	TO2	L	TC	ABC	—	—	—	—	—	—	—	—	—
Offset + 2	Data Length															
Offset + 4	Tx Data Buffer Pointer - A[31:16]															
Offset + 6	Tx Data Buffer Pointer - A[15:0]															

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

Word	Bits	Field Name	Description
Offset + 0	11	L	Last in frame. Written by user. 0 The buffer is not the last in the transmit frame. 1 The buffer is the last in the transmit frame.
Offset + 0	10	TC	Tx CRC. Written by user (only valid if L = 1). 0 End transmission immediately after the last data byte. 1 Transmit the CRC sequence after the last data byte.
Offset + 0	9	ABC	Append bad CRC. Written by user (only valid if L = 1). 0 No effect 1 Transmit the CRC sequence inverted after the last data byte (regardless of TC value).
Offset + 0	8–0	—	Reserved.

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

Transmit Buffer Descriptor (TxBD) 3



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

Word	Bits	Field Name	Description
Offset + 2	15–0	Data Length	Data Length, written by user. Data length is the number of octets the FEC should transmit from this BD's data buffer. It is never modified by the FEC. Bits [15:5] are used by the DMA engine, bits[4:0] are ignored.
Offset + 4	15–0	A[31:16]	Tx data buffer pointer, bits [31:16] ¹
Offset + 6	15–0	A[15:0]	Tx data buffer pointer, bits [15:0].

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

Software Processing Steps for Tx Buffers

1. Software writes application payload into a first Tx buffer.
2. Software writes TCP header into a second Tx buffer.
3. Software writes IP header into a third Tx buffer.
4. Software writes Ethernet header into a fourth Tx buffer.
(Note: Forming this header is actually a MAC function).
5. If the software is going to supply the Ethernet CRC, then the TC bit in all four Tx buffer descriptors must be cleared to 0. If the FEC is going to compute the Ethernet CRC, then the last buffer descriptor must have TC =1, with TC = 0 in all other buffer descriptors.
(Note: Computing the Ethernet CRC is actually a MAC function.)
6. The R bits must be set to 1. The R bit in the first buffer descriptor must be set to 1 last to avoid premature transmission.
7. The FEC hardware is informed by writing (any value) to the ***Transmit Descriptor Active Register*** (TDAR).

FEC Hardware Processing Steps for Tx Buffers

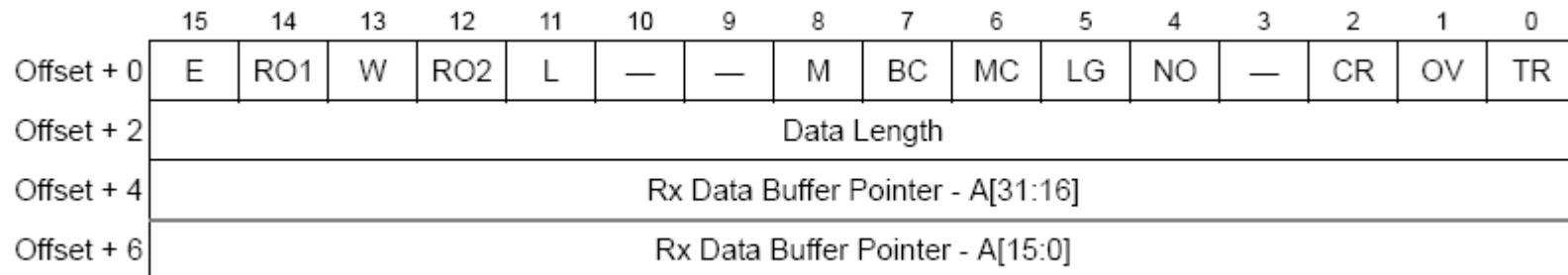
1. The FEC detects that the TDAR has been written.
2. The DMA Controller in the FEC reads the next Tx buffer descriptor in the Tx buffer descriptor ring.
3. If the R bit is 1, then the bytes are transferred from the Tx buffer into the FEC transmitter hardware by DMA.
4. After all bytes have been transferred out of a buffer, the R bit in the corresponding Tx buffer descriptor is cleared to 0. This releases the buffer to be recycled.
5. The next Tx buffer descriptor is inspected. If $R = 1$, then go back to step 3. If $R = 0$, stop further Tx buffer descriptor (and hence also buffer) processing.
6. Wait for the TDAR register to be written next.

FEC Receive Buffer Size Register (EMRBR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	R_BUF_SIZE							0	0	0	0
W																
Reset	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Address	IPSBAR + 0x1188															

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

Receive Buffer Descriptor (RxBD) 1

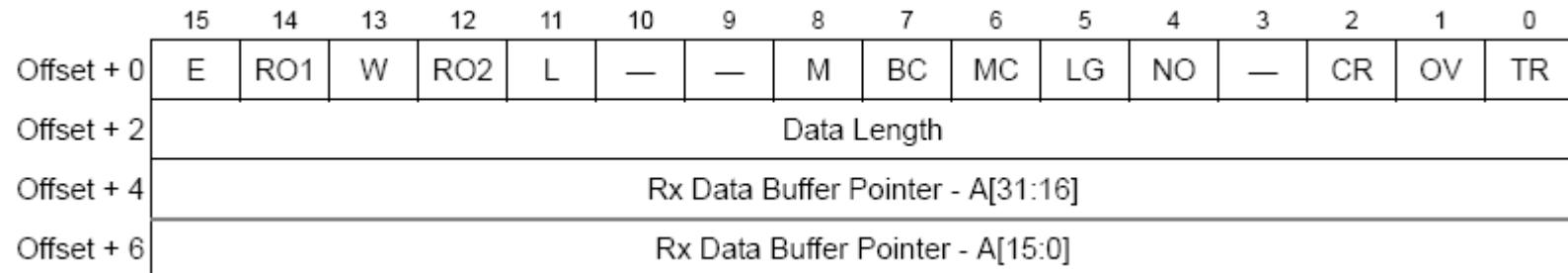


Copyright of Freescale Semiconductor, Inc. 2008. Used by permission.

Word	Bits	Field Name	Description
Offset + 0	15	E	<p>Empty. Written by the FEC (=0) and user (=1).</p> <p>0 The data buffer associated with this BD has been filled with received data, or data reception has been aborted due to an error condition. The status and length fields have been updated as required.</p> <p>1 The data buffer associated with this BD is empty, or reception is currently in progress.</p>
Offset + 0	14	RO1	Receive software ownership. This field is reserved for use by software. This read/write bit will not be modified by hardware, nor will its value affect hardware.
Offset + 0	13	W	<p>Wrap. Written by user.</p> <p>0 The next buffer descriptor is found in the consecutive location</p> <p>1 The next buffer descriptor is found at the location defined in ERDSR.</p>

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

Receive Buffer Descriptor (RxBD) 2



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

Word	Bits	Field Name	Description
Offset + 0	12	RO2	Receive software ownership. This field is reserved for use by software. This read/write bit will not be modified by hardware, nor will its value affect hardware.
Offset + 0	11	L	Last in frame. Written by the FEC. 0 The buffer is not the last in a frame. 1 The buffer is the last in a frame.
Offset + 0	10-9	—	Reserved.

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

Receive Buffer Descriptor (RxBD) 3

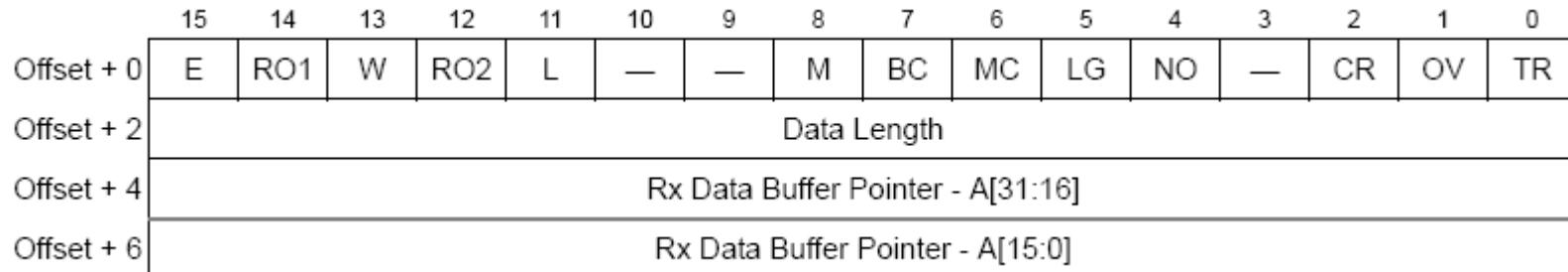
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Offset + 0	E	RO1	W	RO2	L	—	—	M	BC	MC	LG	NO	—	CR	OV	TR
Offset + 2																
Offset + 4																
Offset + 6																

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

Word	Bits	Field Name	Description
Offset + 0	8	M	Miss. Written by the FEC. This bit is set by the FEC for frames that were accepted in promiscuous mode, but were flagged as a "miss" by the internal address recognition. Thus, while in promiscuous mode, the user can use the M-bit to quickly determine whether the frame was destined to this station. This bit is valid only if the L-bit is set and the PROM bit is set. 0 The frame was received because of an address recognition hit. 1 The frame was received because of promiscuous mode.
Offset + 0	7	BC	Will be set if the DA is broadcast (FF-FF-FF-FF-FF-FF).
Offset + 0	6	MC	Will be set if the DA is multicast and not BC.
Offset + 0	5	LG	Rx frame length violation. Written by the FEC. A frame length greater than RCR[MAX_FL] was recognized. This bit is valid only if the L-bit is set. The receive data is not altered in any way unless the length exceeds 2032 bytes.

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

Receive Buffer Descriptor (RxBD) 4



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

Word	Bits	Field Name	Description
Offset + 0	4	NO	Receive non-octet aligned frame. Written by the FEC. A frame that contained a number of bits not divisible by 8 was received, and the CRC check that occurred at the preceding byte boundary generated an error. This bit is valid only if the L-bit is set. If this bit is set the CR bit will not be set.
Offset + 0	3	—	Reserved.
Offset + 0	2	CR	Receive CRC error. Written by the FEC. This frame contains a CRC error and is an integral number of octets in length. This bit is valid only if the L-bit is set.
Offset + 0	1	OV	Overrun. Written by the FEC. A receive FIFO overrun occurred during frame reception. If this bit is set, the other status bits, M, LG, NO, CR, and CL lose their normal meaning and will be zero. This bit is valid only if the L-bit is set.
Offset + 0	0	TR	Will be set if the receive frame is truncated (frame length > 2032 bytes). If the TR bit is set the frame should be discarded and the other error bits should be ignored as they may be incorrect.

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

Receive Buffer Descriptor (RxBD) 5

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Offset + 0	E	RO1	W	RO2	L	—	—	M	BC	MC	LG	NO	—	CR	OV	TR
Offset + 2 Data Length																
Offset + 4 Rx Data Buffer Pointer - A[31:16]																
Offset + 6 Rx Data Buffer Pointer - A[15:0]																

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

Word	Bits	Field Name	Description
Offset + 2	15–0	Data Length	Data length. Written by the FEC. Data length is the number of octets written by the FEC into this BD's data buffer if L = 0 (the value will be equal to EMRBR), or the length of the frame including CRC if L = 1. It is written by the FEC once as the BD is closed.
Offset + 4	15–0	A[31:16]	RX data buffer pointer, bits [31:16] ¹
Offset + 6	15–0	A[15:0]	RX data buffer pointer, bits [15:0]

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

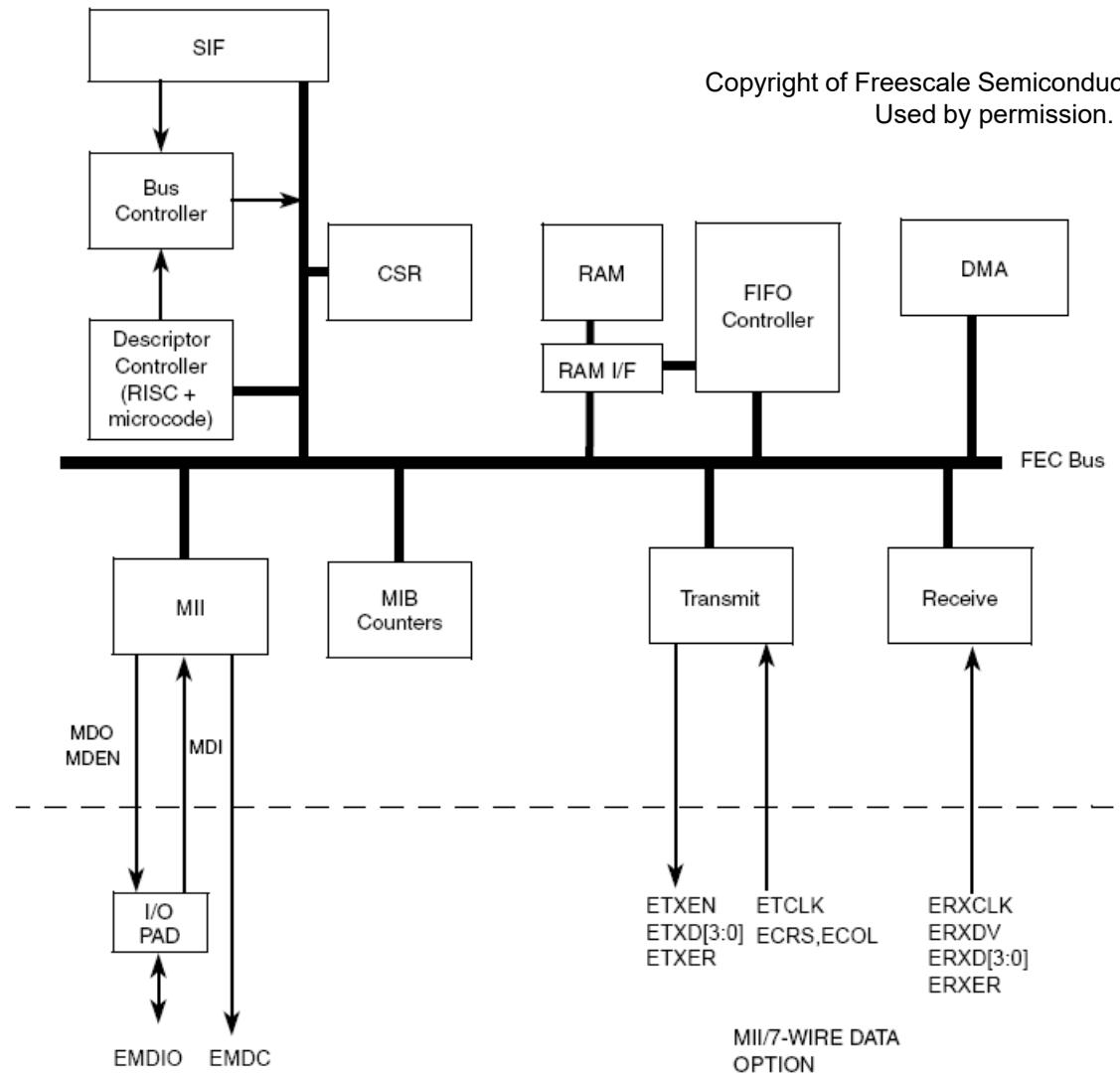
Software Processing Steps for Rx Buffers

1. Software initializes a sufficient number of Rx buffers to hold the frame that is expected to arrive from the PHY. The buffers must all have the length that was written earlier into the EMRBR.
2. The empty bit E must be set to 1 in all of the corresponding Rx buffer descriptors.
3. The FEC hardware is alerted by writing (any value) to the ***Receive Descriptor Active Register*** (RDAR).
4. The software is free to go on to other work; meanwhile, the FEC hardware will take care of transferring arriving data into the buffers.
5. The ***Receive Frame Interrupt*** (RXF) bit in the ***Ethernet Interrupt Event Register*** (EIR) is set to 1 by the FEC hardware as soon as the last buffer in a frame has been received.
6. The FEC interrupt service routine transfers the Rx buffers to Rx software. The Rx buffers are recycled later. The Rx EIMR mask bits must be cleared, or the Rx EIR status bits must be written to 1.

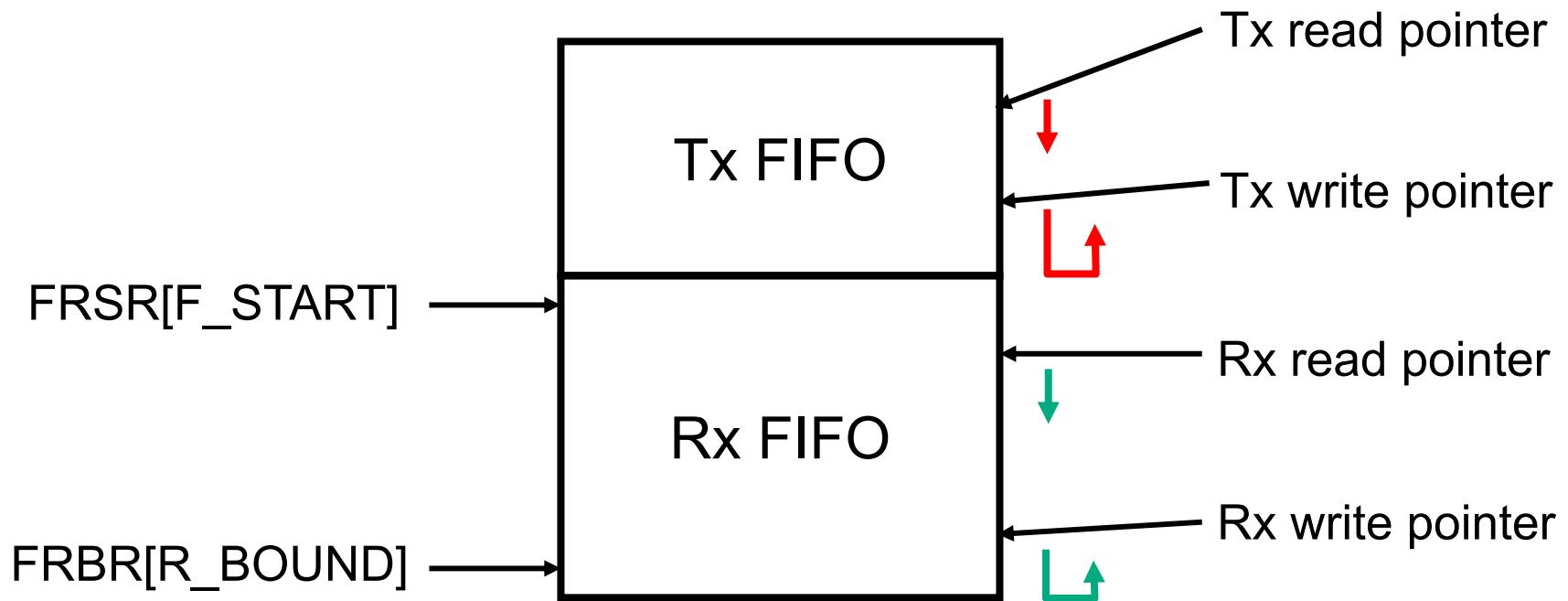
FEC Hardware Processing Steps for Rx Buffers

1. Wait until the FEC RDAR is written by CPU software.
2. The DMA Controller in the FEC reads the next Rx buffer descriptor in the Rx buffer descriptor ring.
3. If the E bit is 1, then arriving bytes are transferred by DMA from the FEC receiver hardware into the Rx buffer. If E = 0 in two read attempts, then stop Rx buffer processing and go back to step 1.
4. After an Rx buffer has been filled up (or the last data byte in the frame has been received), the E bit in the corresponding Rx buffer descriptor is cleared to 0. If the last byte in the frame was received, then go to step 6.
5. Set the ***Receive Buffer Interrupt*** (RBF) bit and clear the ***Receive Frame Interrupt*** (RXF) bit in the ***Ethernet Interrupt Event Register*** (EIR). Go back to step 2.
6. Write the total frame length in the length field of the last Rx buffer.
7. Clear the RBF bit and set the RXF bit in the EIR. Go back to step 1.

Fast Ethernet Controller (FEC) Architecture



First-In First-Out Buffer



Note: The read and write pointers are incremented (with wrap-around) automatically by the FEC ***Direct Memory Access*** (DMA) controller.

FIFO Receive Start Register (FRSR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	0	R_FSTART								0	0
W																
Reset	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x1150															

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

Bits	Name	Descriptions
31–10	—	Reserved, read as 0 (except bit 10, which is read as 1).
9–2	R_FSTART	Address of first receive FIFO location. Acts as delimiter between receive and transmit FIFOs.
1–0	—	Reserved, read as 0.

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

FIFO Receive Bound Register (FRBR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	0	R_BOUND								0	0
W																
Reset	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x114C															

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

Bits	Name	Descriptions
31–10	—	Reserved, read as 0 (except bit 10, which is read as 1).
9–2	R_BOUND	Read-only. Highest valid FIFO RAM address.
1–0	—	Reserved, should be cleared.

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

Transmit FIFO Watermark Register (TFWR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	TFWR	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x1144															

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

Bits	Name	Descriptions
31–2	—	Reserved, should be cleared.
1–0	TFWR	Number of bytes written to transmit FIFO before transmission of a frame begins 0x 64 bytes written 10 128 bytes written 11 192 bytes written

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

FEC MAC Initialization (before ECR[ETHER_EN])

Description
Initialize EIMR
Clear EIR (write 0xFFFF_FFFF)
TFWR (optional)
IALR / IAUR
GAUR / GALR
PALR / PAUR (only needed for full duplex flow control)
OPD (only needed for full duplex flow control)
RCR
TCR
MSCR (optional)
Clear MIB_RAM (locations IPSBAR + 0x1200-0x12FC)

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

FEC DMA Initialization (before ECR[ETHER_EN])

Description
Initialize FRSR (optional)
Initialize EMRBR
Initialize ERDSR
Initialize ETDSR
Initialize (Empty) Transmit Descriptor ring
Initialize (Empty) Receive Descriptor ring

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

FEC Initialization (after ECR[ETHER_EN])

Description
Initialize BackOff Random Number Seed
Activate Receiver
Activate Transmitter
Clear Transmit FIFO
Clear Receive FIFO
Initialize Transmit Ring Pointer
Initialize Receive Ring Pointer
Initialize FIFO Count Registers

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

Data Communication Interfaces and TCP/IP

References:

- Behrouz A. Forouzan, “TCP/IP Protocol Suite”, 2nd ed., McGraw-Hill, 2003, ISBN 0-07-246060-1.
- Numerous white papers that are available on the World Wide Web.

Data Communication Protocols

- Data communication takes place according to strictly defined ***constraints*** and ***rules***, such as:
 - the services that can be requested and provided
 - the legal formats for messages that are exchanged
 - the rules for starting and terminating a session
 - the rules for exchanging control and status information
 - the rules for exchanging data messages
 - the rules for dealing with error conditions
 - etc.
- These constraints and rules are together called a ***protocol***.
- In a ***layered communication interface***, the interactions at each layer are governed by separate protocols.

Five Elements of a Communication Protocol

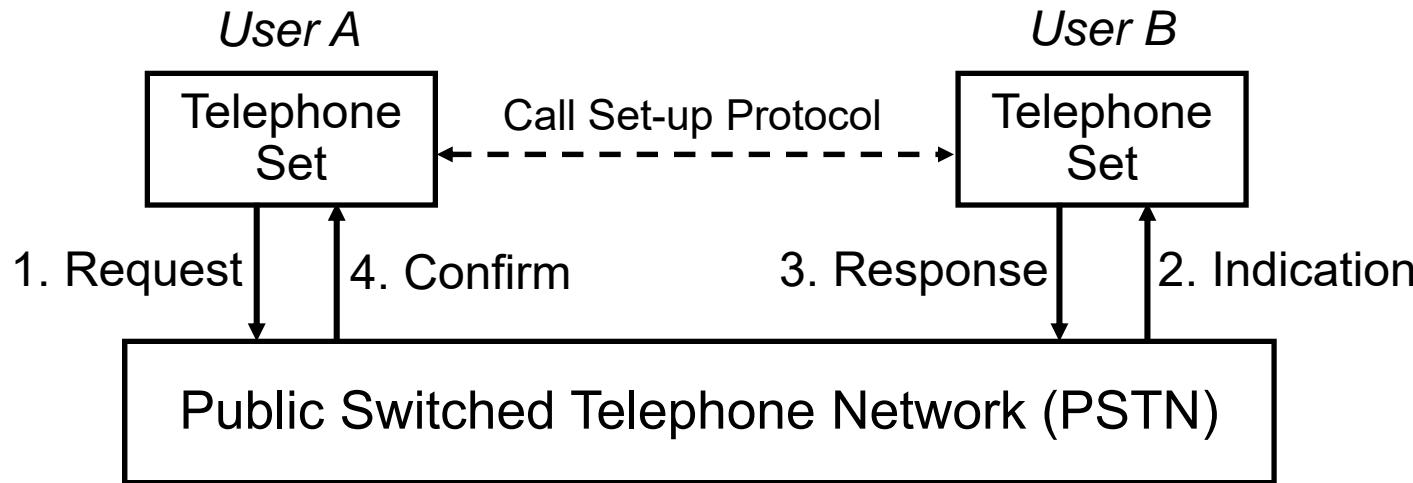
- “1. The **service** to be provided by the protocol.
- 2. The **assumptions** about the environment in which the protocol is executed.
- 3. The **vocabulary** of messages used to implement the protocol.
- 4. The **encoding** (i.e., the format) of each message in the vocabulary.
- 5. The **procedure rules** guarding the consistency of message exchanges.”

Source: G. R. Holzmann, “Design and Validation of Computer Protocols”, Prentice Hall, 1991, ISBN 0-13-539925-4.

Ex: Session and Application Layer Protocols

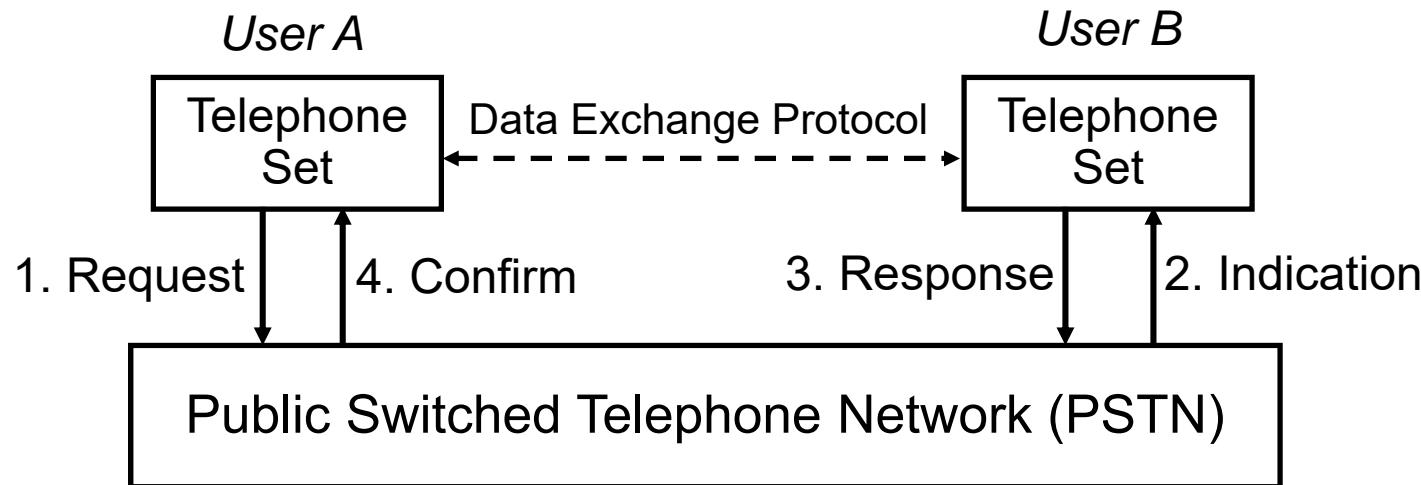
- A **session layer** protocol is responsible for:
 - establishing a new connection
 - choosing full or half-duplex communication mode
 - synchronizing the communication at a high level
 - possibly recovering from lost connections
 - terminating an existing connection
- An **application layer** protocol governs how an application (computer or human) uses an existing reliable communication connection.
- Protocol interactions between two peers at the same layer on different nodes can be viewed as having four steps: (1) **request**, (2) **indication**, (3) **response**, and (4) **confirm**.
- Many of the basic protocol concepts can be illustrated using a human-to-human telephone call using old-style telephones with handsets.

Setting Up a Telephone Call (Session Layer)



- Request:** Local telephone taken off hook. Hear dial tone. Enter telephone number. Hear locally-generated ringing sound signal.
- Indication:** Remote telephone receives the electrical ringing signal. Remote telephone emits audible ringing sound.
- Response:** Remote telephone taken off hook. Ringing stops.
- Confirm:** Local telephone no longer receives a ringing sound signal. Instead, hears silence then remote party saying “Hello!”

Telephone Conversation (Application Layer)



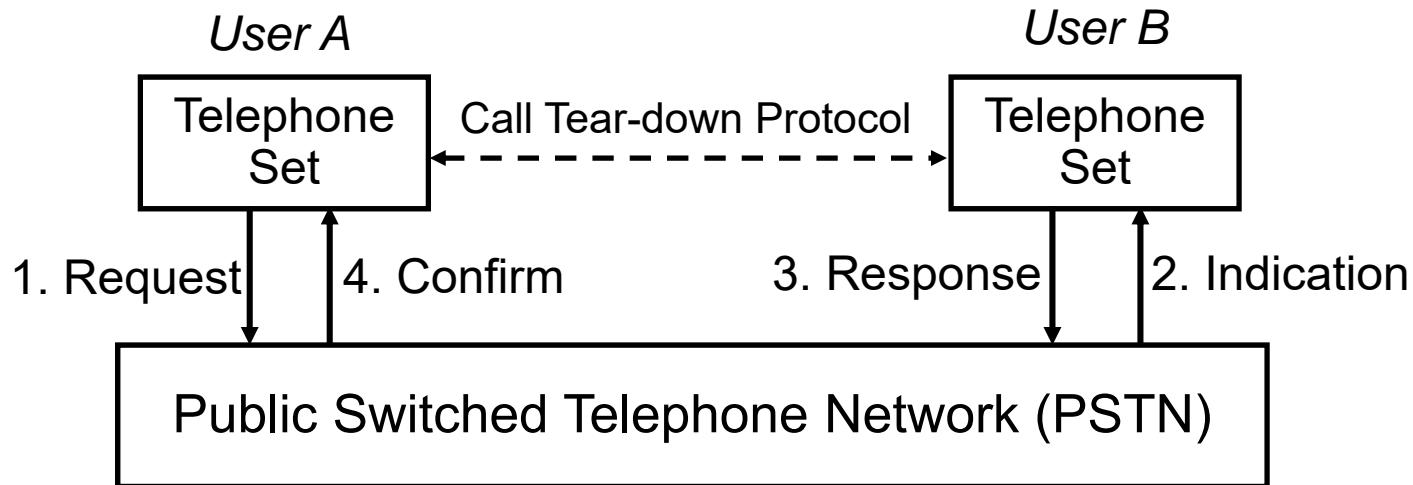
Request: User A speaks a sentence into their telephone set.

Indication: User B hears user A's sentence being spoken.

Response: B's response might be "Yes". Or B's response to A's sentence might be implied by the content of sentences spoken next by user B to user A.

Confirm: User A hears user B's "Yes" or a new sentence from B.

Ending a Telephone Call (Session Layer)



Request: User A says “Good bye B!”.

Indication: User B hears User A say “Good bye B!”.

Response: User B says “Good bye A!”.

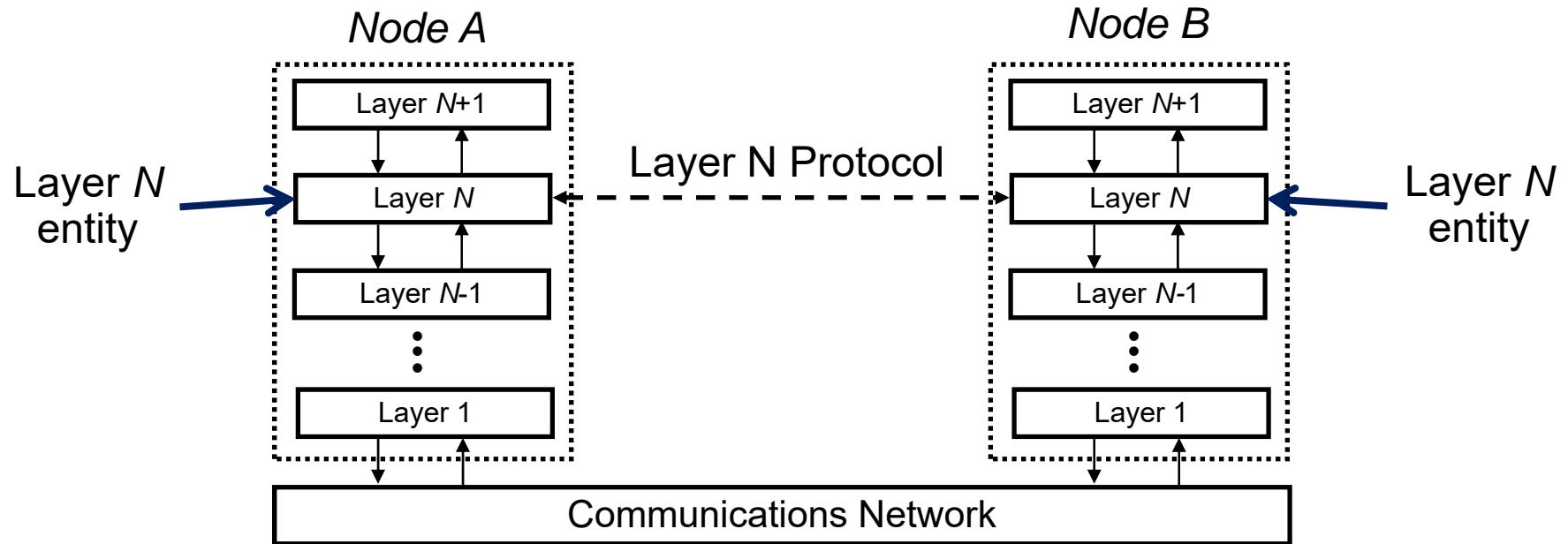
Confirm: User A hears User B say “Good bye A!”.

Request: User A and B both hang up. The telephone network takes down the connection that carried the call.
If one user does not hang up, they hear silence at first.
Eventually a loud off-hook alarm signal will be heard.

Layered Data Communication Interfaces

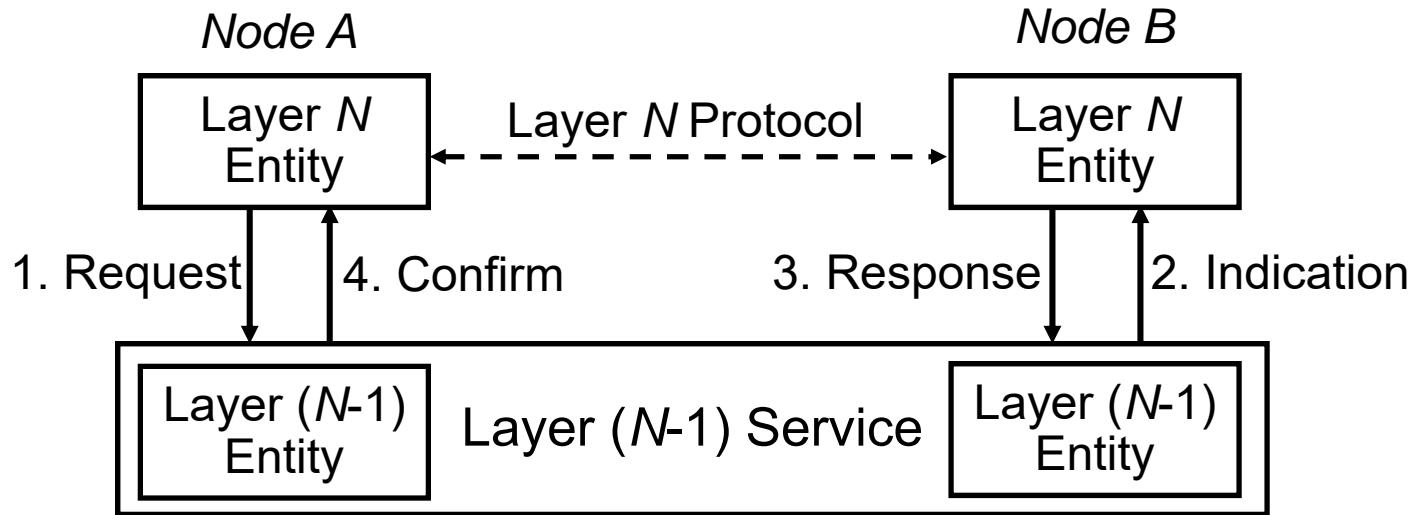
- Layering is used to manage the complexity of data communication interfaces (as well as computer systems).
- Layering provides many advantages in communications:
 - ***Divide-and-conquer.*** Each layer by itself is simpler to specify, design, and verify.
 - One layer can be modified or reworked more easily with minimal impact on the other layers.
 - Application software and the operating system can interact with the communications interface at one or more possible layers (although it is usually best to access the interface at the highest possible layer).

Layered Protocol Stacks



- Layer N provides “services” to layer $N+1$. Typical services include: (1) **set up** a connection; (2) **transfer** data; (3) **take down** a connection.
- Layer N uses the services of layer $N-1$.
- The **layer N entity** in Node A interacts indirectly with its **peer** (i.e., the layer N entity in Node B) according to a precisely defined set of rules called the **layer N protocol**.

Layer N Protocol Primitives



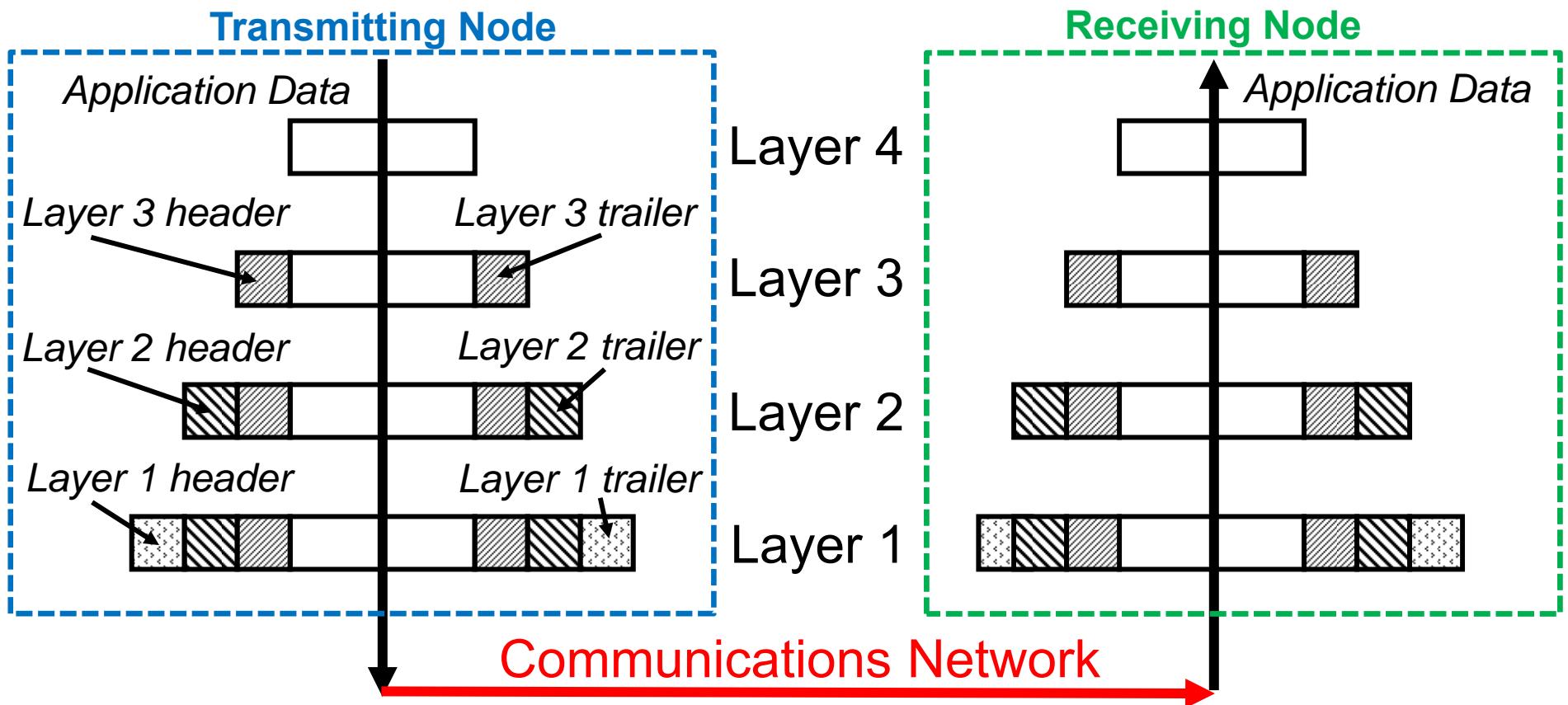
Request: Node A layer-N entity requests service from Node A layer-(N-1) entity, which provides layer-(N-1) services.

Indication: Node B layer-(N-1) entity signals to Node B layer-N entity that its layer-N peer in sys. A is requesting a service.

Response: Node B layer-N entity signals to Node B layer-(N-1) entity that it has provided the layer-N service for its peer in A.

Confirm: Node A layer-(N-1) entity signals to Node A layer-N entity that the requested layer-N service has been done.

Protocol Overhead in Data Packets



- A header and/or trailer is added to the data packet at each layer *going down the protocol stack* in the *transmitting node*.
- The headers and/or trailers are *processed and removed* when *going up the protocol stack* in the *receiving node*.

The ISO OSI-RM 7-layer Protocol Stack (1980)

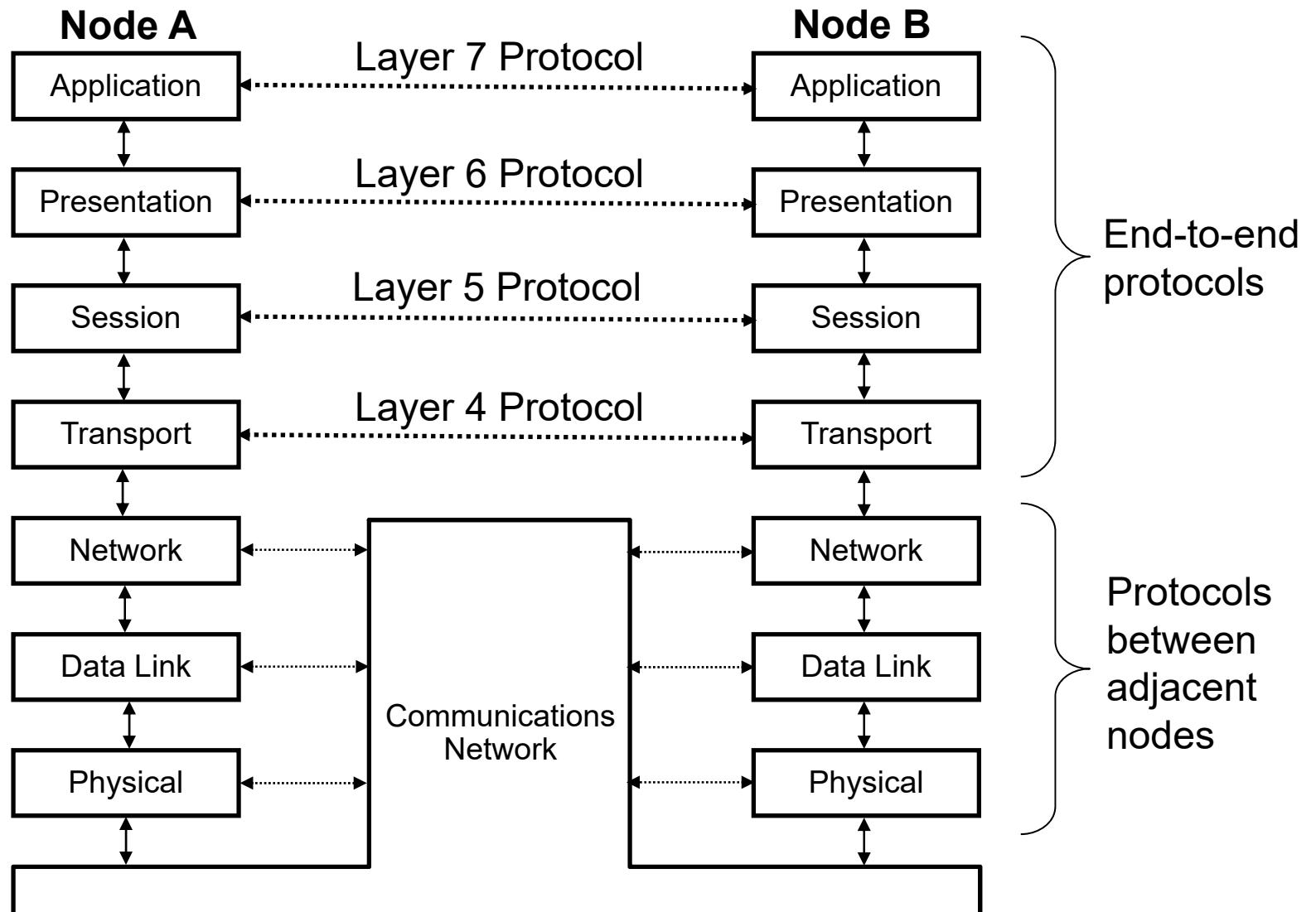
ISO = International Standards Organization

OSI = Open Systems Interconnection

RM = Reference Model

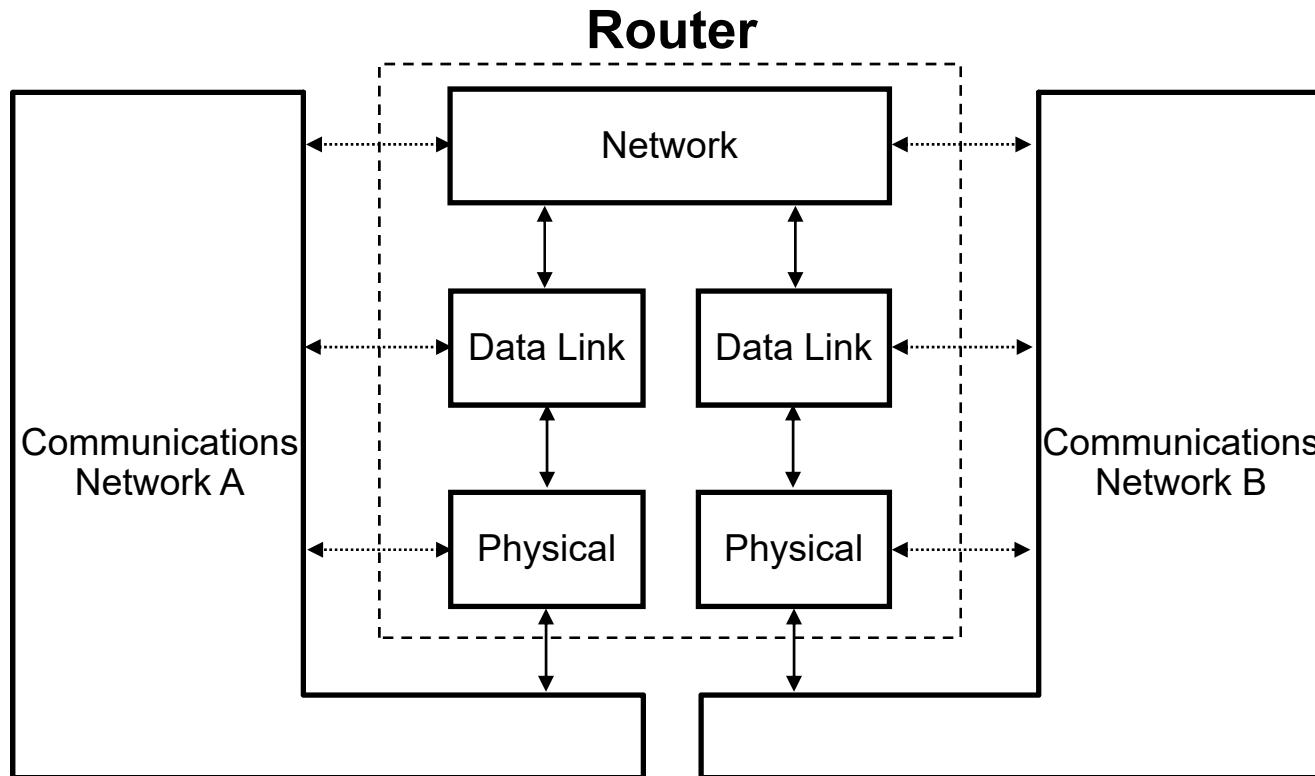
Layer	Name	Typical Services
7.	<i>Application</i>	Generation, processing & consumption of information.
6.	<i>Presentation</i>	Data format conversion, compression, encryption.
5.	<i>Session</i>	Opening and closing of communication channels.
4.	<i>Transport</i>	Control of end-to-end flow of data byte streams.
3.	<i>Network</i>	Addressing and routing of packets through network.
2.	<i>Data Link</i>	Tx. of data frames from one node to the next node.
1.	<i>Physical</i>	Transmission of bits over the communication medium.

OSI Protocol Interfaces



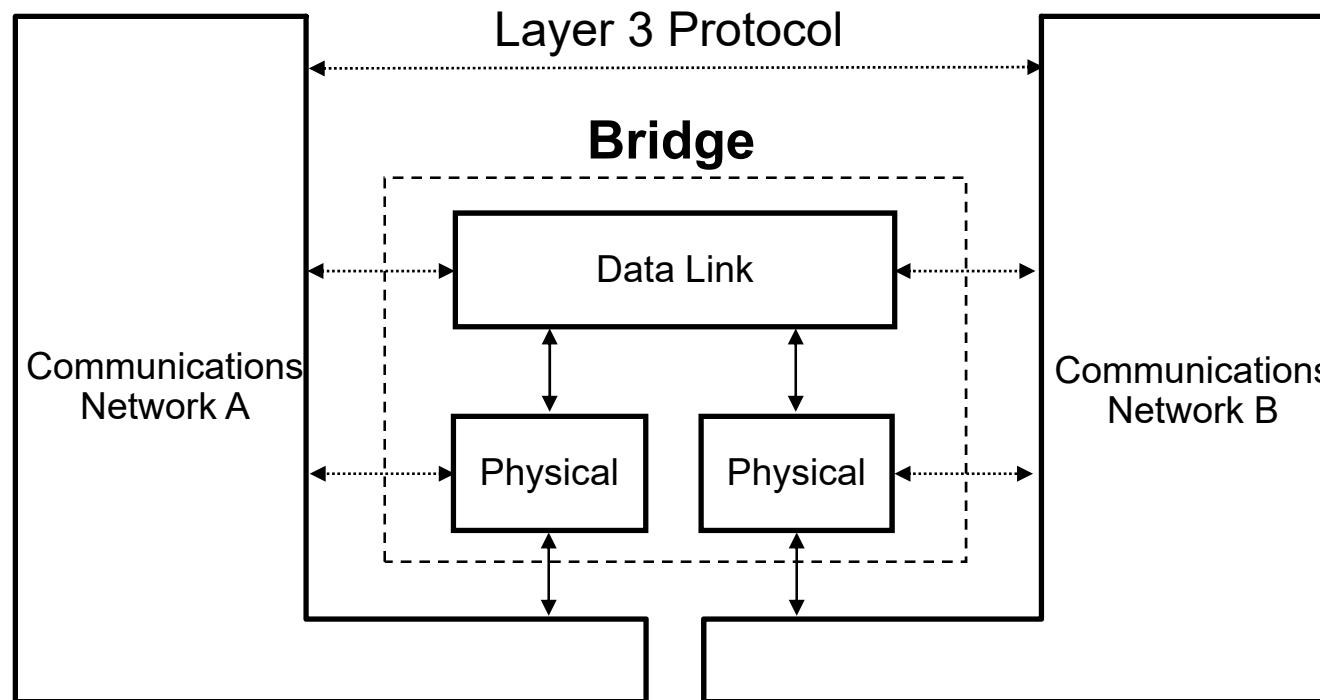
Routers in the OSI Model

- A router is used to interface two networks that operate using different layer 3 protocols and/or that use different layer 3 packet formats.
e.g. FDDI campus backbone to/from the ETLC building
- Typical router functions include: (1) forwarding the packet to the next node on its route to the destination; (2) filtering data packets according to destination address; and (3) any required data packet reformatting.



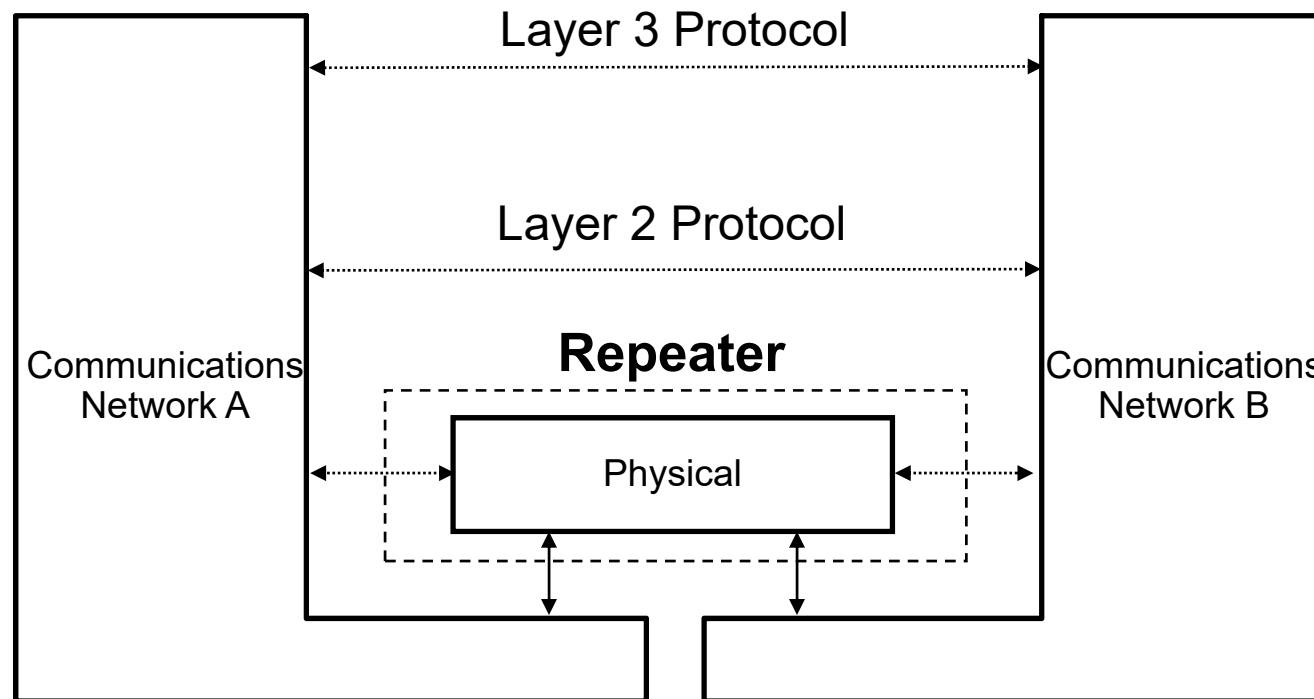
Bridges in the OSI Model

- A *bridge* is usually used to interface two networks that operate using the same layer 3 protocol.
 - e.g. ETLC 2nd floor Ethernet to/from 5th floor Ethernet
- A bridge *filters data packets* according to the destination address. The filtering action is useful for preventing traffic in network A from being sent unnecessarily to network B, and vice versa.

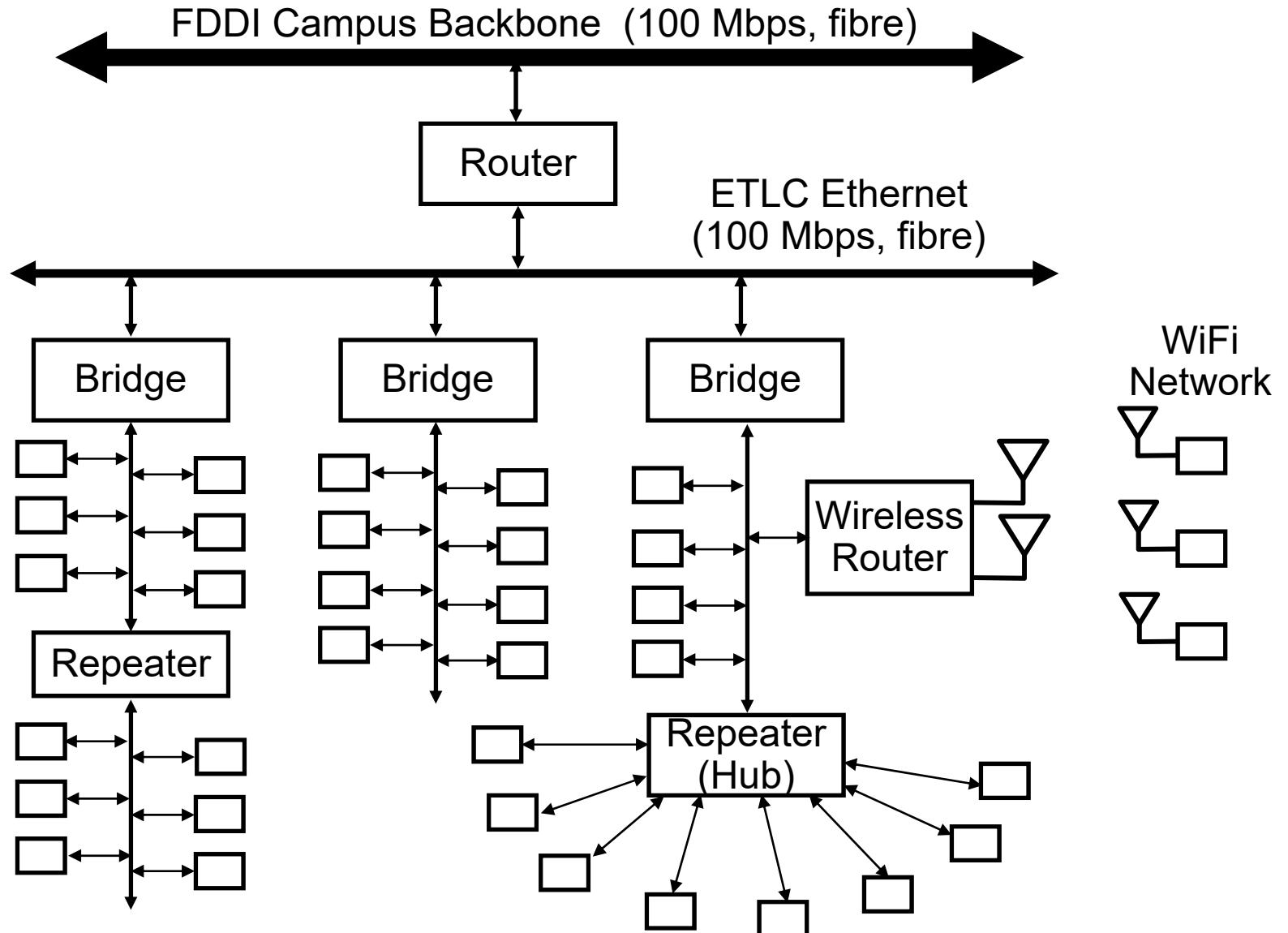


Repeaters & Hubs in the OSI Model

- A *repeater* is used to interface two networks that operate using the same layer 2 and layer 3 protocols.
- A repeater is a “dumb” device that *transfers bits back and forth* between the two networks. A repeater filters out noise and amplifies the signal.
- A *hub* is a repeater that creates a star topology (shown on next slide).



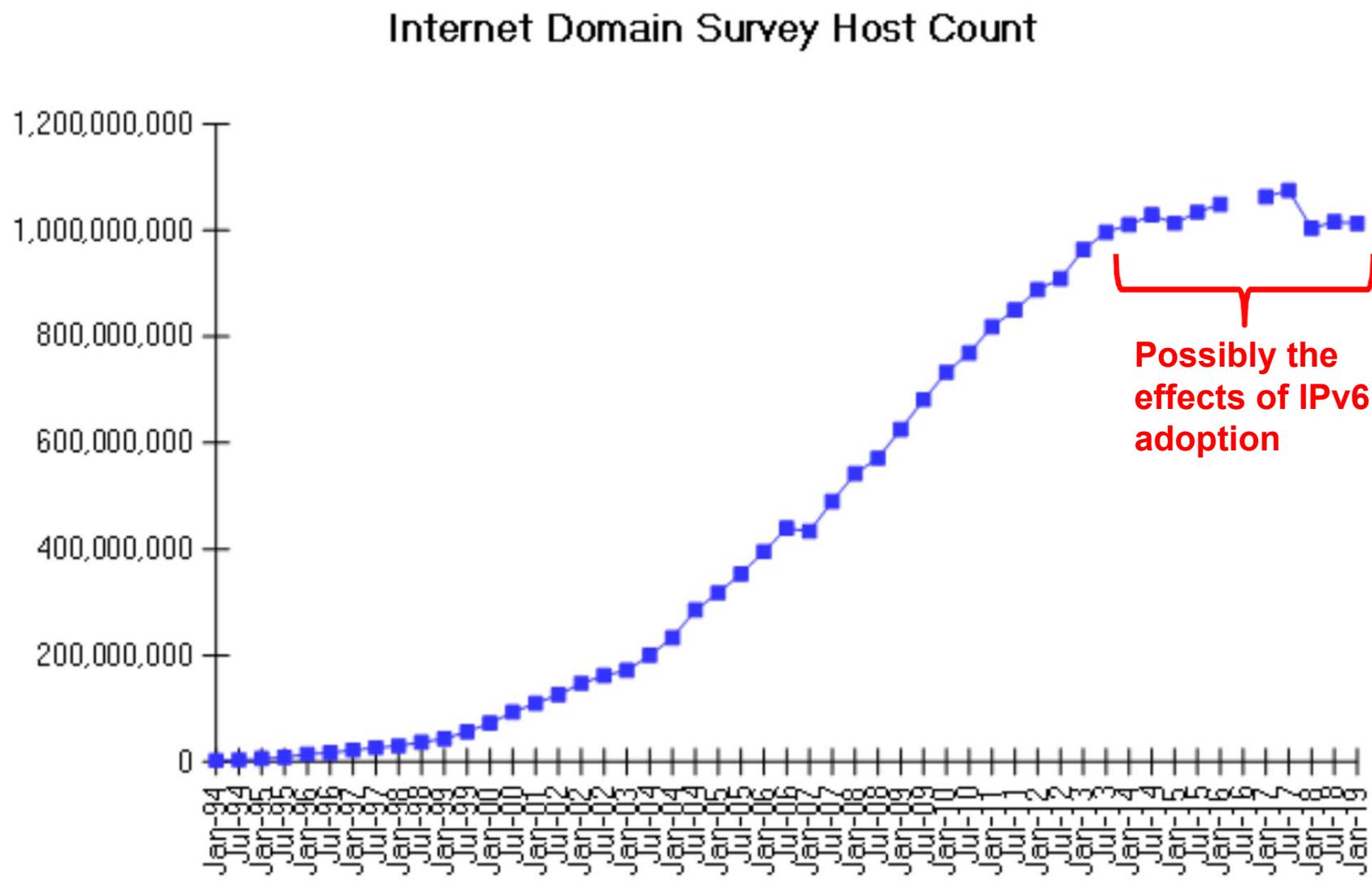
Data Communication Interfaces and TCP/IP



History of Large-Scale Data Networks

- 1963: ReserVec airline reservation system (Ferranti Canada) enters service with Trans-Canada Air Lines (now called Air Canada).
- 1964: SABRE airline reservation system in service with American Airlines.
- 1962: RAND Corporation study recommends deployment of a packet switched data network capable of surviving a nuclear attack.
- 1969: U.S. Dept. of Defense Advanced Research Projects Agency (ARPA) sponsors the construction of a large, general-purpose, packet switching data network. ARPANET starts with 4 nodes: UCLA, SRI (Stanford), UC Santa Barbara, & the University of Utah.
- 1972: First electronic mail program created by Ray Tomlinson.
- 1974: Vincent Cerf and Bob Kahn coin the term Internet in a paper on TCP.
- 1985: The ARPANET connects roughly 1200 nodes.
- 1987: The civilian Internet connects roughly 25,000 nodes.
- 1990: Early version of World-Wide Web developed at CERN in Geneva.
- 1993: Marc Andreessen develops Mosaic, the first WWW browser.
- 2017: The Internet connects over 1,100,000,000 nodes as of July 2017.

Internet Domain Survey IPv4 Host Count (Jan 2019)



Source: Internet Systems Consortium (www.isc.org)

Copyright © 2020 by Bruce Cockburn

8-19

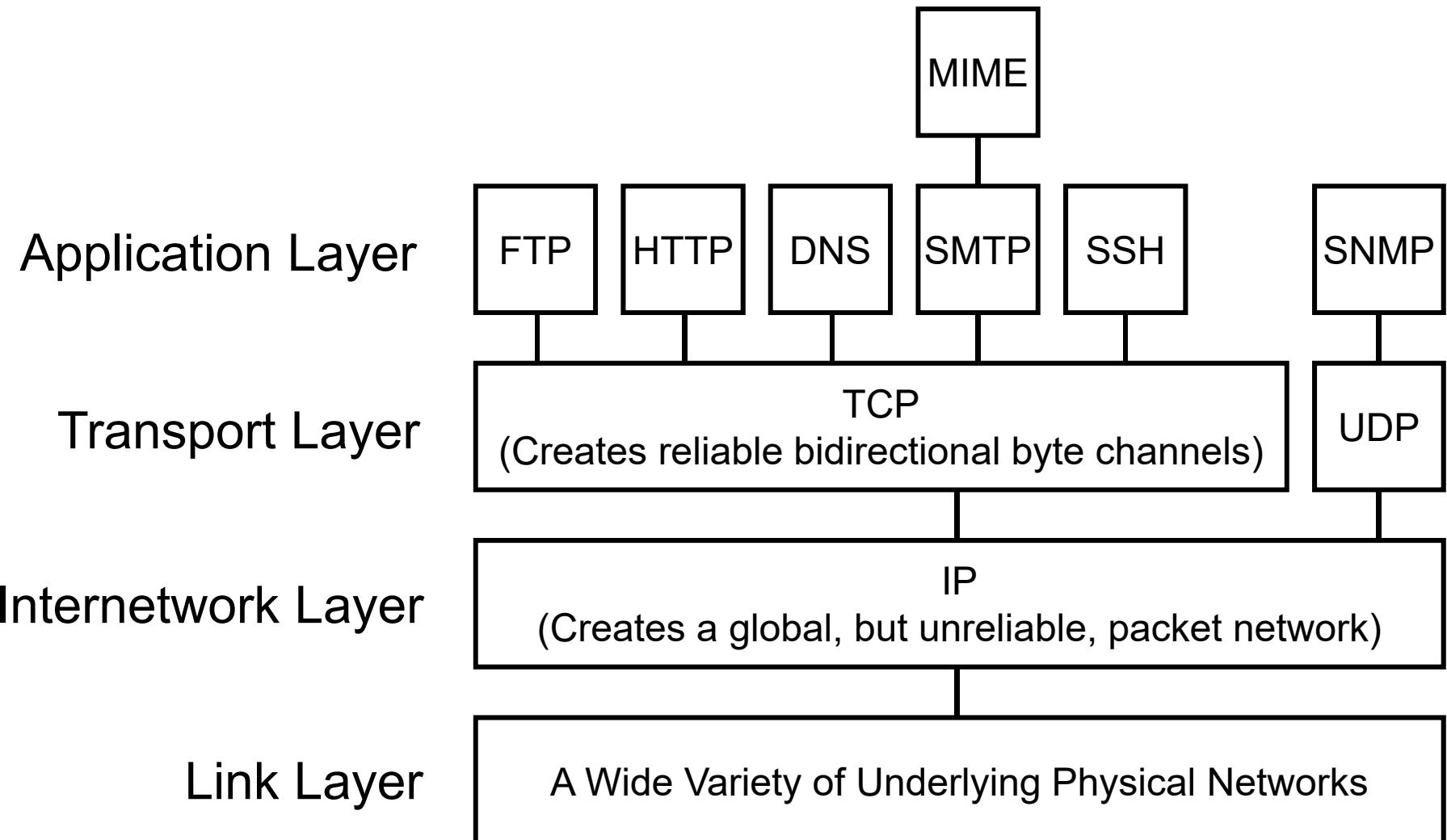
Evolution of TCP/IP

- ***Network Control Protocol*** (NCP): The protocol used in the ARPANET from 1970 until the early 1980s.
- ***Transmission Control Protocol*** (TCP): A successor to NCP first described in 1973 by Vincent Cerf and Bob Kahn.
- TCP/IP: In the late 1970s, TCP was split up into two distinct protocols: A new Transmission Control Protocol (TCP) and the ***Internet Protocol*** (IP).
- In 1981, Univ. Calif. Berkeley modified the UNIX operating system to include an implementation of TCP/IP.
- In 1983, TCP/IP became the exclusive protocol on the ARPANET.

The Structure of TCP/IP

- TCP/IP evolved before the 7-layer ISO Reference Model was available, so the layers in TCP/IP do not correspond exactly to the seven standard layers.
- To *maximize flexibility and portability*, the physical and data link layers are left to the underlying network.
- IP corresponds closely to the ISO network layer. It provides a ***connectionless datagram service*** between two addressed nodes, with no guarantees on delivery or on the specific route that was taken through the network.
- TCP corresponds to the ISO transport layer. It builds on the services of IP to provide a ***reliable, connection-oriented two-way stream of bytes***. The byte order is preserved.
- The functions of the ISO presentation and session layers are provided by the application protocol and TCP.

TCP/IP and Some Related Protocols



TCP/IP-Related Protocols and Applications

FTP = File Transfer Protocol

HTTP = Hypertext Transfer Protocol

DNS = Domain Name Service

SMTP = Simple Mail Transfer Protocol

MIME = Multi-purpose Internet Mail Extension

SSH = Secure Shell

UDP = User Datagram Protocol

SNMP = Simple Network Management Protocol

The details of these, and the many other TCP/IP-related protocols and applications, are beyond the scope of this course. For much more detail see “TCP/IP Protocol Suite” by Behrouz Forouzan (McGraw Hill, 2003).

Services Provided by IP

- IP provides two services to the overlying layers:
 1. **Send** a given data packet to a node with a given address.
 2. **Receive** a data packet that was sent to the present node.
The data packet plus IP header is called a “datagram”.
- IP provides an *unreliable, connectionless* delivery service:
 - No warning is given of failure to deliver a datagram.
 - No guarantee that a series of datagrams sent to the same destination will take the same route. The order of such a sequence of datagrams might not be preserved.
 - IP can break up and send a packet as multiple datagrams.
- IP prepends its own header to each datagram to provide information that is required for IP datagram processing at the intervening (tandem) nodes and at the destination node.

IP Version 4 Datagram Format (IPv4)

31	27	23	15	11	0							
Version	IHL	Service Type	Total Length									
Identification Field		Flags		Fragment Offset								
Hop Limit	Protocol		Header Checksum									
Source IP Address												
Destination IP Address												
Options and Padding												
Data Field (a multiple of bytes)												
<i>Note:</i> maximum total datagram length is 65,535 bytes.												

IHL = Internet Header Length (measured in 32-bit words).
Padding ensures that the header is a multiple of words long.

IP Version 4 Address Formats (IPv4)

Class A: Few networks, each with many hosts

0	Network	Host (24 bits)
---	---------	----------------

Class B: Medium number of hosts and networks

1 0	Network (14 bits)	Host (16 bits)
-----	-------------------	----------------

Class C: Many networks, each with few hosts

1 1 0	Network (21 bits)	Host (8 bits)
-------	-------------------	---------------

Class D: Intended for multicasting to all hosts in defined groups

1 1 1 0	Host Group Identifier (28 bits)
---------	---------------------------------

Class E: Reserved addresses

1 1 1 1	Address (28 bits)
---------	-------------------

More on IPv4 Addresses

- In order to participate in send and receive IP datagrams with a TCP/IP-based Internet, a node needs to have:
 1. A unique IP address e.g. $0x8D0E4818 = 141.14.72.24$
 2. A subnet mask e.g. $0xFFFFC000 = 255.255.192.0$
 3. The IP address of a “gateway” router
 4. The IP address of a “name server”
- A **static IP address** is associated “permanently” with the hardware of the node (e.g., with the 6-byte *Ethernet physical address* of the node).
- A **dynamic IP address** is assigned to a node from a pool of temporary addresses for a limited period of time, usually referred to as a *lease*. When the lease expires, the node must request a dynamic IP address once again. In this way, dynamic IP addresses are returned & reused.
- The **subnet mask** is a 32-bit mask which, when bitwise ANDed with an IP address, yields the **base address** for the block of IP addresses corresponding to the local subnetwork.

e.g. $141.14.72.24 \text{ AND } 255.255.192.0 \Rightarrow 141.14.64.0$
 $0x8D0E4818 \text{ AND } 0xFFFFC000 \Rightarrow 0x8D0E4000$

IP Version 6

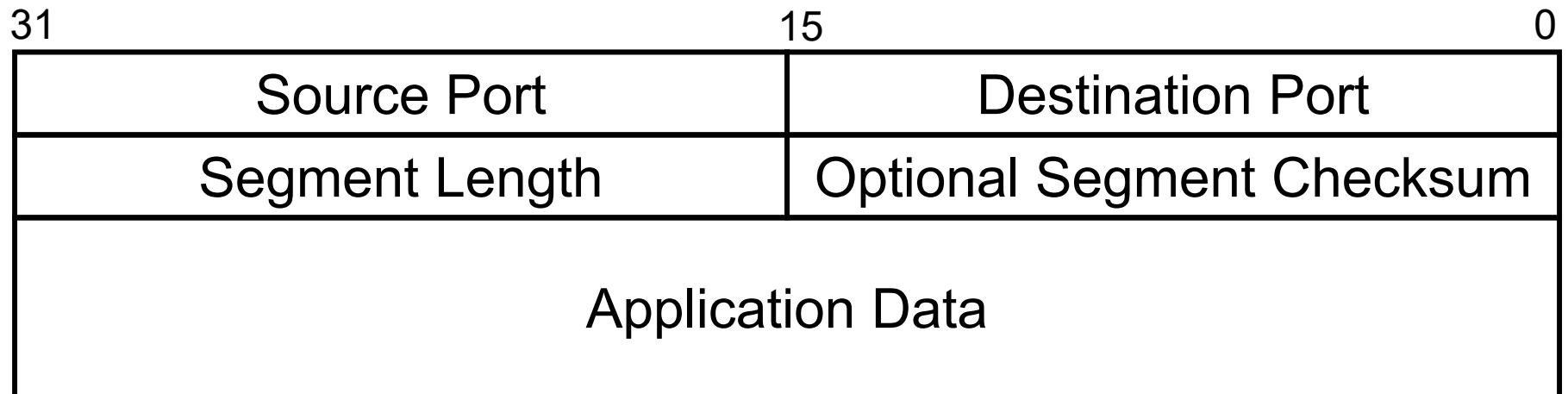
- The rapid growth of the Internet, and the rather rigid way that 32-bit IPv4 addresses have been allocated, has caused some high-growth parts of the world (e.g., Asia) to run out of addresses. IPv4 needs to be replaced with a better protocol.
- IP Version 6 (IPv6), first defined in 1996, uses **128-bit addresses**. IPv6 is gradually replacing IPv4 (still common).
- IPv6 has new options and it has been designed to be more easily expanded with new functionalities.
- IPv6 has a new mechanism (the **flow label**) for handling special types of traffic, like real-time audio and video.
- IPv6 has support for **encryption** and **authentication** options to provide confidentiality and better packet integrity.

Behrouz A. Forouzan, “TCP/IP Protocol Suite”, 2nd ed., McGraw-Hill, 2003.

Transport Layer Protocols: TCP and UDP

- **User Datagram Protocol (UDP)** provides a ***connectionless*** transport service on top of IP:
 - No guarantee of delivery or of delivery order of the data packets, which in UDP and TCP are called “segments”.
 - IP address is augmented with source and destination port numbers (16 bits each).
 - UDP provides a simpler “lightweight” alternative to TCP which is suitable for “streaming multimedia” services.
- **Transmission Control Protocol (TCP)** provides a reliable, ***connection-oriented*** transport service on top of IP:
 - Like UDP, port numbers are provided.
 - A connection must be set up to carry a communication.
 - A feature-rich set of service primitives is provided to the overlying application layer.

UDP Segment Format

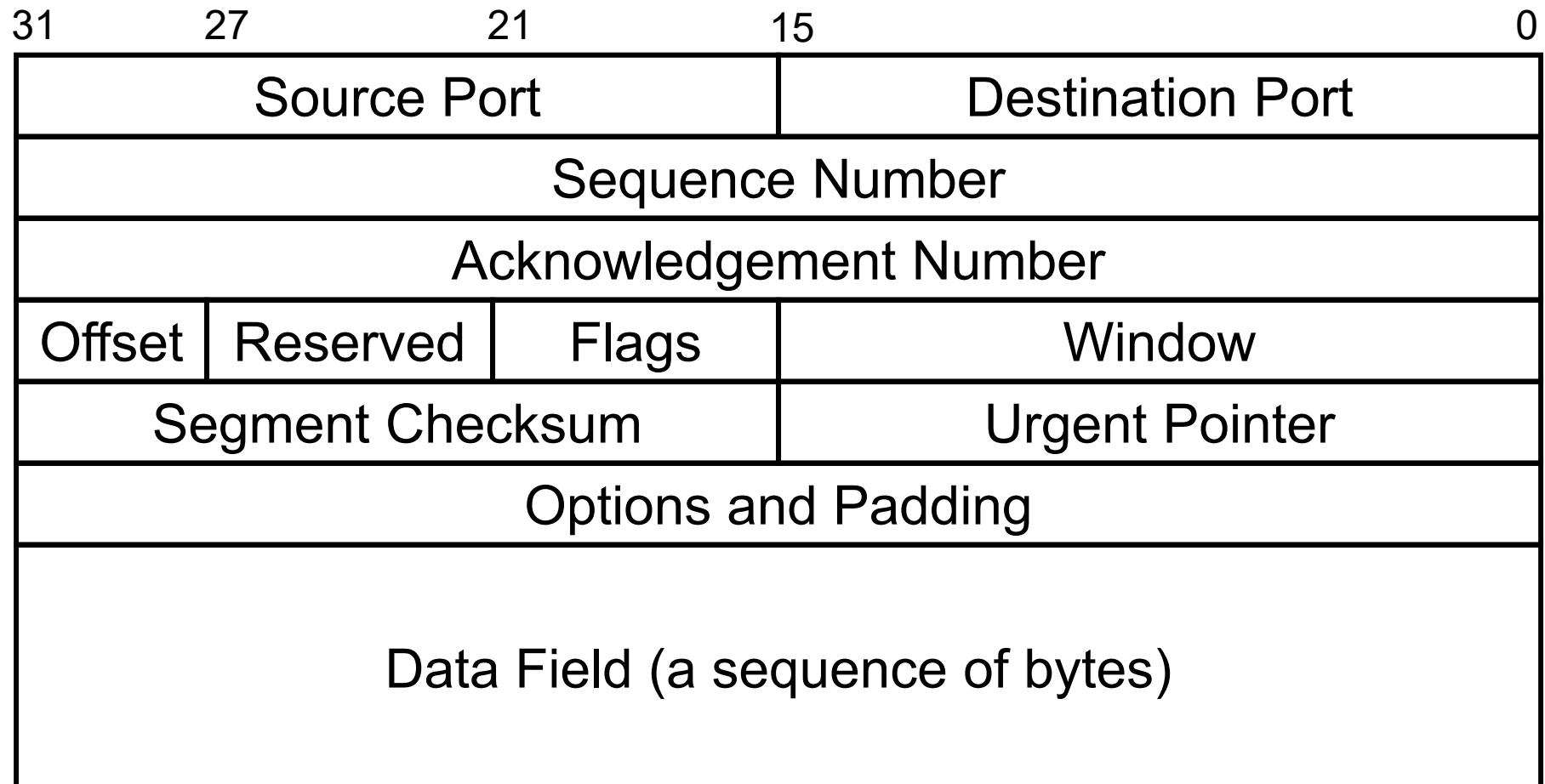


- The ***port numbers*** can be used to identify applications or high-level services on the source and destination nodes.
- If the ***checksum*** is included, then the checksum field is nonzero. If a recomputed checksum does not match the checksum that was sent in the segment, then the entire segment is discarded and no further action is taken.

Some Standard Port Numbers in UDP & TCP

Port	Prot.	Service	Port	Prot.	Service
7	TCP	Echo	70	TCP	Gopher
13	both	Daytime	79	TCP	Finger
19	both	Character gen.	80	TCP	HTTP
20	TCP	FTP data	109	TCP	POP-2
22	TCP	Secure shell	110	TCP	POP-3
23	TCP	Telnet	111	both	RPC
25	TCP	SMTP	161	UDP	SNMP
37	both	Time	162	UDP	SNMP-TRAP
67	UDP	BOOTP-server	179	TCP	BGP
68	UDP	BOOTP-client	520	UDP	RIP
69	UDP	TFTP	>2000		user-defined

TCP Segment Format



TCP Header Fields

- The ***sequence number*** is a number that corresponds to the first data byte. All data bytes have sequence numbers.
- The ***acknowledgement number*** identifies the sequence number of the next data byte that is expected to be received. Contains valid number only used if the ACK bit is set.
- The ***data offset*** gives the number of 32-bit words in the header.
- Six flags are used for various purposes:
 - URG:** Urgent pointer field enabled (MSB)
 - ACK:** Acknowledgement field enabled
 - PSH:** Push function. Force transmission of segment.
 - RST:** Reset the connection.
 - SYN:** Synchronize the sequence numbers.
 - FIN:** No more data will be sent from sender. (LSB)

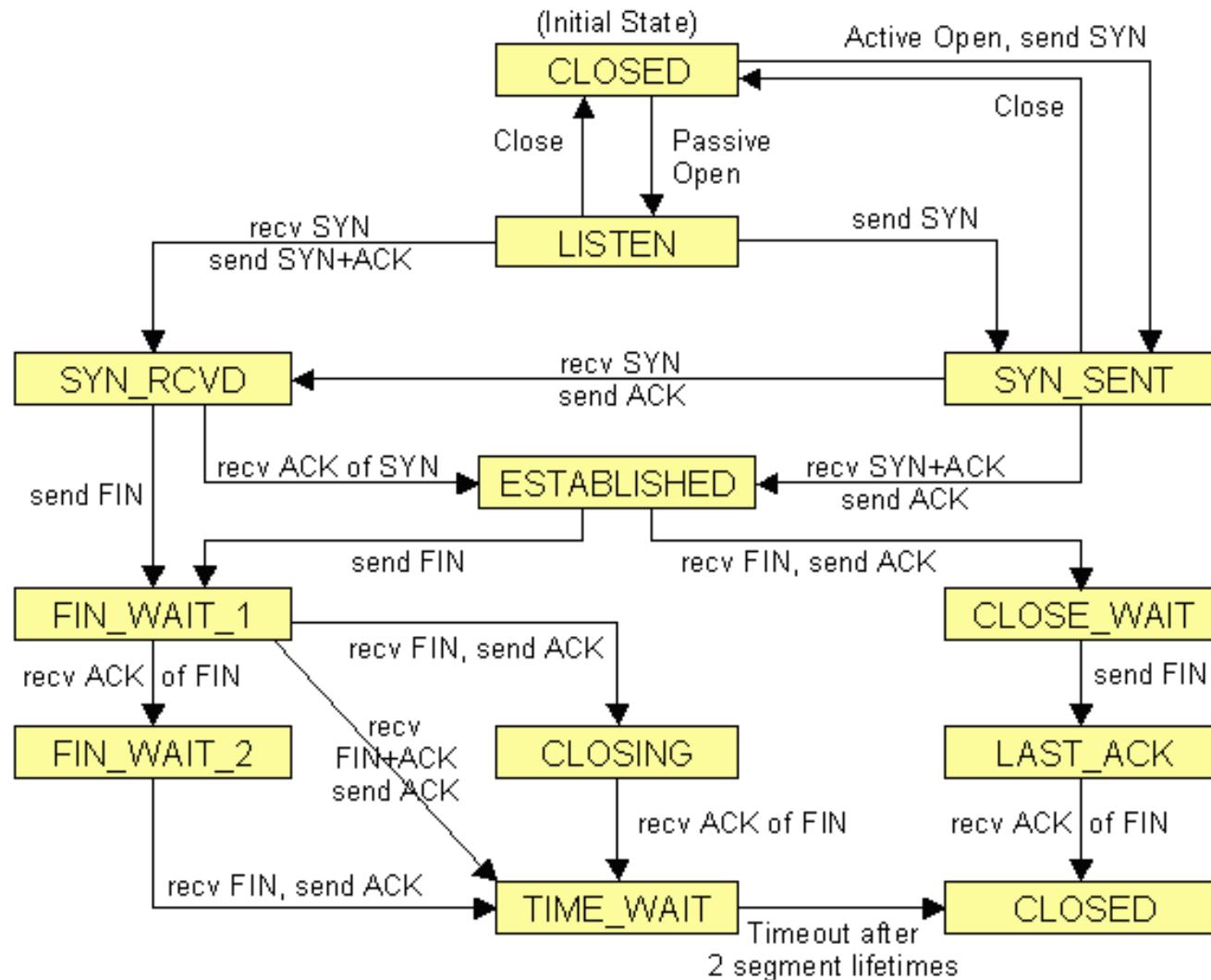
TCP Header Fields (cont'd)

- The **window** indicates the maximum number of data bytes that the other TCP entity must be prepared to accept without an acknowledgement. The buffer size in the receiver must be taken into account when negotiating the window.
- The **checksum** is computed over the TCP segment and allows the receiving TCP entity to detect transmission errors.
- The **urgent pointer** gives the sequence number of the first nonurgent data byte. The urgent bytes are located at the start of the payload for fast access. Only used if URG = 1.
- Only one **option** is currently defined in TCP. It specifies the **maximum segment size**, in bytes, that will be accepted. This size is the largest TCP payload allowed in one segment. The value is determined when the connection is established.

The TCP Finite State Machine

- An 11-state ***finite state machine*** (FSM) is used to structure the status and input/output behaviour of the TCP entity.
- The TCP FSM controls: (a) ***connection establishment***, (b) ***data transfer***, and (c) ***connection termination***.
- At any one time within a host, a TCP connection exists in exactly one state in the TCP FSM. The TCP states at the two ends of the same TCP connection can be different.
- ***Transitions are made from state to state*** within the TCP FSM as a result of ***events***, such as receiving commands from the host's software, timer timeouts, and receiving TCP segments from the other end of the connection.
- In addition to causing state transitions, ***events can also cause TCP segments to be transmitted*** to the other end.

The State Diagram of the TCP FSM



Aside: Client-Server Communication

- Many communications between processes on networked nodes can be structured as client-server interactions.
- A ***server process*** resides on a networked node and waits for other processes (clients) to make connections to it and to request services from it.
Ex: Unix systems will have server processes for each of the standard port numbers to handle incoming connection requests to those port numbers.
- A ***client process*** resides on a networked node and can make connections and send service requests to remote server processes.
- TCP/IP connections are commonly used to support client-server communication over the Internet.

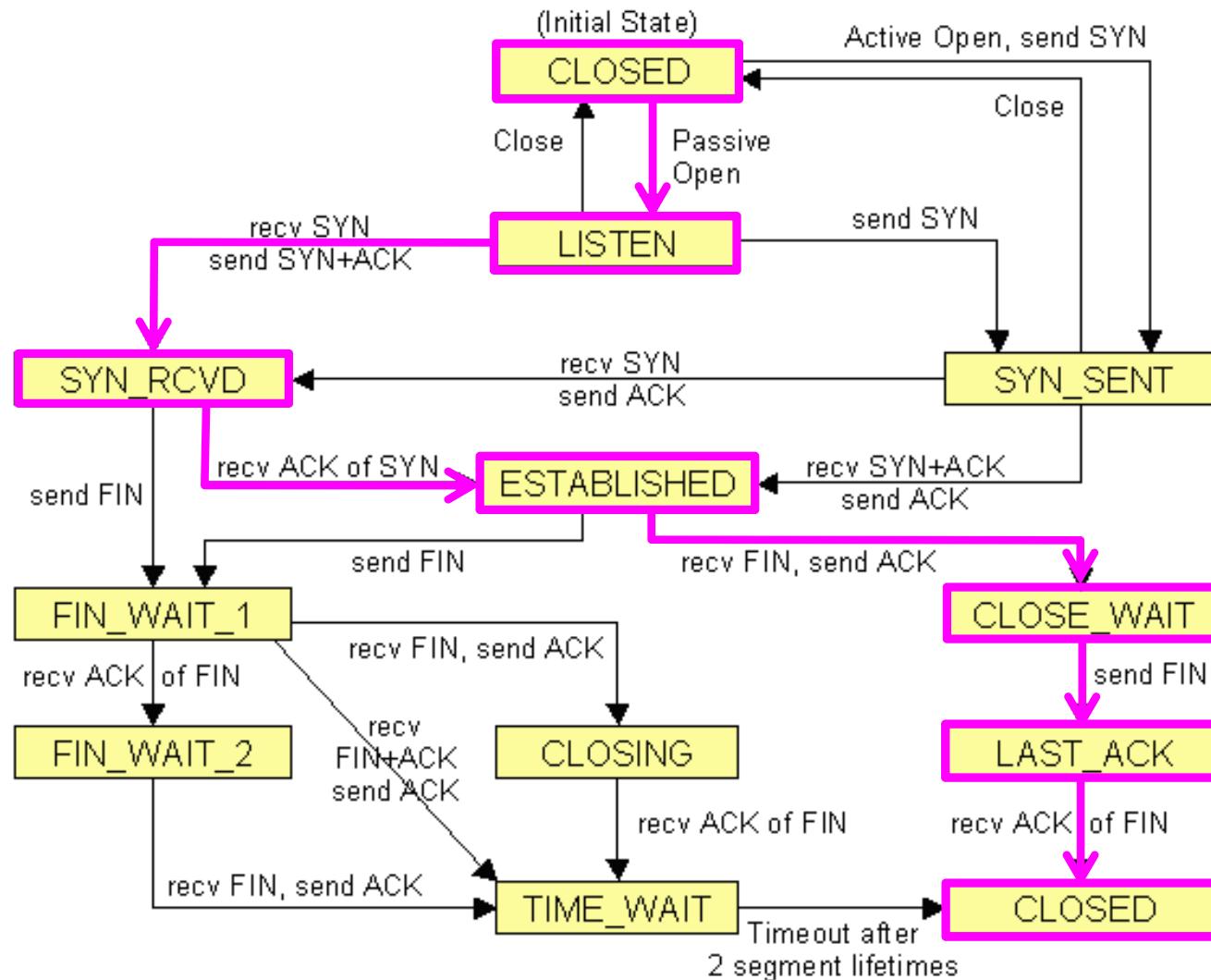
The 11 TCP States

CLOSED	There is no connection.
LISTEN	The server is waiting for a call from a client.
SYN-SENT	Connection request was sent; waiting for ack.
SYN-RCVD	A connection request was received.
ESTABLISHED	A connection is established.
FIN-WAIT-1	The host requests to close the connection.
FIN-WAIT-2	The other side accepts the conn. closing.
CLOSING	Both ends have decided to close the conn.
TIME-WAIT	Waiting for retransmitted segments to die.
CLOSE-WAIT	Server is waiting for the application to close.
LAST-ACK	Server is waiting for the last ack.

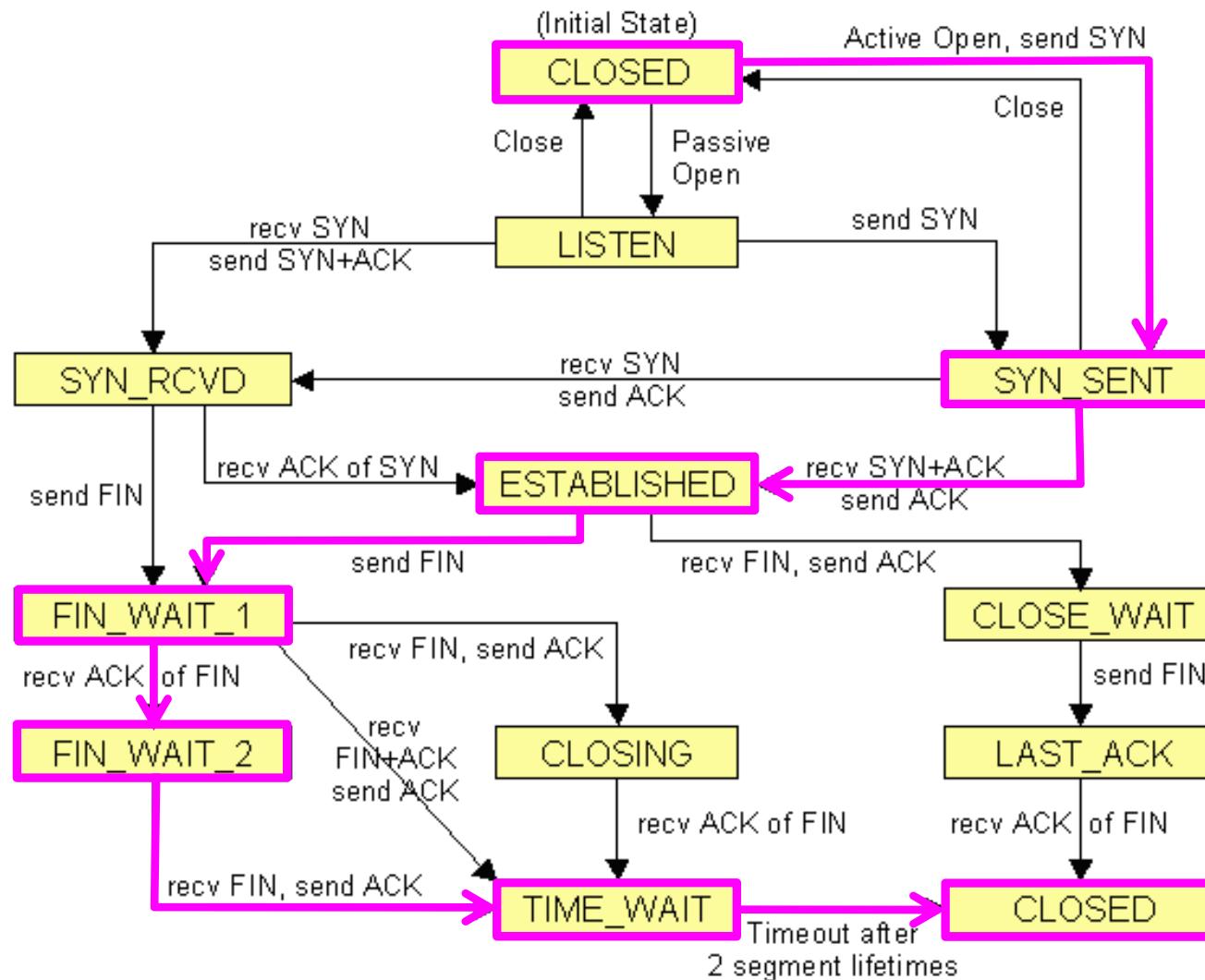
TCP Events Communicated on Segments

- A SYN segment sent by a client has the SYN bit set to 1. The *source and destination port numbers* are specified. The *first sequence number from the client side* is specified so that the server will be synchronized.
- A SYN+ACK segment sent by a server has the SYN and ACK bits set to 1. The *acknowledgement number* is the client sequence number plus 1. The segment also specifies the *first sequence number from the server*, so the client will be synchronized. The *client window size* is specified.
- An ACK segment is sent by the client with the ACK bit set to 1. The *acknowledgement number* is the server sequence number plus 1. The client specifies the *server window size*.
- A FIN segment has the FIN bit set to 1. It is sent by a host to *finish* (take down, terminate) the current connection.
- A RST segment has the RST bit set to 1. It is used by a TCP host to warn the other side that the connection is being *reset*.

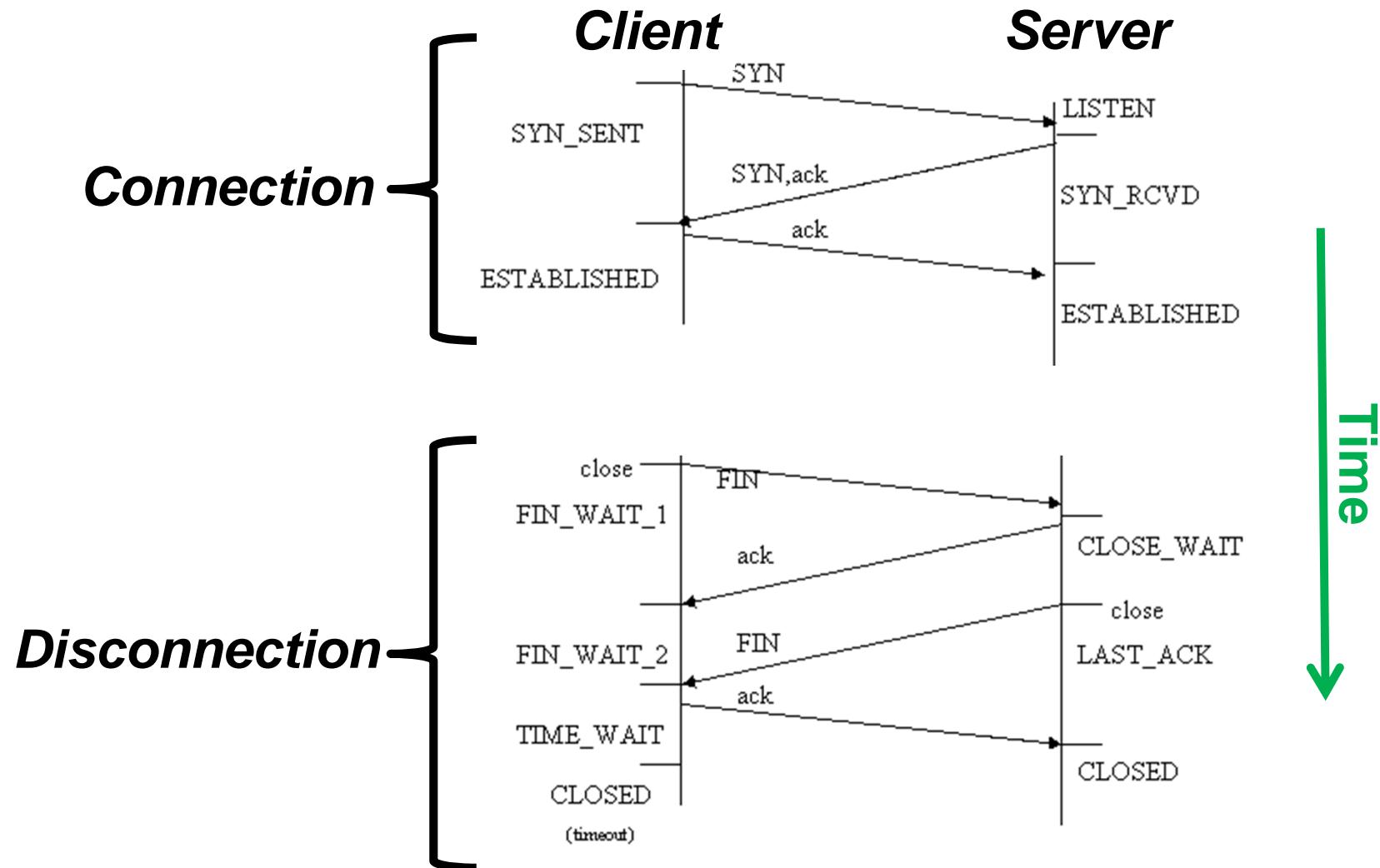
Normal TCP State Sequence in Servers



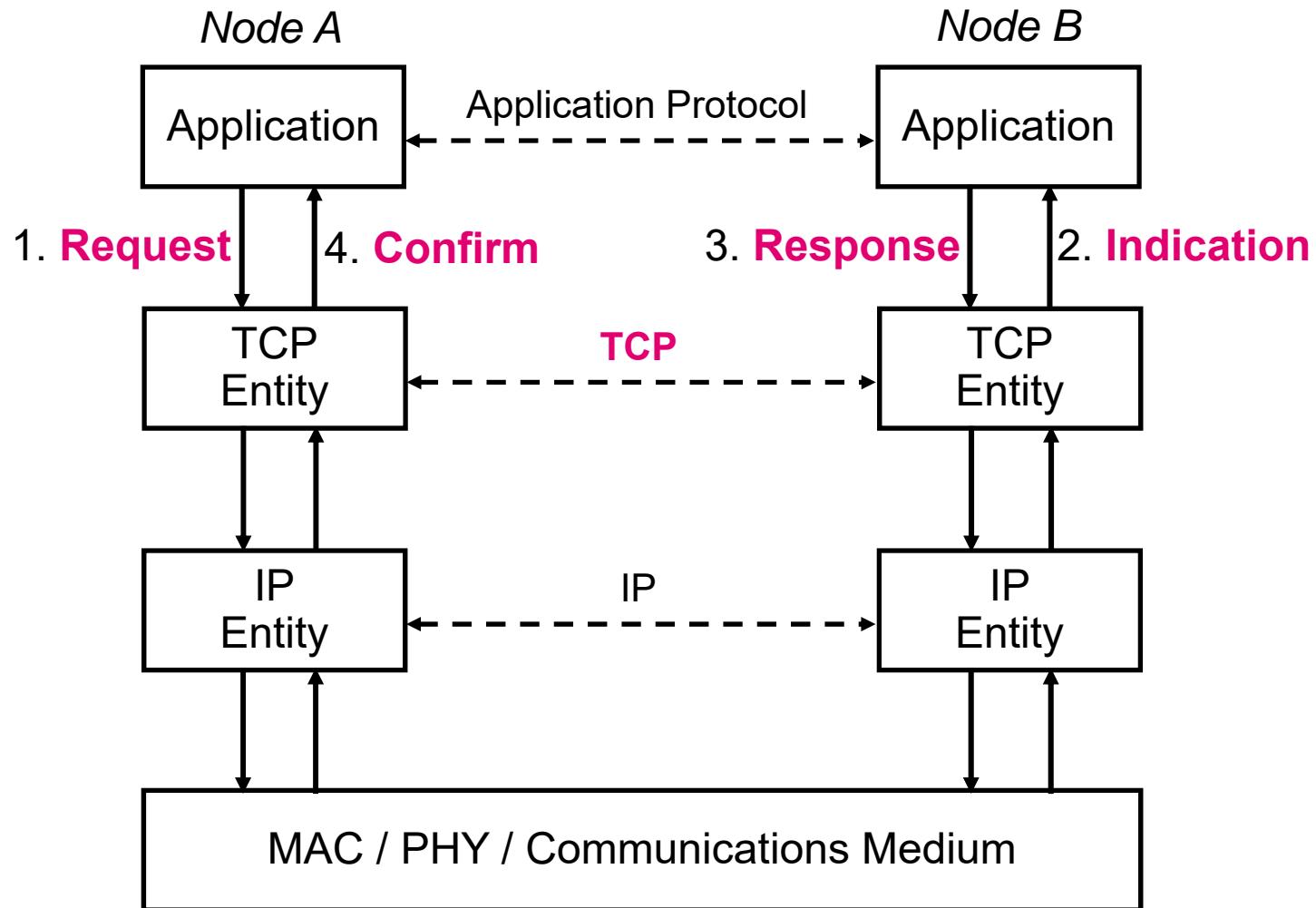
Normal TCP State Sequence in Clients



TCP Connection and Disconnection



Accessing the Services of TCP



TCP Service Request & Response Primitives

1. Listen for connection attempt at specified security and precedence from *any* remote destination.
2. Listen for connection attempt at specified security and precedence from a *specified* remote destination.
3. Request an active connection at a specified security and precedence to a *specified* destination.
4. Request a connection at a particular security and precedence to a *specified* destination and transmit data with the request.
5. Transfer data across the named connection.
6. Issue incremental allocation for receive data to TCP.
7. Close the connection gracefully.
8. Close the connection abruptly.
9. Query the connection status.

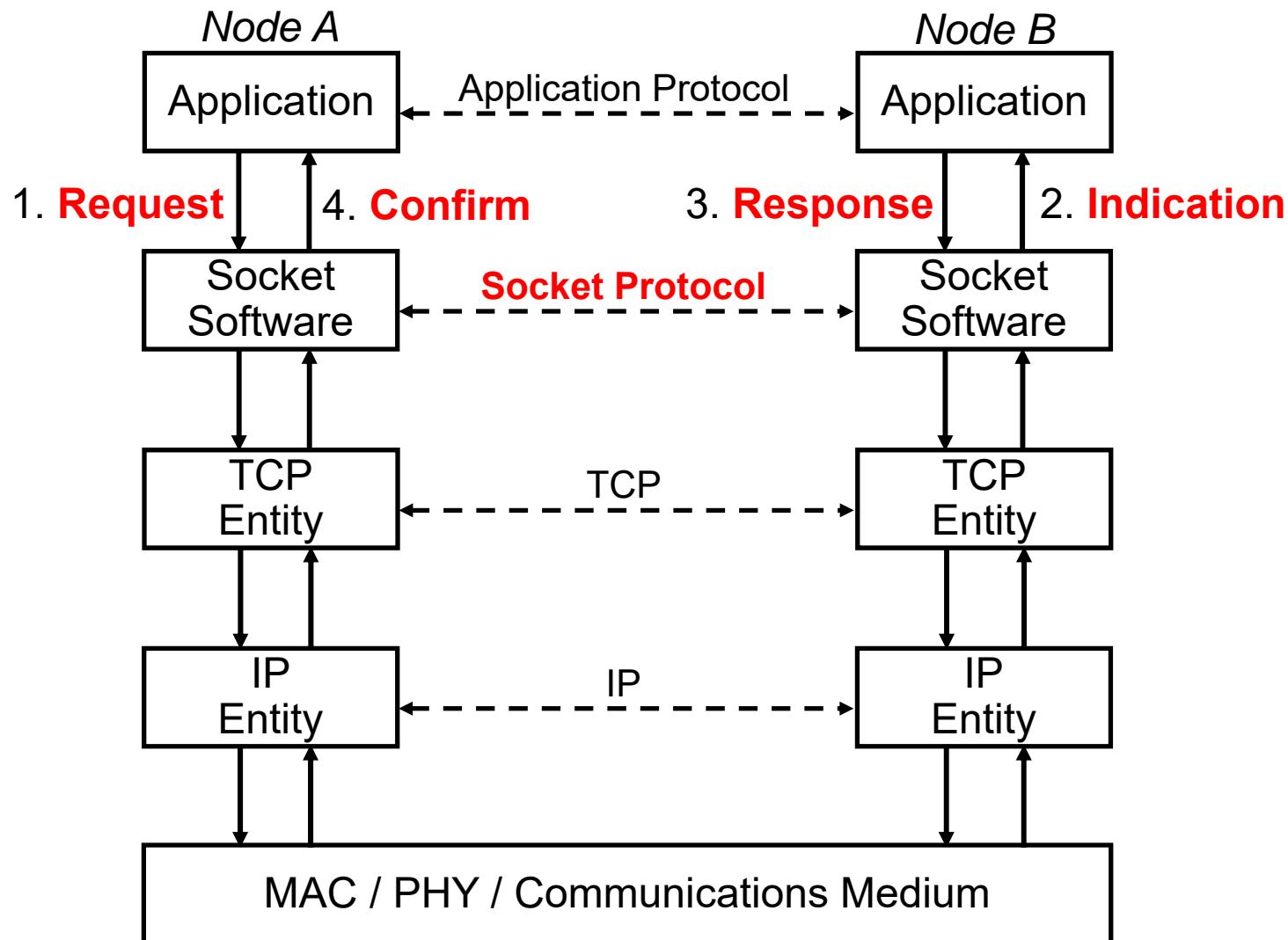
TCP Service Confirm & Indication Primitives

1. Informs TCP user of the connection name that has been assigned to the pending connection that was just requested.
2. Report failure to open an active connection.
3. Report success at obtaining an active connection.
4. Report arrival of data. Data is included.
5. Reports that the remote (peer) TCP user has requested to close the connection gracefully. All remote data has now been sent.
6. Reports that the connection has been terminated. The reason for the termination is included.
7. Reports the current status of the connection.
8. Reports a service request error or an internal error.

Accessing the TCP/IP Services in Unix

- The services of TCP and IP can be accessed directly by implementation-specific function calls or system calls.
- However, because of the complexity of TCP/IP, it is common to provide a simplified ***Application Program Interface*** (API).
- In Unix (including Linux) systems, the API uses software abstractions called “**sockets**”, which can be used like files.
- A socket is really a data structure that ties together a TCP connection with a process or task.
- Sockets must exist and be properly initialized at both ends of a TCP connection for communication to proceed.

Introducing a Socket Layer into the Stack



Client-Server Communication with Sockets

- A ***server process*** gets ready for incoming connection requests by creating and initializing a socket using the API.
 1. `socket()` - create a new socket on local node
 2. `bind()` - bind a socket to a local port number
 3. `listen()` - listen for connection requests to socket
 4. `accept()` - block the process until connection occurs
- A ***client process*** initiates a communication channel by:
 1. `socket()` - create a new socket on local node
 2. `connect()` - connect the socket to specified server
- After the connection has been set up, both processes can exchange data by using the following API functions:
 1. `write()` - send data bytes to the remote process
 2. `read()` - receive data bytes from remote process

Example of Client-Server Code on Unix

- Here the server creates and initializes a socket, then waits for a connection request from a client.
- When a connection establishment request is received, the server will: (1) output the next received message to its output stream; (2) send an acknowledgement back to the client; and then (3) terminate itself.
- The client (1) creates and initializes a socket; (2) requests a connection to the server; (3) prompts the user for a test message to send; (4) sends that message to the server; (5) waits for an acknowledgement message from the server; (6) outputs the acknowledgement; and then (7) terminates itself.

Example Server Code

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno, clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;
    if (argc < 2) {
        fprintf(stderr,"ERROR, no port provided\n");
        exit(1);
    }
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
        error("ERROR on binding");
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd,
                      (struct sockaddr *) &cli_addr,
                      &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    bzero(buffer,256);
    n = read(newsockfd,buffer,255);
    if (n < 0) error("ERROR reading from socket");
    printf("Here is the message: %s\n",buffer);
    n = write(newsockfd,"I got your message",18);
    if (n < 0) error("ERROR writing to socket");
    return 0;
}
```

Source: Sockets Tutorial, www.cs.rpi.edu/courses/sysprog/socket/sock.html

Example Client Code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
        fprintf(stderr,"usage %s hostname port\n", argv[0]);
        exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");

    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
          (char *)&serv_addr.sin_addr.s_addr,
          server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0)
        error("ERROR writing to socket");
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    if (n < 0)
        error("ERROR reading from socket");
    printf("%s\n",buffer);
    return 0;
}
```

Output from Client-Server Example

```
nyquist >
nyquist > server 2000
```

```
itanium >
itanium > client nyquist 2000
Please enter the message:
```

Output from Client-Server Example

```
nyquist >
nyquist > server 2000
```

```
itanium >
itanium > client nyquist 2000
Please enter the message: How's the weather in NY?
itanium>
```

Output from Client-Server Example

```
nyquist >
nyquist > server 2000
Here is the message: How's the weather in NY?
nyquist >
```

```
itanium >
itanium > client nyquist 2000
Please enter the message: How's the weather in NY?
itanium >
```

Output from Client-Server Example

```
nyquist >
nyquist > server 2000
Here is the message: How's the weather in NY?
nyquist >
```

```
itanium >
itanium > client nyquist 2000
Please enter the message: How's the weather in NY?
itanium > I got your message
itanium >
```

BOOTP and DHCP

- ***Bootstrap Protocol*** (BOOTP) is a protocol that is used by a BOOTP server to provide a newly connecting client with (1) its IP address, (2) the subnet mask, (3) the IP address of the gateway, and (4) the IP address of the name server.
- The BOOTP server is accessed using UDP to port 67.
- A static IP address is retrieved from a fixed table using the physical address of the client.
- ***Dynamic Host Configuration Protocol*** (DHCP) is an extension to BOOTP that serves out both static and dynamic IP addresses.
- The DHCP server first attempts to find a static IP address for the physical address. If one is not found, then a dynamic IP address is assigned for a negotiable lease time to the client from a pool of available unused dynamic IP addresses.

File Transfer Protocol (FTP)

- The ***File Transfer Protocol*** (FTP) uses the services of TCP to transfer human-readable text (ASCII or EBCDIC-encoded) or binary files between a client and a server.
- FTP servers listen for control connections to Port 21. Such connections can then allow files to be transferred over separate data connections with the server's Port 20.
- The client-side control and data ports do not have to be 20 or 21, respectively.
- FTP provides a relatively large number, currently 51, of different commands. There are 45 three-digit decimal return codes that are used to respond to FTP commands.
- The major strength of FTP is that it is a widely supported file transfer protocol on the Internet; however, it provides no encryption or protection against snooping.

Frequently Used FTP Commands

USER – enter username for authentication purposes

PASS – enter password for authentication purposes

PORT – specify the client port for the data connection

CWD – change the working directory on the server

MKD – make a directory on the server

RMD – remove a directory from the server

TYPE – set transfer mode to either ASCII or binary

GET / RETR – retrieve (download) a remote file

PUT / STOR – store (upload) a file to the remote server

DELE – delete a file from the server

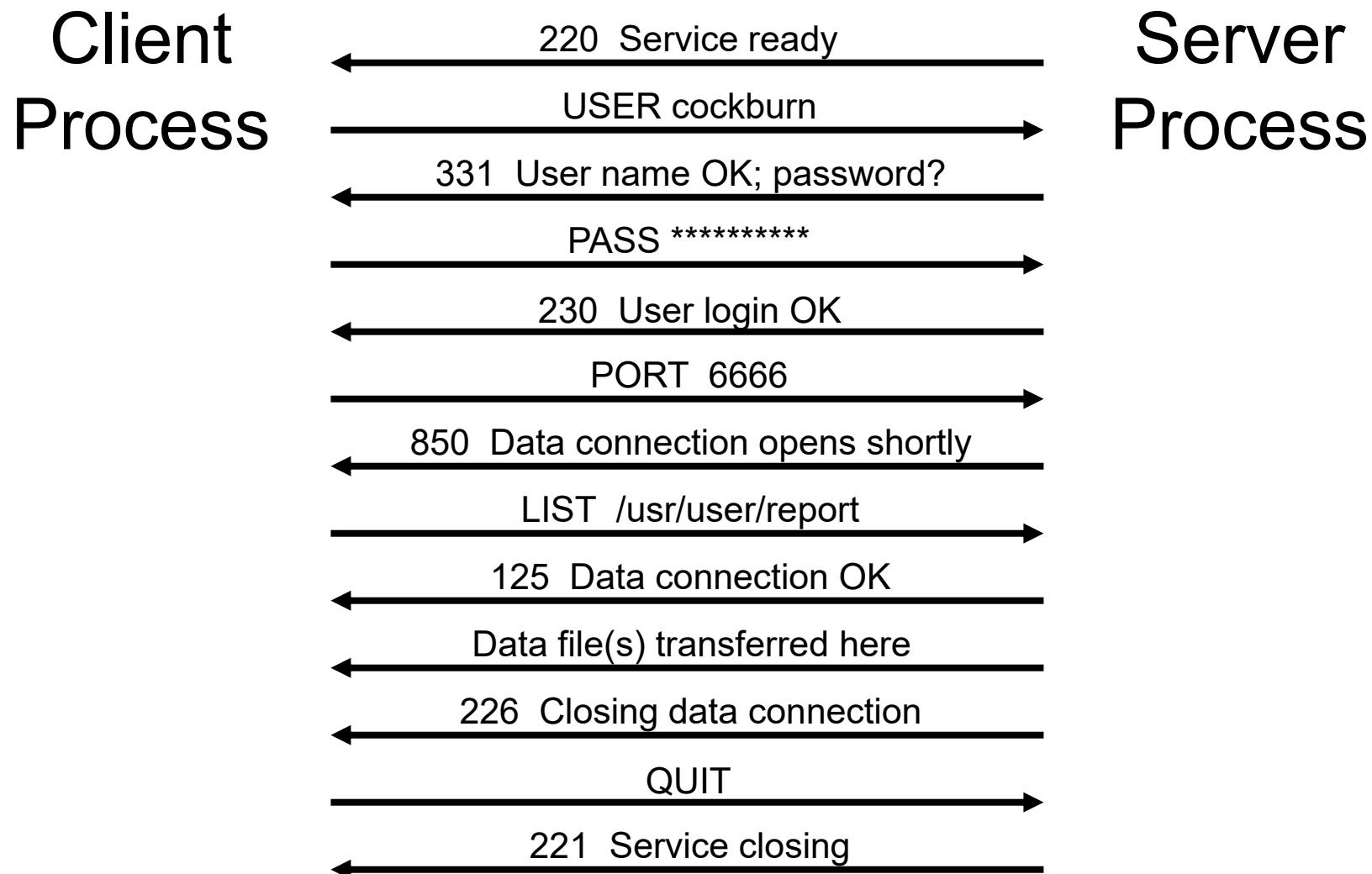
LIST – return information on a specified file or directory

QUIT – terminate the FTP session

Some Common FTP Return Codes

- 150 File status is OK; data connection will be open shortly
- 200 Command OK
- 220 Service ready
- 221 Service closing
- 225 Data connection open
- 226 Closing data connection
- 230 User login OK
- 331 User name OK; password is needed
- 425 Cannot open data connection
- 426 Connection closed; transfer aborted
- 550 Action is not done; file unavailable
- 553 Requested action not taken; file name not allowed

Example: A Brief FTP Session



TELNET

- It is often convenient to be able to log in to a remote server to access software applications as easily as if they were located physically at the site of the server.
- The Internet application TELNET allows one to create a “virtual terminal” to a remote server using a single TCP/IP connection.
- Port 23 is reserved for TELNET on the server side. The client can choose to use another port number.
- TELNET uses an intermediate Network Virtual Terminal (NVT) character set to safely translate characters between the character set of the client and the character set of the server.
- NVT is an extension of ASCII. All 128 7-bit ASCII codes are included, with a fixed “0” bit added as an eighth most significant bit (MSB).
- NVT control codes have been defined, with a fixed “1” as the MSB.
- A special “Interpret as control” code (0xFF) must be sent before each control code to avoid confusion between data and control codes. Binary data 0xFF is sent as one 0xFF followed by a second 0xFF.

Hypertext Transfer Protocol (HTTP)

- The ***Hypertext Transfer Protocol (HTTP)*** is an application layer protocol that is widely used on the TCP/IP-based Internet.
- HTTP is used to exchange information (text, hypertext, images, sounds, video) over the ***World-Wide Web*** (WWW). “Hypertext” is text that is provided with clickable links that simplify the selection and retrieval of related information. A webpage is a document that can contain hypertext.
- The ***Hypertext Markup Language (HTML)*** is widely used to encode the appearance of webpages. HTML is used in the body of an HTTP packet.
- HTTP is a simple request-response protocol:
 - the client sends a request message to the server
 - the server sends a response message to the client
 - a TCP/IP connection is created automatically
- TCP/IP Port 80 on the server is reserved for HTTP.
- At the client machine, a user interacts with HTTP using a browser application (e.g. Internet Explorer, Firefox, Safari, etc.). At the server, HTTP requests are interpreted by a “Webserver” application.

HTTP Request Messages

- Request messages have four parts:
 - Request line = Request type + URL + HTTP version
 - One or more HTTP header lines
 - A blank separator line
 - An optional body (one or more lines)
- A ***Uniform Resource Locator*** (URL) is an Internet file address. The older expression Universal Resource Locator is a synonym.

The URL format is used by several TCP/IP applications.

URL = Method :// HostAddress : Port / Path

Method = HTTP, FTP, TELNET, Gopher, News, etc.

HostAddress = IP address or symbolic address of the host

Port = TCP port number at the destination (e.g. 80 for HTTP)

Path = Pathname on the destination host to the desired file

HTTP Request Line

- Request Line = Request type + URL + HTTP version

Ex: GET http://www.cern.org:80/usr/bin/image1 HTTP/1.1

Ex: GET http://www.cern.org/usr/bin/image1 HTTP/1.1

Ex: GET /usr/bin/image1 HTTP/1.1

Ex: GET http://www.ece.ualberta.ca/~cmpe401/ HTTP/1.1

- Some of the other possible request types/methods in HTTP:

POST Send input data to server.

PUT Send a new file to server.

OPTIONS Obtain a list of the options of the destination node.

DELETE Remove the specified file from the server.

TRACE Perform msg. loopback test to the destination node.

HTTP Header Lines

- HTTP request and response messages can contain header lines.
- Header lines communicate additional information, such as formatting, data handling capabilities, software version numbers, date + time, etc.
- Header line format = Name : (space) header-value
- **General headers** (used in both request and response messages)
Ex: Date, MIME-version, Cache-control, Connection, Upgrade
- **Request headers** (used only in request messages)
Ex: Accept, Accept-charset, Accept-encoding, Accept-language
Authorization, If-modified-since, User-agent, etc.
- **Response headers** (used only in response messages)
Ex: Accept-range, Age, Public, Retry-after, Server
- **Body Headers** (used in messages that contain bodies)
Ex: Content-encoding, Content-length, Expires, Last-modified, etc.

HTTP Response Messages

- Response messages have four parts:
 - Status line
 - One or more HTTP header lines
 - A blank separator line
 - An optional body (one or more lines)
- Status line = HTTP version + status code + status phrase
- Some typical status codes and phrases (and meanings):

200 OK	(The request was successful.)
302 Moved permanently	(The requested URL has been removed.)
400 Bad request	(There is a syntax error in the request.)
403 Forbidden	(The requested service is denied.)
404 Not found	(The file was not found.)
503 Service unavailable	(The service is temporarily unavailable.)

HTTP Example #1

Request message sent from client to server:

```
GET /usr/bin/image1 HTTP/1.1  
Accept: image/gif  
Accept: image/jpeg
```

Response message sent from server to client:

```
HTTP/1.1 200 OK  
Date: Mon, 21-Oct-04 13:37:14 GMT  
Server: Nyquist  
MIME-version: 1.0  
Content-length: 1024
```

(Body of the document)

HTTP Example #2

Request message sent from client to server:

```
HEAD /usr/homepage.html HTTP/1.1  
Accept: */*
```

Response message sent from server to client:

```
HTTP/1.1 200 OK  
Date: Tue, 09-Oct-04 15:12:10 GMT  
Server: Nyquist  
MIME-version: 1.0  
Content-type: text/html  
Content-length: 2048
```

HTTP Example #3

Request message sent from client to server:

```
POST /cgi-bin/doc.pl HTTP/1.1  
Accept: image/gif  
Accept: image/jpeg  
Content-length: 50
```

(Input information)

Response message sent from server to client:

```
HTTP/1.1 200 OK  
Date: Fri, 22-Oct-04 18:32:14 GMT  
Server: Nyquist  
MIME-version: 1.0  
Content-length: 1000
```

(Body of the document)

Dynamic Webpages

- A webpage whose content does not change often can be encoded as an occasionally updated ***static HTML*** file.
- However, there are many situations where it is desirable for the server and/or the client to have the ability to change the content and/or the appearance of a webpage.
- At the server, an HTML file can be modified (e.g., ASCII edits) at the time that it is sent back in response to a client request. However, this is a rather inflexible approach.
- ***Server-side scripting languages*** (e.g., ASP, JSP, PHP, ColdFusion, Perl, etc.) are now commonly used to provide dynamic webpages on the server.
- ***Client-side scripting languages*** (e.g., JavaScript, Flash) are used to provide dynamic effects in webpages that are displayed by the browser at the client.

Interfacing MicroC/OS to TCP/IP

Interfacing Small Systems to TCP/IP

- TCP/IP is a complicated protocol that requires substantial memory space and program complexity.
- Special care is required when interfacing small systems, with limited memory and CPU resources, to TCP/IP.
- As implementation examples, we will consider how MicroC/OS has been interfaced to two TCP/IP stacks:
 1. NetBurner's TCP/IP stack (the ECE 315 lab system)
 2. The Lightweight IP (lwIP) portable, open-source TCP/IP stack. [Adam Dunkels, "Design and Implementation of the LWIP TCP/IP Stack," Swedish Institute of Computer Science, Feb. 20, 2001.]

Opportunities for Saving Time and Space

- Strict partitioning of the TCP/IP interface according to the protocol layers can be inefficient.
Ex. It may be more efficient at times for TCP to be able to look directly into the IP header without always having to go through the IP layer software.
- We want to avoid unnecessary copying of data and packet buffers between the application and the communication subsystem, or between different layers in the protocol stack.
Ex. We may want the application to use the same buffers as the communication subsystem even if this violates strict partitioning between applications and the TCP/IP stack and operating system.

Implement the Protocol Layers as Tasks?

- One could employ separate tasks for each protocol layer:
 - (4) Application task(s)
 - (3) Transport (TCP) layer task
 - (2) Internetworking (IP) layer task
 - (1) Device driver task (for Ethernet, RS-232C, etc.)This approach was taken by NetBurner's TCP/IP stack.
- *Advantages:*
 - interlayer interfaces strictly defined and enforceable
 - debugging and code maintenance is easier
 - can more easily modify protocol settings at run-time
- *Disadvantages:*
 - larger required number of task context switches
 - need to carefully control access to shared buffers
 - access to multi-layer protocol data is slowed down

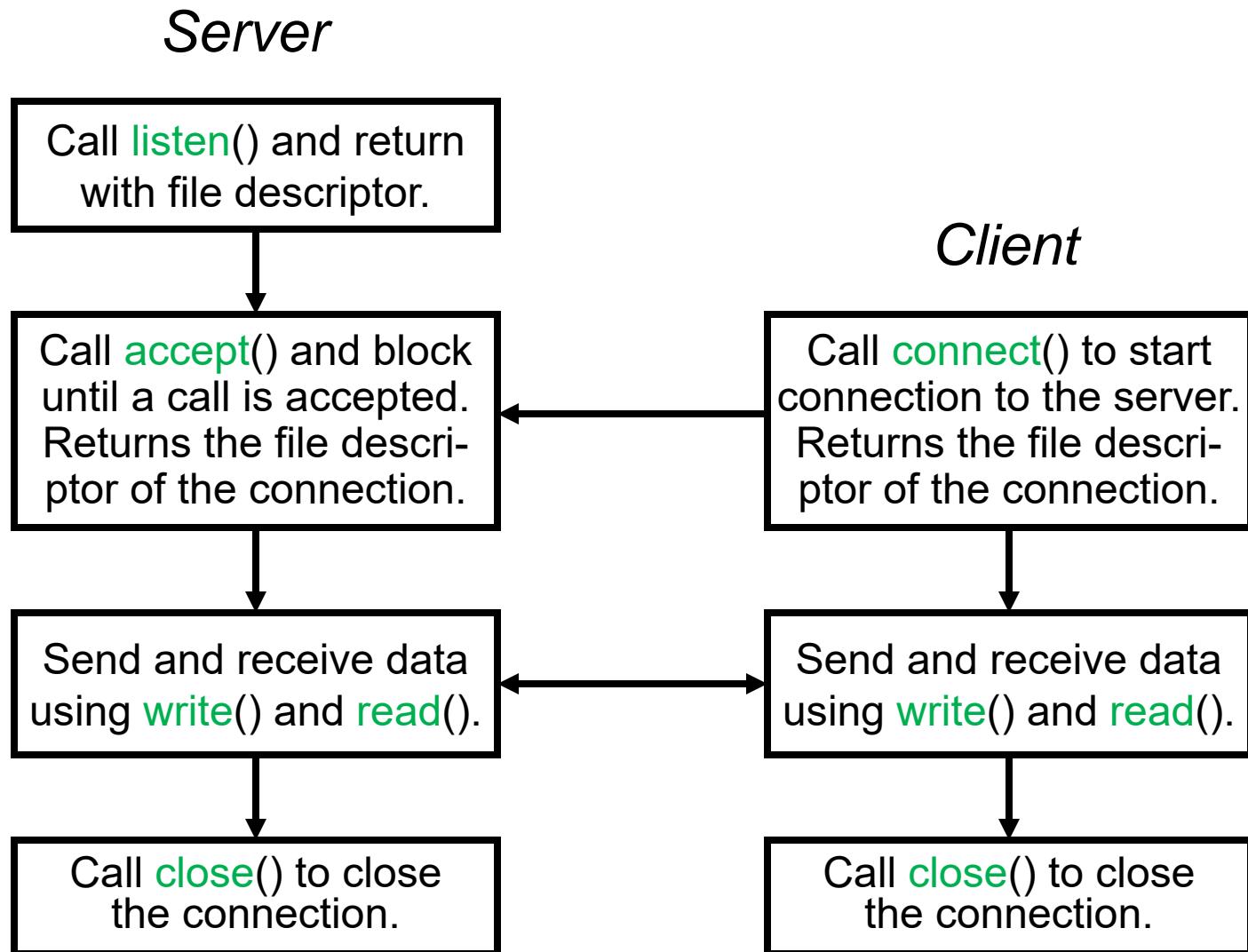
Some Other Implementation Options

- Embed the TCP/IP stack inside the kernel. User tasks must access the stack using system calls (also called “traps”).
Advantages: Fast & efficient; users shielded from stack details.
Disadvantages: More difficult to change or modify the stack.
- Combine protocol layers into one task, outside the kernel. User code may, in some cases, be combined with this same task.
Advantages: Protocol stack is more portable across kernels.
 Opportunities to gain efficiency in space & time.
Disadvantages: Stack software is less clearly partitioned apart from user code.
- Lightweight IP takes the last approach, with the option of keeping user code in the same or a separate task from the stack task.

The NetBurner TCP/IP Stack in MicroC/OS

- NetBurner's TCP/IP stack is partitioned across several tasks that handle the different protocol layers:
 - Priority 38: Ethernet hardware (FEC) driver task
 - Priority 39: IP entity task
 - Priority 40: TCP entity task
 - Priority 44: PPP entity task (for TCP/IP over RS-232C)
 - Priority 45: HTTP web server task (created by StartHTTP)
 - Priority 50: Default priority of the UserMain() task
- The FEC driver task has the highest priority (i.e., lowest priority number in MicroC/OS) so that Ethernet hardware actions will be handled promptly before IP and TCP.

NetBurner's Client-Server Architecture



TCP/IP Application Programming Interface (1)

New data types introduced in the TCP/IP stack:

- IPADDR - data structure that holds one IP address or mask

File descriptor error return codes (all negative values):

- TCP_ERR_TIMEOUT - connection timed out
- TCP_ERR_NOCON - connection not yet established
- TCP_ERR_CLOSING - connection closed and not available
- TCP_ERR_NOSUCH_SOCKET - socket does not exist
- TCP_ERR_NONE_AVAIL - no more sockets are available
- TCP_ERR_CON_RESET - connection reset by other side
- TCP_ERR_CON_ABORT - internal stack error

TCP/IP Application Programming Interface (2)

```
void InitializeStack( IPADDR IP_addr = 0,  
                      IPADDR IP_mask = 0,  
                      IPADDR IP_gateway = 0 );
```

- function that initializes the TCP/IP stack
- typically called by UserMain()
- "IP_addr" is the IP address of the local node
- "IP_mask" is the IP mask of the local node
- "IP_gateway" is the IP address of the gateway for the local node
- If called with no inputs, then default IP values are copied from the system configuration record.

```
void KillStack( );
```

- function that shuts down the TCP/IP stack
- typically called by UserMain()

TCP/IP Application Programming Interface (3)

int listen(IPADDR IP_addr, WORD port_num, BYTE max_pends);

- called by the server to put the TCP entity into the LISTEN state
- “IP_addr” specifies a client IP; value INADDR_ANY means that a connection/call will be accepted from any client IP
- “port_num” specifies the listening port number on the server
- “max_pends” specifies maximum number of pending connections
- return value is the listening file descriptor, i.e. a dummy socket

**int accept(int fdl, IPADDR *Pclient_IP, WORD *Pclient_port,
WORD timeout);**

- called by the server to block until a client call is received
- “fdl” is the listening file descriptor, i.e. the dummy listening socket
- “Pclient_IP” points to the IP address of a new calling client
- “Pclient_port” points to the port number on the client
- Pclient_IP and Pclient_port can be left null if information not wanted
- “timeout” is the maximum number of ticks waiting for a call; a “timeout” value of 0 means wait forever for the next call
- return value is the network file descriptor, i.e. the working socket

TCP/IP Application Programming Interface (4)

int read(int fdnet, char *buf, int buf_siz);

- used by servers & clients to receive a char array payload
- “fdnet” is the network file descriptor, i.e. the working socket
- “buf” stores the retrieved payload, which is a char array
- “buf_siz” is the maximum number of chars to be read
- positive return value is the actual number of new chars in “buf”
- negative return code means the socket was closed by the other side.

int write(int fdnet, char *buf, buf_siz);

- used by servers & clients to send a char array payload
- “fdnet” is the network file descriptor, i.e. the working socket to write
- “buf” holds the payload to be written, which is a char array
- “buf_siz” is the number of chars to be written
- return code is the number of chars actually written

TCP/IP Application Programming Interface (5)

```
int ReadWithTimeout( int fdnet, char *buf, int buf_siz,  
                      unsigned long timeout );
```

- used by servers & clients to receive a char array payload
- “fdnet” is the network file descriptor, i.e. the working socket
- “buf” stores the received payload, which is a char array
- “buf_siz” is the maximum number of chars to be read
- “timeout” is a timeout to socket closure, expressed in timer ticks
- a positive return value is the number of chars in char_buf
- a negative return value means the other party closed the socket

```
void writestring( int fdnet, char *buf );
```

- an alternative to the write() function
- “fdnet” is the network file descriptor, i.e. the working socket
- writes a null-terminated string to a network file descriptor

TCP/IP Application Programming Interface (6)

```
int connect( IPADDR remote_IP, WORD local_port, WORD remote_port,  
          DWORD timeout );
```

- connects a client to the specified port on a remote server
- “remote_IP” specifies the IP address of the remote server
- “local_port” specifies the port number to use for the connection; a value of 0 causes the stack to choose an unused port
- “remote_port” specifies the port number on the server
- “timeout” is a timeout, expressed in timer ticks; a value of 0 means that the function will wait forever
- return value is the network file descriptor, i.e. the working socket

```
void close( fdnet );
```

- called by both servers & clients to end a TCP connection
- “fdnet” is the network file descriptor, i.e. the working socket

Example: A Simple Echo Server (1)

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

#define ECHO_LISTEN_PORT 23      // use the TELNET port
#define RX_BUFSIZE 4096

DWORD EchoServerTaskStack[USER_TASK_STK_SIZE];
    __attribute__(( aligned(4)));


const char *Echo_server = "Simple echo server";

char RXBuffer[RX_BUFSIZE];
int ListenPortNumVar = ECHO_LISTEN_PORT;
```

Example: A Simple Echo Server (2)

```
void EchoServerTask( void *pd )
{
    IPADDR client_IP;
    WORD port;
    int ListenPort = *(int *)pd;
    int fdListen = listen(INADDR_ANY, ListenPort, 7);
    if (fdListen > 0) {
        while (1) {
            int fdnet = accept(fdListen, &client_IP, &port, 0);
            while (fdnet > 0) {
                writestring( fdnet, "Welcome to the server\r\n" );
                int n = 0;
                do {
                    n = read( fdnet, RXBuffer, RX_BUFSIZE );
                    write( fdnet, RXBuffer, n );
                } while ((n > 0) && (RXBuffer[0] != '.' ))
                close( fdnet );
                fdnet = 0;
            } // end while (fdnet > 0)
        } // end while (1)
    } // if (fdListen > 0)
}
```

Example: A Simple Echo Server (3)

```
void UserMain( void *pd)
{
    InitializeStack();

    if (EthernetIP == 0) { GetDHCPAddress(); }
    EnableAutoUpdate();

    OSChangePrio(MAIN_PRIO);

    OSTaskCreate( EchoServerTask,
        (void *) &ListenPortNumVar,
        (void *) &EchoServerTaskStack[USER_TASK_STK_SIZE],
        (void *) EchoServerTaskStack,
        MAINPRIO-1) ; // higher priority than UserMain()

    while (1) {
        OSTimeDly( TICKS_PER_SECOND * 5 );
    } // end while (1)
}
```

NetBurner's HTTP Server (1)

- The simple echo server could be used to build an HTTP server; however, NetBurner already provides an extensible HTTP server.
- NetBurner's HTTP server (a “webserver”) is implemented as a task at priority `HTTP_PRIO`, which is 45 by default.
- Required steps to use NetBurner's webserver:
 - include the header file “`http.h`”
 - include the header file “`htmlfiles.h`”
 - initialize the TCP/IP stack by calling “`InitializeStack()`”
 - start up the HTTP server by calling “`StartHTTP()`”
- A default web page will be returned for GET requests to port 80.
- The HTTP server is shut down by calling “`StopHTTP()`”.

NetBurner's HTTP Server (2)

New types that are defined and used in the HTTP server:

```
typedef char * PSTR // pointer to a string  
typedef const char * PCSTR // pointer to a constant string
```

void StartHTTP(WORD port = 80);

- starts up the HTTP server task
- default HTTP port of 80 can be overridden
- usually called by UserMain();
- must have previously called InitializeStack();

void StopHTTP();

- shuts down the HTTP server task

NetBurner's HTTP Server (3)

```
// Default simple handler of HTTP GET requests
int BaseDoGet( int sock, PSTR url, PSTR rxBuffer )
{
    if (*url == 0) { // null url, so GET default webpage
        RedirectResponse( sock, url_of_default_webpage );
        // url_of_default_webpage is typically "HTML/index.htm"
        // All other webserver .htm files are in the HTML dir.
        return 1;
    }
    if ( httpstrcmp(url,"ECHO") ) { // ECHO server loop
        while (*url==0) url++; // Advance past nul's
        SendTextHeader( sock ); // Send HTML header
        *url = ' ';
        writestring( sock, rxBuffer ); // Echo back the message
        return 1;
    }
    if (!SendFullResponse( url, sock )) { // GET webpage
        NotFoundResponse( sock, url ); // else 404 not found
    }
    return 0;
}
```

NetBurner's HTTP Server (4)

void RedirectResponse(int fd, PCSTR url);

- redirect the current GET request to the webpage at "url"
- send the webpage to socket "fd"

int SendFullResponse(PCSTR url, int fd);

- responding by GET-ing the stored webpage for the given "url"
- the webpage must have been stored previously in the html subdirectory of the project directory before building the project
- the webpage may have dynamic parts that change at runtime
- send the HTTP header plus webpage back to socket "fd"
- returns 1 if webpage was found; otherwise, returns 0

void NotFoundResponse(int fd, PCSTR url);

- no webpage was found on the server with the given "url"
- send back an HTTP "not found" response to socket "fd"

NetBurner's HTTP Server (5)

int httpstrcmp(PCSTR string, PCSTR pattern);

- "string" contains an array of chars
- "pattern" contains an array of UPPER CASE chars
- returns 1 if the shorter "string" or "pattern" appears as a prefix of the longer array of "string" or "pattern"; otherwise, returns 0

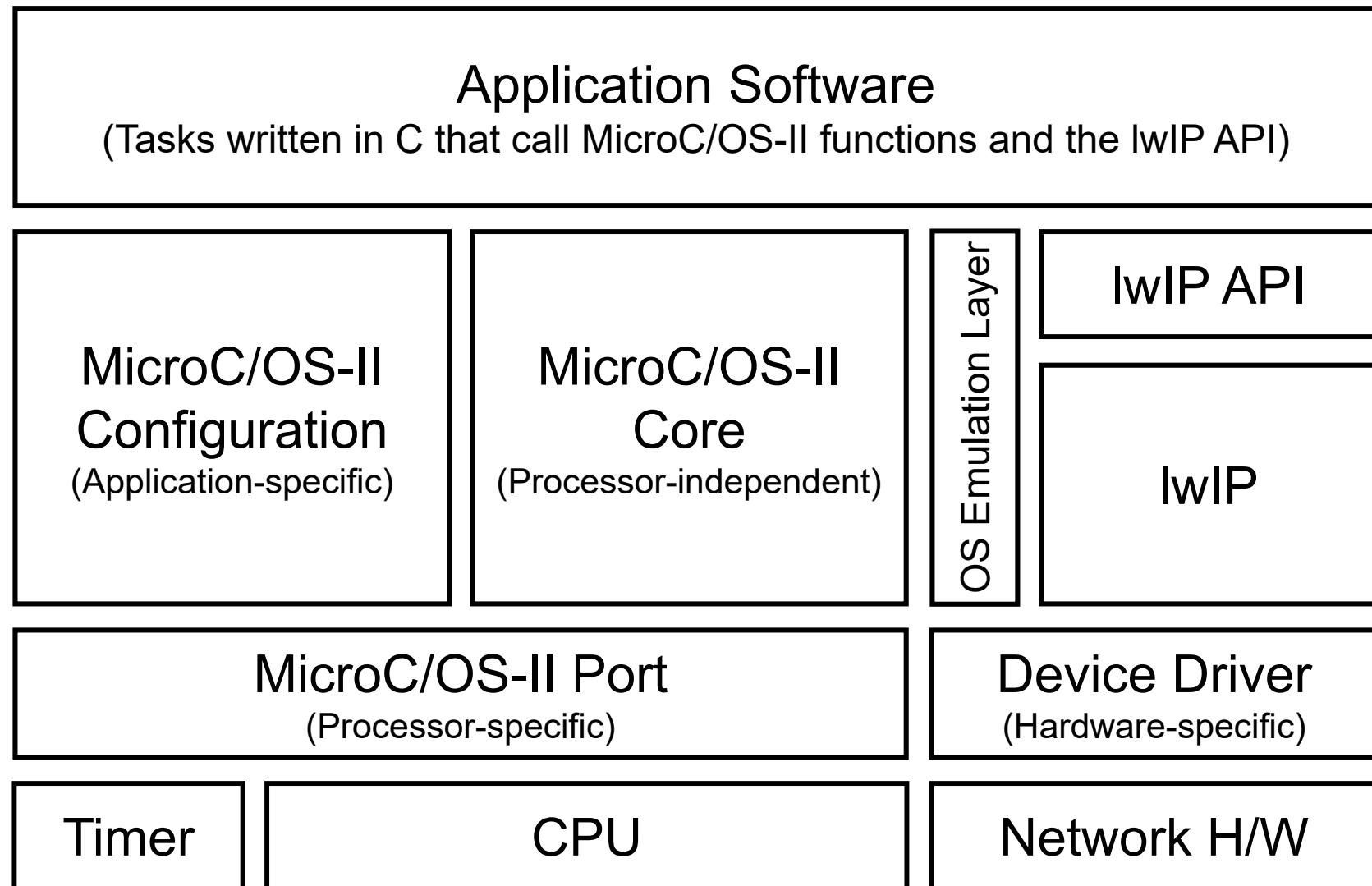
void SendTextHeader(int fd);

- send the appropriate HTTP response header for a text reply
- must be the first part of the HTTP response to socket "fd"
- followed by a write of a plain text file
- used to response to HTTP ECHO requests

void SendHTMLHeader(int fd);

- send the appropriate HTTP response header for an HTML file
- followed by a write of an HTML file

Architecture showing MicroC/OS-II with lwIP



Components of Lightweight IP

IwIP Application Program Interface (API):

- simplified function calls for accessing TCP/IP network
- very similar to the socket API in Unix

IwIP Process:

- kernel/OS-independent implementation of TCP/IP

Operating System Emulation Layer:

- minimal set of functions required of the O.S. by IwIP

Network Hardware Interface:

- provides access to physical layer: Ethernet, RS-232C, Bluetooth, wireless LAN (IEEE 802.11b, 802.11a), etc.

Device Driver:

- software that initializes and operates the hardware

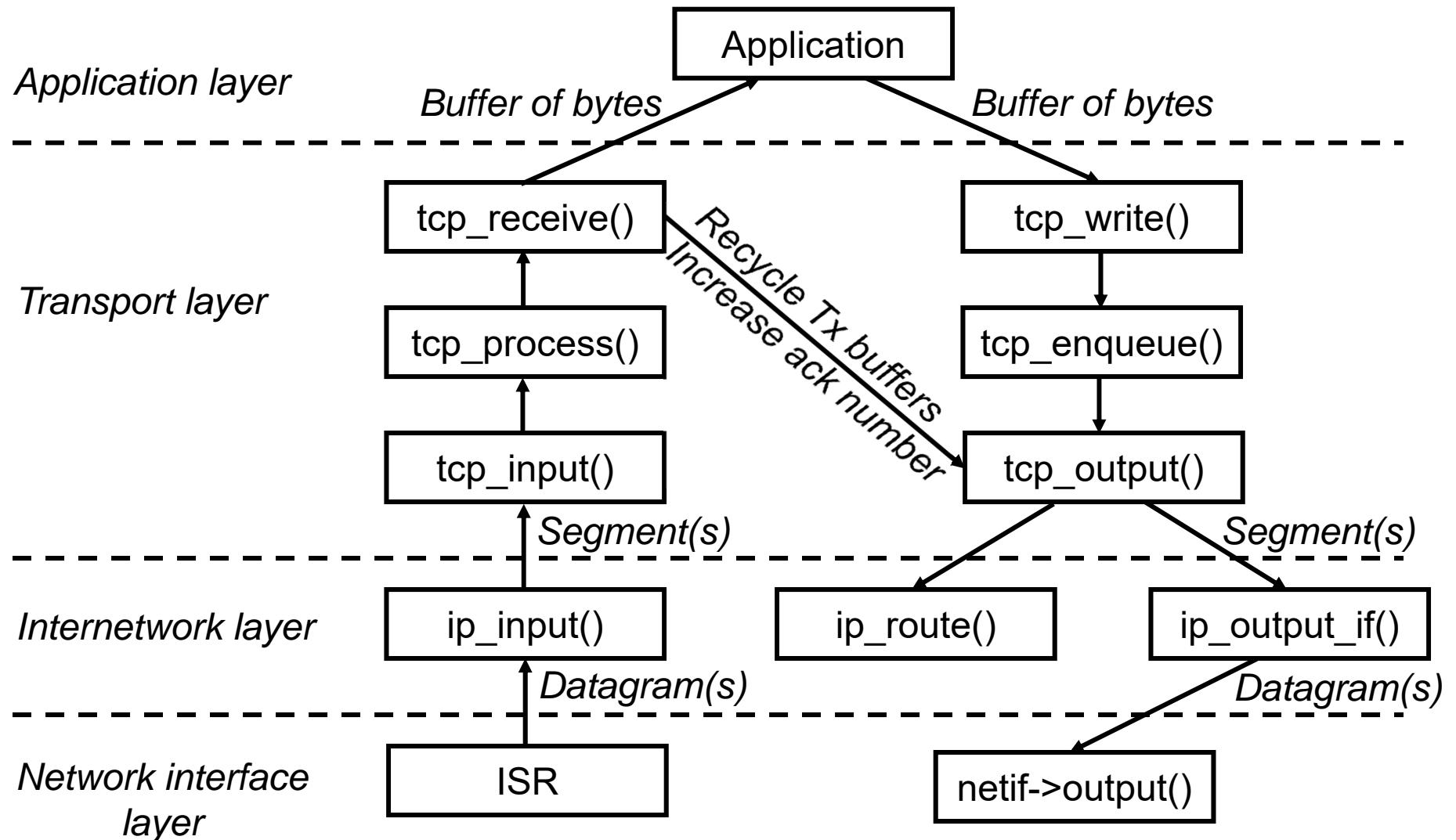
Operating System Emulation Layer

- Operating System specific functions are kept together in this layer to make it easier to port lwIP to different OS's.
- Required functionality in the operating system or kernel:
 - One-shot timer with at least 200 ms resolution
 - Semaphore process synchronization mechanism
 - Mailbox system with:
 - (1) nonblocking “post to message” queue function
 - (2) blocking “receive from message” queue function
- Lightweight IP has its own buffer management system so that it can exploit potential efficiencies and also be more independent of the other parts of the system.

Device Driver

- A **device driver** is a body of software that takes care of initializing and operating a hardware device or interface.
Typical components in a device driver:
 - initialization routine
 - data output routine
 - data input routine
 - interrupt service routine(s)
 - data buffers, state variables
 - query interface status
- In lwIP, as in Unix systems, there can be several device drivers available for operating different network interfaces.
- All device drivers can be used in the same way by higher-level lwIP software, so lwIP is shielded from device-specific details.

TCP/IP Data Processing in lwIP



Processing of Incoming Datagrams in lwIP

- ip_input()
 - verifies the IP header of the datagram
 - datagram discarded if error detected
 - verifies if datagram destination is host node
 - passes resulting segment up to the appropriate higher layer protocol (TCP, UDP, ICMP)
- tcp_input()
 - verifies TCP header checksum in segments
 - parses options in the TCP header
 - from port number, determines TCP connection
- tcp_process()
 - make any state transitions in TCP protocol
 - call tcp_receive() if connection ready for data
- tcp_receive()
 - strip off TCP header and pass data to application
 - recycle any ACKed outgoing segments in buffer
 - call tcp_output() if ACKed transmitted bytes allow more outgoing segments to be transmitted

Processing of Outgoing Data Bytes in lwIP

- tcp_write() - called to pass data byte buffer to TCP layer
- tcp_enqueue() - if necessary, break up data buffer into pieces
 - queue up the resulting TCP segments
- tcp_output() - check to see if more segments can be sent
 - if yes, call ip_route then ip_output_if()
- ip_route() - find the next node in the datagram's route and
 select the outgoing port from the present node
- ip_output_if() - assemble datagram, and call device driver
- ifnet->output() - the device driver that loads the datagram into
 the transmitter hardware

Servo and Stepper Motor Control

References:

- STMicroelectronics, “L298 Dual Full Bridge Driver,” datasheet, Jan. 2000.
- STMicroelectronics, “L297 Stepper Motor Controllers,” datasheet, Dec. 2001.

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 in any design.

Actuators, Relays, Servos and Stepper Motors

- Microcomputers are often required to control the movement of mechanical devices or to set other physical parameters.
- An ***actuator*** is a device that is used by a microcomputer to control an external physical quantity, such as position.
- A ***relay*** is a mechanical switch that is closed and opened by driving or not driving current through a magnetic coil.
 Normally open (n.o.) relay: close the switch by energizing
 Normally closed (n.c.) relay: open the switch by energizing
- A ***servo*** is an actuator that is used to cause mechanical rotation within some fixed range of angles (< 360 deg.)
- A ***stepper motor*** is an actuator whose rotational position (360 deg.) and speed can be controlled by digital signals.

Servos

- A **servo** is an actuator that is used to cause mechanical rotation within some fixed range of angles (< 360 deg.)
- Example application: Servos are used in radio controlled aircraft to control the flight surfaces: *rudder* (yaw control), *elevator* (pitch control), and *ailersons* (roll control).
- A widely used control interface for servos uses an analog signaling method called **pulse width modulation (PWM)**. Many microcontrollers have PWM interfaces (e.g., the MCF54415 has 8 PWM channels).
- A train of positive pulses is sent at some fixed repetition rate, for example, at 50 Hz. The pulses have a fixed amplitude (e.g., 3 to 5 V) and a width that is controlled to be within some range about a neutral width (e.g., 1.5 ms +/- 0.5 ms). The pulse width determines the angle of a rotor.

A Typical Small Servo: Futaba S3003

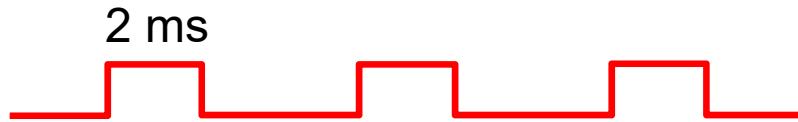


FUTM0031		
Volts	Torque	Speed
4.8V	44 oz-in (3.2 kg-cm)	0.23 sec/60°
6.0V	57 oz-in (4.1 kg-cm)	0.19 sec/60°
Dimensions		Weight
1-9/16 x 13/16 x 1-7/16 in (40 x 20 x 36 mm)		1.3 oz (37 g)
		3P

Figures courtesy of FutabaUSA

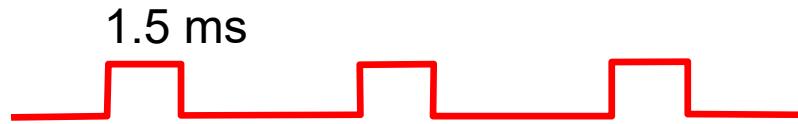
- The voltage given in the table is the supply for the servo motor.
- The given rotational speed is for an unloaded servo. A loaded servo will be slower to rotate in response to PWM width changes.
- The servo motor provides a holding torque that keeps the rotor at the controlled angle, despite any externally applied torque.

PWM Signals for Controlling the Rotor Angle



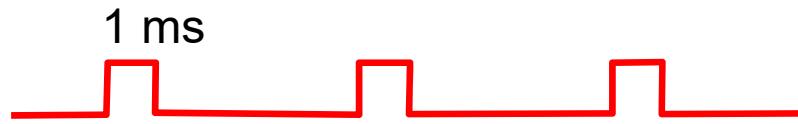
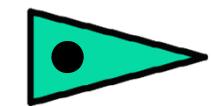
Maximum width/angle

+90 deg.



Neutral width/angle

0 deg.



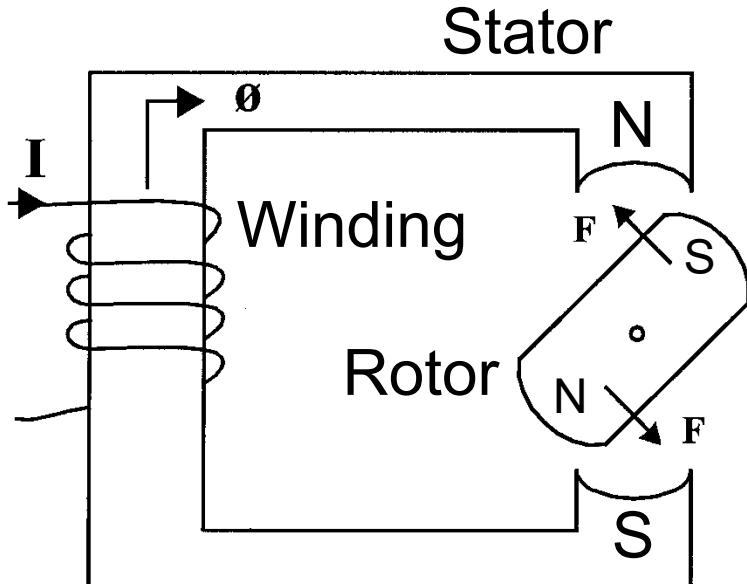
Minimum width/angle

-90 deg.



Note: The duty cycle (i.e., pulse width to pulse period ratio) is not drawn to scale. A 50 Hz pulse rate implies that pulses arrive every 20 ms.

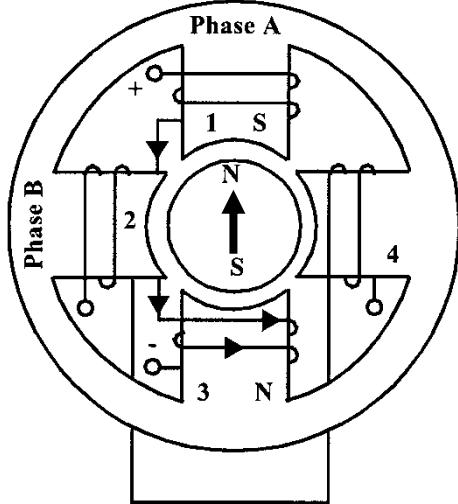
Basic Principles behind the Stepper Motor



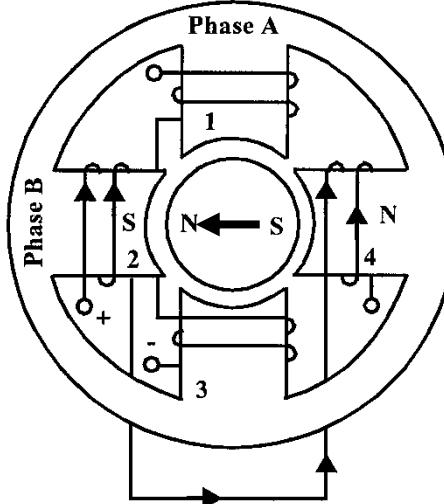
- A rotating **rotor** contains a permanent magnet, which has fixed north (N) and south (S) poles.
- A stationary **stator** can be energized with current to produce an electromagnet. The location of the N and S poles of this magnet are determined by the direction of the current driven through the one or more winding(s).
- The rotor experiences a torque that acts to bring the rotor into alignment with the stator (the N and S poles attract one another). The torque disappears when the stator is fully aligned with the rotor.
- Controlled rotation of the rotor can be obtained by energizing *on* and de-energizing *off* multiple stator windings in the correct sequence.

Simplified View of Stepper Motor Operation

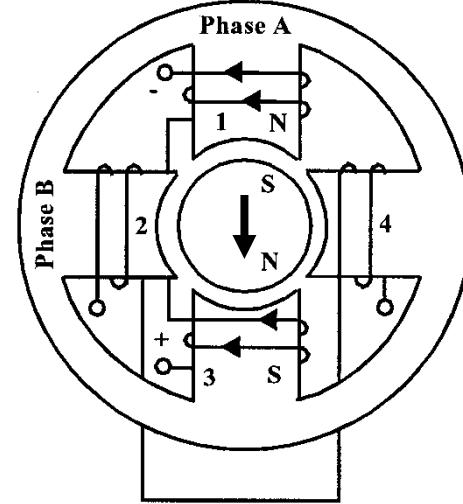
Step #1



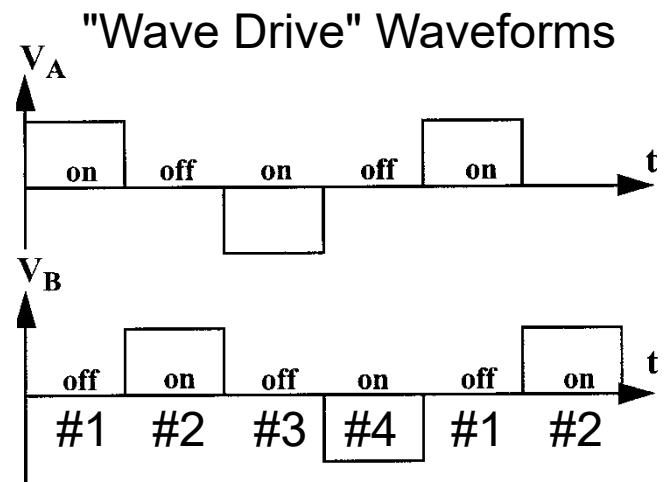
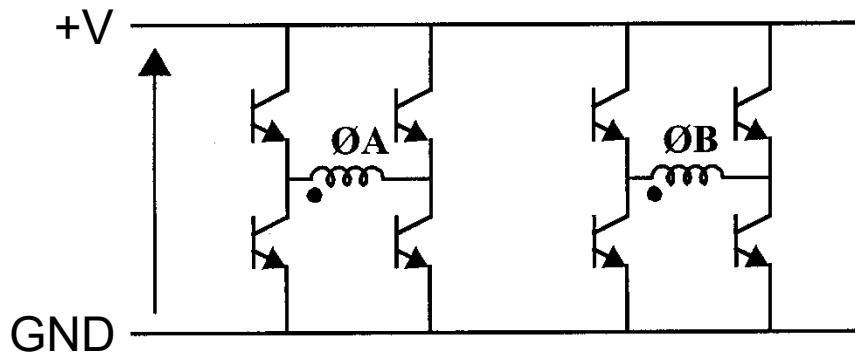
Step #2



Step #3



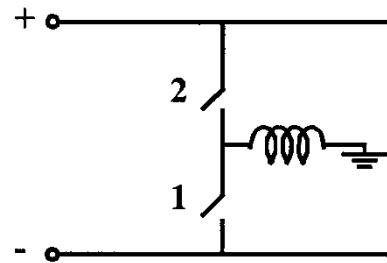
Bipolar Full H-Bridge Circuit



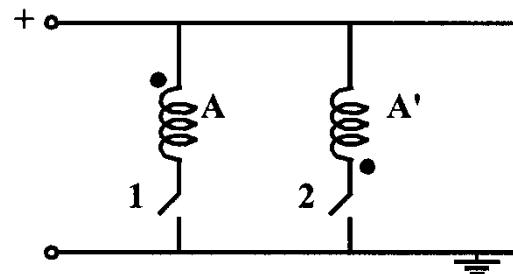
Two Additional Drive Configurations

Bipolar, one winding

(Also called “Half-H Bridge”,
“Half-Bridge” & ‘Totem Pole’)



Unipolar, two windings



Pro: Minimum number of windings.

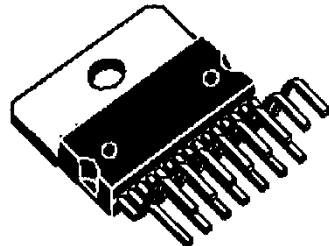
Pro: Simple bridge circuit.

Con: Needs three supply voltages.

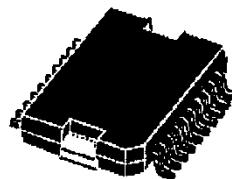
Con: Twice as many windings required on the stator.

Conclusion: The bipolar full H-bridge circuit is more common.

STMicro L298 Dual H-Bridge Driver IC

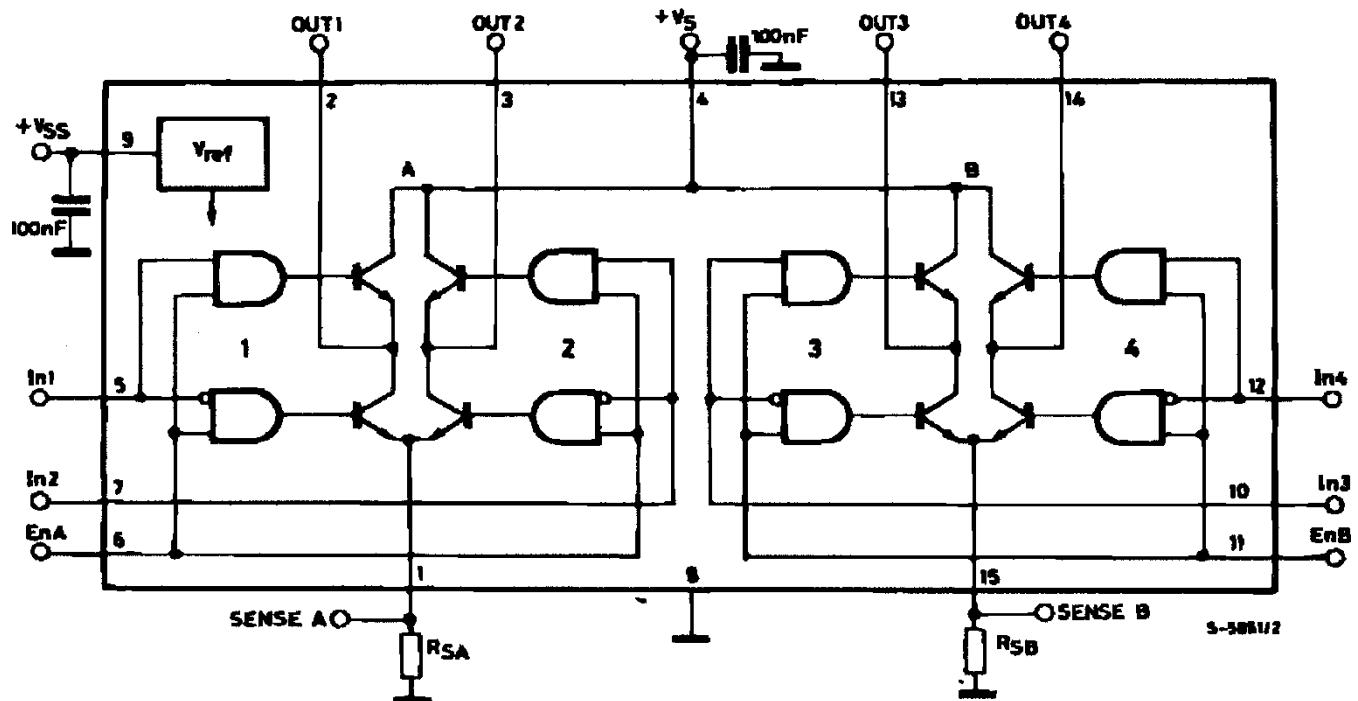


Multiwatt15



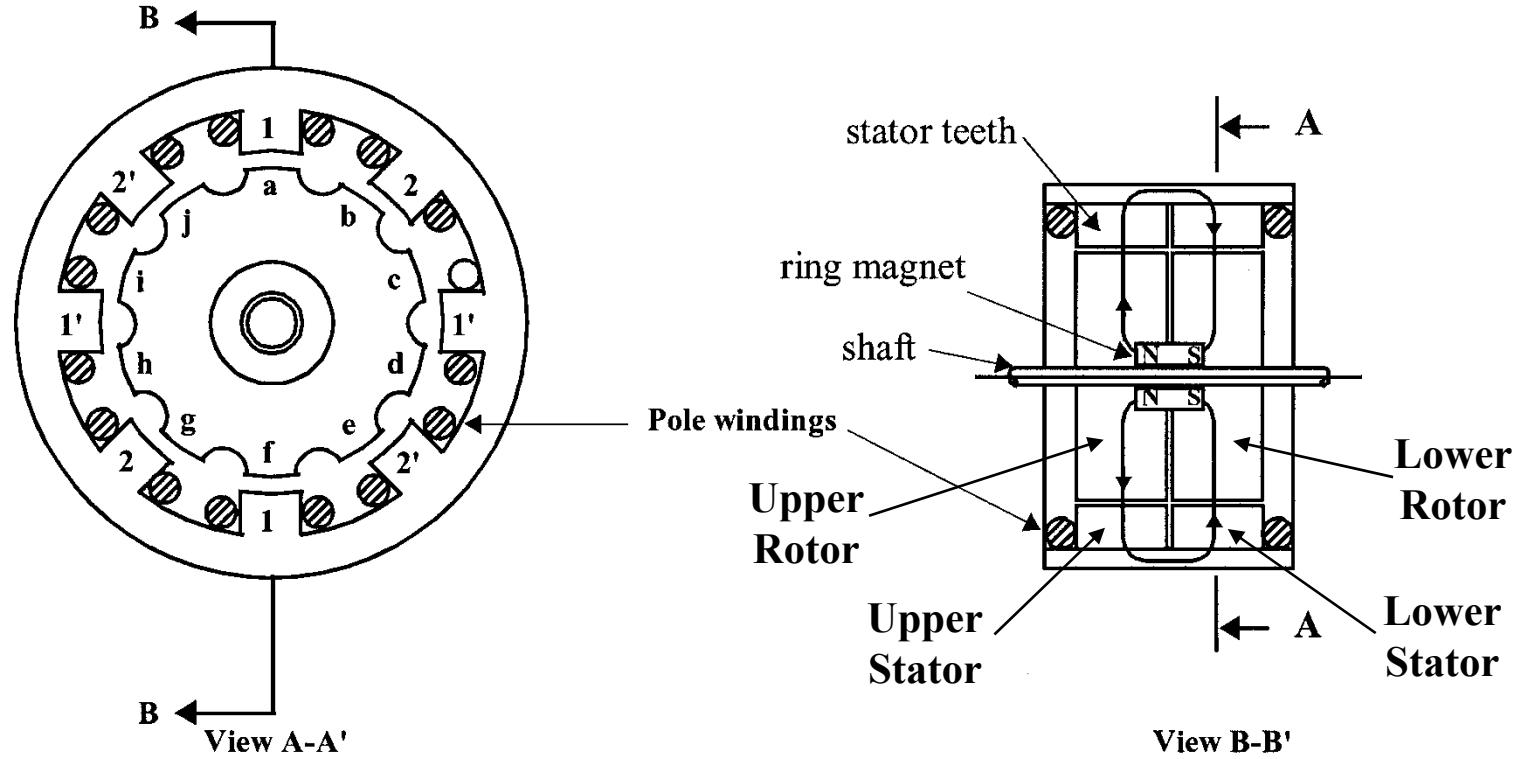
PowerSO20

- TTL-compatible control inputs
- Drives inductive loads at up to 46 V and a total of 4 A DC



Copyright © 2000 STMicroelectronics

Hybrid Stepper Motor with Two Stator Stacks

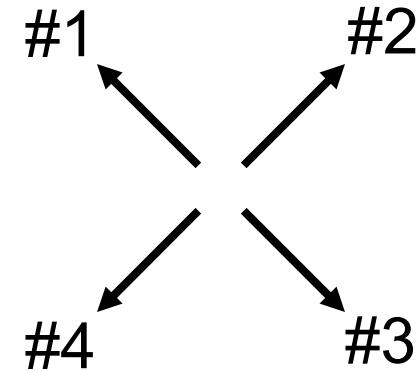
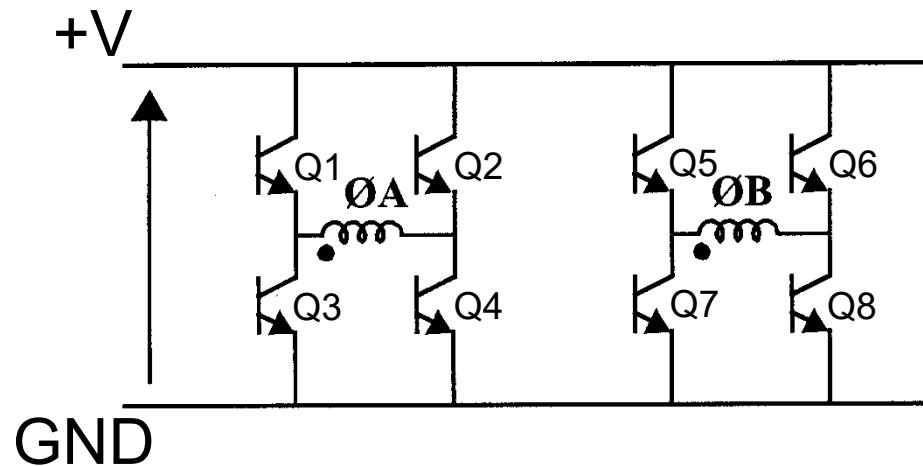


Number of rotor teeth, $n_r = 10$

Number of stator teeth, $n_s = 8$

$$\text{Step angle} = (360/n_s) - (360/n_r) = 45 - 36 = 9 \text{ degrees}$$

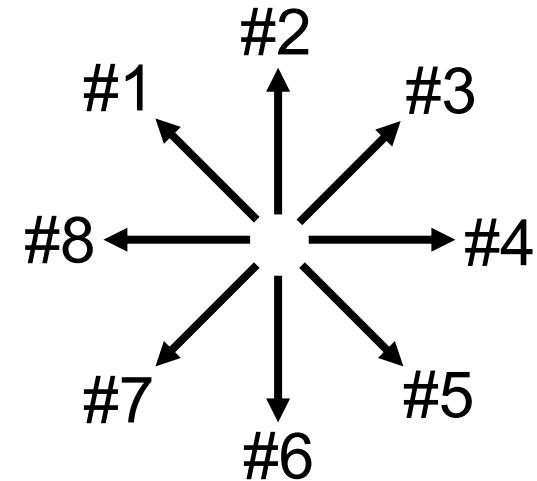
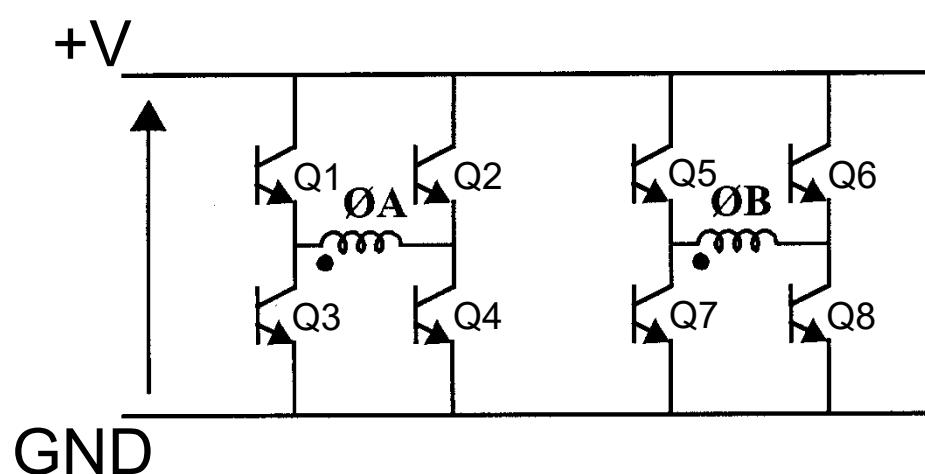
Full-Step Sequence (constant current, bipolar drive)



Step	Q1-Q4	Q2-Q3	Q5-Q8	Q6-Q7	Rotor Angle (mod 360 deg.)
#1	on	off	on	off	315 = -45
#2	on	off	off	on	45 = -45 + 90
#3	off	on	off	on	135 = 45 + 90
#4	off	on	on	off	225 = 135 + 90
#1	on	off	on	off	315 = 225 + 90
#2	on	off	off	on	45 = 405 = 315 + 90

Note: The step angle in real stepper motors is never 90 degrees.
A step angle of 1.8 degrees (200 steps per rev.) is fairly common.

Half-Step Sequence (bipolar drive)

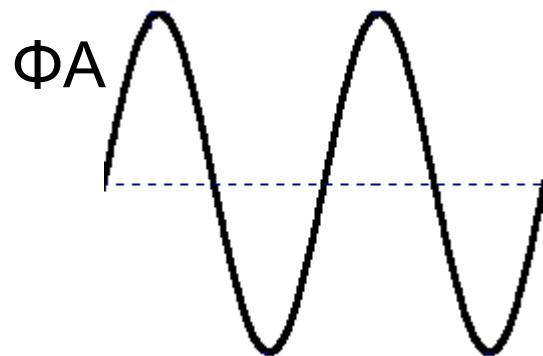


Step	Q1-Q4	Q2-Q3	Q5-Q8	Q6-Q7	Rotor Angle (mod 360 deg.)
#1	on	off	on	off	315 = -45
#2	on	off	off	off	0 = 360 = -45 + 45
#3	on	off	off	on	45 = 0 + 45
#4	off	off	off	on	90 = 45 + 45
#5	off	on	off	on	135 = 90 + 45
#6	off	on	off	off	180 = 135 + 45
#7	off	on	on	off	225 = 180 + 45
#8	off	off	on	off	270 = 225 + 45
#1	on	off	on	off	315 = 270 + 45

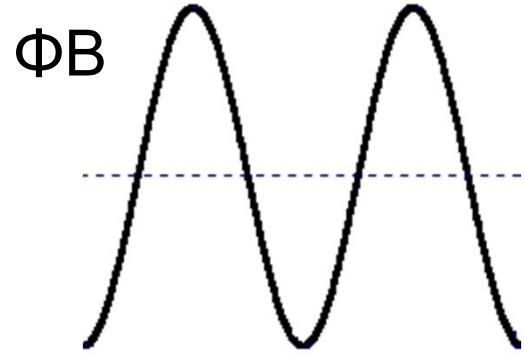
Synchronous, Full-Step and Half-Step Drive Waveforms

Synchronous Drive

Constant torque & current

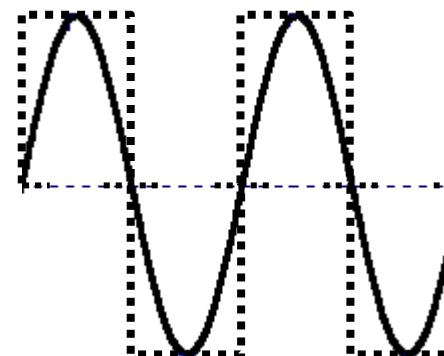


Sinusoids 90 degrees
out of phase



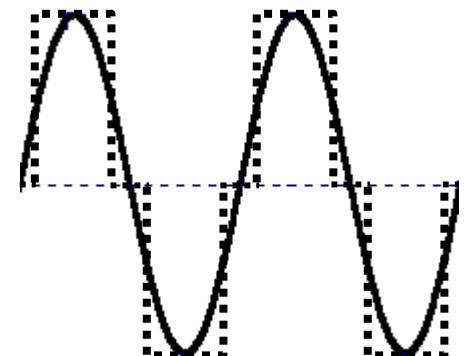
Full-Step Drive

Pulsing torque



Half-Step Drive

Pulsing torque



Stator Drive Waveforms

- The simple wave drive waveform is rarely used. The full-step waveform provides stronger holding torque, with the same number of step positions as wave drive.
- Note: Both the full-step and half-step waveforms are rough approximations to the sinusoidal waveforms used in synchronous motors.
- **Half-step drive** provides twice as many stable rotor positions compared to full-step drive, at the cost of an only slightly more complicated drive waveform.
- **Full-step drive** provides more constant total winding current. (The current is still not steady in a rotating motor because of the effects of the inductance of the stator windings.)
- In **microstepping** waveforms, even finer control of the rotor position is achieved by using a digital-to-analogue (DAC) converter to produce two variable-amplitude drive waveforms that closely approximate sinusoids. Each full step could be split into, say, 256 microsteps.

Practical Stepper Motors

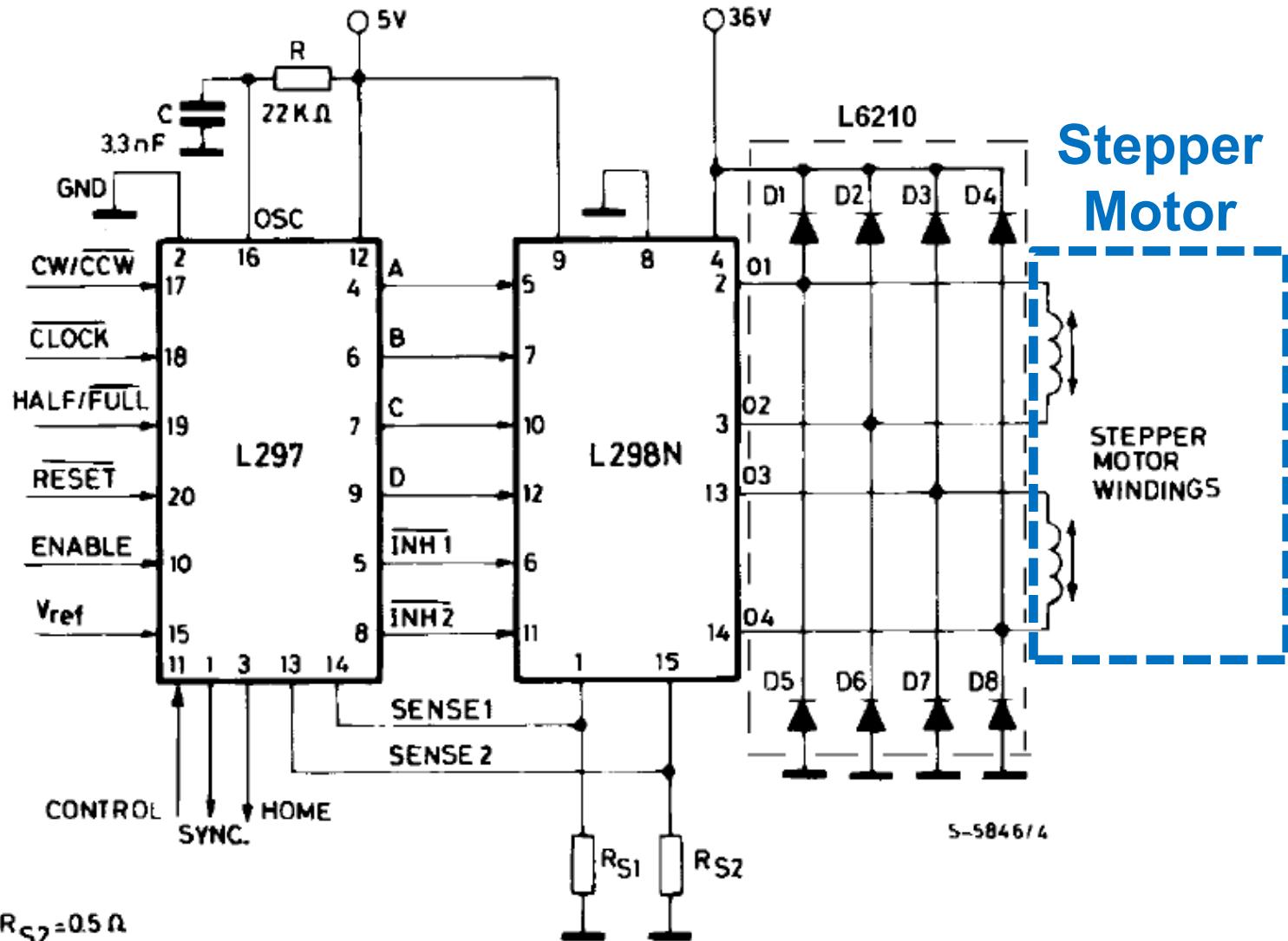
- Real stepper motors come in a variety of configurations:
 - different numbers of phase windings
 - unipolar (one dir.) or bipolar (both dirs.) current feeds
 - different numbers of steps (e.g., 200 steps per 360 deg.)
 - different values of generated mechanical torque
- The bridge circuit controls the switching of current through the windings. The microcontroller must produce transistor control signals with the correct sequence and timing. These control signals may need to be buffered (e.g., to produce sufficiently large base-emitter current).
- The rotational speed must not be changed too suddenly; otherwise, the motor shaft position will slip away from its expected position to another (incorrect) stable position.

STMicro L297 Stepper Motor Controller

- Some microcontroller units (e.g., Freescale MCF5234 and MCF54415) include subsystems that allow complex timing waveforms to be produced.
- The Enhanced Time Processing Unit (eTPU) in the MCF5234 has 16 hardware “channels” that can be associated with software functions that can be used to control stepper motors and many other high-speed timing applications.
- In our laboratory, we will be using the L297 integrated circuit from STMicroelectronics, together with the MCF54415 MCU and an L298 H-bridge driver, to control a stepper motor.

L297 Stepper Motor Controller Connections

From MCU



$$R_{S1}, R_{S2} = 0.5 \Omega$$

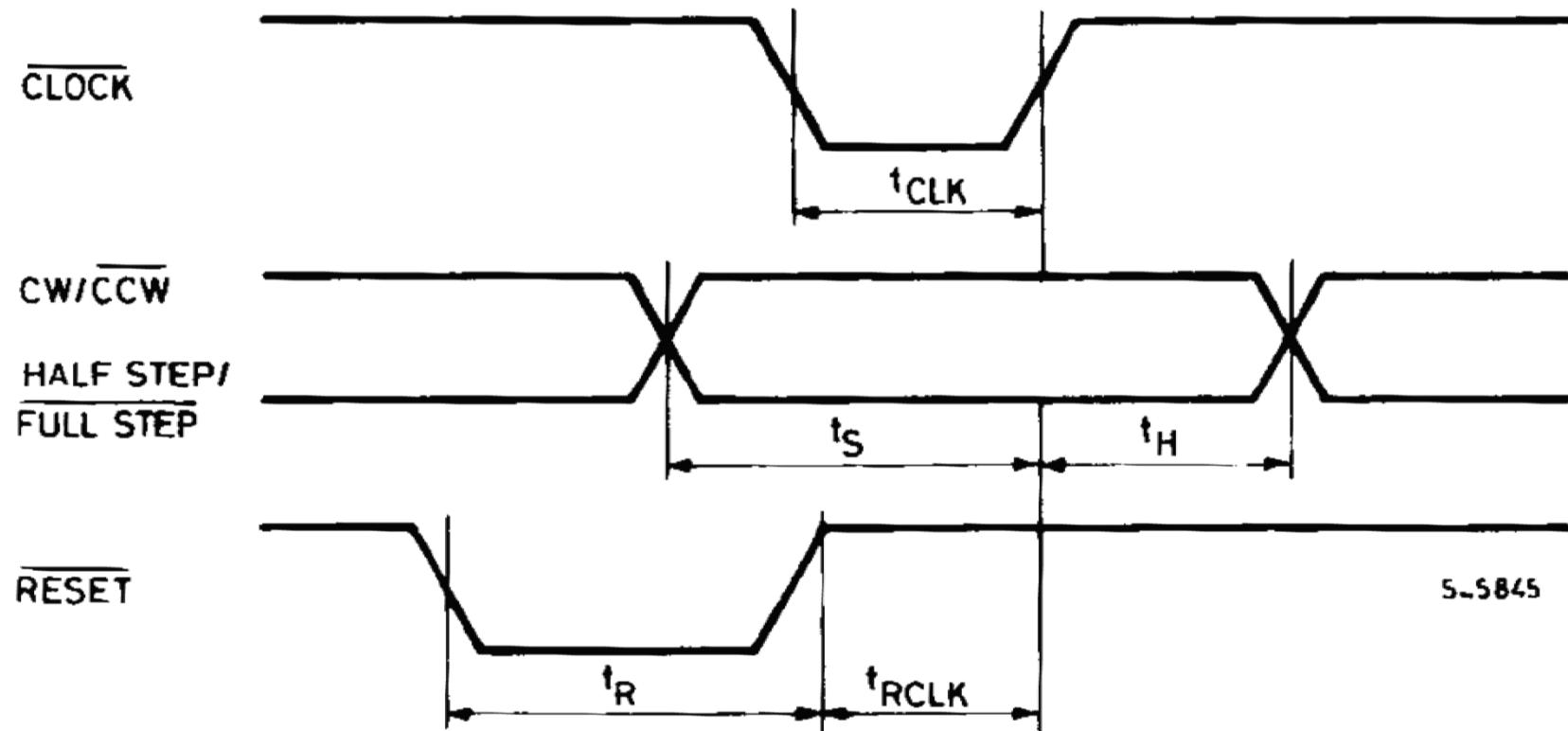
D1 to D8 = 2A FAST DIODES

Copywrite © 2001 STMicroelectronics

STMicro L297 Features

- Generates four-phase drive signals that are suitable for controlling both (a) two-phase bipolar stepper motors, and (b) four-phase unipolar stepper motors.
- The full-step, half-step and wave drive waveforms can be generated without detailed intervention from the micro-controller unit (MCU).
- The MCU must provide the step clock (**CLOCK***), motor direction (**CW/CCW***) and step mode (**HALF/FULL***) input signals.
- An active low pulse on **CLOCK*** advances the motor by one increment on the rising edge of **CLOCK***.
- The active low **RESET*** signal forces the L297 into the output state **ABCD = 0101** (state 1).

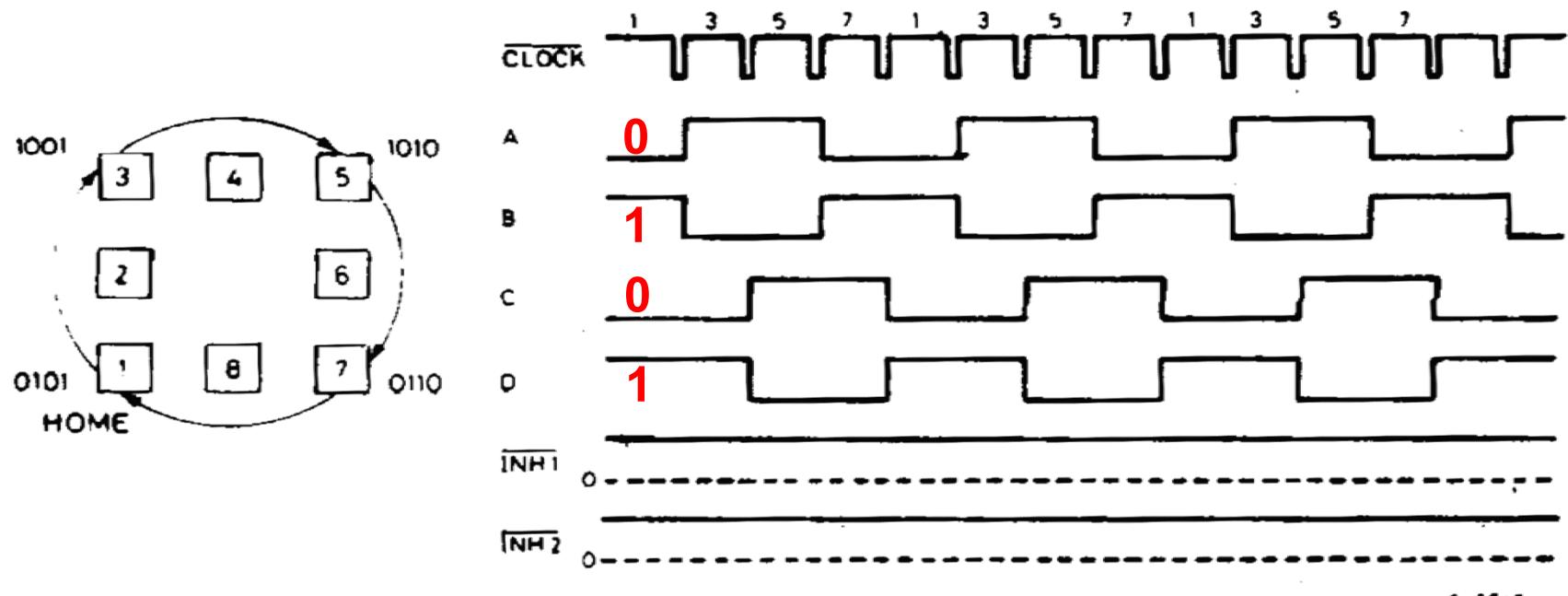
L297 Input Waveforms



Copyright © 2001 STMicroelectronics

L297 Full-step Output Waveforms

Note: HALF/FULL* is low when the state is 1 (ABCD = 0101) or 3 (1001), 5 (1010) or 7 (0110).

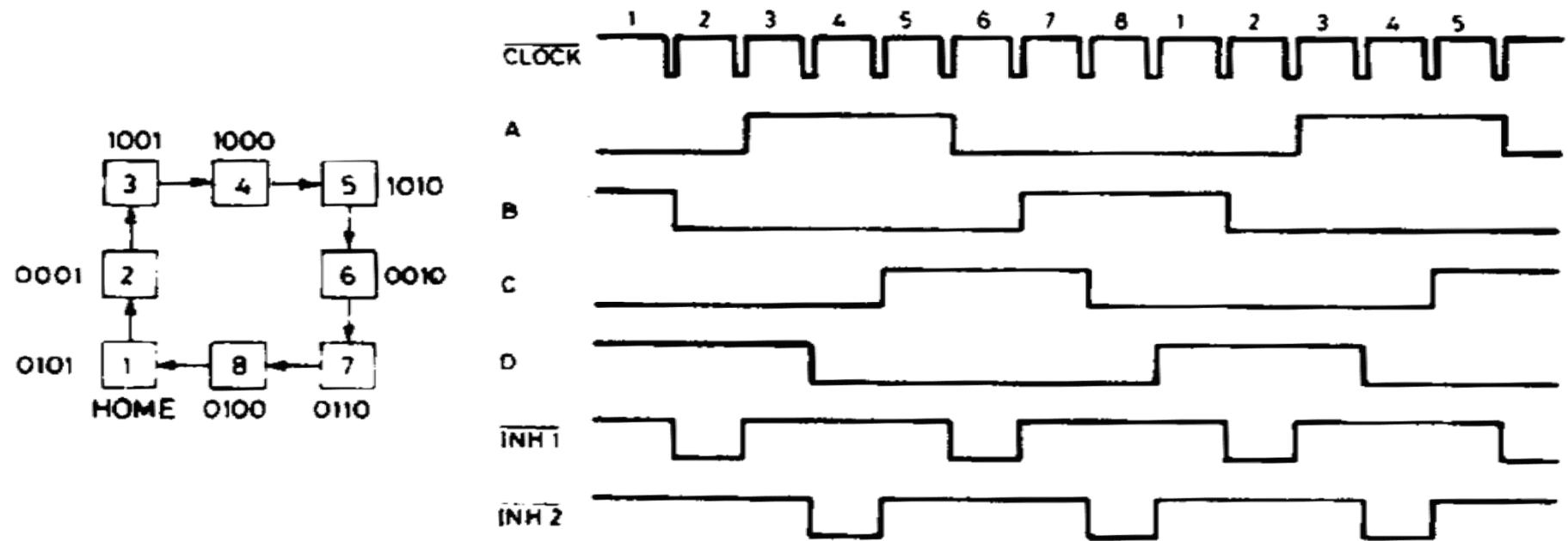


S-5842

Copyright © 2001 STMicroelectronics

L297 Half-step Output Waveforms

Note: HALF/FULL* is held high. The state sequence is 1 (ABCD = 0101), 2 (0001), 3 (1001), 4 (1000), 5 (1010), 6 (0010), 7 (0110) and 8 (0100).

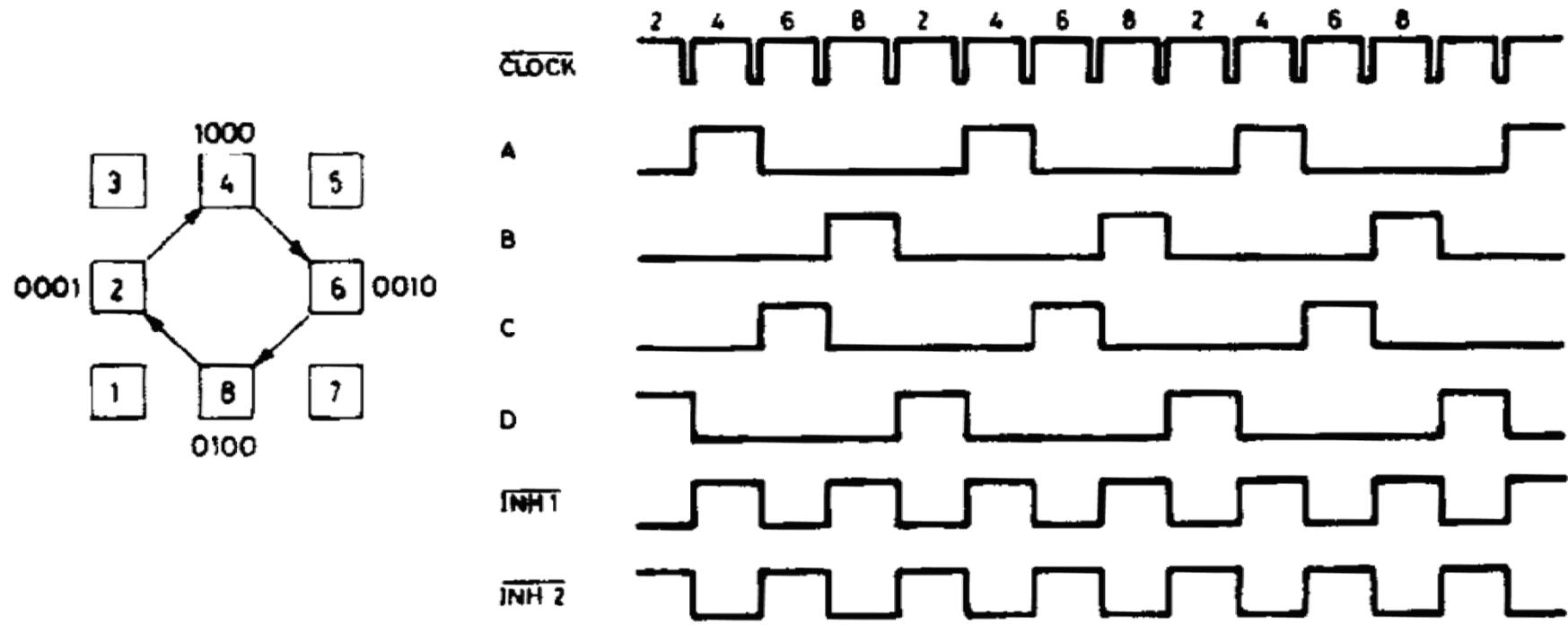


S-5841

Copyright © 2001 STMicroelectronics

L297 Wave Drive Output Waveforms

Note: HALF/FULL* is low when the state is 2 (ABCD = 0001) or 4 (1000), 6 (0010) or 8 (0100).

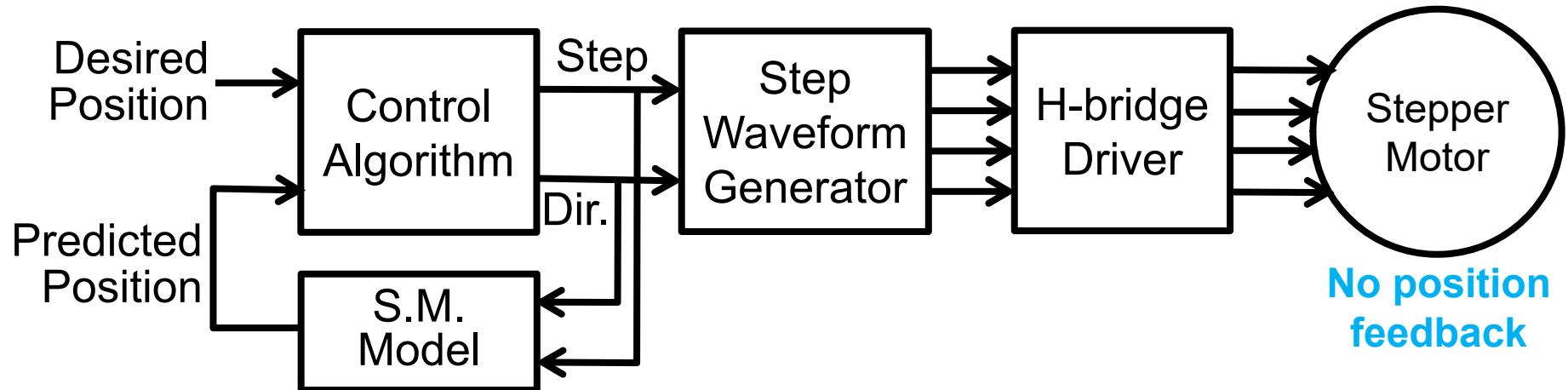


5-5843

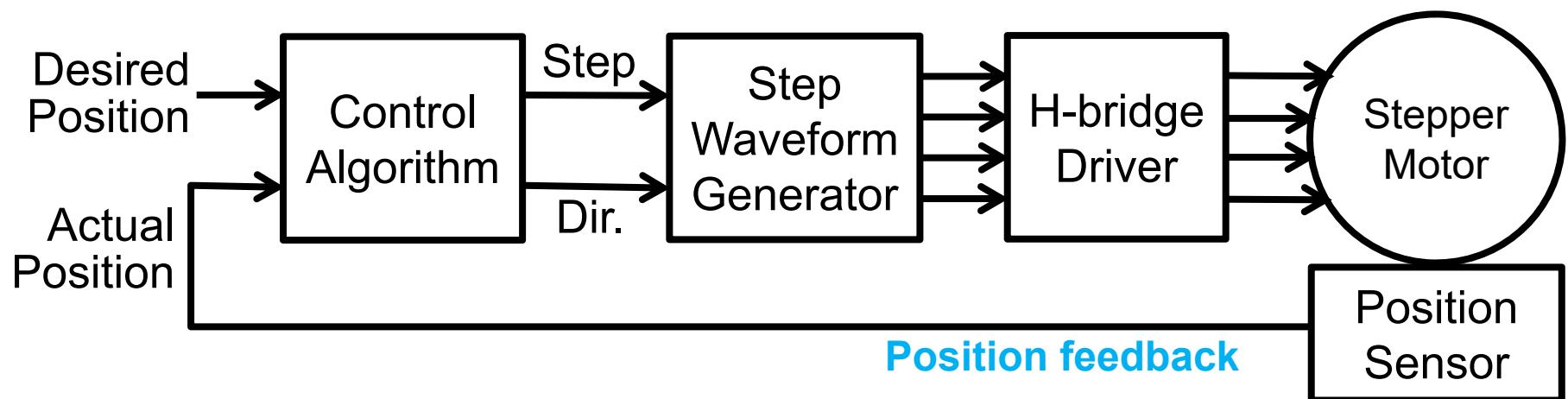
Copywrite © 2001 STMicroelectronics

Open-loop vs. Closed-loop Control (1)

Open-loop Stepper Motor Control Configuration:



Closed-loop Stepper Motor Control Configuration:



Open-loop vs. Closed-loop Control (2)

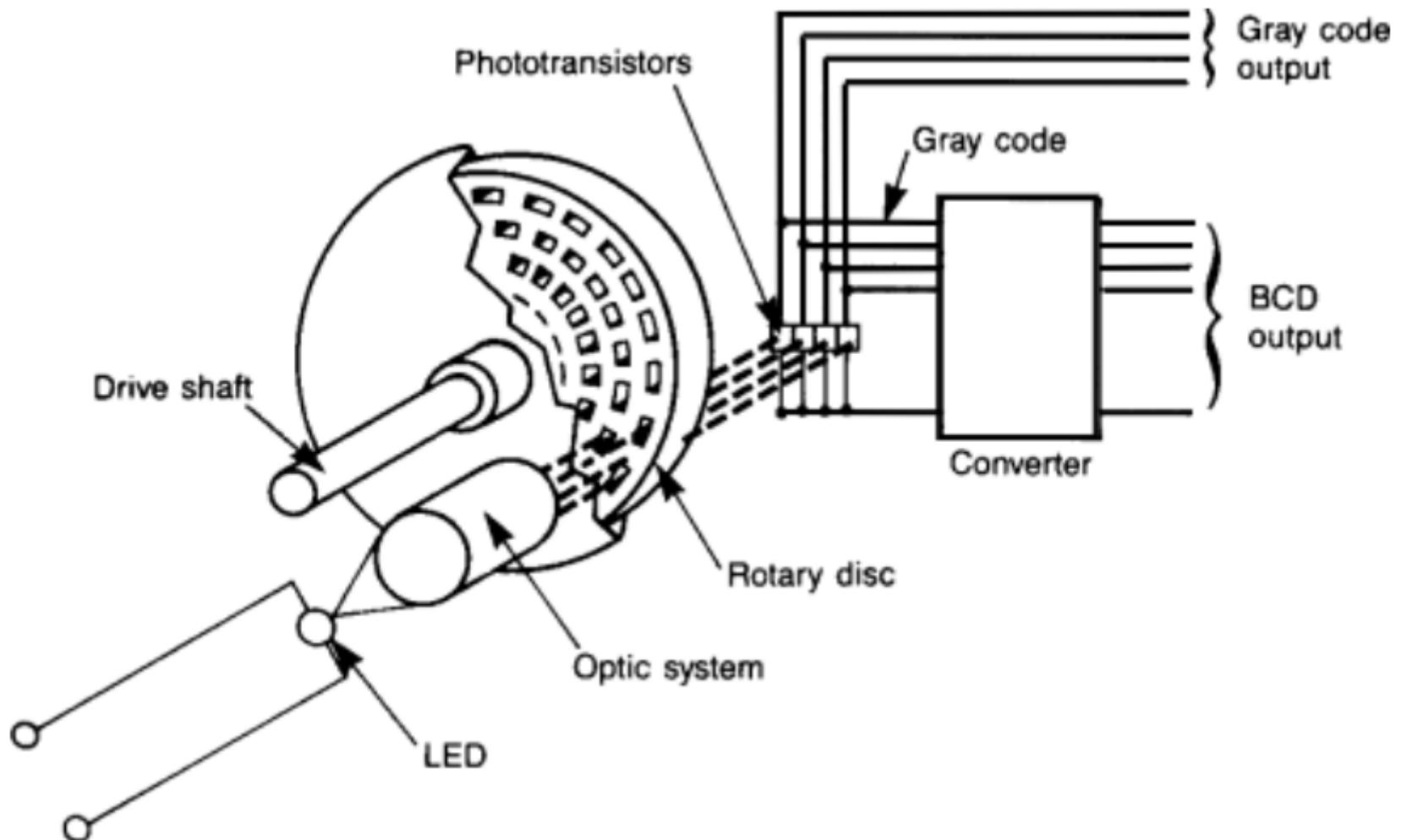
Open-loop Stepper Motor Control Configuration:

- A method is required to put the rotor into the initial position.
- If the rotor fails to keep up with step commands (e.g., the step commands were too fast and the inertia of the rotor and load caused the rotor to slip behind), the actual rotor position might differ from the desired position.
- The control algorithm is blind and cannot make corrections to the motor position to account for slippage.

Closed-loop Stepper Motor Control Configuration:

- A position sensor (e.g., a shaft encoder) at the motor determines the actual rotor position and sends this information as feedback to the control algorithm.
- The control algorithm first computes the difference between the desired position and the *actual position* before it determines what step commands to produce.

Shaft Encoder



From www.plcdev.com

Gray (or Reflected) Code (1)

- The Gray code (invented by Frank Gray in 1947 at Bell Laboratories) is a sequence of binary codewords where adjacent codewords differ in one bit.
- The Gray code is useful in position sensors because, even if the sensor of one bit is slightly slow or fast, the detected codeword is always accurate to within one position.
- The codewords in a Gray code are constructed using an iterative reflection algorithm, starting with 0 and 1:

0, 1

00, 01, 11, 10

000, 001, 011, 010, 110, 111, 101, 100

etc.

Gray (or Reflected) Code (2)

- A Gray code can be used as a pattern that is painted on a disc (or painted on a shaft directly).
- The Gray code words can then be read using a linear array of light sensors (e.g., phototransistors).

#0: 0000000000

#1: 0000000001

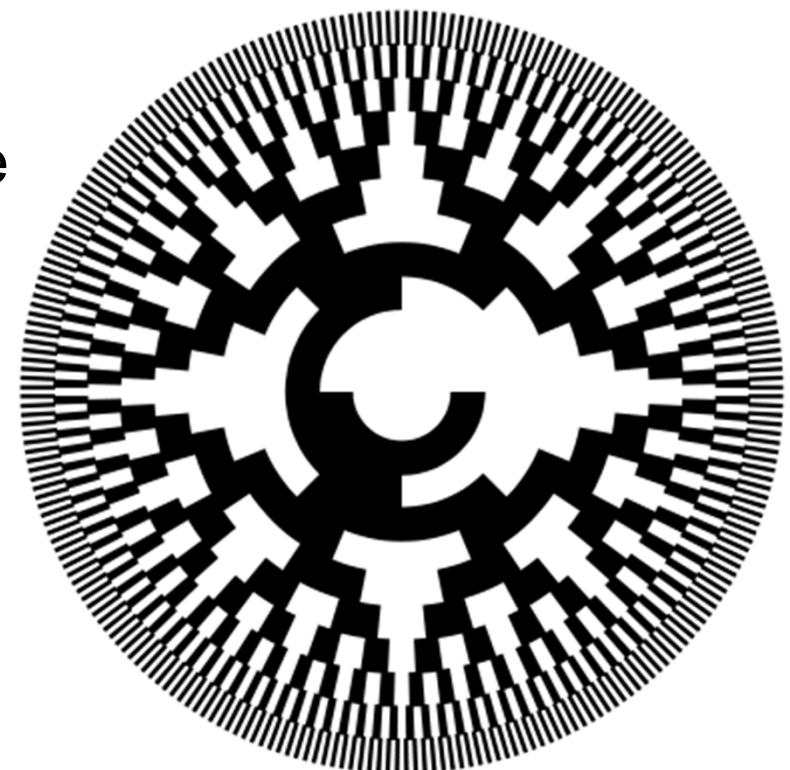
#2: 0000000011

etc.

#1021: 1000000011

#1022: 1000000001

#1023: 1000000000



A Gray code disc
with 1024 positions

Stepper Motor Control Algorithms

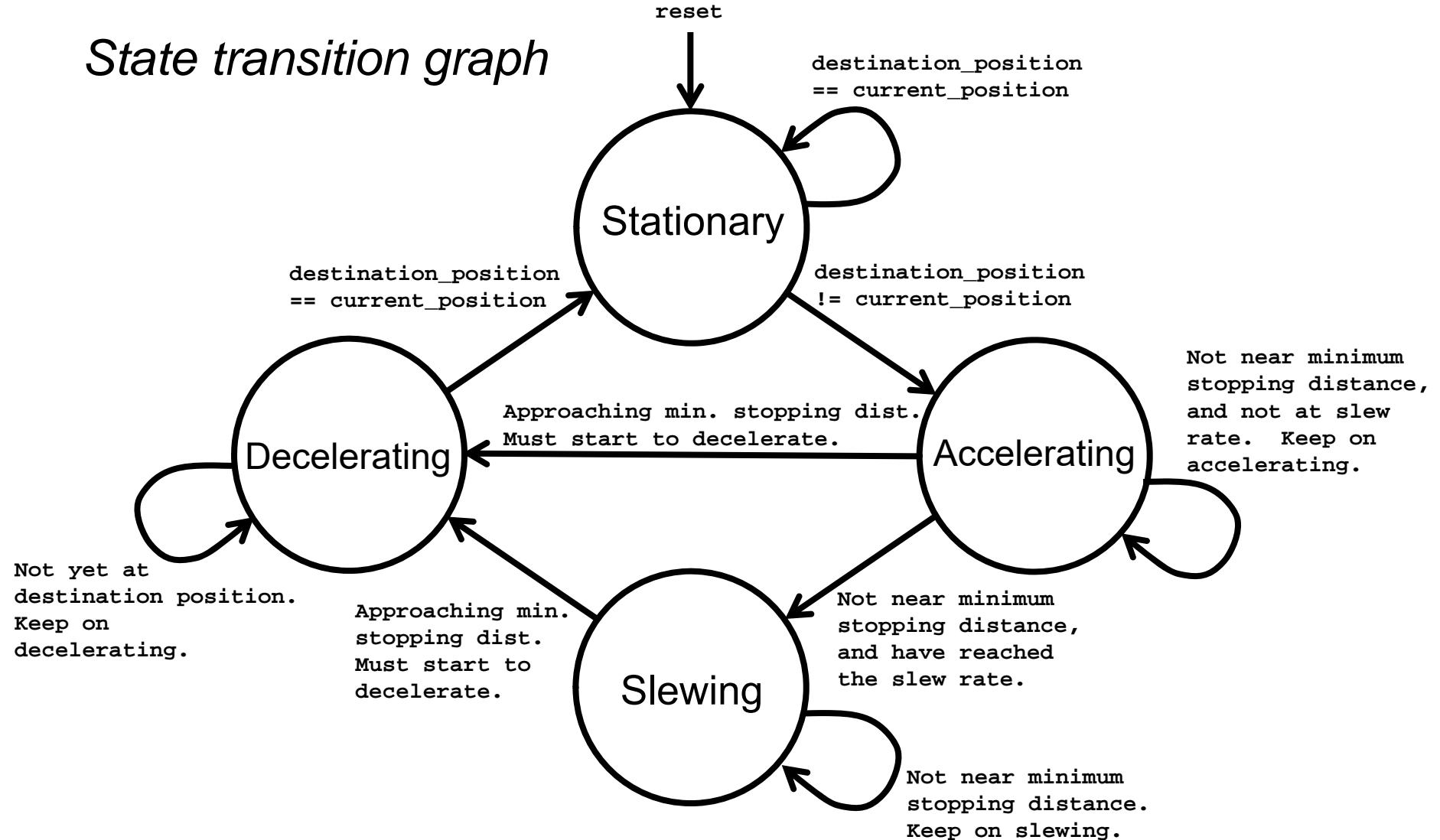
- Modern microcontroller units (MCUs) can implement a wide variety of possible algorithms in software for controlling stepper motors (SMs) using either the open-loop or closed loop configurations.
- A SM control algorithm must take into account practical constraints:
 - The rotor and attached load have *angular inertia* and the motor has a *maximum torque*. The control algorithm should not attempt to exceed the resulting maximum possible angular acceleration.
 - The motor will have a maximum allowed rotational speed (*slew rate*). The controller should not attempt to cause the motor to exceed that maximum speed. This means there is a *maximum allowed step rate*.

Open-loop Stepper Motor Control using the eTPU

- The *Enhanced Time Processing Unit* (eTPU) block in the MCF5234 MCU allows a list of increasingly fast step rates to be defined in advance, ranging from a slowest step rate up to a maximum step rate (the slew rate).
- When the desired position is changed by the CPU, the eTPU determines the direction of movement, and then starts producing step signals at the slowest step rate.
- The step rate is then increased gradually up to the next fastest step rate, and then the next step rate, right up to possibly the fastest step rate.
- When the destination position is approached, the step rates are ramped down in the opposite order until the rotor reaches the desired position at the slowest step rate.
- In the MCF54415, the same controller behaviour can be achieved using software.

Open-loop Stepper Motor Control Algorithm (1)

State transition graph



Open-loop Stepper Motor Control Algorithm (2)

```
present_state = ST_STATIONARY;
present_position = STARTING_POSITION;
step_period = STATIONARY_POLLING_PERIOD;

while (1) {
    switch ( present_state ) {
        case ST_STATIONARY:   do_stationary();
                               break;
        case ST_ACCELERATING: do_accelerating();
                               break;
        case ST_SLEWING:       do_slewing();
                               break;
        case ST_DECELERATING: do_decelerating();
                               break;
    } // end switch

    OS_delay( step_period );
} // end while
```

Open-loop Stepper Motor Control Algorithm (3)

```
void do_stationary()
{
    if ( motor_enabled )
        if ( destination_position == current_position )
            present_state = ST_STATIONARY;
            step_period = STATIONARY_POLLING_PERIOD;
        else
            if ( destination_position > current_position )
                delta_position = +1;
                output_forward_signal_to_motor_driver;
            else
                delta_position = -1;
                output_reverse_signal_to_motor_driver;
            endif;
            present_state = ST_ACCELERATING;
            step_period = MAXIMUM_STEP_PERIOD; // start slowly
        endif;
    else
        present_state = ST_STATIONARY;
        step_period = STATIONARY_POLLING_PERIOD;
    endif;
}
```

Open-loop Stepper Motor Control Algorithm (4)

```
void do_accelerating()
{
    output_step_signal_to_motor_driver;

    current_position = current_position + delta_position;
    distance_remaining = abs(destination_position -
                                current_position);
    look_up min_stopping_distance for step_period;
    if (min_stopping_distance > distance_remaining - MARGIN)
        look_up period_increase for step_period;
    step_period = step_period + period_increase;
    present_state = ST_DECELERATING;

    elseif (step_period <= SLEW_PERIOD + MIN_PERIOD_DECREASE)
        step_period = SLEW_PERIOD;
        present_state = ST_SLEWING;

    else
        look_up period_decrease for step_period;
        step_period = step_period - period_decrease;
        present_state = ST_ACCELERATING;
    endif;
}
```

Open-loop Stepper Motor Control Algorithm (5)

```
void do_slewing()
{
    output_step_signal_to_motor_driver;

    current_position = current_position + delta_position;
    distance_remaining = abs(destination_position -
                               current_position);
    look_up min_stopping_distance from step_period;

    if (min_stopping_distance > distance_remaining - MARGIN)
        look_up period_increase for SLEWING_PERIOD;
        step_period = SLEWING_PERIOD + period_increase;
        present_state = ST_DECELERATING;

    else
        step_period = SLEWING_PERIOD;
        present_state = ST_SLEWING;
    endif;
}
```

Open-loop Stepper Motor Control Algorithm (6)

```
void do_decelerating()
{
    output_step_signal_to_motor_driver;

    current_position = current_position + delta_position;
    distance_remaining = abs(destination_position -
                               current_position);
    look_up min_stopping_distance for step_period;

    if (current_position == destination_position)
        step_period = STATIONARY_POLLING_PERIOD;
        present_state = ST_STATIONARY;

    else
        look_up period_increase for step_period;
        step_period = step_period + period_increase;
        present_state = ST_DECELERATING;
    endif;
}
```

Open-loop Stepper Motor Control Algorithm (7)

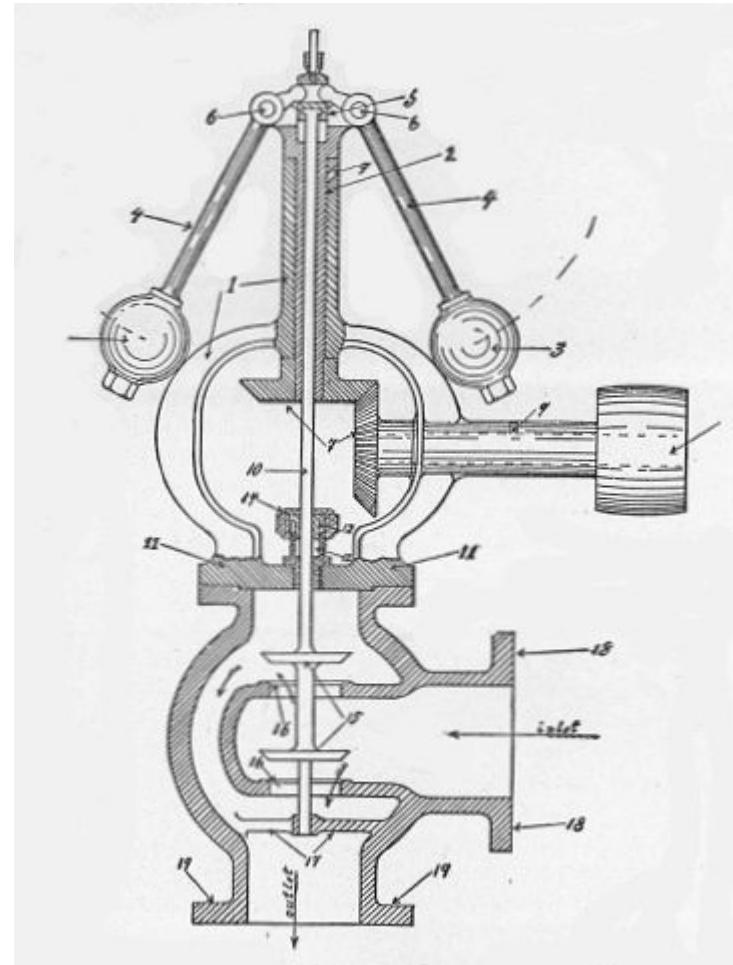
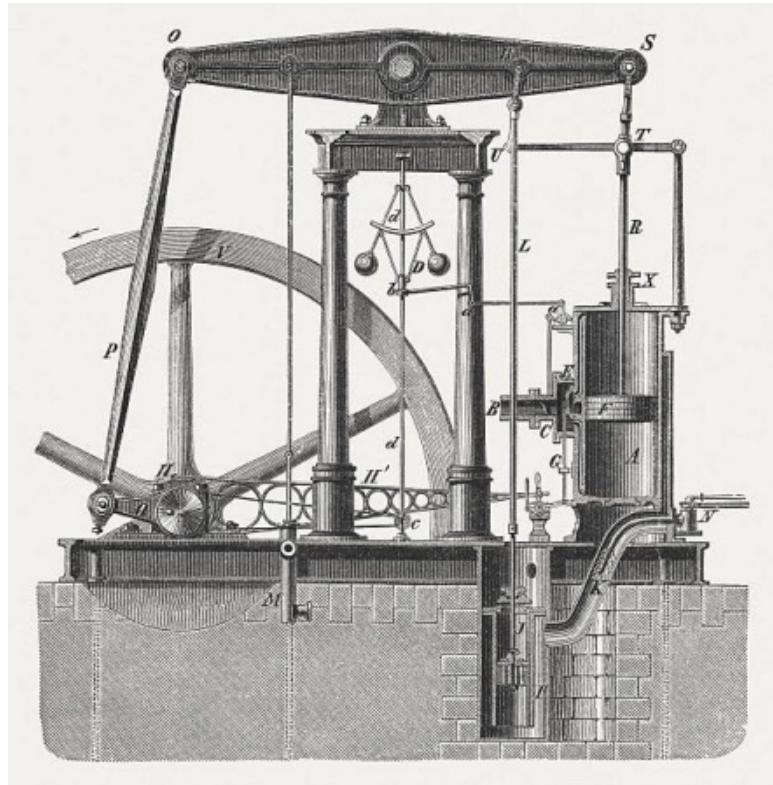
Possible improvements to this SM control algorithm:

- Provide the ability to change the direction of motion by allowing the entry, at any time, of a destination position that differs from the current position in the opposite direction from the current direction of motion.
- Provide an emergency stop sequence.
- Allow the slew rate to be changed.
- Provide position feedback and implement a closed-loop control algorithm.

Early Feedback Control Systems

- Embedded systems are widely used today to implement **feedback control systems**, which are systems that respond to feedback signals and produce control signals that cause a larger engineering system to produce the desired behaviour.
- For example, the Scottish inventor James Watt (1736-1819) made many improvements to Thomas Newcomen's original steam engine. These improvements greatly improved the efficiency and power of the steam engine, which was key to starting the Industrial Revolution in 18th century Great Britain.
- One of these improvements was the **centrifugal governor** (1788), in which a rotating pendulum assembly with two iron weights was used to measure the rotational speed of the engine (a **measured process variable**) to provide feedback to a valve (a **control signal**) that regulated the flow of steam and thus regulated the engine speed.
- The speed of the engine could thus be automatically kept near to the desired speed (the **set point**) for the steam engine.

James Watt's Centrifugal Governor



From N. Hawkins, *New Catechism of the Steam Engine*, Theo Audel, New York, 1904.

Feedback Control Theory

- Feedback control theory is the branch of applied mathematics that deals with modeling the behaviour of systems with feedback and the design of control algorithms that cause a controller to produce control signals that induce the system to have a desired behaviour.
- Typical desired system behaviours:
 - Stable and efficient operation at a selected set point (e.g., speed, input rate, output rate) despite the presence of perturbations.
 - Fast and smooth transition from one set point to a new set point.
- In 1868 James Clerk Maxwell wrote a classic paper (“On Governors”, *Proc. Royal Society of London*, vol. 16, pp. 270-283) that provided a mathematical framework for the centrifugal governor and other control mechanisms that had been developed to control machinery in the first 100 years of the Industrial Revolution.
- In the early 20th century, the problem of steering large boats was influential in the development of modern PID control theory.

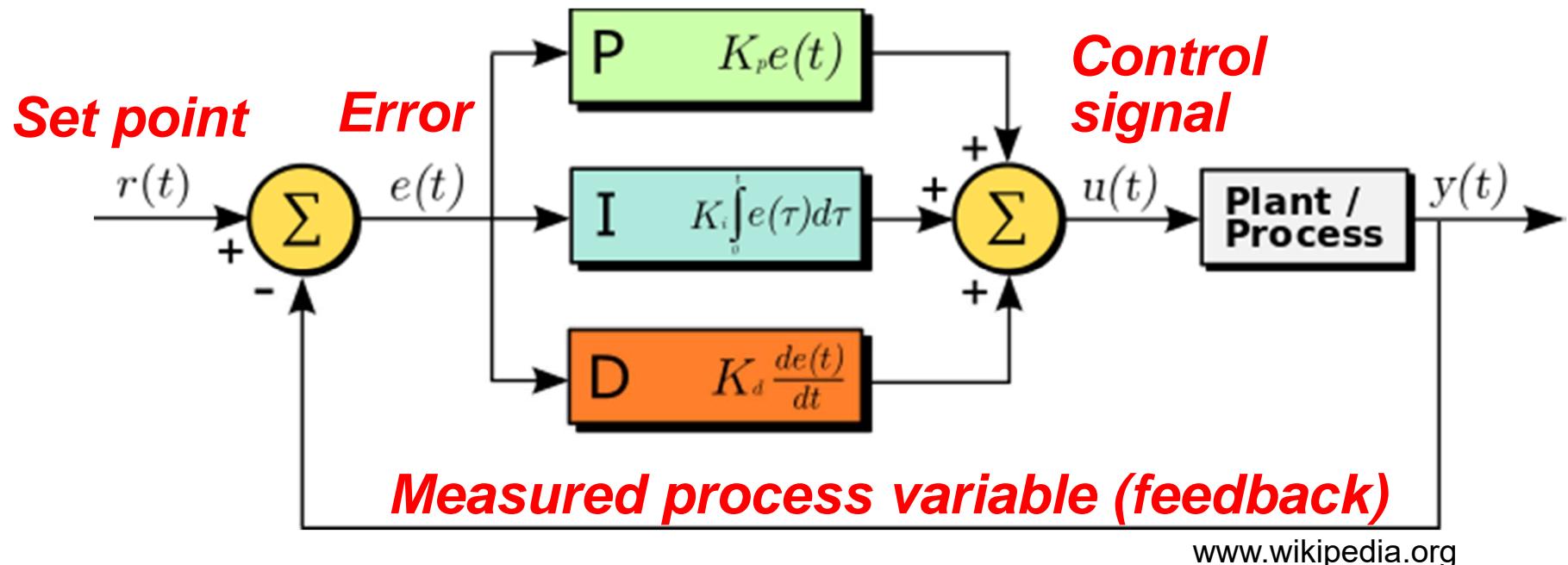
A Control Example: Steering a Large Boat

Consider the problem of using the rudder to change the direction of motion of a boat from one direction (a straight line or an arc) to a new direction (a straight line or an arc):

- **Proportional term:** When the boat is moving in the new direction (the new steady state), the rudder's position is in direct proportion to the new direction (that it, either go straight, turning left, or turning right). A sharper turn requires a greater deviation in the rudder's position.
- **Integral term:** However, the boat is sluggish (slow to respond to a new rudder position) because of its heavy weight (that is, its rotational inertia). As the boat slowly starts to turn, an error builds up between the desired new boat direction and the actual boat direction. The rudder should be oversteered to overcome this error in the boat's direction.
- **Derivative term:** If the boat starts to turn too fast, then a future error can be predicted to build up between the actual position of the boat and the desired position. The rudder should be understeered a bit to slow down the boat's rotation so that the predicted error is avoided.

Proportional-Integral-Derivative (PID) Controllers

- PID Controllers are widely used in industrial controller applications when there is a closed-loop configuration.



Proportional term: overcomes steady-state **present error**

Integral term: overcomes accumulated **past errors**

Derivative term: avoids predicted **future errors**

The PID Controller Equation

$$u(t) = MV(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

- $u(t)$ is the ***control signal***, or manipulated variable, that is to be computed. For a stepper motor, $u(t)$ is the step rate and the rotational direction (i.e., the sign of the step rate).
- $e(t)$ is the ***error signal***, which is computed as the difference of the desired ***set point*** $r(t)$ and the ***measured process variable*** $y(t)$. For a stepper motor, $r(t)$ is an angular position that is increasing (or decreasing) at a constant rate if the motor is to be rotating at a constant speed.
- K_p is the ***proportional gain***, which is a first tuning parameter.
- K_i is the ***integral gain***, which is a second tuning parameter.
- K_d is the ***differential gain***, which is a third tuning parameter.

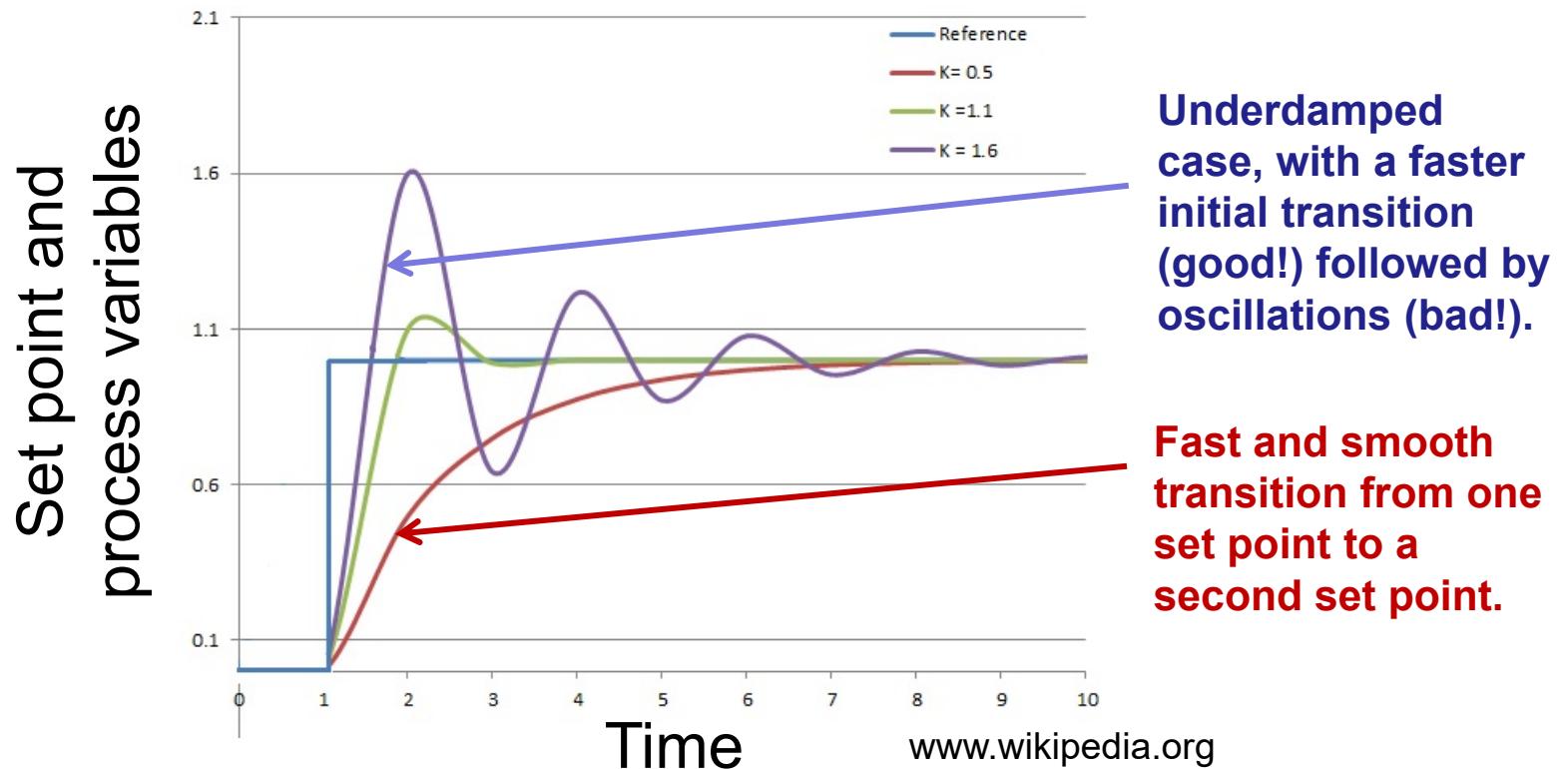
PID Controller Tuning (1)

$$u(t) = \text{MV}(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

- PID controller theory is a complex topic that is the subject of many books, papers and entire courses in control theory.
- For a shorter summary consult: K. H. Ang, G.C.Y. Chong and Y. Li, “PID control system analysis, design, and technology,” *IEEE Transactions on Control Systems Technology*, July 2005, vol. 13, no. 4, pp. 559-576.
- The three parameters K_p , K_i and K_d must be tuned to produce acceptable controller performance. We want to avoid large errors and oscillations while still having quick response to new set points. A common strategy is to adjust K_p first, then K_i , and then finally K_d in that order.

PID Controller Tuning (2)

- The figure below shows the change in the measured process variable $y(t)$ when the reference set point $r(t)$ is suddenly changed, for three different values of K_p and with K_i and K_d held at constant values.



Using a PID Controller for a Stepper Motor

- Any stepper motor control algorithm must take into account practical and safety limits, such as maximum allowed acceleration and deceleration, and maximum allowed speed.
- The basic PID control algorithm must be modified to incorporate these constraints when controlling SMs.
- Closed loop SM controllers will often use a simpler algorithm than a PID controller.
- More complex control scenarios (e.g., control of high-speed robot arms, jet aircraft) will benefit from the superior performance of tuned PID controllers.

Synchronization and Flow Control

A Wide Range of Possible I/O Rates

Device	Behaviour	Partner	Data Rate (KB/s)
Keyboard	Input	Human	0.01
Mouse / tablet	Input	Human	0.02
Voice Input	Input	Human	0.02
Scanner	Input	Human	200.00
Voice Output	Output	Human	0.60
Line Printer	Output	Human	1.00
Laser Printer	Output	Human	100.00
CRT	Output	Human	30,000.00
Network Terminal	I/O	Machine	0.05
Network LAN	I/O	Machine	200.00
Floppy Disc	Storage	Machine	50.00
CD-ROM	Storage	Machine	500.00
Magnetic Tape	Storage	Machine	2000.00
Hard Disc	Storage	Machine	2000.00

Source: Computer Organization & Design: The Hardware/Software Interface,
by David A. Patterson and John L. Hennessy, (Morgan Kaufmann, 1994), p. 513.

Mismatch between CPU and I/O Devices

Observation: A ColdFire clocked at 150 MHz executes roughly 50 million instructions per second.

Example #1: Keyboard input

$$\begin{aligned}\text{maximum data rate} &= 0.01 \text{ KB/s} = 10 \text{ bytes / sec} \\ &= 10 \text{ ASCII characters / second}\end{aligned}$$

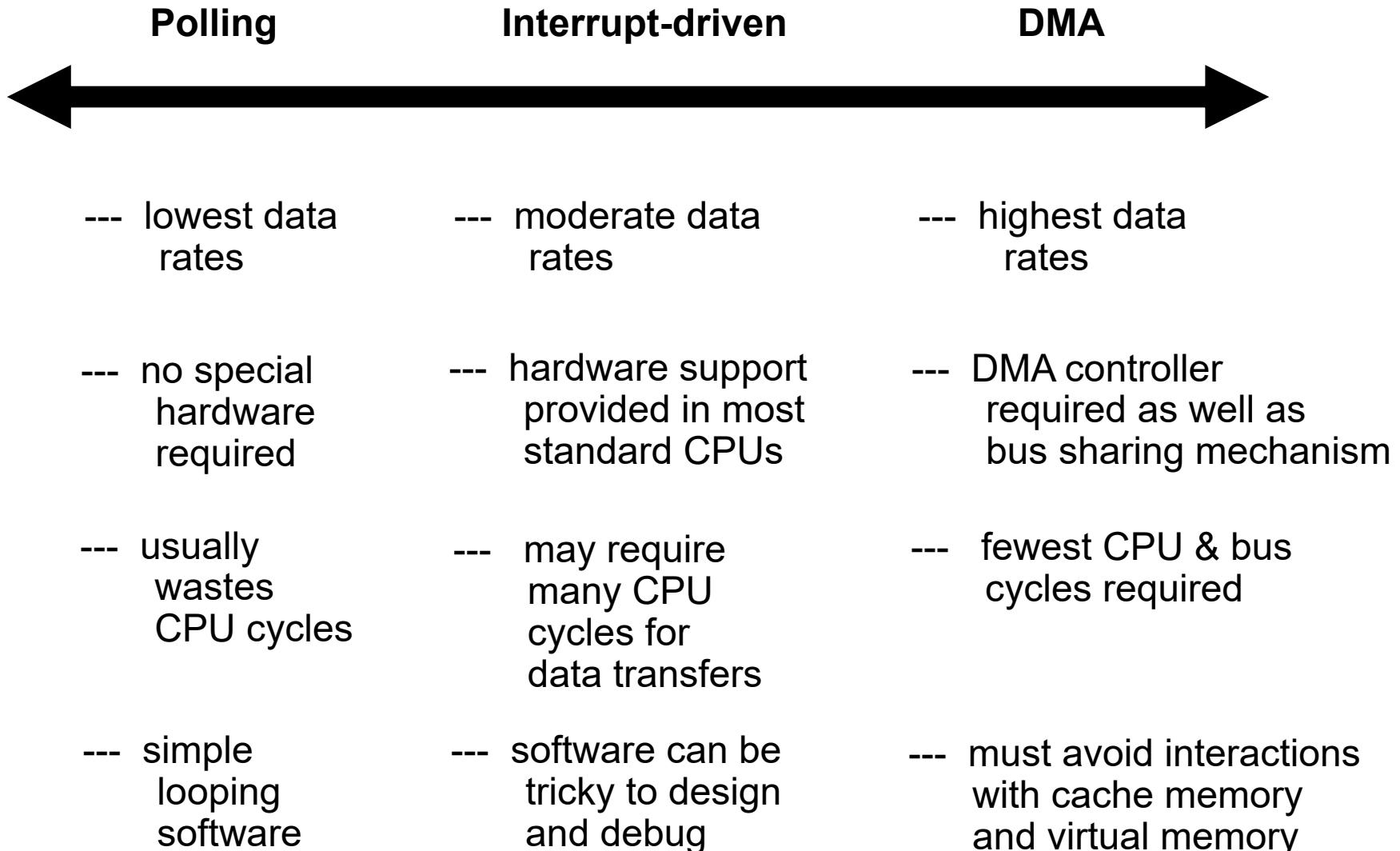
If a CPU takes roughly 100 instructions to process a character, then the CPU has the capacity to process keystrokes 500,000 times faster than they can be entered by a human typist.

Example #2: CRT (Video) Output

$$\text{data rate} = 30,000 \text{ KB/s} = 15 \text{ million 16-bit words / sec}$$

A 150-MHz ColdFire can move blocks of data at a maximum rate of about 75 million longwords per second (150 Mwords/sec).

Spectrum of I/O Data Transfer Methods



Examples of Polling

Determine the "polling overhead" for three different I/O devices. Assume that 100 clock cycles are required to perform one polling operation, and that the CPU executes using a 150-MHz clock. Determine the fraction of CPU time consumed for the following three cases, where it is a requirement that no data be lost.

- 1) The mouse input must be polled at least 30 times per second to smoothly track user movements.
- 2) The floppy disc transfers data to the CPU in 16-bit words and the data rate is 500 KB/s.
- 3) The hard disc transfers data to the CPU in 512-byte bursts at an average data rate of 20 MB/s. Assume that two clock cycles are required to transfer one longword.

Example of Polling (cont'd)

1) Mouse

Number of clock cycles required to poll the mouse

$$= 30 \times 100$$

$$= 3000 \text{ cycles per second}$$

Thus, fraction of CPU cycles consumed by the mouse

$$= 3000 / (150 \times 10^6)$$

$$= 0.002 \%$$

Conclusion: Impact of mouse polling on CPU performance is insignificant.

Example of Polling (cont'd)

2) Floppy Disc

Number of clock cycles per second used up polling
the floppy disc controller

$$\begin{aligned} &= 0.5 \times (500 \times 2^{10}) \times 100 \\ &= 25.6 \times 10^6 \text{ cycles per second} \end{aligned}$$

Thus, fraction of CPU time consumed

$$\begin{aligned} &= \frac{25.6 \times 10^6}{150 \times 10^6} \\ &= 17.1 \% \end{aligned}$$

Conclusion: Impact of polling on CPU performance is significant. Transferring individual words by polling is very inefficient. Consider using a DMA controller.

Example of Polling (cont'd)

3) Hard disc

Number of clock cycles per second used up transferring bytes and polling the hard disc controller

$$\begin{aligned} &= ((20 \times 2^{20}) \times 2 / 4) + ((20 \times 2^{20}) \times 100 / 512) \\ &= 10,485,650 + 4,096,000 \text{ cycles per second} \end{aligned}$$

Thus, fraction of CPU time consumed

$$\begin{aligned} &= \frac{14,581,650}{150 \times 10^6} \\ &= 9.7 \% \end{aligned}$$

Conclusion: Impact of polling on CPU performance is significant. Direct Memory Access (DMA) should be considered to speed up the block transfers.

Interrupt Management

- An interrupt-handling mechanism is a standard feature on all modern microprocessors.
- The interface designer must make several important design decisions with respect to interrupt handling:
 - How many different independent interrupt sources?
 - How should the possible interrupts be prioritized?
 - Will the interrupts will be “autovectored” or “user vectored”? Or are the exception vectors for interrupts stored in an Interrupt Controller Module (INTC)? The MCF54415 has three INTCs that manage interrupts from 173 different possible interrupt sources.
 - How will the interrupt vectors be updated when device drivers are installed or changed?

Direct Memory Access (DMA)

- A suitable method when large quantities of data need to be transferred between an I/O device and main memory
- A *DMA controller* generates bus signals that cause data blocks to be transferred over the bus without the CPU. Multiple CPU "MOVE" instructions are thus avoided.
- Interrupts are used in DMA to signal important events:
 - initial call to device driver to set up the DMA transfer
e.g., source and/or destination addresses, byte count
 - IRQ to signal that an error has occurred during DMA
 - IRQ to signal the end of a DMA transfer
- A digital system may contain more than one controller capable of performing DMA. However, using more than one DMA controller at a time will produce declining benefits since they will all be sharing the one system bus.

Direct Memory Access Controllers (DMACs)

- A DMAC is a special-purpose processor for performing Direct Memory Access operations over the shared system bus.
- The data transfer algorithm in a DMAC is hard-coded—it behaves like one *block move instruction* that avoids a large number of fetch-decode-execute cycles for many CPU moves.
- The details of the DMA block move can be configured by the CPU by loading control registers in the DMAC.
- The DMAC can produce interrupts for selected events, such as when the block move completes successfully, or when the block move encounters errors.
- DMACs can exist as separate modules in a microcomputer, or they can be incorporated as part of peripheral modules (e.g., network interfaces, video controllers, memory controllers, etc.)

Potential Complications with DMA

Observation: When a digital system contains a CPU and other controllers capable of performing DMA, then there are two or more processors accessing the memory:

Potential Problems:

1) *CPU with Cache Memory*

- Some words in physical memory may be out of date (the new data is only in the cache).
- If DMA updates memory words, then some lines in the CPU's cache may need to be marked invalid.

2) *Virtual Memory*

- Memory locations have both virtual and physical addresses. Which addresses should DMA use?
- At page boundaries in physical memory, adjacent physical addresses are not necessarily adjacent virtual addresses (i.e., addresses used by CPU S/W).

Options for Solving these Complications

- 1) *Provide DMA controller with address translation H/W.*
 - CPU and DMAC use the same virtual addresses.
 - Both CPU and DMAC cause page faults when virtual addresses are used that belong to pages that are not already loaded into physical memory.
- 2) *Do DMA only within page boundaries in physical RAM.*
 - CPU uses virtual addresses with address translation.
 - DMAC uses physical addresses without translation.
 - Operating system must refrain from moving affected pages during the DMA transfer.

Data Structures Used in I/O Software

- 1) Buffers
- 2) Double Buffers
- 3) FILO Stacks
- 4) FIFO Queues
- 5) Circular Buffers
- 6) Semaphores, flags, etc.

1) Buffers

- It is usually more efficient to process larger blocks of data rather than individual words.
- Why? The overhead per block is often not that different from the overhead per byte or per word.

A *buffer* is the term used to refer to the following:

- 1) a block of data that is processed together
- 2) a region in main memory used to hold such a block of data
- 3) a dedicated memory device, separate from main memory, that is used to store such a block of data

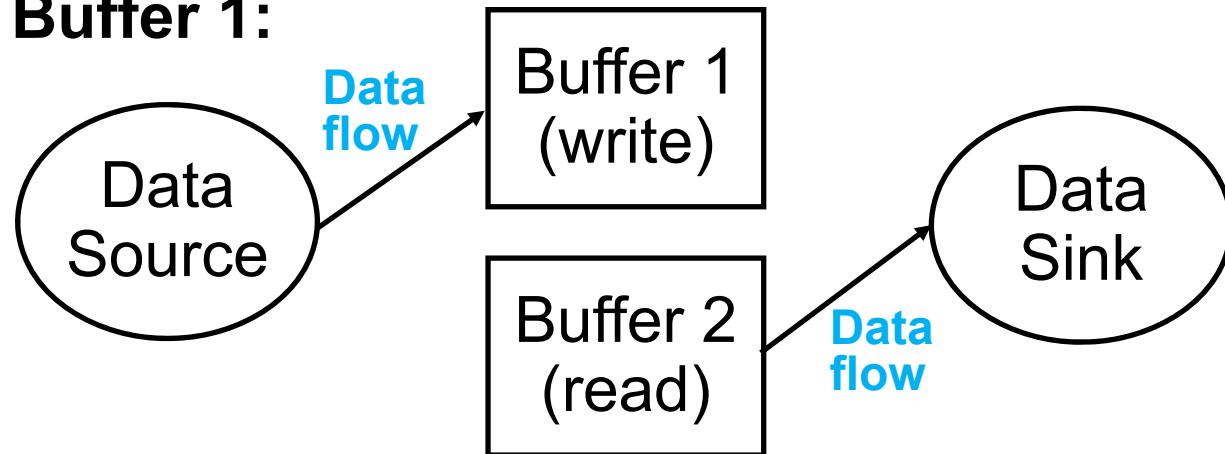
2) Double Buffers

- Two equal-sized storage areas are provided.
- At any one time, one area is the *write buffer*, where newly generated data can be written by one agent.
- At the same time, the other area is the *read buffer*, where previously written data can be read without the danger of having new data written into the area.
- After all of the data in the read buffer has been read, all or some of the contents of the write buffer can be transferred into the read buffer.

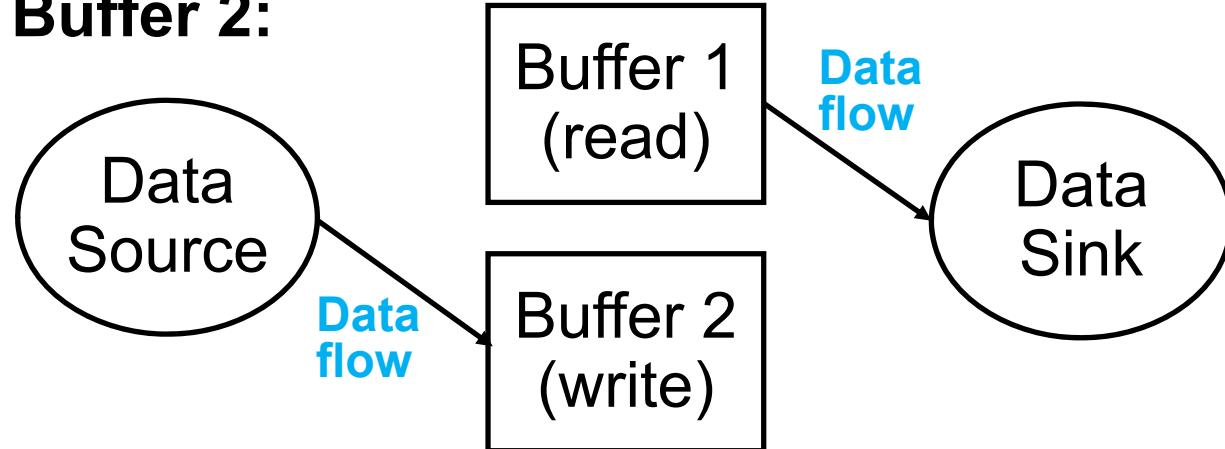
Note: The “transfer” can usually be accomplished by simply swapping the roles of the two buffers, say by swapping the base pointers to the two buffers.

Double Buffers / Ping-Pong Buffers

Writing to Buffer 1:



Writing to Buffer 2:



3) Stacks

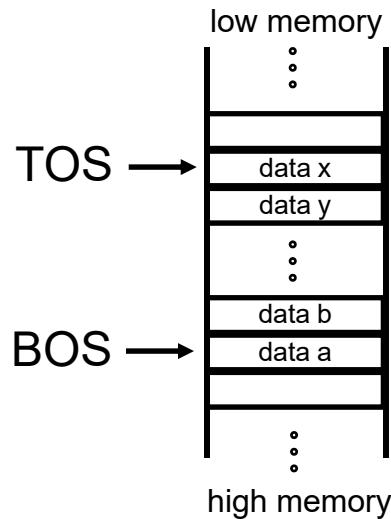
- It is a common requirement to access stored data on a *first-in last-out* (FILO) basis
 - e.g. store the CPU state prior to starting an exception handling routine or a subroutine
- The *FILO stack* is a data structure that provides the following three basic operations:
 - 1) PUSH (datum) -- add a new datum onto the stack
 - 2) TOP -- examine the most recently stored datum in the stack
 - 3) POP -- remove the most recently stored datum from the stack

Note: Sometimes the TOP and POP operations are combined into one operation (POP).

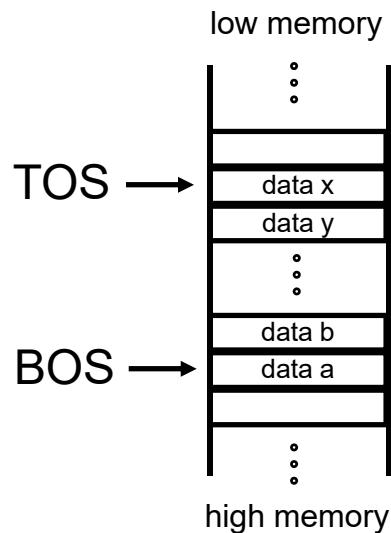
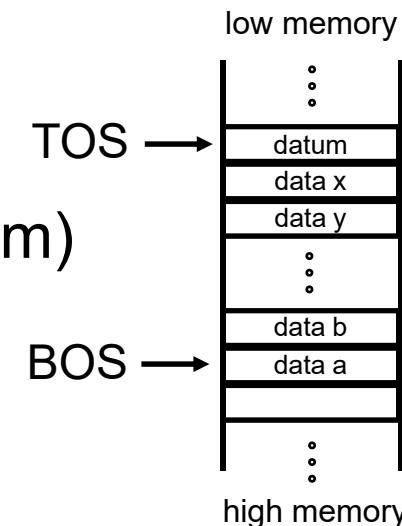
3) Stacks (cont'd)

- Additional stack terminology:
 - **Top of Stack** (TOS) is the storage area occupied by the most recently stored datum.
 - **Bottom of Stack** (BOS) is the storage area occupied by the first datum that was stored.
- A stack may be implemented either as:
 - a) a special-purpose hardware device
 - b) a region in main memory, together with TOS and BOS stack pointers
- Data items are not moved around in memory once they have been pushed onto the stack.
- Instead, a fixed pointer records the BOS and a changing pointer records the TOS.

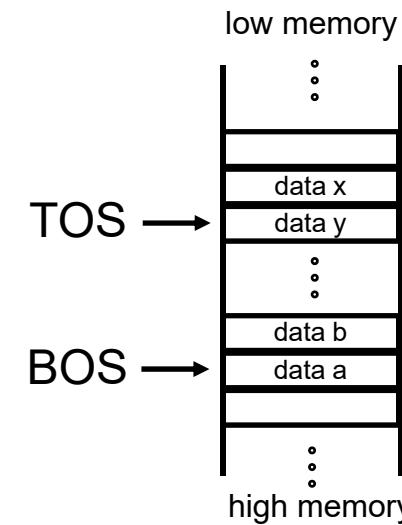
Stacks Growing Toward Low Memory



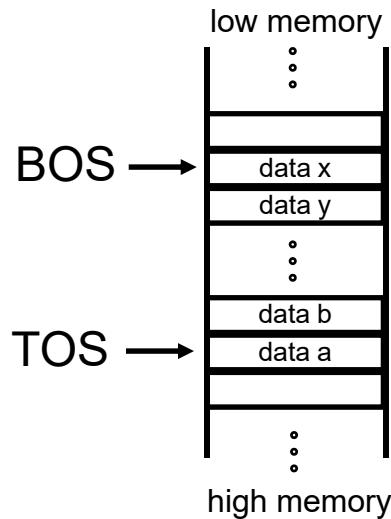
after PUSH(datum)



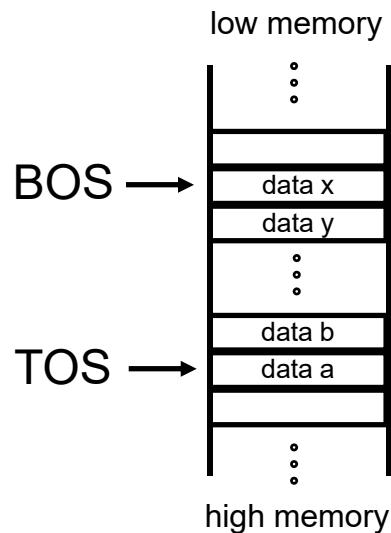
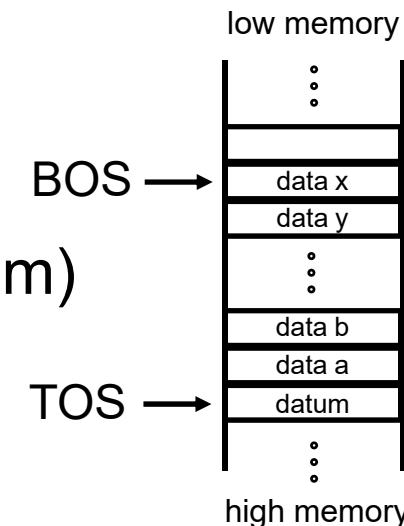
after POP



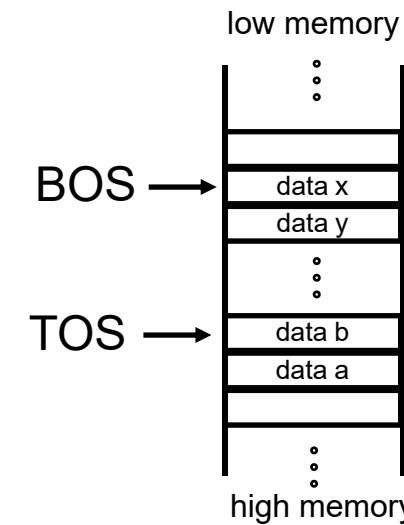
Stacks Growing Toward High Memory



after PUSH(datum)



after POP



4) FIFO Queues

- A queue is a data structure in which the data items are accessed on a *first-in first-out* (FIFO) basis.
- A *FIFO queue* provides the following three basic operations:
 - 1) PUSH (datum) -- Add a datum to the back end of the queue.
 - 2) FRONT -- Examine the datum at the front end of the queue.
 - 3) PULL -- Remove the datum from the front end of the queue.

Note: Sometimes the FRONT and PULL operations are combined into one operation (PULL).

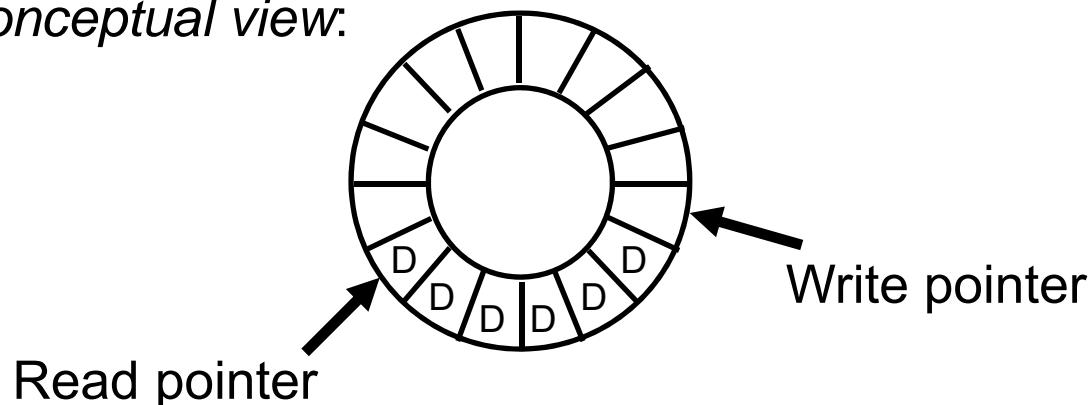
4) FIFO Queues (cont'd)

- Additional queue terminology
 - **Front of Queue** (FOQ) is the storage area occupied by the earliest stored data item that has yet to be removed from the queue.
 - **Back of Queue** (BOQ) is the storage area occupied by the most recently stored data item.
- A queue may be implemented either as:
 - a) a region in main memory, together with FOQ and BOQ stack pointers
 - b) a special-purpose hardware device

5) Circular Buffers

- One practical problem with implementing a queue is to contain the movement of the queue to a limited region in main memory.
- A *circular buffer* is a data structure that provides the same functionality as a queue, within a region in memory defined by two fixed address limits.
- The capacity of the circular buffer is fixed.

Conceptual view:

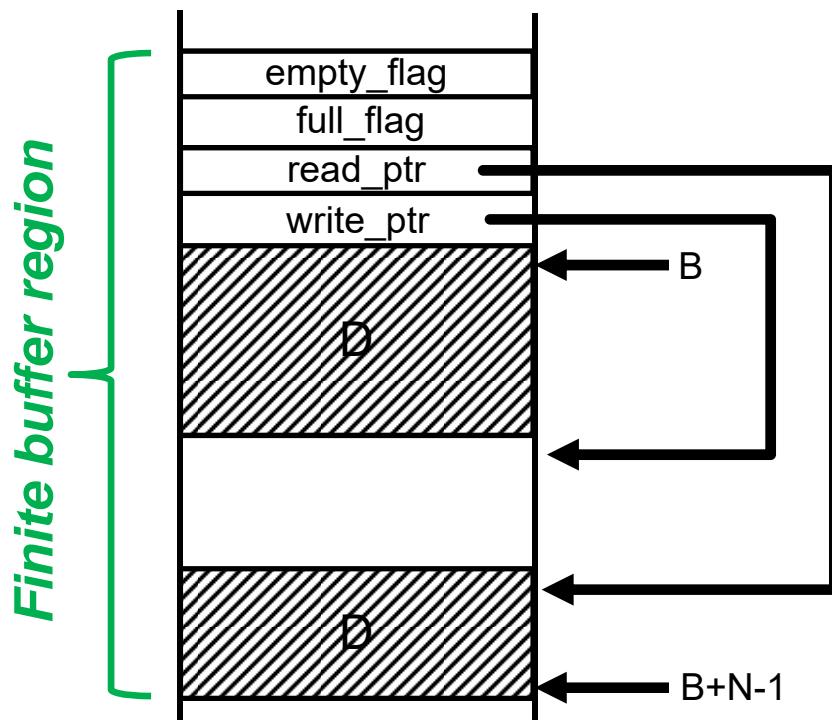


5) Circular Buffers

- Two pointers are used to keep track of data input/output:
 - a) **read pointer** = pointer to the front of the queue
= address where the next datum to be *read* from the buffer is located
 - b) **write pointer** = pointer to the back of the queue
= address where the next datum will be *written* into the buffer
- The read and write pointers advance in the same direction through a finite-sized buffer region in memory.
- When a pointer must be moved beyond the first address limit, it is set to the other address limit, (i.e., wrapped around) instead of being incremented (or decremented) outside of the allocated buffer region in memory.

5) Circular Buffers (cont'd)

Practical Implementation of an N-word Circular Buffer



Initialization routine:

```
/* base address is B */  
init_circular_buffer();  
begin  
    empty_flag = true;  
    full_flag = false;  
    write_ptr = B;  
    read_ptr = B;  
end;
```

5) Circular Buffers (cont'd)

Write routine for the circular buffer:

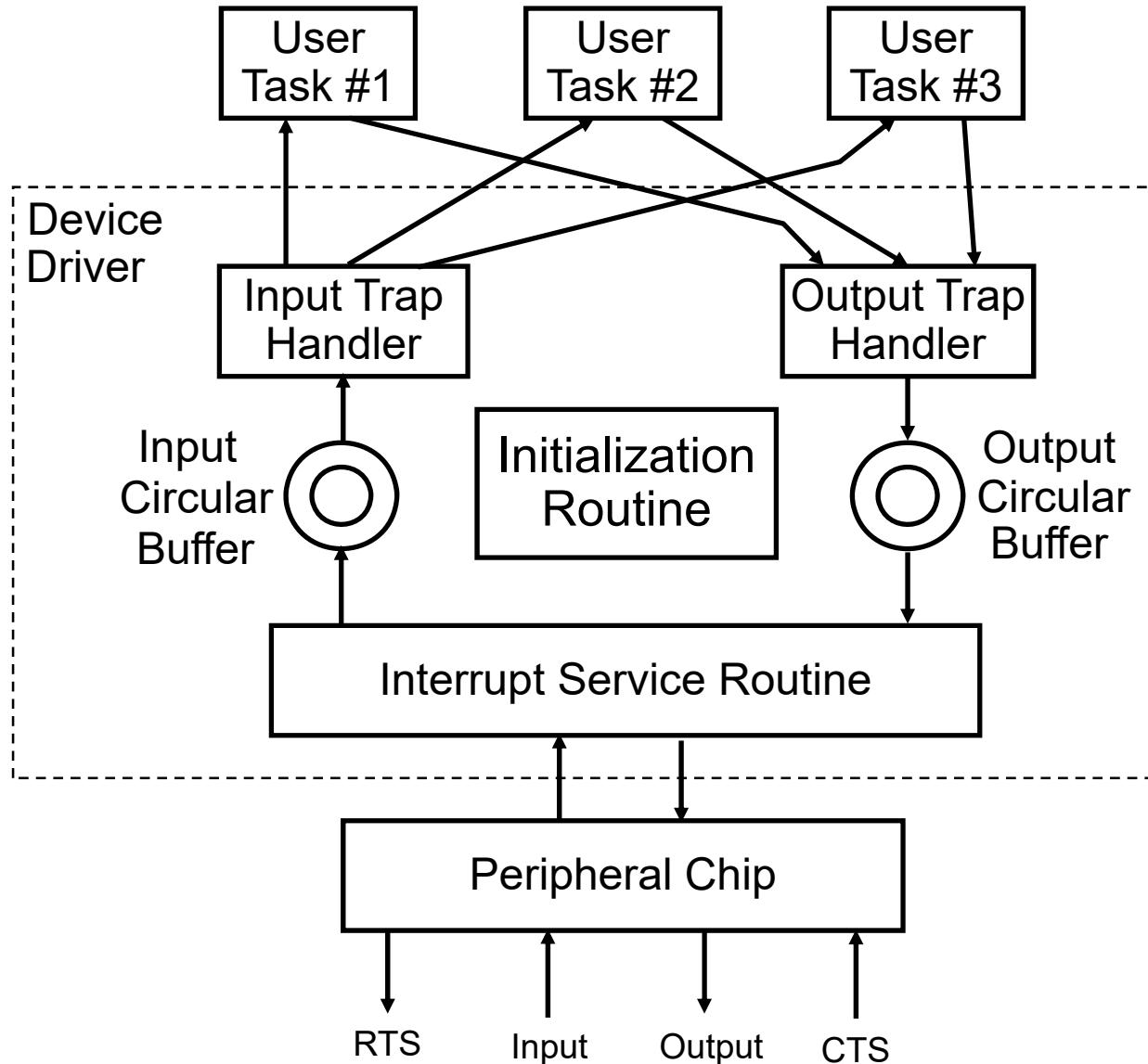
```
err_type procedure write( data_type data );
begin
    if full_flag then return( ERR_BUF_FULL );
    else
        memory(write_ptr) = data;
        empty_flag = false;
        write_ptr = (((write_ptr - B) +1) mod N) + B;
        if ( read_ptr == write_ptr ) then
            full_flag = true
        endif;
        return( ERR_OK );
    endif;
end;
```

5) Circular Buffers (cont'd)

Read routine for the circular buffer:

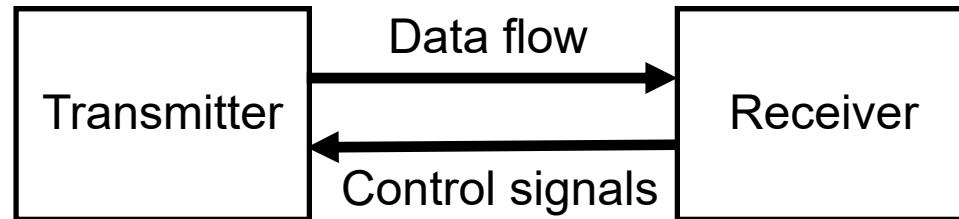
```
err_type procedure read( data_type *Pdata );
begin
    if empty_flag then return( ERR_BUF_EMPTY );
    else
        *Pdata = memory(read_ptr);
        full_flag = false;
        read_ptr = (((read_ptr - B) +1) mod N) + B;
        if ( read_ptr == write_ptr ) then
            empty_flag = true
        endif;
        return( ERR_OK );
    endif;
end;
```

Circular Buffers in a Device Driver



Note: The device driver must manage the flow of data to and from each user task so that the data flows are not mixed up. For example, a buffer (or file) structure could be used so that the device driver can identify the user task for each buffer and the size of each buffer.

Flow Control



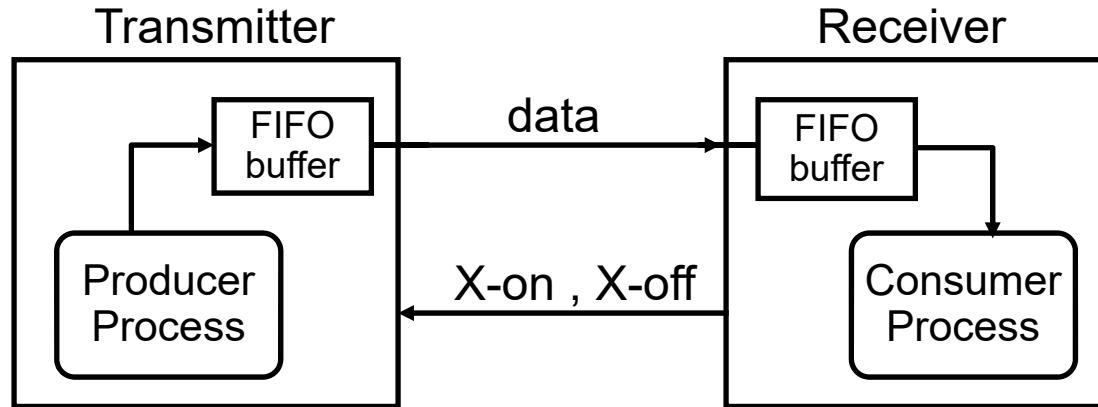
- In an asynchronous communications channel from a transmitter to a receiver, some control mechanism is required to prevent the transmitter from overwhelming the receiver with more data than it can handle.
- A buffer in the receiver helps deal with only short-term fluctuations in data rates, not sustained mismatches.
- Also required is a buffer in the transmitter and some signalling method, called *flow control*, to permit the receiver to stop (or slow down) and restart (or speed up) the flow of data from the transmitter.

Flow Control Mechanisms

Five common flow control mechanisms:

- 1) X-on / X-off for RS-232C
- 2) Ethernet Flow Control using Pause Frames
- 3) Hardware Flow Control
- 4) ENQ / ACK for RS-232C
- 5) Window-based methods, including TCP

1) X-on / X-off Flow Control



- The channel is duplex (bi-directional) and carries ASCII data.
- The transmitter stops sending more data after its receiver gets an ASCII DC3 character, which is also known as X-off (code \$13). On a keyboard, X-off is generated by typing control-S.
- The transmitter can resume sending data to the receiver after its receiver gets an ASCII DC1 character, which is also known as X-on (code \$11). On a keyboard, an X-on is generated by typing control-Q.

Receiver Using X-on/X-off Flow Control

```
/* Receiver polling loop process */
repeat
    if receive_buffer_too_full then
        /* too full could mean 90% full */
        send X-off character back to transmitter;
        repeat
            process some characters in receive buffer;
        until receive_buffer_sufficiently_empty;
        /* sufficiently empty could mean 40% full */
        send X-on character back to transmitter;
    endif;
    if a_character_has_been_received then
        add new character to receive buffer;
    endif;
    if receive_buffer_not_empty then
        process some characters in receive buffer;
    endif;
forever;
```

Transmitter Using X-on/X-off Flow Control

```
/* Transmitter polling loop process */
repeat
    if transmit_buffer_full then
        stop producing more data characters;
    endif;
    if X-off_character_received_from_receiver then
        repeat
            stop transmitting data to receiver;
            if transmit_buffer_full then
                stop producing more data characters;
            else
                produce more data and add to buffer;
            endif;
            do something else, or busy wait, for a while;
        until X-on_received_from_receiver;
    endif;
    if transmit_buffer_not_empty then
        transmit some characters from buffer to receiver;
    endif;
    if transmit_buffer_not_full then
        produce more data characters;
        add data characters to transmit buffer;
    endif;
forever;
```

Pros and Cons of RS-232C X-on/X-off

Pros:

- Simple
- Can easily be implemented in software
- No need for special hardware features

Cons:

- Relatively slow response time if done in software
- X-on/X-off characters consume data bandwidth
- How to recover if X-on/X-off chars are lost?

2) Ethernet Flow Control Using Pause Frames

- Ethernet standard IEEE 802.3x (1997) provides a simple flow control method, which allows a receiving node to send a special “pause frame” to a transmitting node that causes that node to stop transmitting frames to the receiving node for a finite time given in the pause frame.
- The transmission pause time is expressed in units of “quanta”, where a quanta is equal to 512 bit times (64 serial byte times).
- When the 16-bit Length/Type field parameter in the Ethernet frame header is 0x8808 (decimal 34,824), the data field in an Ethernet frame is interpreted as a “MAC Control” command.
- A “pause frame” is encoded with the 16-bit opcode 0x0001, followed a 16-bit pause time (in units of quanta), and then 42 all-zero bytes.
- Either the actual destination address (for the transmitter to be stopped) or the reserved multicast address 01:80:C2:00:00:01 must be used in the header of the pause frame. The source address in the pause frame is the address of the node that sends the pause frame.

Structure of an Ethernet Pause Frame

```
Frame 2 (64 bytes on wire, 64 bytes captured)
Arrival Time: Jan 30, 2008 10:25:52.012139558
[Time delta from previous captured frame: 0.036914825 seconds]
[Time delta from previous displayed frame: 0.036914825 seconds]
[Time since reference or first frame: 0.036914825 seconds]
Frame Number: 2
Frame Length: 64 bytes
Capture Length: 64 bytes
[Frame is marked: False]
[Protocols in frame: eth:macc]
[Coloring Rule Name: Broadcast]
[Coloring Rule String: eth[0] & 1]
Ethernet II, Src: 42networ_30:41:50 (00:0f:5d:30:41:50), Dst: Spanning-tree-(for-bridges)_01 (01:80:c2:00:00:01)
Destination: Spanning-tree-(for-bridges)_01 (01:80:c2:00:00:01)
    Address: Spanning-tree-(for-bridges)_01 (01:80:c2:00:00:01)
        ....1 .... .... .... = IG bit: Group address (multicast/broadcast)
        ....0. .... .... .... = LG bit: Globally unique address (factory default)
Source: 42networ_30:41:50 (00:0f:5d:30:41:50)
    Address: 42networ_30:41:50 (00:0f:5d:30:41:50)
        ....0 .... .... .... = IG bit: Individual address (unicast)
        ....0. .... .... .... = LG bit: Globally unique address (factory default)
Type: MAC Control (0x8808)
MAC Control
    Pause: 0x0001
    Quanta: 65535
```

0000 01 80 c2 00 00 01 00 0f 5d 30 41 50 88 08 00 01]0AP...
0010 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030 00 00 00 00 00 00 00 00 00 3f ab 2a 6b ?:*k

Destination address = 01:80:c2:00:00:01

MAC Control opcode = 00:01 “Pause frame”

Source address = 00:0f:5d:30:41:50

Pause time = ff:ff 65535 x 512 bit times

Length/type = 88:08 “MAC Control”

Frame checksum = 3f:ab:2a:6b

Pros and Cons of Ethernet Pause Frames

Pros: • Simple mechanism, like X-off with timeout to X-on.

Cons: • The delay to stop transmission is at least the transmission time of the pause frame. This is the time to transmit 84 bytes (12-byte minimum interframe gap, 7-byte preamble, 1-byte SFD, 64 bytes in the rest of the frame).
• The pause frame stops *all* transmission from one node to a second node. This mechanism does not distinguish between different kinds of traffic, and so it can be disruptive to operation.

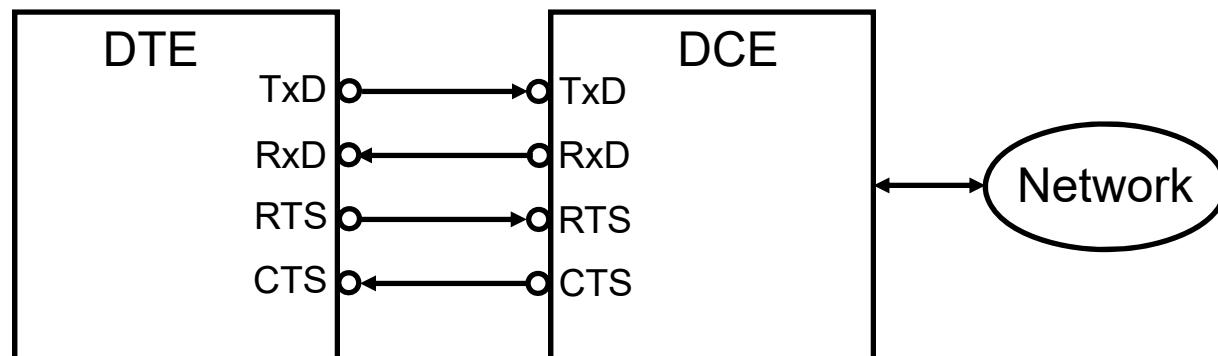
Ethernet standard IEEE 802.1Qbb (2011) defines an enhanced “Priority pause frame” that provides separate pause times for traffic at eight different “classes of service”.

3) Hardware Flow Control

- Instead of using special data characters for flow control signalling, use hardware handshake signals.
- Appropriate handshake signals are provided in the RS232c asynchronous communications standard.

Request to Send (RTS) is used by the DTE to signal to the DCE its desire to transmit more data.

Clear to Send (CTS) is used by the DCE to signal its ability to receive more data from the DTE.



Pros and Cons of Hardware Flow Control

- Pros:**
- Faster than X-on/X-off flow control
 - Simple handshake protocol is easily implemented

- Cons:**
- Extra hardware signals (and wires) are required to carry the control signals (e.g., CTS, RTS) in parallel with the data signals (e.g., TxD, RxD).

4) ENQ / ACK Flow Control

- A block-oriented mechanism: i.e., data is sent from transmitter to receiver in fixed-sized blocks.
- Transmitter sends ASCII “ENQ” control character (hex value \$05) to ask the receiver whether it is ready to receive the next block of data.
- Receiver sends ASCII “ACK” control character (hex value \$06) to transmitter when it is ready to receive the block of data.
- After the transmitter receives the “ACK”, it sends one block of data to the receiver.
- ENQ-ACK cycle begins again for the new block of data to be transmitted.

Pros and Cons of ENQ/ACK Flow Control

Similar pros and cons as in X-on/X-off flow control.

Pros:

- Simple
- Can easily be implemented in software
- No need for special hardware features

Cons:

- Relatively slow response time if done in software
- ENQ/ACK characters consume data bandwidth
- How to recover if ENQ/ACK chars are lost?
- The data flow is limited by the control communication delay (sending ENQ & then receiving ACK).

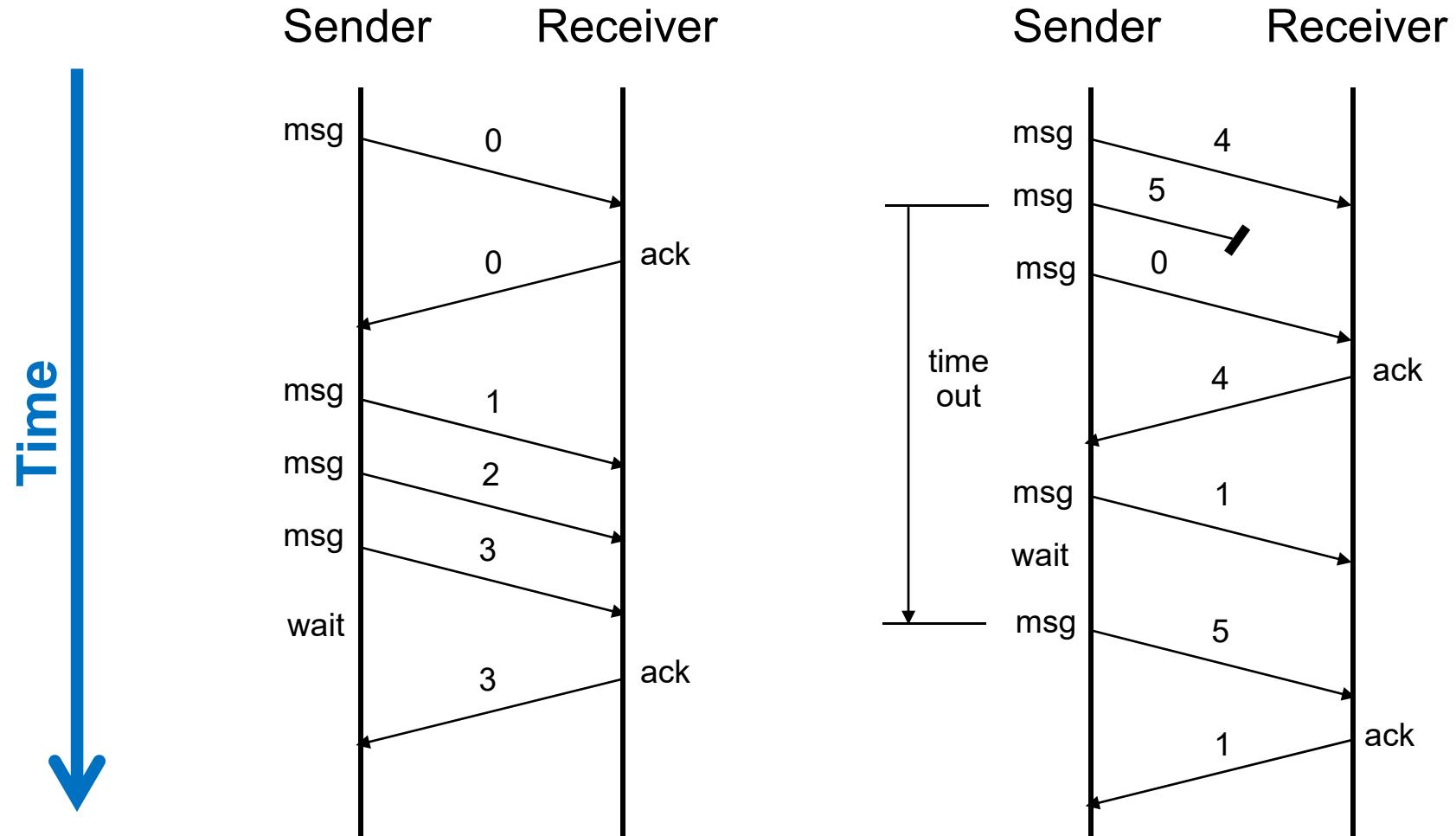
5) Window-based Flow Control

- A method of flow control together with a mechanism for retransmitting lost messages, and detecting out-of-order and duplicate received messages.
- The sender tags its message with a sequence number $0, \dots, N-1$ that is incremented after each message.
- The receiver acknowledges single messages or groups of messages by sending back the sequence number of the last message/byte that was correctly received in order.
- The sender is allowed to have up to $W \leq N$ messages sent to the receiver but not yet acknowledged.
 W is the *window size*, which may be fixed or adjustable.

Handling Lost Messages / Data Packets

- It is possible for data packets to get lost
 - they may get discarded because of detected errors
 - they may get discarded because of buffer overflow
 - they may get greatly delayed because of congestion
- Once the transmitter has sent out W data packets, it must stop and wait for an acknowledgement. But it should not wait forever.
- A timer is used to keep track of the time elapsed since each data packet was transmitted. If the elapsed time exceeds some value (e.g., twice the expected round trip delay), then the transmitter assumes that the data packet was lost, and then retransmits all unacknowledged packets.

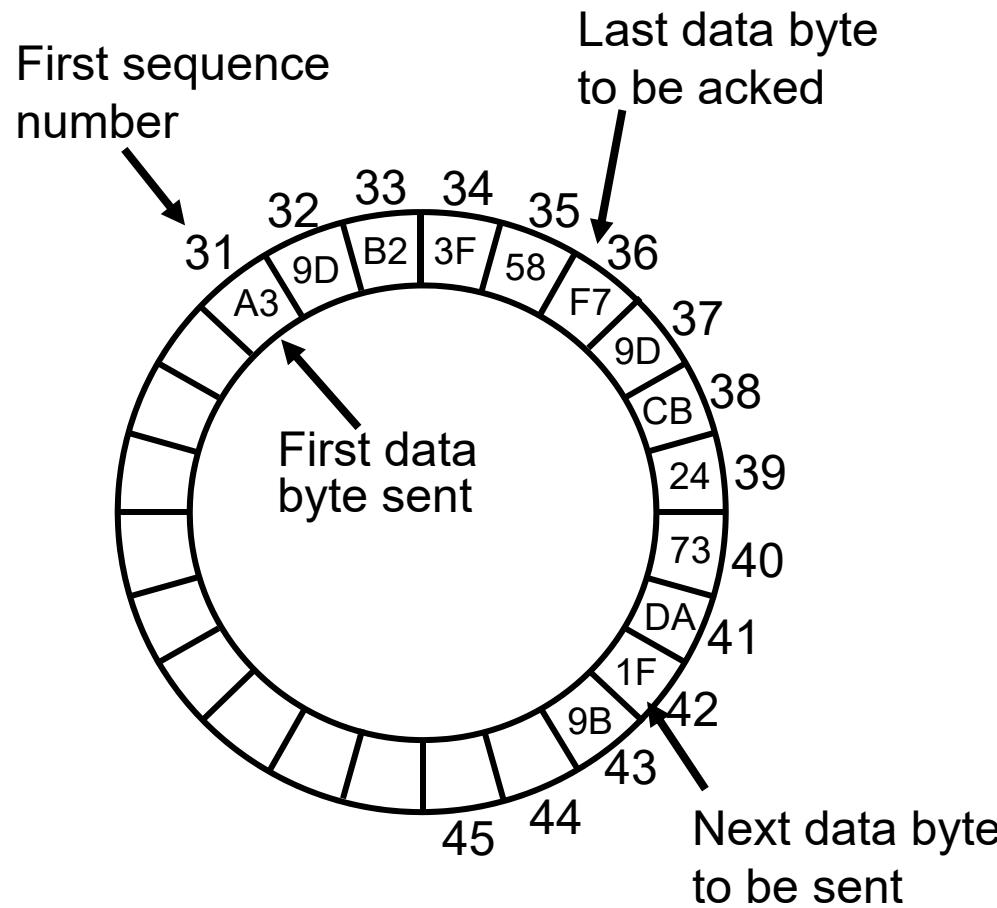
Time Sequence Diagrams



Ex: Flow Control in TCP

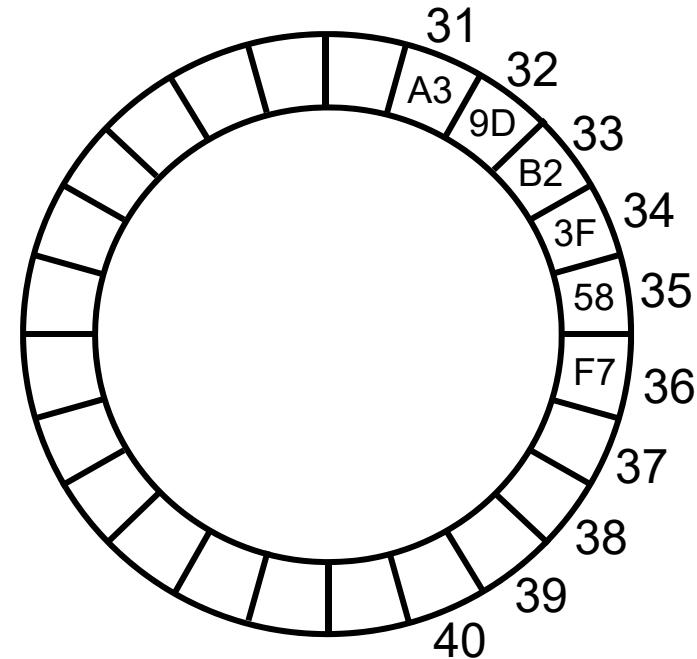
- The *data bytes* are numbered by TCP to allow it to:
 - (1) detect the loss of segments (loss of data bytes)
 - (2) detect out-of-order and duplicate segments
 - (3) implement flow control
- Data bytes (not the data packets) in the transmit and receive directions are numbered separately.
- The first data byte in either direction is assigned a random initial 32-bit unsigned integer number.
- Each TCP segment is assigned a 32-bit sequence number, which is the number of first data byte in the segment.
- The 32-bit acknowledgment number in the transmit direction is the byte number of the next expected data byte in the receive direction.

Ex: Flow Control in TCP



Buffer in Sender

Bytes in transit: 9D, CB, 24, 73, DA



Buffer in Receiver

Ex: Flow Control in TCP

- The receiver (with respect to one of the two directions) controls the window size in the sender.
- The sender window size is communicated in a field in the segments that are travelling in the opposite direction.
- The sender window could be set equal to the current number of free byte locations in the receiver's buffer.
- A sender window value of zero prevents the transmitter from sending any more data (like an X-off command).
- A sender window value of greater than zero allows the transmitter to send data bytes (but not exceeding the window size if there are unacknowledged data bytes).

Pros and Cons of Window-based Flow Control

Pros:

- Flexible
- Accommodates flow control, lost segments, out-of-order segments, duplicate segments

Cons:

- More complicated than other flow control methods
- Need timers to detect lost segments (without timeouts, the TCP connection could “hang” and be prevented from transmitting any more data).

Microcomputer Busses and Direct Memory Access (DMA)

References:

- Freescale Semiconductor, Inc., “MCF5235 Reference Manual”.
- Freescale Semiconductor, Inc., “MCF5441X Reference Manual”.

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 Freescale Semiconductor, Inc.

Definitions

Definition from “Microcomputer Busses”, by R.M. Cram, (Academic Press, 1991):

“A *bus* is a tool designed to interconnect the functional blocks of a microcomputer in a systematic manner. It provides for standardization in mechanical form, electrical specifications, and communication protocols between board-level devices.”

Processor-specific bus: a bus that is intended for use with only one processor or with members of one family of compatible processors.

Ex: Freescale FlexBus, ARM AMBA bus, Altera Avalon bus, IBM CoreConnect, Barco External Bus Interface (EBI)

Standardized processor-independent bus: a bus that is intended to promote interchangeability among a class of board-level products based on possibly different processors.

Ex: PCI, PCIe

Historical busses: S-100, Unibus, Std bus, VME bus, Multibus, etc.

A Functional Classification of Busses

1) Local Processor-Memory & Graphics Busses

- short, synchronous, high-speed
- overriding priority: maximize the data throughput
 - e.g. Rambus, DDR, DDR2, DDR3, VESA, AGP, PCIe

2) Input/Output (I/O), Peripheral and Instrument Busses

- maximum flexibility is the priority
- must accommodate a variety of data rates and latencies
- open standards are used to maximize the potential market
 - e.g. SCSI, GPIB (IEEE-Std-488), USB, Firewire

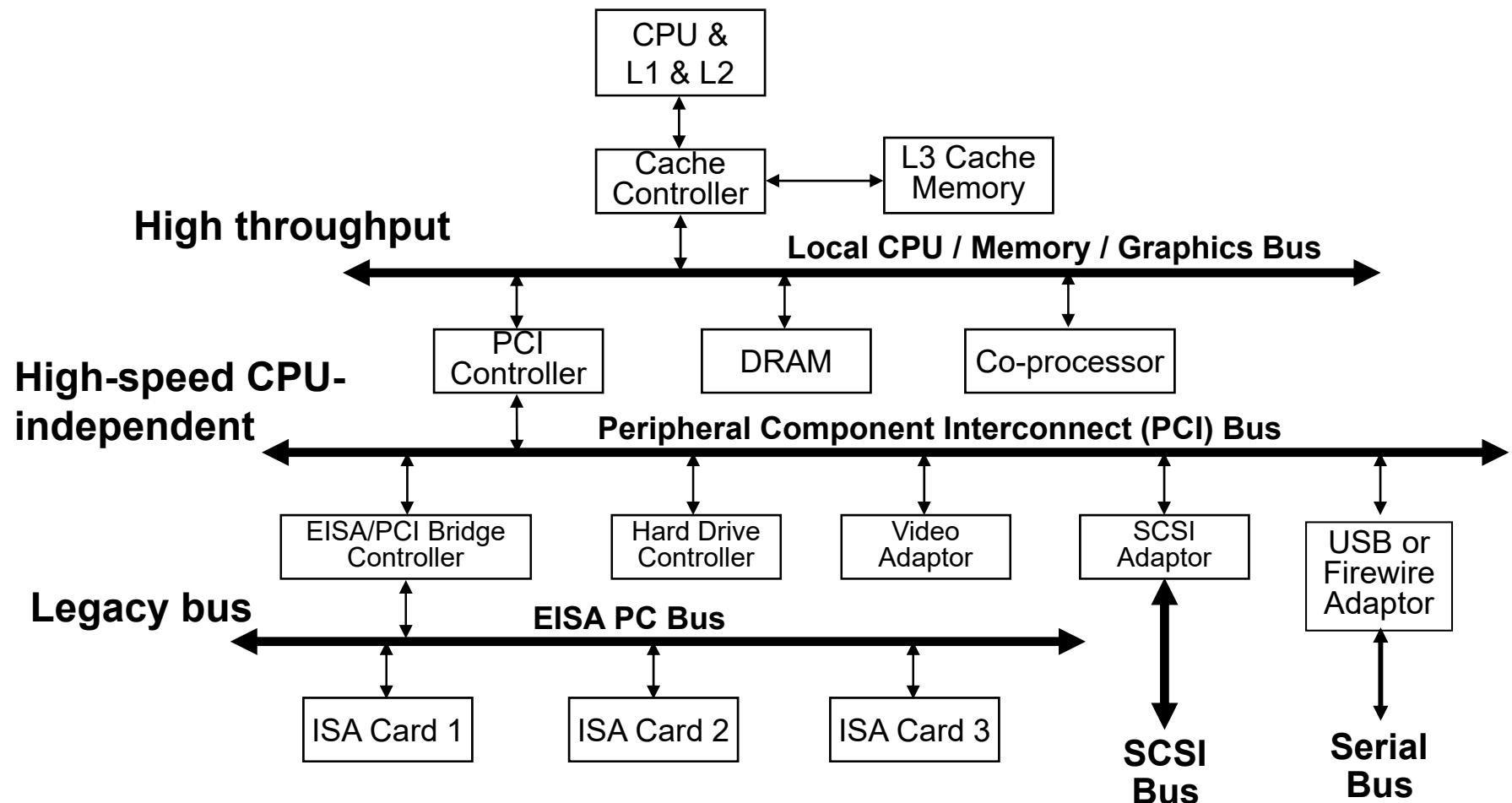
3) Expansion Busses (formerly called Backplane Busses)

- often midway in performance between processor-memory busses and I/O, Peripheral & Instrument busses
- expansion busses are used to reduce system design cost and to reduce the time-to-market for new computer systems
 - e.g. S100, Std bus, VME, Multibus, PCI, CompactPCI

Hierarchy of Busses (early 1990s)

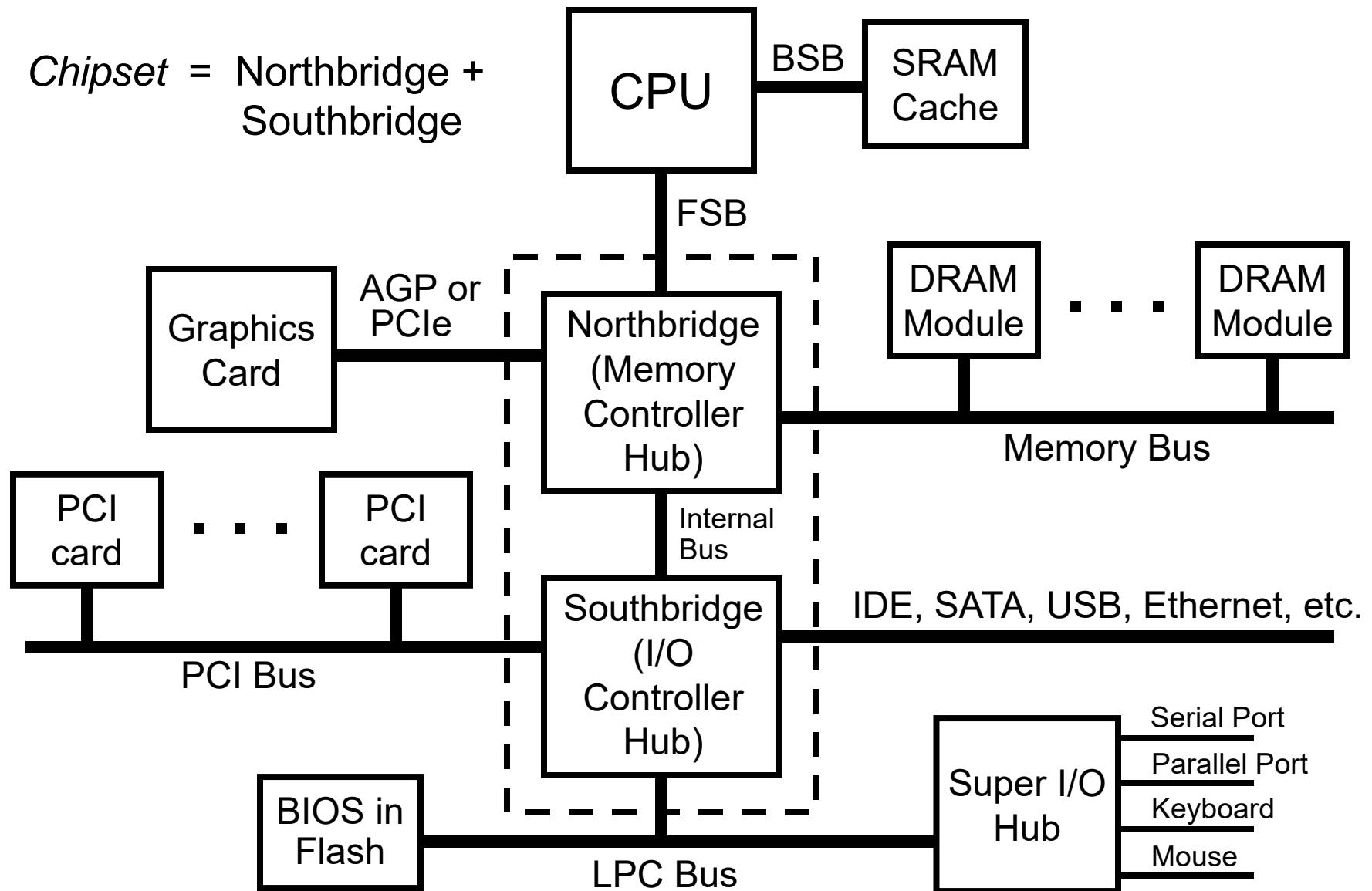
- To exploit the strengths (and avoid the weaknesses) of different kinds of busses, high-performance systems often use a hierarchy of busses.

Example: Personal Computer Bus Architecture (old-fashioned)



Chipset-based PC Bus Architecture (late 1990s)

Chipset = Northbridge + Southbridge



Some PC Bus Acronyms

BSB = Back Side Bus (connects CPU to an external cache)

FSB = Front-Side Bus

SRAM = Static Random-Access Memory (fast & expensive)

DRAM = Dynamic Random-Access Memory (slower & cheap)

AGP = Accelerated Graphics Port

PCI = Peripheral Component Interconnect (bus-based)

PCIe = PCI Express (faster point-to-point version of PCI)

IDE = Integrated Drive Electronics (for connecting hard disks)

SCSI = Small Computer System Interface

SATA = Serial Advanced Technology Attachment

USB = Universal Serial Bus

BIOS = Basic Input/Output System (self-test, autoconfiguration)

LPC = Low Pin Count (for low-speed interfaces to humans)

Intel X299 High-end Chipset (June 2017)

INTEL® X299 CHIPSET BLOCK DIAGRAM

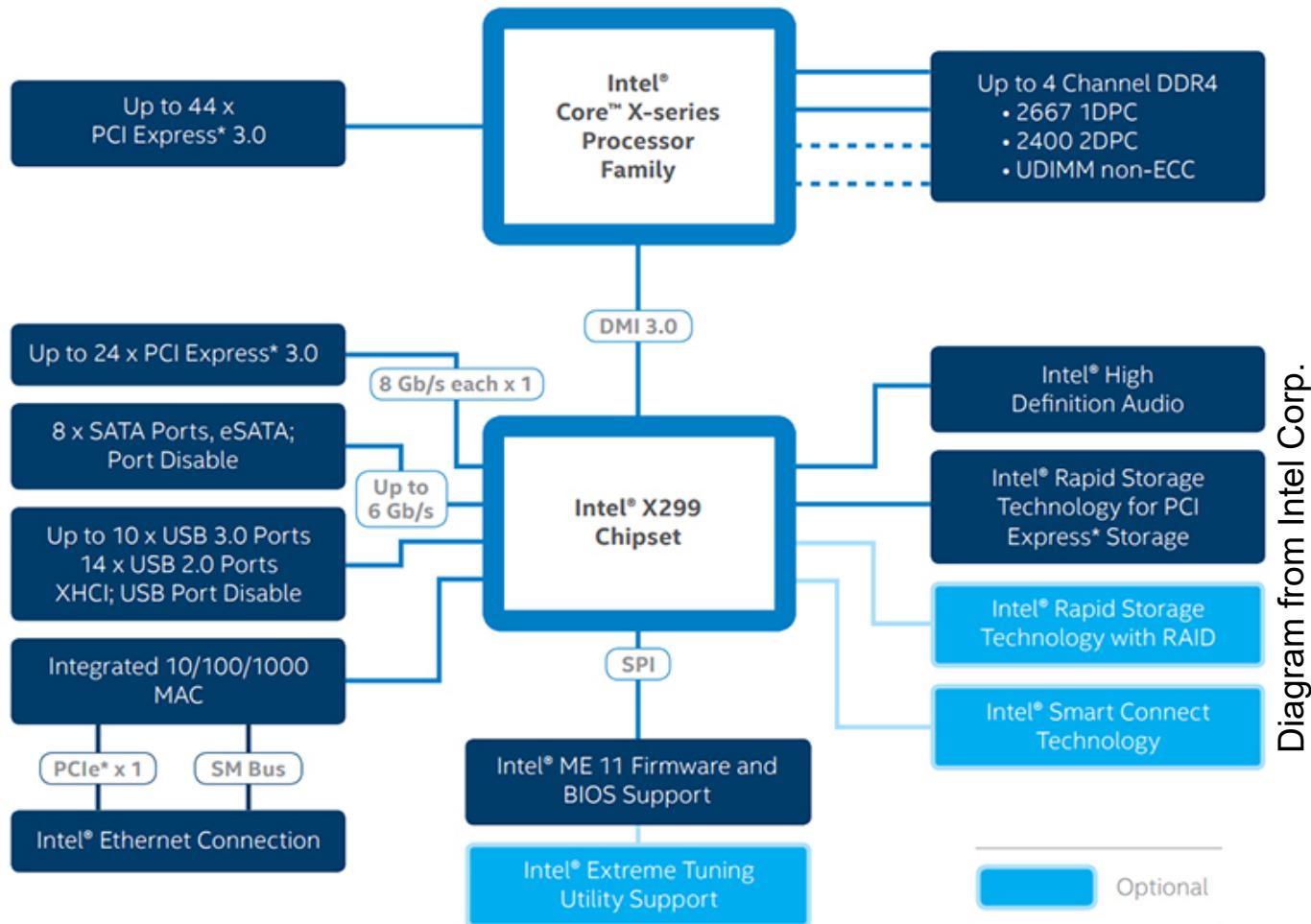


Diagram from Intel Corp.

Note: Direct Media Interface 3.0 (DMI 3.0) was originally a Northbridge-Southbridge interface bus at Intel.

Copyright © 2020 by Bruce Cockburn

12-7

Some Bus Terminology

Bus protocol : A set of allowed bus signal transition sequences and required timing constraints.

Bus operation / transaction : A data transfer or control transfer operation that takes place using bus signals according to a bus protocol.

Bus master : A subsystem connected to the bus that can determine the bus operations. More than one bus master can be present on the same bus, but only one bus master can have control (i.e., be active) at a time.

e.g. **CPUs, DMACs, Graphics Accelerator**

Bus slave : A subsystem connected to the bus that responds to bus operations initiated by the currently active bus master.

e.g. **RAM, ROM, Peripheral / Interface Chips**

Bus arbitration : The process of determining which one of two or more contending bus masters will be awarded control of the bus (and thereby become the active bus master).

Arbiter : A circuit (possibly in the CPU or MCU) that performs arbitration.

Basic Bus Functions

1. Data transfer

- master-to-slave(s), slave-to-master, slave-to-slave
- bytes, words, longwords, larger blocks (e.g. cache lines)

2. Interrupt handling

- communication of active interrupt signals to the bus master
- arbitration between multiple sources of interrupts
- communication of interrupt vector information (e.g. IACK cycle)

3. Arbitration among multiple bus masters

- arbitration protocol to choose one bus master from among two or more possible contending bus masters
- procedure for transferring bus control from the current bus master to a new bus master

4. Utility functions

- power (including the voltages used as the reference for digital signals)
- clocks
- system reset & test signals (e.g., JTAG port)
- power failure detection & battery back-up

Signal Groups Within a Typical Bus

1. Data signals

- encode the data that is passed between the bus master and bus slaves/targets
- number of data signals determines the “bit width” of the system
- parity bits or other error correction and control (ECC) bits may be transmitted with each data word so that errors can possibly be detected and corrected at the destination

2. Address signals

- used to identify locations in memory and memory-mapped registers in peripheral & interface chips
- the number of address signals determines the maximum size of the (virtual) memory space

Note: Some or all of the data and address signals may be time-multiplexed on the same bus lines to reduce the pins and wiring.

Signal Groups Within a Typical Bus (cont'd)

3. Control signals

- used to co-ordinate bus transactions
- used to arbitrate among:
 - multiple possible bus masters
 - multiple sources of interrupts
- power failure handling
- entry into and exit from test modes; distribute test data
- access to system state (e.g. CPU registers and memory) to support software debugging

4. Power signals

- typical: +5 VDC, +3.3 VDC, +1.8 VDC, +1.2 VDC, etc.
- optional: +12 VDC, -12 VDC, -5 VDC

Sharing a Bus Among Multiple Bus Masters

Coarsest granularity

- 
1. **Exclusive Control (Burst Mode):** Each bus master retains exclusive control of the bus for several bus transactions.
e.g. CPU, CPU, DMAC1, DMAC1, DMAC1, CPU, CPU, . . .
 2. **Cycle Stealing:** Bus transactions from different bus masters are interleaved on an *ad hoc* basis or on a strictly round-robin basis.
e.g. CPU, DMAC1, DMAC2, CPU, DMAC1, DMAC2, . . .
 3. **Split Transactions (Pipelined Bus)**

Read transactions are split into two transactions:

- 1) master sends read command & target address
- 2) slave sends a return block containing data

The bus is available to be used for other transactions (possibly by other bus masters) during the memory read access time.

e.g. RAMBUS, Synchronous DRAMs (DDR, DDR2, DDR3, etc.)

Finest granularity

	PROS	CONS
Exclusive Control	<ul style="list-style-type: none"> -- simplicity -- software method -- no special hardware required 	<ul style="list-style-type: none"> -- coarse granularity -- bus time may not be shared fairly or efficiently
Cycle Stealing	<ul style="list-style-type: none"> -- fairer sharing of the bus 	<ul style="list-style-type: none"> -- requires hardware support -- however this support is available in most CPU's
Split Transactions	<ul style="list-style-type: none"> -- high-speed buses do not have to wait for slowly responding devices 	<ul style="list-style-type: none"> -- requires hardware support in the bus and all connected devices

Arbitration Among Contending Bus Masters

1. Fixed Priority

- Each bus master is assigned a unique priority.
- The contending bus master with the highest priority is awarded control of the bus.

2. Rotating Priority

- At any one time, each bus master has a unique priority.
- The priority assignments are periodically rotated so that each bus master takes a turn at having each of the available priorities.

3. Pseudo-random Selection

- The winning bus master is selected "randomly" from among the currently contending masters.
- The selection algorithm is not truly random because it is an entirely predictable digital algorithm that mimics the statistics of a truly random process. Such an algorithm is called "pseudo-random".

Synchronous, Asynchronous, Semi-synchronous

Synchronous Busses:

- All subsystems coordinate data transfers with respect to the edges of one common (possibly multi-phase) system clock.
- All peripheral chips must be able to respond to bus transactions (reads and writes) within the same time constraints.

Asynchronous Busses:

- Responses to bus transactions can arrive at any time.
- Need special handshake lines in the control bus to ensure that data transfers take place correctly.
- Bus masters can work with different peripheral devices with a variety of response times.

Semi-synchronous Busses:

- A compromise between purely synchronous and purely asynchronous busses.
- Bus transactions can take a variable number of system clock cycles.
- System can accommodate a variety of response times from the peripherals.
- Need a handshake/acknowledge signal to end each bus transaction.

Data Transfer Protocols

Unilateral :

- The bus master initiates the bus operation and then assumes that the corresponding bus slave(s) will respond within predetermined time constraints.
- The duration of the bus operation is determined entirely according to the operation selected by the bus master and timing provided by a shared system clock signal.

Bilateral:

- The bus master initiates the bus operation, but the duration of the subsequent bus operation depends on the responses of both the bus slave and the bus master.
- The bus master and the bus slave use handshake lines to communicate timing information to each other.

Multilateral:

- Extension of the bilateral transfer protocol to allow for more than one active bus slave.

Bus Classification

Data Transfer Protocol	Underlying Timing		No Clock Present
	Shared System Clock	No Clock Present	
Unilateral	Synchronous e.g. 6800		
Bilateral	Semi-synchronous e.g. 68000, Pentium, MCF523x	Asynchronous e.g. Unibus	
Multilateral		Asynchronous GPIB (IEEE-488)	

Bus Trade-Offs

Source: J. Hennessy and D. Patterson, “Computer Architecture: A Quantitative Approach”, (Morgan Kaufmann, 1990)

Option	Higher Performance	Lower Cost
1) Bus Width	Separate data & address busses	Multiplexed data & address
2) Data Width	Wider is faster e.g. 8, 16, 32, 64	Narrower is < \$ e.g. 16, 8
3) Transfer Size	Block transfers using DMA	Single word using CPU
4) Bus Masters	Multiple masters (requires arbitration)	One master, the CPU, no arbit.
5) Split Transactions?	Yes, to get more pipelining	No, too complex
6) Clocking	Synchronous with matched elements	Asynchronous or semisynchronous

Timing Terminology

Caution: Terminology may vary slightly between vendors.

Double check by checking the data sheets

Set-up time , t_{su} : the minimum length of time that a signal must be valid at a circuit input *before* a second triggering signal arrives at a second input.
Usually a clock

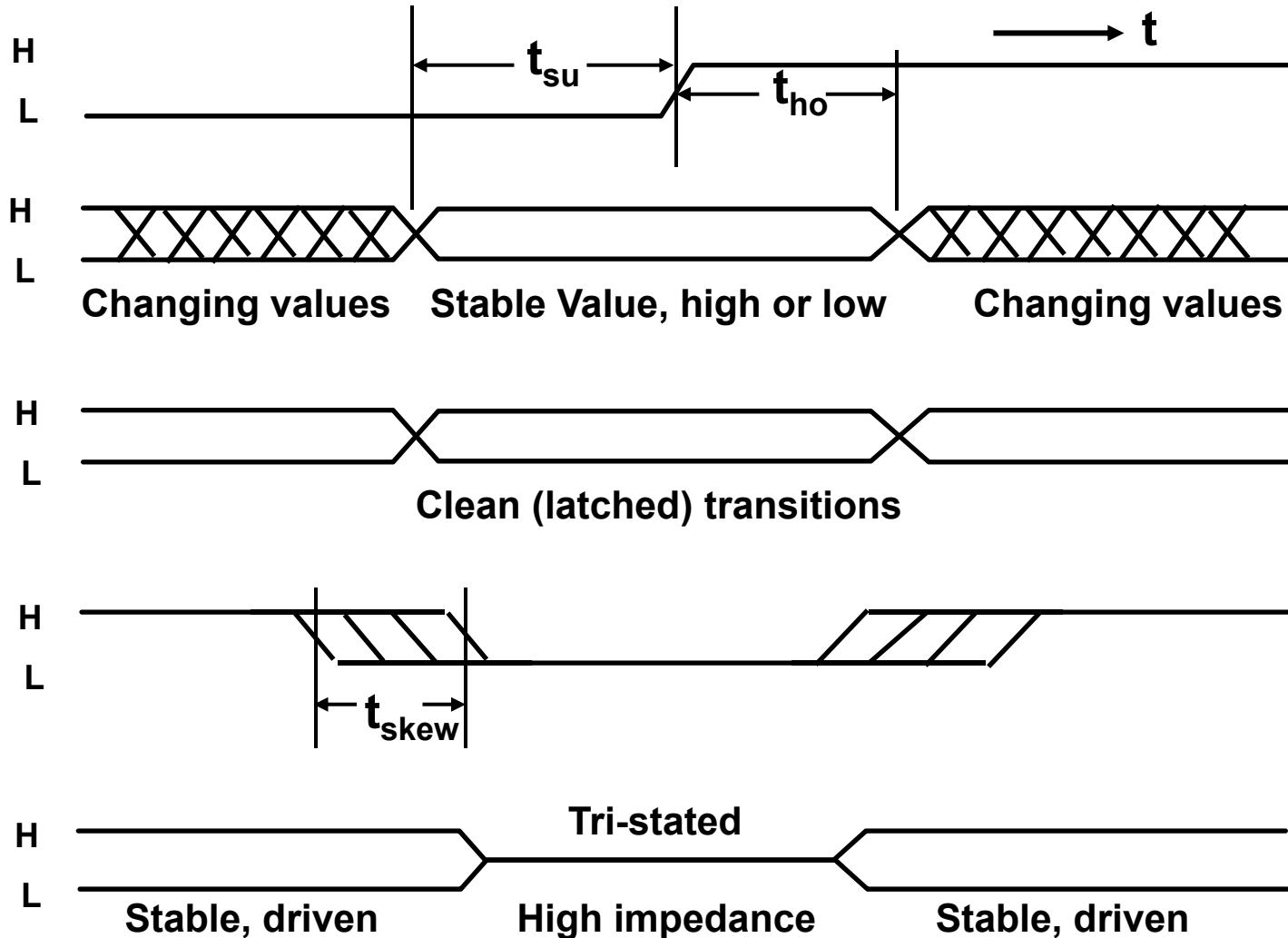
Delay time , t_{co} : the length of time that a circuit requires for its output(s) to begin to change in response to a triggering signal arriving at a second input.

Hold time , t_{ho} : the minimum length of time that a signal must be kept valid at a circuit input *after* a triggering signal has been received at a second input.

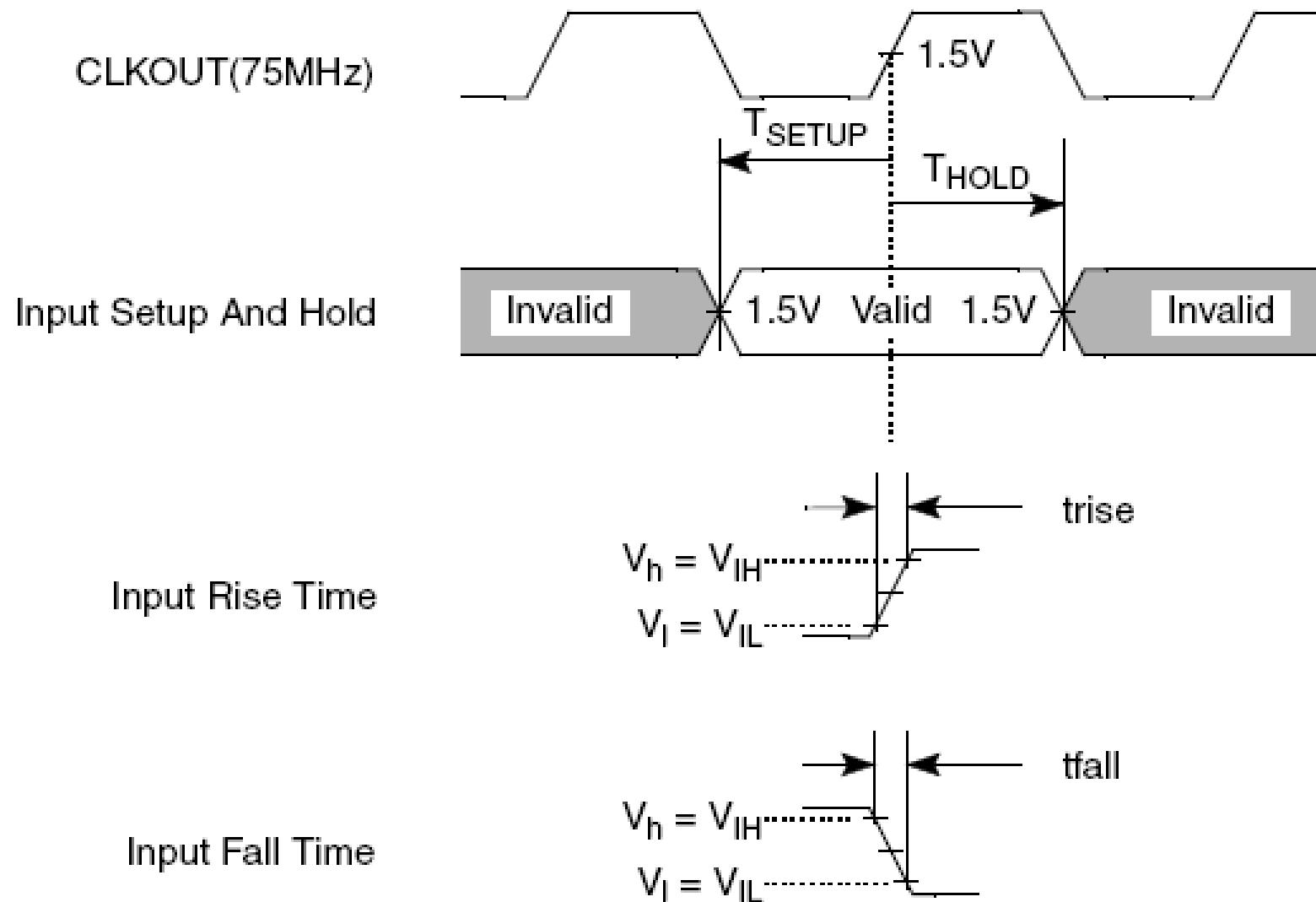
Timing skew , t_{skew} : the maximum range of times over which a particular signal transition (L -> H or H -> L) can occur.

- Due to variations in driver output resistance
- Combinational logic outputs take a while to stabilize

Timing Diagram Notation



Ex: Setup and Hold Time Def'ns for the MCF523x



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

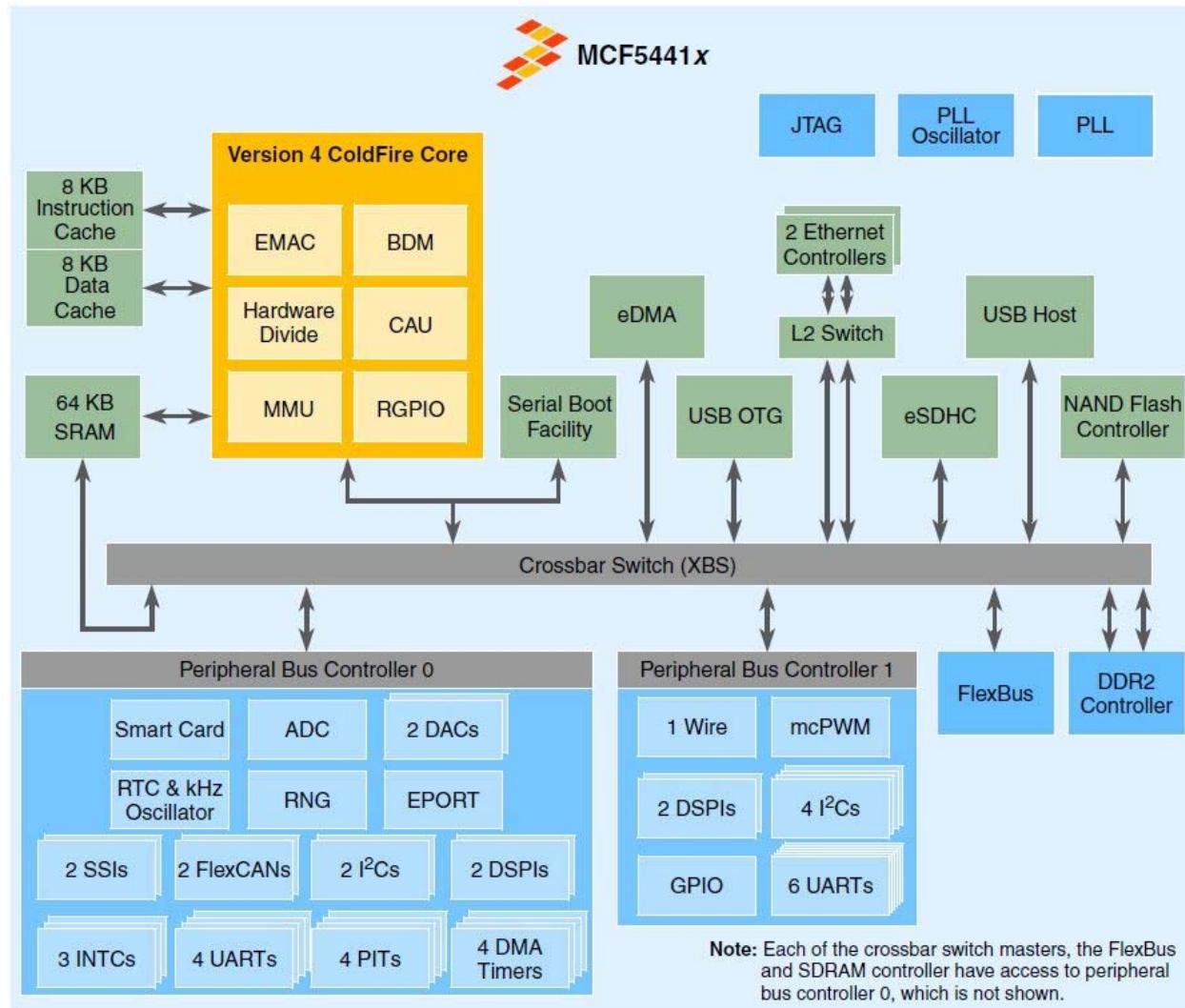
Copyright © 2020 by Bruce Cockburn

12-21

The M68000 Bus

- **Processor-specific** (Note: the 68000 bus later became the basis for the VME backplane bus standard).
- **Semisynchronous**: Bus transactions take a variable number of CPU clock cycles. This provides timing flexibility.
- **Bilateral**: The bus master (CPU) initiates the transaction, but the slave uses data acknowledge (DTACK) to fix the number of cycles that are required for the transaction.
- **16 or 32-bit Data Bus**: Width depends on the processor.
- **24-bit Address Bus**: A0 may be decoded on some CPUs.
e.g. 68000 uses Upper and Lower Data Strobes, no A0.
Only UDS is active when A0 = 0 and one byte is being transferred
Only LDS is active when A0 = 1 and one byte is being transferred.
Both UDS and LDS are active for word and long word transfers.

MCF5441X Architecture



Copyright of Freescale Semiconductor, Inc. 2012.

Copyright © 2020 by Bruce Cockburn

12-23

MCF5441X Pinout Diagram

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	VSS	FB _{AD3}	FB _{AD13}	FB _{AD14}	FB _{AD16}	FB _{AD20}	FB _{AD22}	FB _{AD26}	FB _{AD29}	SDHC _{CLK}	SIMO _{CLK}	SSI0 _{MCLK}	SSI0 _{BCLK}	USBO _{DM}	USBH _{DM}	VSS
B	FB _{CS4}	FB _{AD2}	FB _{AD8}	FB _{AD11}	FB _{AD15}	FB _{AD19}	FB _{AD24}	FB _{AD28}	FB _{AD31}	UART0 _{RXD}	UART0 _{RTS}	SDHC _{DAT0}	SDHC _{DAT3}	USBO _{DP}	USBH _{DP}	RTC _{EXTAL}
C	FB _{BE/BWE3}	FB _{AD1}	FB _{AD7}	FB _{AD9}	FB _{AD10}	FB _{AD17}	FB _{AD23}	FB _{AD30}	UART1 _{RXD}	UART1 _{CTS}	SDHC _{CMD}	SSI0 _{RXD}	SSI0 _{TXD}	SIMO _{PD}	SIMO _{RST}	RTC _{Xtal}
D	FB _{BE/BWE1}	FB _{ALE}	FB _{AD5}	FB _{AD12}	FB _{AD18}	FB _{AD21}	FB _{AD25}	FB _{AD27}	UART1 _{TXD}	UART1 _{RTS}	UART0 _{TXD}	SDHC _{DAT1}	SIMO _{VEN}	CAN1 _{TX}	CAN1 _{RX}	VSS
E	FB _{CS1}	FB _{BE/BW E2}	FB _{AD4}	FB _{AD6}	FB _{VDD}	FB _{VDD}	FB _{VDD}	VSS	IVDD	IVDD	IVDD	SIMO _{XMT}	UART0 _{CTS}	SDHC _{DAT2}	SSI0 _{FS}	VSTBY _{RTC}
F	FB _{OE}	FB _{CS5}	FB _{AD0}	FB _{BE/BWE0}	FB _{VDD}	FB _{VDD}	VSS	VSS	IVDD	IVDD	IVDD	TRQ7	TRQ1	TRQ4	VDD _{OSC_A_PLL}	VSS _{OSC_A_PLL}
G	FB _{CLK}	FB _{R/W}	FB _{CS0}	ADC _{IN4}	FB _{VDD}	VSS	VSS	VSS	VSS	VSS	VSS	VDD _{USB0}	T3IN	I2C0 _{SDA}	I2C0 _{SCL}	EXTAL
H	ADC _{IN0}	ADC _{IN6}	FB _{TA}	AVDD _{ADC}	AVSS _{ADC}	VSS	VSS	EVDD	VSS	VSS	VSS	VDD _{USBH}	T1IN	T2IN	TOIN	XTAL
J	ADC _{IN1}	ADC _{IN2}	ADC _{IN5}	VDDA _{DAC}	VSSA _{DAC}	VSS	EVDD	EVDD	EVDD	EVDD	VSS	VSS	PST3	PST0	PST1	PST2
K	DSPI0 _{SOUT}	DSPI0 _{PCS0}	ADC _{IN7}	ADC _{IN3}	BOOT _{MOD1}	EVDD	EVDD	EVDD	EVDD	EVDD	VSS	TRST	TDO	RESET	TMS	
L	DSPI0 _{PCS1}	DSPI0 _{SCK}	DSPI0 _{SIN}	VSS	BOOT _{MOD0}	EVDD	VSS	VSS	VSS	VSS	VSS	TDI	DDATA0	DDATA3	RST _{OUT}	
M	TRQ3	TRQ2	UART2 _{RTS}	UART2 _{CTS}	VSS	VSS	SD _{VDD}	SD _{VDD}	SD _{VDD}	SD _{VDD}	SD _{VDD}	DDATA2	MII0 _{RXCLK}	DDATA1	TCLK	
N	TRQ6	UART2 _{TXD}	SD _{A5}	SD _{A10}	SD _{A2}	SD _{BA1}	SD _{CS}	SD _{CAS}	SD _{D3}	SD _{VTT}	OW _{IO}	MII0 _{TXD2}	MII0 _{RXD2}	MII0 _{RXER}	JTAG _{EN}	MII0 _{MDI0}
P	UART2 _{RXD}	SD _{A1}	SD _{A9}	SD _{A3}	SD _{A4}	SD _{A14}	SD _{BA2}	SD _{ODT}	SD _{D1}	SD _{VREF}	MII0 _{CRS}	MII0 _{TXEN}	MII0 _{TXD0}	MII0 _{RXDV}	MII0 _{RXD3}	MII0 _{MDC}
R	SD _{A12}	SD _{A7}	SD _{A11}	SD _{A13}	SD _{BA0}	SD _{RAS}	SD _{CKE}	SD _{WE}	SD _{D0}	SD _{D4}	SD _{D6}	MII0 _{COL}	MII0 _{TXD1}	MII0 _{TXER}	MII0 _{RXD1}	TEST
T	VSS	SD _{A6}	SD _{A0}	SD _{A8}	SD _{CLK}	SD _{CLK}	SD _{DM}	SD _{DQS}	SD _{DQS}	SD _{D2}	SD _{D5}	SD _{D7}	MII0 _{TXD3}	MII0 _{TXCLK}	MII0 _{RXD0}	VSS

Copyright of Freescale Semiconductor, Inc. 2012.

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

MCF5441X FlexBus

- The FlexBus interface allows the MCF5441X to connect with a wide variety of external devices (e.g., boot ROM, program code in flash memory, FPGAs, simple peripherals) without the need for additional “glue logic”.
- There is a *time-multiplexed* 32-bit address/data bus, which can be configured to pass data to 8-bit, 16-bit or 32-bit external devices. In the first clock cycle, the multiplexed bus carries the address; in the second (and more) clock cycle(s), the data is passed on the selected number of bytes.
- Four byte strobes are provided for the address/data bus.
- Six programmable chip selects (CSs) are available that are asserted for different address ranges, data bit widths (8, 16 or 32 bits), different address set and hold times w.r.t. CS assertion, and different numbers of wait states.

MCF5441X FlexBus Signals

Table 20-1. FlexBus Signal Summary

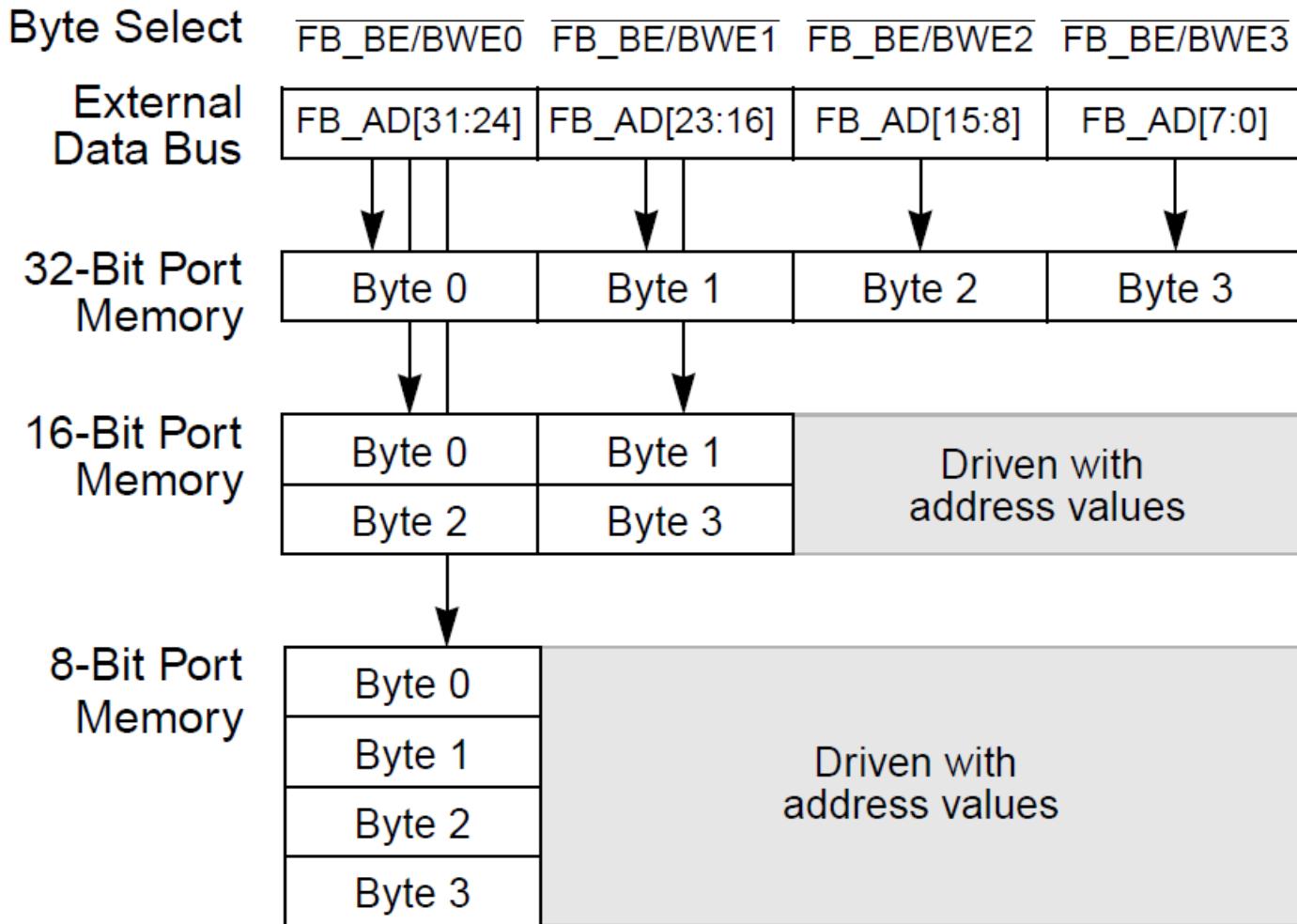
Signal Name	I/O ¹	Description
FB_AD[31:0]	I/O	Address/data bus, FB_AD[31:0].
FB_CS[5:0]	O	General purpose chip-selects. The actual number of chip selects available depends upon the device and its pin configuration. See Table 2-2 for more details.
FB_BE/BWE[3:0]	O	Byte enable/byte write enable
FB_OE	O	Output enable
FB_R/W	O	Read/write. 1 = Read, 0 = Write
FB_ALE	O	Address latch enable
FB_TSIZ[1:0]	O	Transfer size
FB_TBST	O	Burst transfer indicator
FB_TA	I	Transfer acknowledge

FB_TSIZ[1:0]	Transfer Size
00	4 bytes (longword)
01	1 byte
10	2 bytes (word)
11	16 bytes (line)

¹ Because this device shares the FlexBus signals with the NAND flash controller, these signal directions are only valid when the FlexBus controls them. The directions may change during NAND flash cycles.

Copyright of Freescale Semiconductor, Inc. 2012.

MCF5441X FlexBus Signals



Copyright of Freescale Semiconductor, Inc. 2012.

Multiplexed Address/Data Bus Patterns

Port Size and Phase		FB_AD			
		[31:24]	[23:16]	[15:8]	[7:0]
32-bit	Address phase	Address			
	Data phase	Data			
16-bit	Address phase	Address			
	Data phase	Data		Address	
8-bit	Address phase	Address			
	Data phase	Data	Address		

Copyright of Freescale Semiconductor, Inc. 2012.

FlexBus Timing Specs

Table 16. FlexBus timing specifications

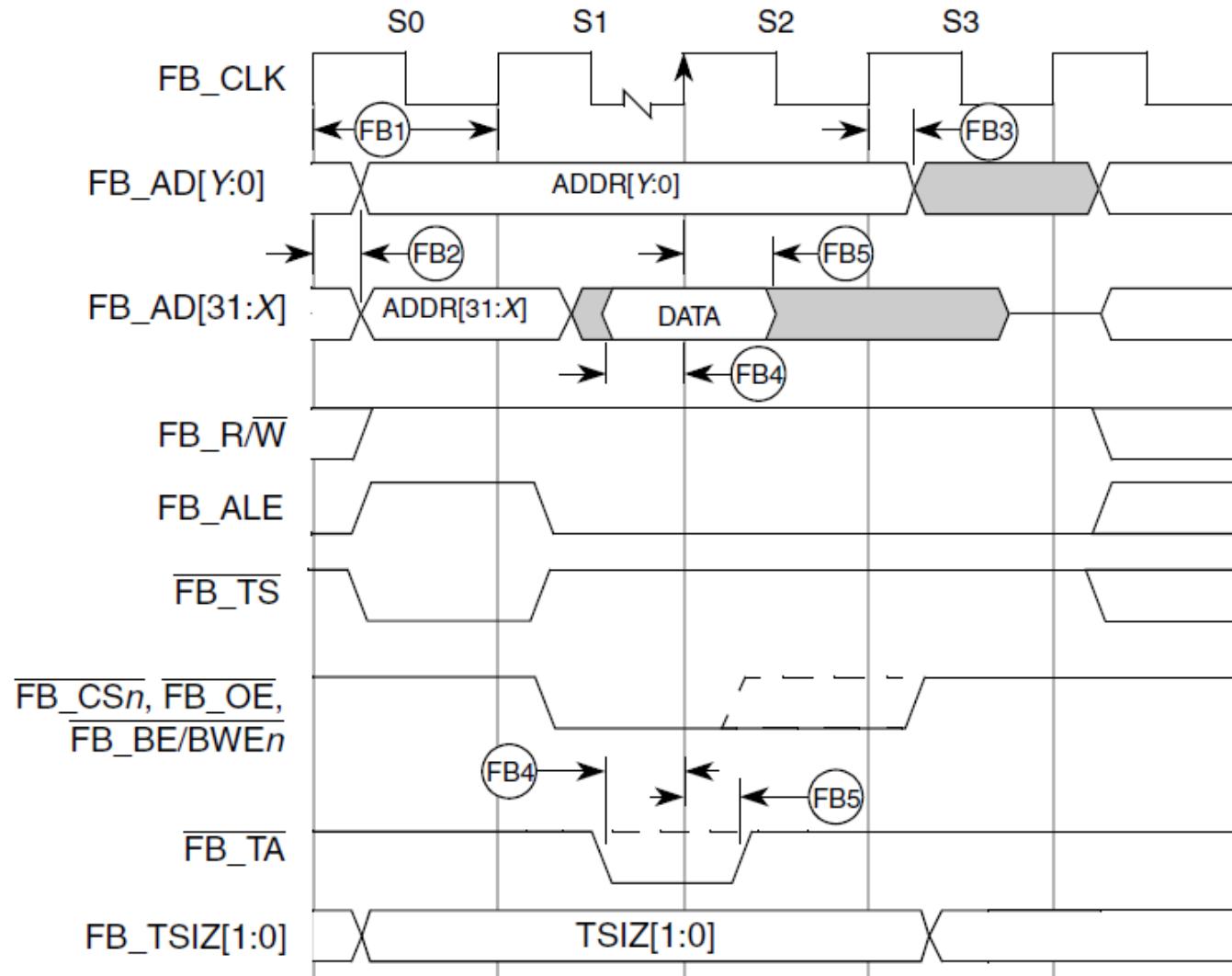
Num	Characteristic	Min	Max	Unit	Notes
	Frequency of operation	—	62.5	MHz	
FB1	Clock period	16	—	ns	
FB2	Output valid	—	6.0	ns	1
FB3	Output hold	0.5	—	ns	1
FB4	Input setup	5.5	—	ns	2
FB5	Input hold	0	—	ns	2

¹ Specification is valid for all FB_AD[31:0], FB_R/W, FB_ALE, FB_TS, FB_CS_n, FB_OE, FB_BE/BWE_n, and FB_TSIZ[1:0].

² Specification is valid for all FB_AD[31:0] and FB_TA.

Copyright of Freescale Semiconductor, Inc. 2012.

FlexBus Read Cycle (1)



Copyright of Freescale Semiconductor, Inc. 2012.

FlexBus Read Cycle (2)

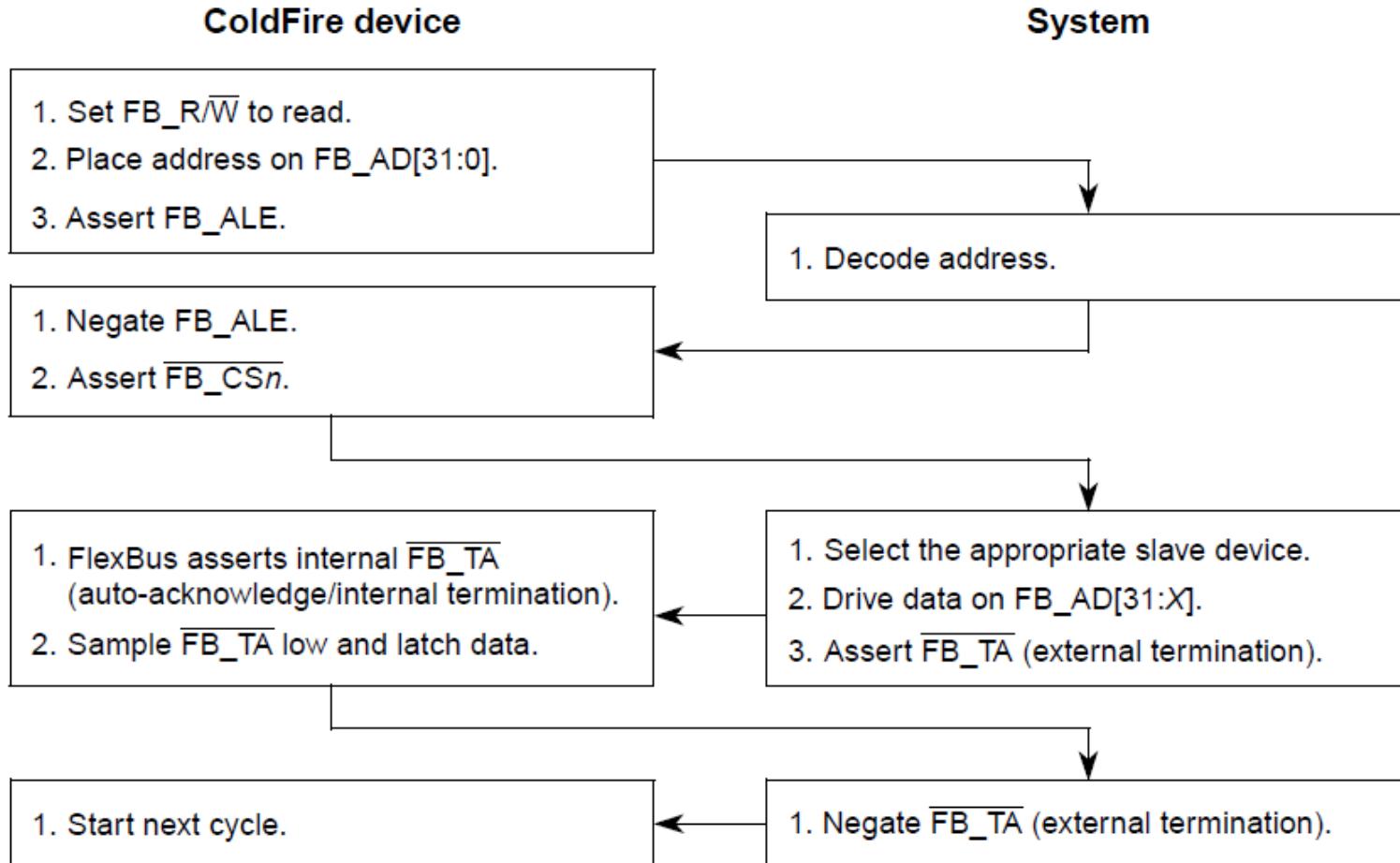
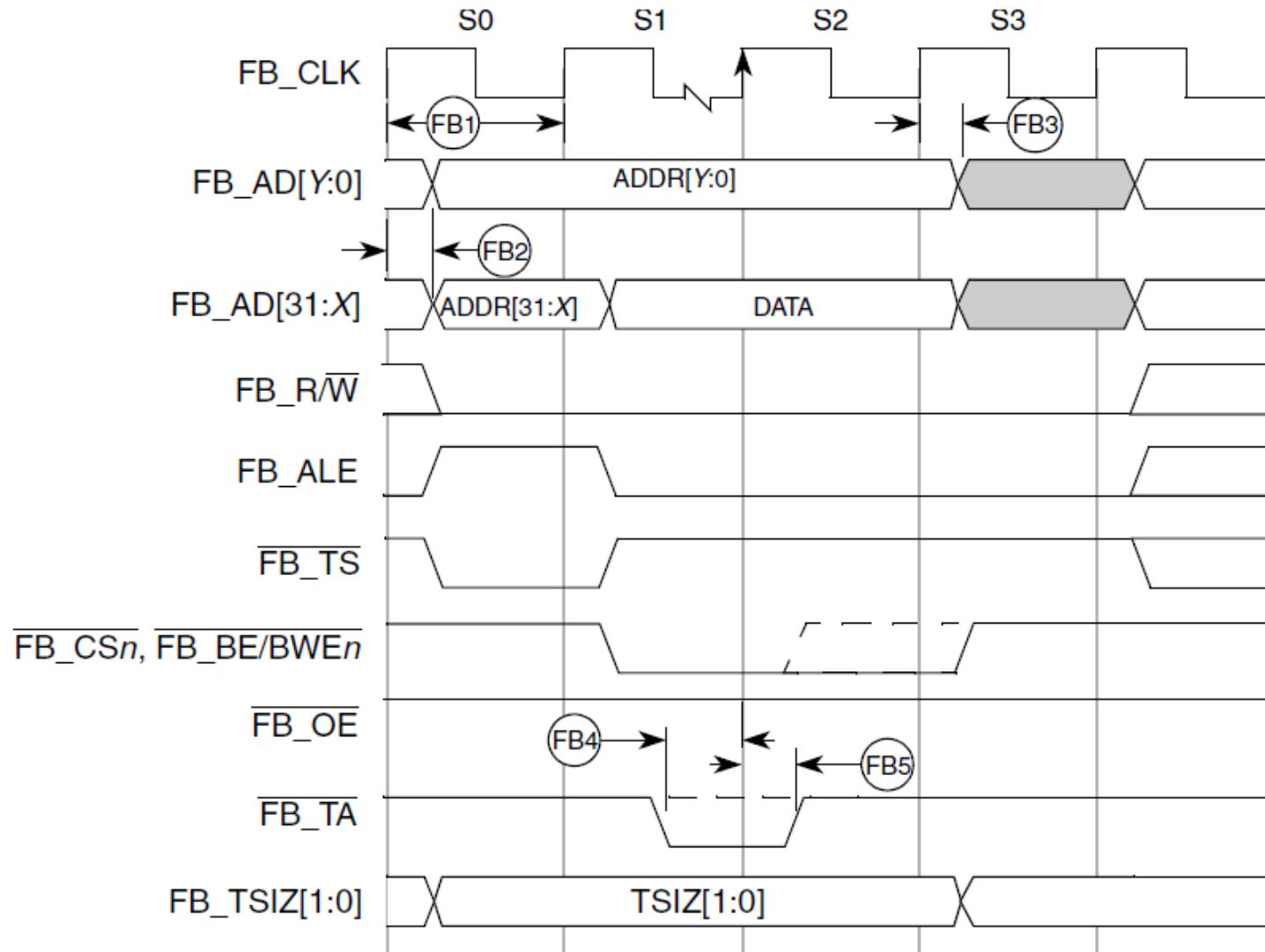


Figure 20-7. Read Cycle Flowchart

Copyright of Freescale Semiconductor, Inc. 2012.

FlexBus Write Cycle (1)



Copyright of Freescale Semiconductor, Inc. 2012.

FlexBus Write Cycle (2)

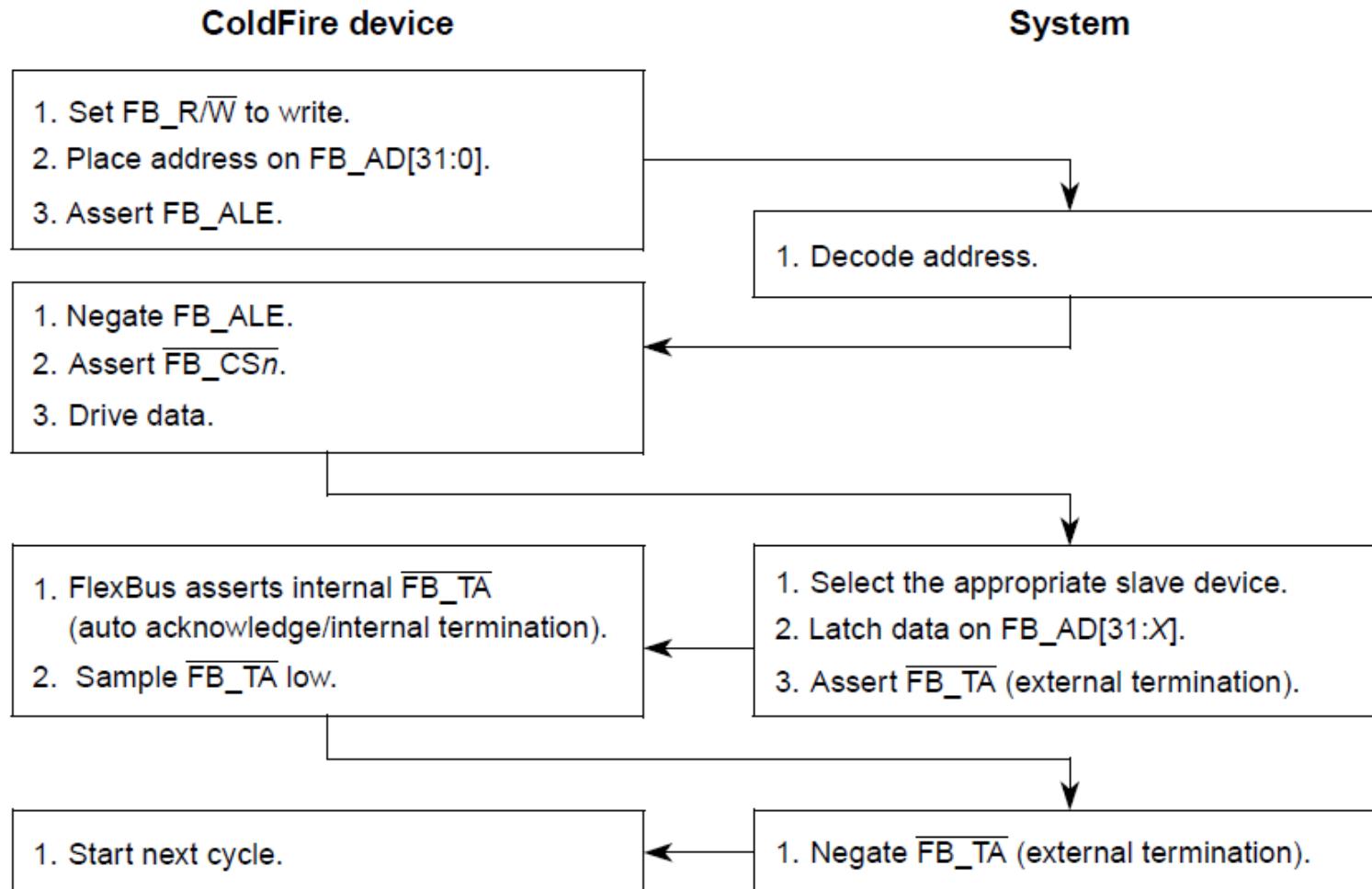


Figure 20-9. Write-Cycle Flowchart

Copyright of Freescale Semiconductor, Inc. 2012.

Copyright © 2020 by Bruce Cockburn

12-33

FlexBus Longword Burst Read from 8-bit Port

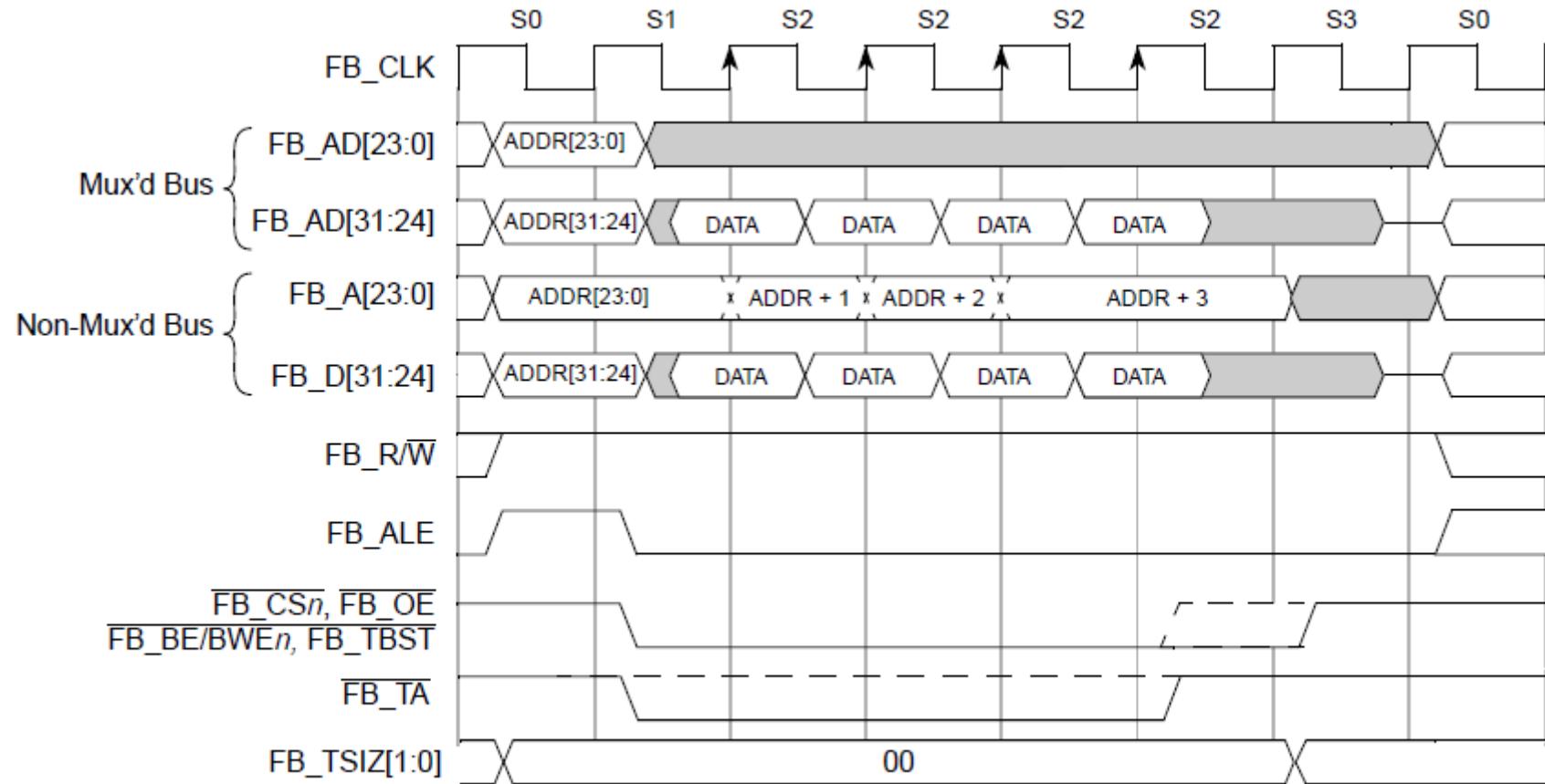


Figure 20-26. Longword-Read Burst from 8-Bit Port 2-1-1-1 (No Wait States)

Copyright of Freescale Semiconductor, Inc. 2012.

FlexBus Longword Burst Write to 8-bit Port

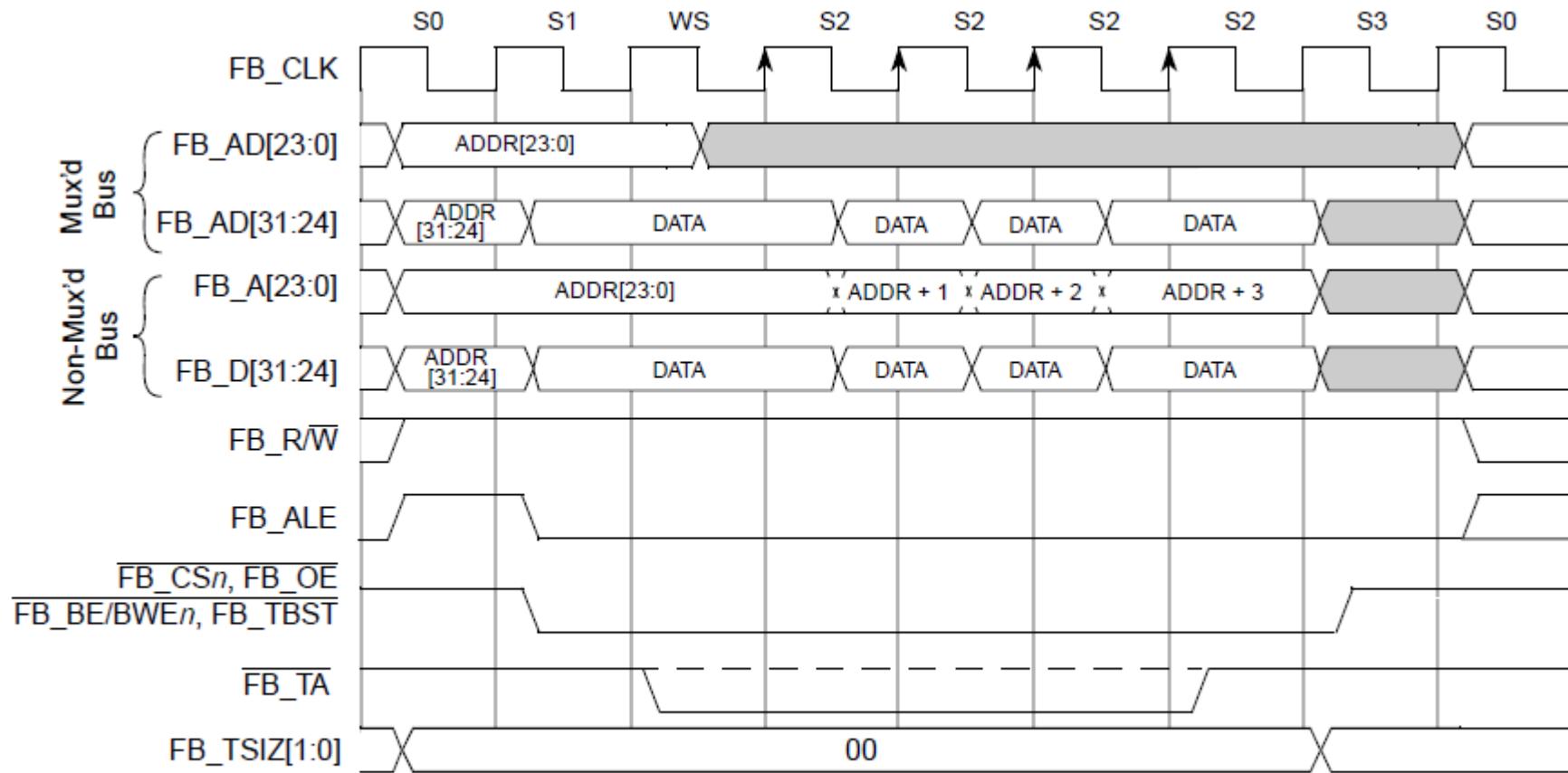


Figure 20-27. Longword-Write Burst to 8-Bit Port 3-1-1-1 (No Wait States)

Copyright of Freescale Semiconductor, Inc. 2012.

The Joint Test Action Group (JTAG) Port

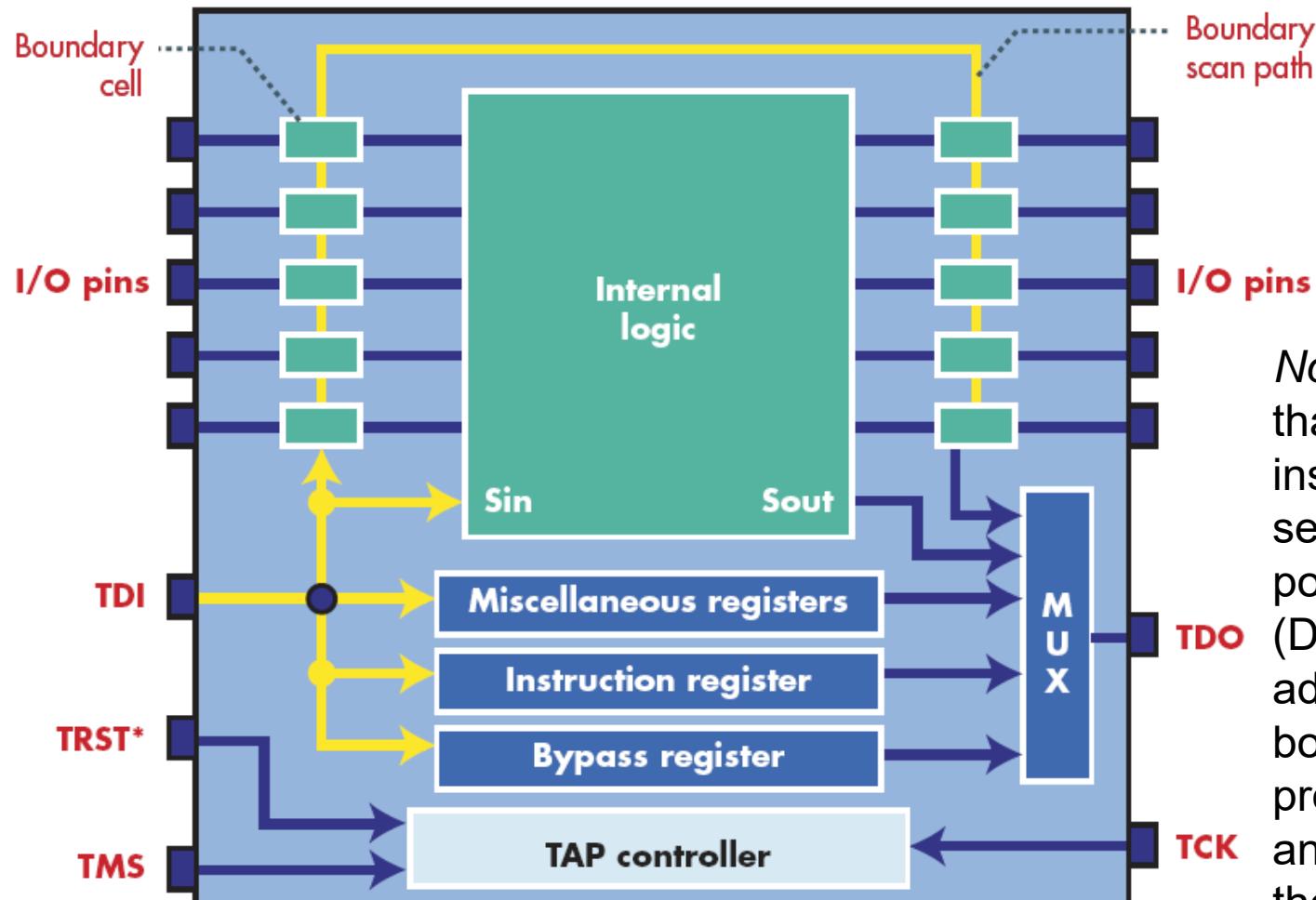
- In 1985 an industry group, called the "Joint Test Action Group" began the development of standard way of accessing chips for production test purposes (to enter test modes, apply tests & retrieve test results).
- The result in 1990 was *IEEE Std. 1149.1-1990 "Standard Test Access Port and Boundary Scan Architecture"*.
- The standard specifies a 4 (optionally a 5) pin test bus interface that is commonly called a "JTAG port". The MCF523x has the following:

Signal Name	Abbreviation	Function	I/O
Test Reset	$\overline{\text{TRST}}$	This active-low signal is used to initialize the JTAG logic asynchronously.	I
Test Clock	TCLK	Used to synchronize the JTAG logic.	I
Test Mode Select	TMS	Used to sequence the JTAG state machine. TMS is sampled on the rising edge of TCLK.	I
Test Data Input	TDI	Serial input for test instructions and data. TDI is sampled on the rising edge of TCLK.	I
Test Data Output	TDO	Serial output for test instructions and data. TDO is three-stateable and is actively driven in the shift-IR and shift-DR controller states. TDO changes on the falling edge of TCLK.	O

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

JTAG Port (IEEE Std 1149.1) Architecture

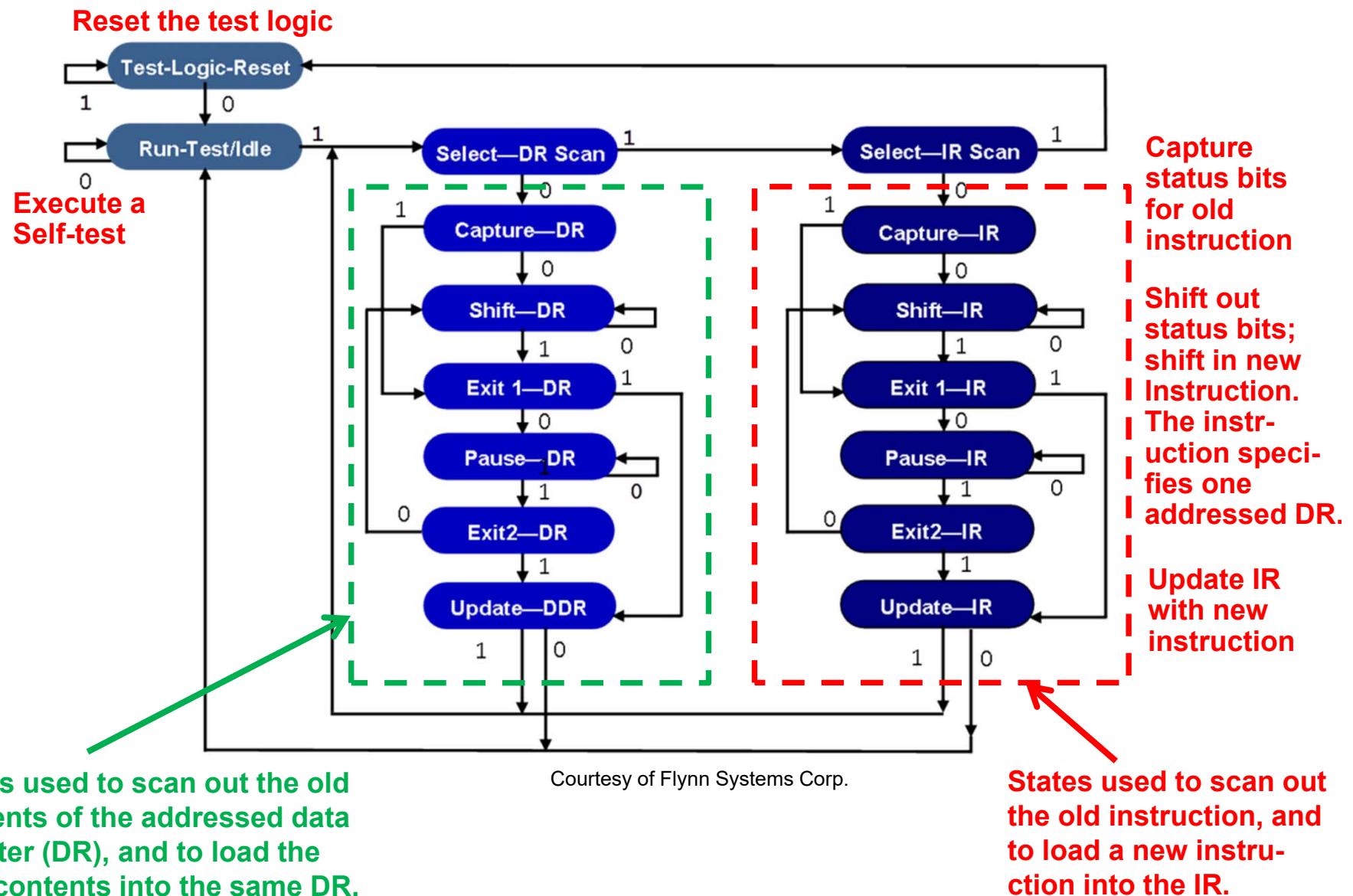
Basic IC architecture of IEEE 1149.1.



Courtesy of Texas Instruments Inc., 2008.

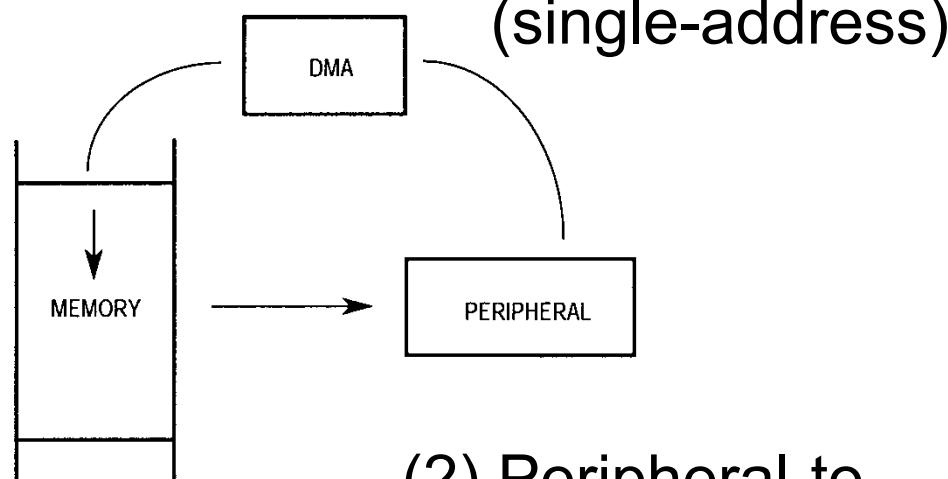
Note: The instruction that is loaded into the instruction register (IR) selects one of several possible data registers (DRs) to be the one addressed DR. The boundary cells, which provide observability and controllability over the I/O pins, form the boundary scan data register.

The JTAG TAP Controller State Machine

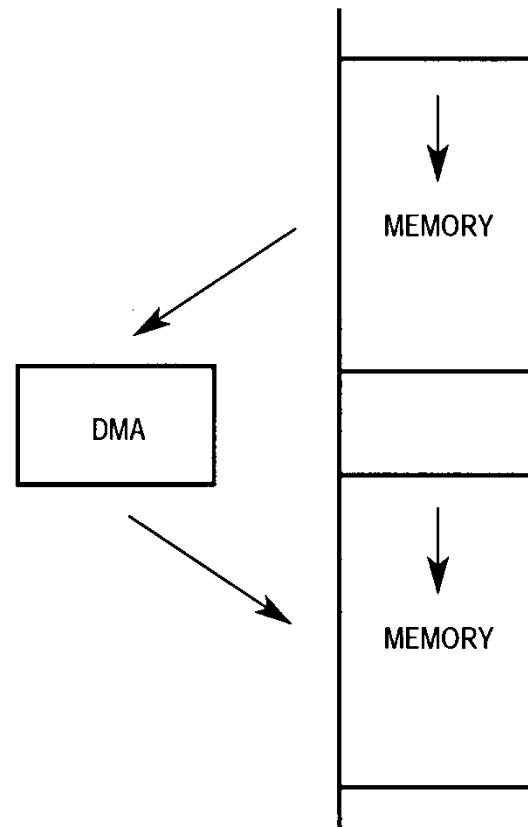


Direct Memory Access (DMA) Data Movements

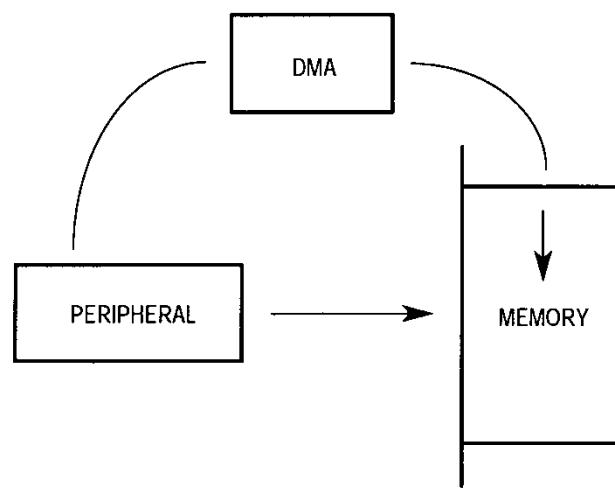
(1) Memory-to-peripheral
(single-address)



(3) Memory-to-memory
(dual-address)



(2) Peripheral-to-
memory
(single-
address)



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

Copyright © 2020 by Bruce Cockburn

12-39

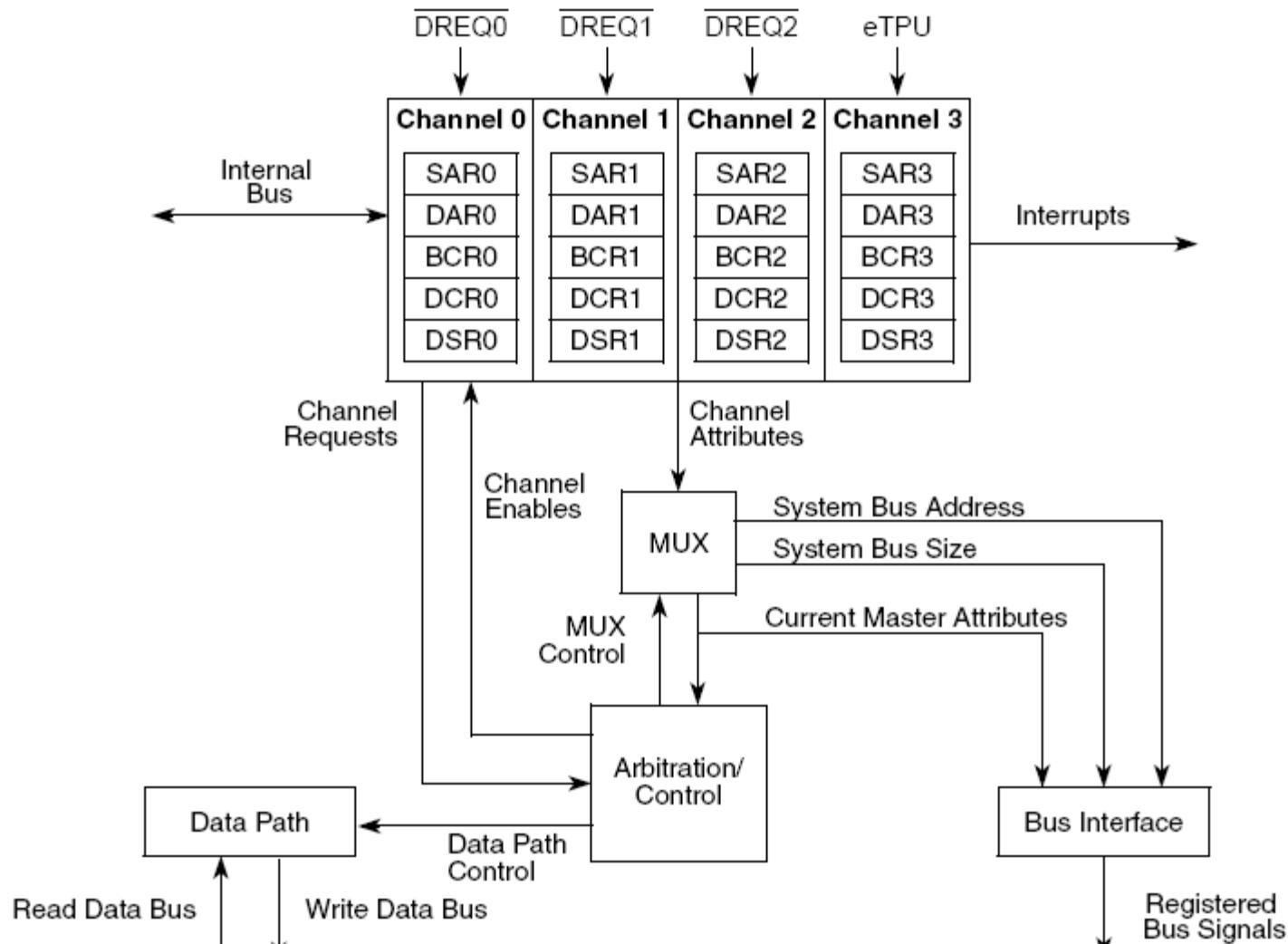
Direct Memory Access in the MCF5441X

- 64 fully programmable channels with independent control descriptors.
- Data movement using dual-address (source & destination) transfers for 8-, 16-, 32- and 128-bit data values.
- Support for nested minor loops (byte counts) and major loops (number of minor loop iterations).
- Can be linked to external request/acknowledge pins.
- Support for channel-to-channel linking for continuous data transfers.
- Support for scatter/gather data patterns.
- Channel transfers can be assigned fixed priorities, or can be assigned round-robin priorities.
- DMA transfers can be triggered by (1) CPU requests, (2) linked channel transfers, and (3) external DMA requests.

Direct Memory Access in the MCF5234

- Four general-purpose Direct Memory Access channels are provided in the MCF5234, which are called: DMA0, DMA1, DMA2 and DMA3.
- Each channel can independently move data over the system bus as bytes, words, longwords, or 16-byte "lines". The source and destinations in each DMA channel can have different data widths.
- Each channel has: (1) a *source address register* (SARn), (2) a *destination address register* (DARn), (3) a *byte count register* (BCRn), (4) a *control register* (DCRn), and (5) a *status register* (DSRn).
- The four DMA channels can receive DMA requests from a variety of possible sources:
 - by the CPU writing the START bit in a DCRn, n = 0, 1, 2 or 3
 - three on-chip UART (serial communications interfaces)
 - four on-chip 32-bit hardware timers
 - the eTPU (treated as if it were an off-chip DMA request)
 - three DMA request signals from off-chip: DREQ0, DREQ1, DREQ2

Architecture of the MCF5234 DMA Module



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

Steps in a DMA Data Transfer

1. **Channel Initialization:** The channel registers are loaded with control information, the initial source address, the initial destination address, and the byte count.
2. **Data Transfer:**
 - Requests for data transfer are received from the CPU, the eTPU, a UART, a hardware timer, or an external peripheral.
 - For each request, data bytes are transferred from the source address to the destination address.
 - One or both addresses are incremented.
 - The data width may be different for the source and destination.
 - The system bus is shared during the transfer using either the cycle-stealing mode or the continuous transfer mode.
 - A 24-bit byte counter is decremented by 1, 2, 4 or 16 to eventually determine when the data transfer has finished.
3. **Channel Termination:** The channel status register is updated. A hardware interrupt may be produced to alert the CPU.

DMA Module Register Map

DMA Channel	IPSBAR Offset	[31:24]	[23:16]	[15:8]	[7:0]			
—	0x00_0014	DMA Request Control Register (DMAREQC) ¹						
0	0x00_0100	Source Address Register 0 (SAR0)						
	0x00_0104	Destination Address Register 0 (DAR0)						
	0x00_0108	Status Register 0 (DSR0)	Byte Count Register 0 (BCR0)					
	0x00_010C	Control Register 0 (DCR0)						
1	0x00_0110	Source Address Register 1 (SAR1)						
	0x00_0114	Destination Address Register 1 (DAR1)						
	0x00_0118	Status Register 1 (DSR1)	Byte Count Register 1 (BCR1)					
	0x00_011C	Control Register 1 (DCR1)						
2	0x00_0120	Source Address Register 2 (SAR2)						
	0x00_0124	Destination Address Register 2 (DAR2)						
	0x00_0128	Status Register 2 (DSR2)	Byte Count Register 2 (BCR2)					
	0x00_012C	Control Register 2 (DCR2)						
3	0x00_0130	Source Address Register 3 (SAR3)						
	0x00_0134	Destination Address Register 3 (DAR3)						
	0x00_0138	Status Register 3 (DSR3)	Byte Count Register 3 (BCR3)					
	0x00_013C	Control Register 3 (DCR3)						

¹ Located within the SCM, but listed here for clarity.

DMA Request Control Register (DMAREQC)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	DMAREQC_EXT
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	DMAC3				DMAC2				DMAC1				DMAC0			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x00_0014															

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

- Entries of value 1 in the DMAREQC_EXT field determine which of the eTPU and off-chip signals DREQ2-0 can request DMA transfers using channels DMAC3-0, respectively.
- If a DMARREQC_EXT bit is 0, then the source of DMA requests for that channel is determined by the corresponding DMACn field.

DMAREQC Bits 31-16

Bits	Name	Description															
31–20	—	Reserved, should be cleared.															
19-16	DMAREQC_EXT	<p>DMA request control for external (off-chip) and eTPU requests. The DMAREQC_EXT[3:0] bits correspond to DMA channels 3, 2, 1, and 0. If set, the corresponding DMACn bit field is ignored. If cleared, refer to the appropriate DMACn bit field for configuring the internal DMA requestor.</p> <p>DMAREQC_EXT[3] controls the eTPU request, while DMAREQC_EXT[2:0] controls the external DMA request/acknowledge signals. In order for an external or eTPU request to activate a DMA channel the corresponding DCRn[EEXT] bit must be set as well.</p> <table border="1"><thead><tr><th></th><th>DMAREQC_EXT[3]</th><th>DMAREQC_EXT[2]</th><th>DMAREQC_EXT[1]</th><th>DMAREQC_EXT[0]</th></tr></thead><tbody><tr><td>0</td><td>See DMAC3</td><td>See DMAC2</td><td>See DMAC1</td><td>See DMAC0</td></tr><tr><td>1</td><td>eTPU</td><td>External DREQ2</td><td>External DREQ1</td><td>External DREQ0</td></tr></tbody></table> <p>Note: GPIO must be configured to enable external DMA requests.</p>		DMAREQC_EXT[3]	DMAREQC_EXT[2]	DMAREQC_EXT[1]	DMAREQC_EXT[0]	0	See DMAC3	See DMAC2	See DMAC1	See DMAC0	1	eTPU	External DREQ2	External DREQ1	External DREQ0
	DMAREQC_EXT[3]	DMAREQC_EXT[2]	DMAREQC_EXT[1]	DMAREQC_EXT[0]													
0	See DMAC3	See DMAC2	See DMAC1	See DMAC0													
1	eTPU	External DREQ2	External DREQ1	External DREQ0													

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

DMAREQC Bits 15-0

Bits	Name	Description
15–0	DMAC n	<p>DMA channel n. Each four bit field defines the logical connection between the DMA requestors and that DMA channel. There are ten possible requesters (4 DMA Timers and 6 UARTs). Any request can be routed to any of the DMA channels. Effectively, the DMAREQC provides a software-controlled routing matrix of the 10 DMA request signals to the 4 channels of the DMA module. DMAC3 controls DMA channel 3, DMAC2 controls DMA channel 2, etc.</p> <p>0100 DMA Timer 0. 0101 DMA Timer 1. 0110 DMA Timer 2. 0111 DMA Timer 3. 1000 UART0 Receive. 1001 UART1 Receive. 1010 UART2 Receive. 1100 UART0 Transmit. 1101 UART1 Transmit. 1110 UART2 Transmit.</p> <p>All other values are reserved and will not generate a DMA request.</p>

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

DMA Channel Control Register, DCRn

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INT	EEXT	CS	AA	BWC			0	0	SINC	SSIZE	DINC	DSIZE		0	
W															START	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SMOD				DMOD				D_REQ	0	LINKCC	LCH1	LCH2			
W										0						
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x00_010C (DMA0); IPSBAR + 0x011C (DMA1); IPSBAR + 0x012C (DMA2); IPSBAR + 0x013C (DMA3)															

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

Bits	Name	Description
31	INT	Interrupt on completion of transfer. Determines whether an interrupt is generated by completing a transfer or by the occurrence of an error condition. 0 No interrupt is generated. 1 Internal interrupt signal is enabled.
30	EEXT	Enable external request. Care should be taken because a collision can occur between the START bit and DREQn when EEXT = 1. 0 External request is ignored. 1 Enables external request to initiate transfer. The internal request (initiated by setting the START bit) is always enabled.

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

DMA Channel Control Register, DCRn

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INT	EEXT	CS	AA	BWC			0	0	SINC	SSIZE	DINC	DSIZE	0		
W														START		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SMOD				DMOD				D_REQ	0	LINKCC	LCH1		LCH2		
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x00_010C (DMA0); IPSBAR + 0x011C (DMA1); IPSBAR + 0x012C (DMA2); IPSBAR + 0x013C (DMA3)															

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

Bits	Name	Description
29	CS	Cycle steal. 0 DMA continuously makes read/write transfers until the BCR decrements to 0. 1 Forces a single read/write transfer per <u>request</u> . The request may be internal by setting the START bit, or external by asserting DREQn.
28	AA	Auto-align. AA and SIZE determine whether the source or destination is auto-aligned, that is, transfers are optimized based on the address and size. See Section 14.4.4.2, "Auto-Alignment." 0 Auto-align disabled 1 If SSIZE indicates a transfer no smaller than DSIZE, source accesses are auto-aligned; otherwise, destination accesses are auto-aligned. Source alignment takes precedence over destination alignment. If auto-alignment is enabled, the appropriate address register increments, regardless of DINC or SINC.

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

DMA Channel Control Register, DCRn

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INT	EEXT	CS	AA	BWC			0	0	SINC	SSIZE	DINC	DSIZE	0		
W														START		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SMOD				DMOD			D_REQ	0	LINKCC	LCH1	LCH2				
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x00_010C (DMA0); IPSBAR + 0x011C (DMA1); IPSBAR + 0x012C (DMA2); IPSBAR + 0x013C (DMA3)															

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

27–25	BWC	Bandwidth control. Indicates the number of bytes in a block transfer. When the byte count reaches a multiple of the BWC value, the DMA releases the bus.	
		BWC	Number of kilobytes per block
		000	DMA has priority and does not negate its request until transfer completes.
		001	16 Kbytes
		010	32 Kbytes
		011	64 Kbytes
		100	128 Kbytes
		101	256 Kbytes
		110	512 Kbytes
		111	1024 Kbytes

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

DMA Channel Control Register, DCRn

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INT	EEXT	CS	AA	BWC			0	0	SINC	SSIZE		DINC	DSIZE	0	
W															START	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SMOD				DMOD			D_REQ	0	LINKCC	LCH1		LCH2			
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x00_010C (DMA0); IPSBAR + 0x011C (DMA1); IPSBAR + 0x012C (DMA2); IPSBAR + 0x013C (DMA3)															

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

24-23	—	Reserved, should be cleared.
22	SINC	Source increment. Controls whether a source address increments after each successful transfer. 0 No change to SAR after a successful transfer. 1 The SAR increments by 1, 2, 4, or 16, as determined by the transfer size.
21–20	SSIZE	Source size. Determines the data size of the source bus cycle for the DMA control module. 00 Longword 01 Byte 10 Word 11 Line (16-byte burst)
19	DINC	Destination increment. Controls whether a destination address increments after each successful transfer. 0 No change to the DAR after a successful transfer. 1 The DAR increments by 1, 2, 4, or 16, depending upon the size of the transfer.

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

DMA Channel Control Register, DCRn

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INT	EEXT	CS	AA	BWC		0	0	SINC	SSIZE	DINC	DSIZE	0			
W														START		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SMOD				DMOD			D_REQ	0	LINKCC	LCH1		LCH2			
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x00_010C (DMA0); IPSBAR + 0x011C (DMA1); IPSBAR + 0x012C (DMA2); IPSBAR + 0x013C (DMA3)															

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

Bits	Name	Description
18–17	DSIZE	Destination size. Determines the data size of the destination bus cycle for the DMA controller. 00 Longword 01 Byte 10 Word 11 Line (16-byte burst)
16	START	Start transfer. 0 DMA inactive 1 The DMA begins the transfer in accordance to the values in the control registers. START is cleared automatically after one system clock and is always read as logic 0.

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

DMA Channel Control Register, DCRn

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INT	EEXT	CS	AA	BWC			0	0	SINC	SSIZE		DINC	DSIZE	0	
W															START	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SMOD				DMOD			D_REQ	0	LINKCC	LCH1		LCH2			
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x00_010C (DMA0); IPSBAR + 0x011C (DMA1); IPSBAR + 0x012C (DMA2); IPSBAR + 0x013C (DMA3)															

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

15–12	SMOD	Source address modulo. Defines the size of the source data circular buffer used by the DMA Controller. If enabled (SMOD is non-zero), the buffer base address will be located on a boundary of the buffer size. The value of this boundary is based upon the initial source address (SAR).												
		<table border="1"> <thead> <tr> <th>SMOD</th><th>Circular Buffer Size</th></tr> </thead> <tbody> <tr> <td>0000</td><td>Buffer Disabled</td></tr> <tr> <td>0001</td><td>16 Bytes</td></tr> <tr> <td>0010</td><td>32 Bytes</td></tr> <tr> <td>...</td><td>...</td></tr> <tr> <td>1111</td><td>256 Kbytes</td></tr> </tbody> </table>	SMOD	Circular Buffer Size	0000	Buffer Disabled	0001	16 Bytes	0010	32 Bytes	1111	256 Kbytes
SMOD	Circular Buffer Size													
0000	Buffer Disabled													
0001	16 Bytes													
0010	32 Bytes													
...	...													
1111	256 Kbytes													

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

DMA Channel Control Register, DCRn

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INT	EEXT	CS	AA	BWC		0	0	SINC	SSIZE	DINC	DSIZE			0	
W															START	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SMOD				DMOD			D_REQ	0	LINKCC	LCH1		LCH2			
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address	IPSBAR + 0x00_010C (DMA0); IPSBAR + 0x011C (DMA1); IPSBAR + 0x012C (DMA2); IPSBAR + 0x013C (DMA3)															

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

11-8	DMOD	Destination address modulo. Defines the size of the destination data circular buffer used by the DMA Controller. If enabled (DMOD value is non-zero), the buffer base address will be located on a boundary of the buffer size. The value of this boundary depends on the initial destination address (DAR).												
		<table border="1"> <thead> <tr> <th>DMOD</th> <th>Circular Buffer Size</th> </tr> </thead> <tbody> <tr> <td>0000</td> <td>Buffer Disabled</td> </tr> <tr> <td>0001</td> <td>16 Bytes</td> </tr> <tr> <td>0010</td> <td>32 Bytes</td> </tr> <tr> <td>...</td> <td>...</td> </tr> <tr> <td>1111</td> <td>256 Kbytes</td> </tr> </tbody> </table>	DMOD	Circular Buffer Size	0000	Buffer Disabled	0001	16 Bytes	0010	32 Bytes	1111	256 Kbytes
DMOD	Circular Buffer Size													
0000	Buffer Disabled													
0001	16 Bytes													
0010	32 Bytes													
...	...													
1111	256 Kbytes													

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

DMA Channel Status Registers, DSRn

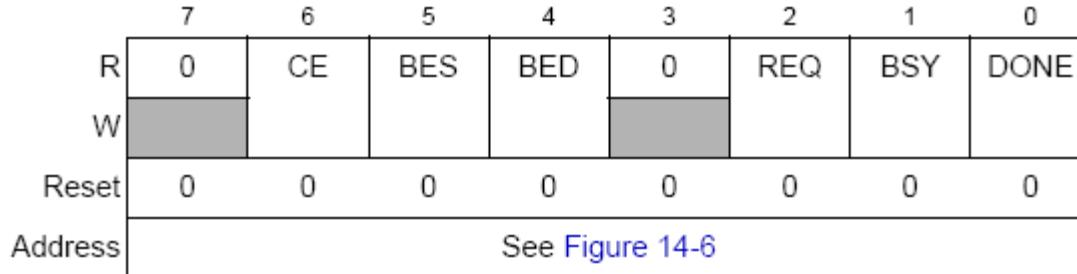
	7	6	5	4	3	2	1	0
R	0	CE	BES	BED	0	REQ	BSY	DONE
W								
Reset	0	0	0	0	0	0	0	0
Address	See Figure 14-6							

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

Bits	Name	Description
7	—	Reserved, should be cleared.
6	CE	Configuration error. Occurs when BCR, SAR, or DAR does not match the requested transfer size, or if BCR = 0 when the DMA receives a start condition. CE is cleared at hardware reset or by writing a 1 to DSR[DONE]. 0 No configuration error exists. 1 A configuration error has occurred.
5	BES	Bus error on source 0 No bus error occurred. 1 The DMA channel terminated with a bus error during the read portion of a transfer.
4	BED	Bus error on destination 0 No bus error occurred. 1 The DMA channel terminated with a bus error during the write portion of a transfer.
3	—	Reserved, should be cleared.
2	REQ	Request 0 No request is pending or the channel is currently active. Cleared when the channel is selected. 1 The DMA channel has a transfer remaining and the channel is not selected.

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

DMA Channel Status Registers, DSRn



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

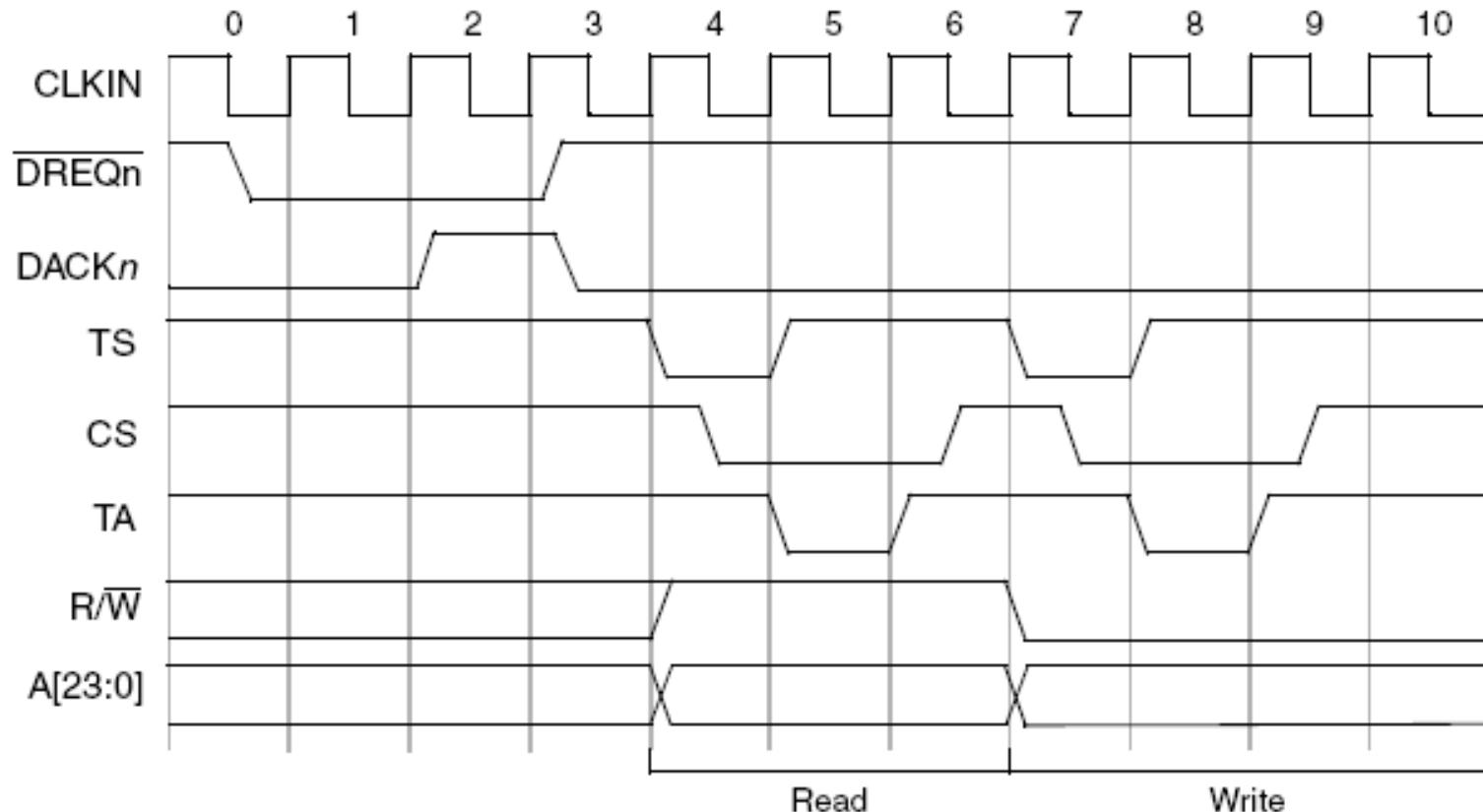
Bits	Name	Description
1	BSY	Busy 0 DMA channel is inactive. Cleared when the DMA has finished the last transaction. 1 BSY is set the first time the channel is enabled after a transfer is initiated.
0	DONE	Transactions done. Set when all DMA controller transactions complete, as determined by transfer count or error conditions. When BCR reaches zero, DONE is set when the final transfer completes successfully. DONE can also be used to abort a transfer by resetting the status bits. When a transfer completes, software must clear DONE before reprogramming the DMA. 0 Writing or reading a 0 has no effect. 1 DMA transfer completed. Writing a 1 to this bit clears all DMA status bits and can be used in an interrupt handler to clear the DMA interrupt and error bits.

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

Cycle-Stealing Mode vs. Continuous Mode

- DMA transfers must share the system bus with the CPU, and the sharing mechanism must be fast to preserve the advantages of fast DMA data transfer
- Two different bus sharing modes are present in the MCF5234 microcontroller:
- **Cycle-Stealing Mode** ($DCRn[CS] = 1$): Only one complete transfer occurs from the source to destination for each request. If an external DREQn is held asserted (low), then a continuous burst of data transfers will occur.
- **Continuous Mode** ($DCRn[CS] = 0$): Once a request is made, bytes are transferred continuously until either: (1) the BCRn reaches zero or a multiple of $DCRn[BWC]$; or (2) $DSRn[DONE]$ is written to 1 by the CPU to terminate the DMA.
- The DMA channels are prioritized, with DMA0 having the highest priority and DMA3 having the lowest priority.

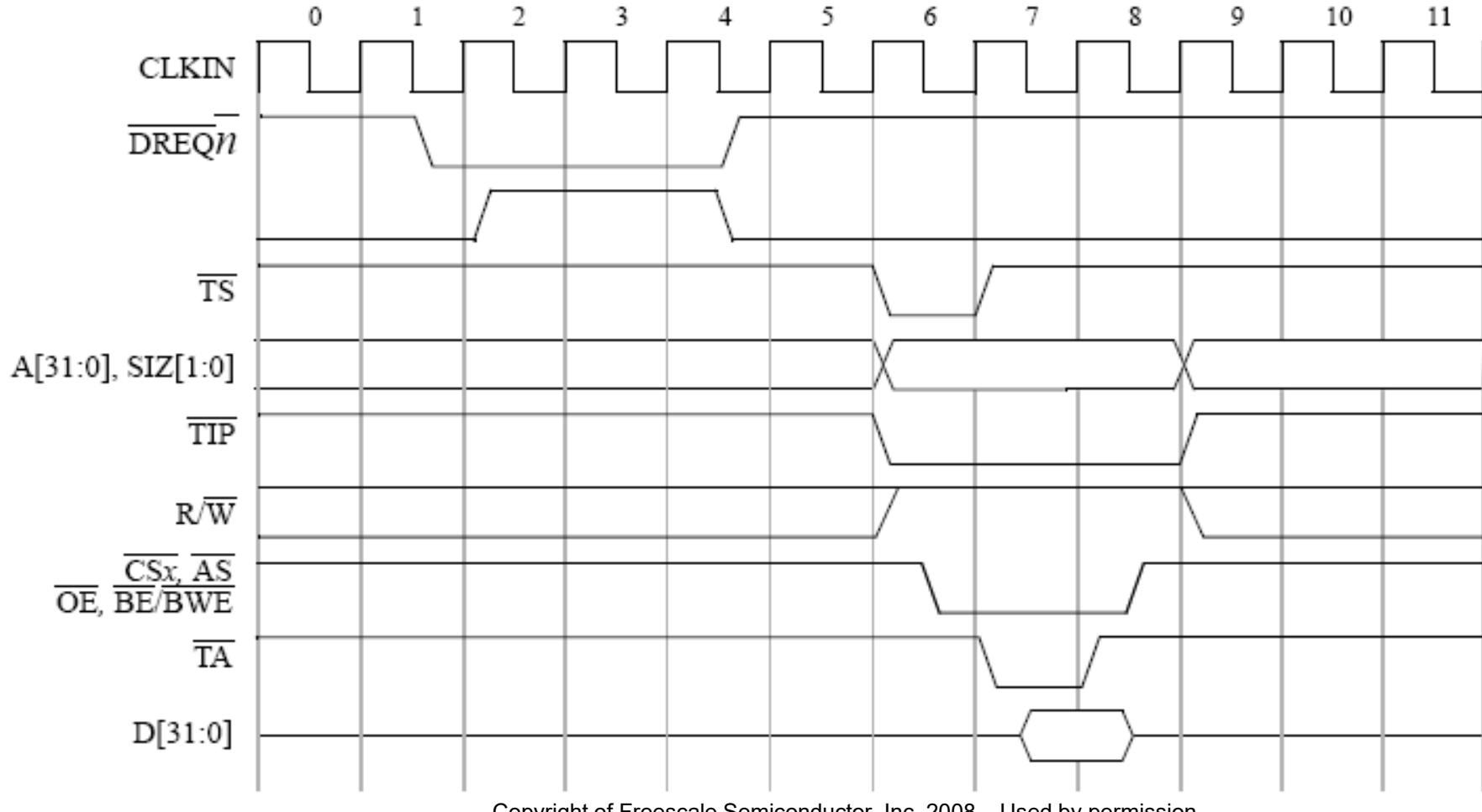
Dual-Address DMA Bus Waveforms



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

Note: Data is flowing here from one memory buffer to another.

Single-Address DMA Bus Waveforms



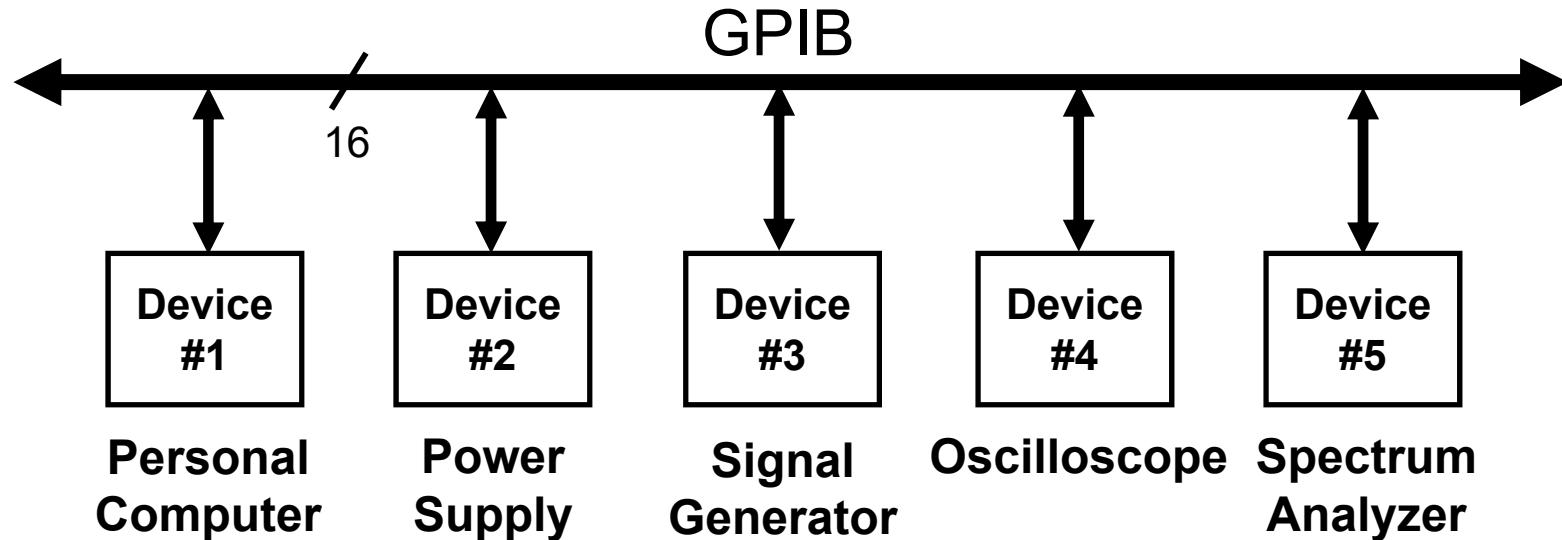
Note: Data is flowing here from a memory buffer to a peripheral.

A Few More Representative Busses

General-Purpose Interface Bus (GPIB)

- The GPIB instrument bus originated from the *Hewlett-Packard Interface Bus* (HPIB), which was defined in the late 1960s as a way of facilitating the interconnection of digital computers with HP's "intelligent" instruments.
- GPIB standardizes the electrical, mechanical and (to a lesser extent) the software interfaces for a bus that interconnects up to 15 devices.
- From 1972 until the present, the GPIB has been accepted as a standard by several major standards organizations:
 - *Institute of Electrical and Electronics Engineers* (IEEE)
 - *American National Standards Institute* (ANSI)
 - *International Electrotechnical Commission* (IEC)
 - *Electronic Industry Association* (EIA)
- GPIB interfaces are now present on many medium to large-sized instruments (e.g., oscilloscopes, signal generators, plotters, etc.). Note that the maximum data rate of GPIB (1 Mbyte/sec in theory, <750 Kbyte/sec in practice) is rather slow by today's standards.

Typical GPIB System



- A sophisticated, programmable instrument system can be rapidly constructed if all of the devices are GPIB-compatible.
- A star-type bus topology is also supported.

Chronology of GPIB and IEEE-488

- 1965-72 Hewlett-Packard develops the Hewlett-Packard Instrument Bus for use in its own instrument products.
- 1972-74 HP lobbies to have the HP-IB adopted as an industry standard.
- 1975 The HP-IB, renamed the GPIB, is adopted by the IEEE as standard IEEE-488-1975. This specification standardizes only the mechanical and electrical aspects of the bus.
- 1976 ANSI adopts the GPIB as standard MC1.1.
- 1978 IEEE releases a major revision of the standard: IEEE-488-1978.
- 1982 IEEE defined “recommended practice” for codes and format conventions to be used on the GPIB.
- 1987 Standard IEEE-488.2 standardizes many low-level, device-independent commands and protocols.
- 1990 A consortium of companies releases the *Standard Commands for Programmable Instruments* (SCPI) to promote consistency in software commands for instruments of the same type.

Main Features of GPIB

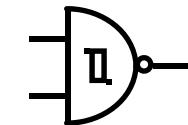
- A parallel bus containing 16 signal lines (8 data + 8 control).
Active low control lines: EOI, REN, SRQ, ATN, IFC, NDAC, NRFD, DAV.
Eight ground lines are also provided in the bus.
- A maximum of 15 devices can be connected in one basic linear bus.
- Maximum cable length (without a bus extender) is 20 meters.
- Maximum data transfer rate is 1 Mbyte/sec (300 Kbyte/sec is typical).
- An asynchronous, multilateral data transfer protocol is used.
- Synchronization is provided using a 3-wire handshake. No clock is used!
- Devices on the bus can be *Controllers*, *Talkers-only*, *Listeners-only*, and *Talkers-and-Listeners*. Only one controller can be active at a time (the System or *Master Controller*).
- The primary (one-byte) address space allows for up to 31 Talkers and 31 Listeners. A secondary, extended (two-byte) address space allows for up to 961 Talkers and 961 Listeners.

Electrical Aspects of GPIB

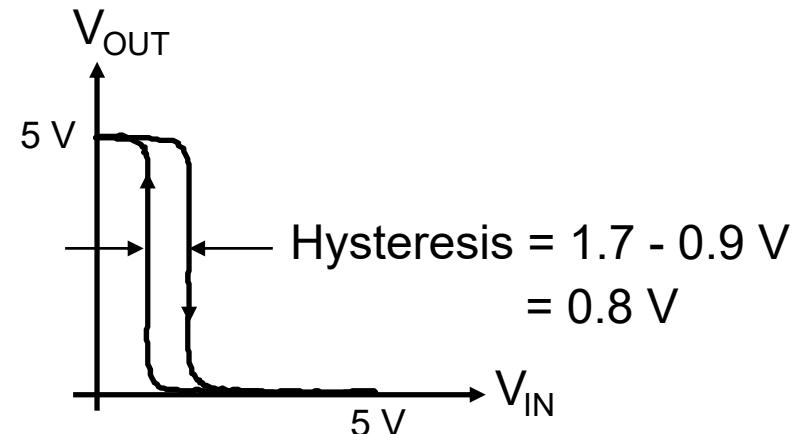
- Bus drivers and receivers are TTL-compatible.
- Input logic 0 (false) is any signal between +2.0 V and 5.0 V.
Input logic 1(true) is any signal between 0 V and +0.8 V.
- Schmitt-trigger receivers, with at least 0.4 V hysteresis, are preferred.
This provides additional noise immunity for bus signals.
- Open collector drivers required for: SRQ, NRFD, NDAC
Open collector or tri-state drivers for: ATN, IFC, REN, EOI, DAV
Open collector drivers required for DIO1-8 for parallel poll devices
Open collector or tri-state drivers for DIO1-8 for other devices.
- V_{OL} is no more than 0.5 V while sinking 48 mA.
 V_{OH} is at least 2.4 V while sourcing 5.2 mA.
- The capacitive load of each bus connection to a device is to be no greater than 100 pF for input voltages of less than 2 V.

Schmitt Trigger

Example: TTL 74LS132



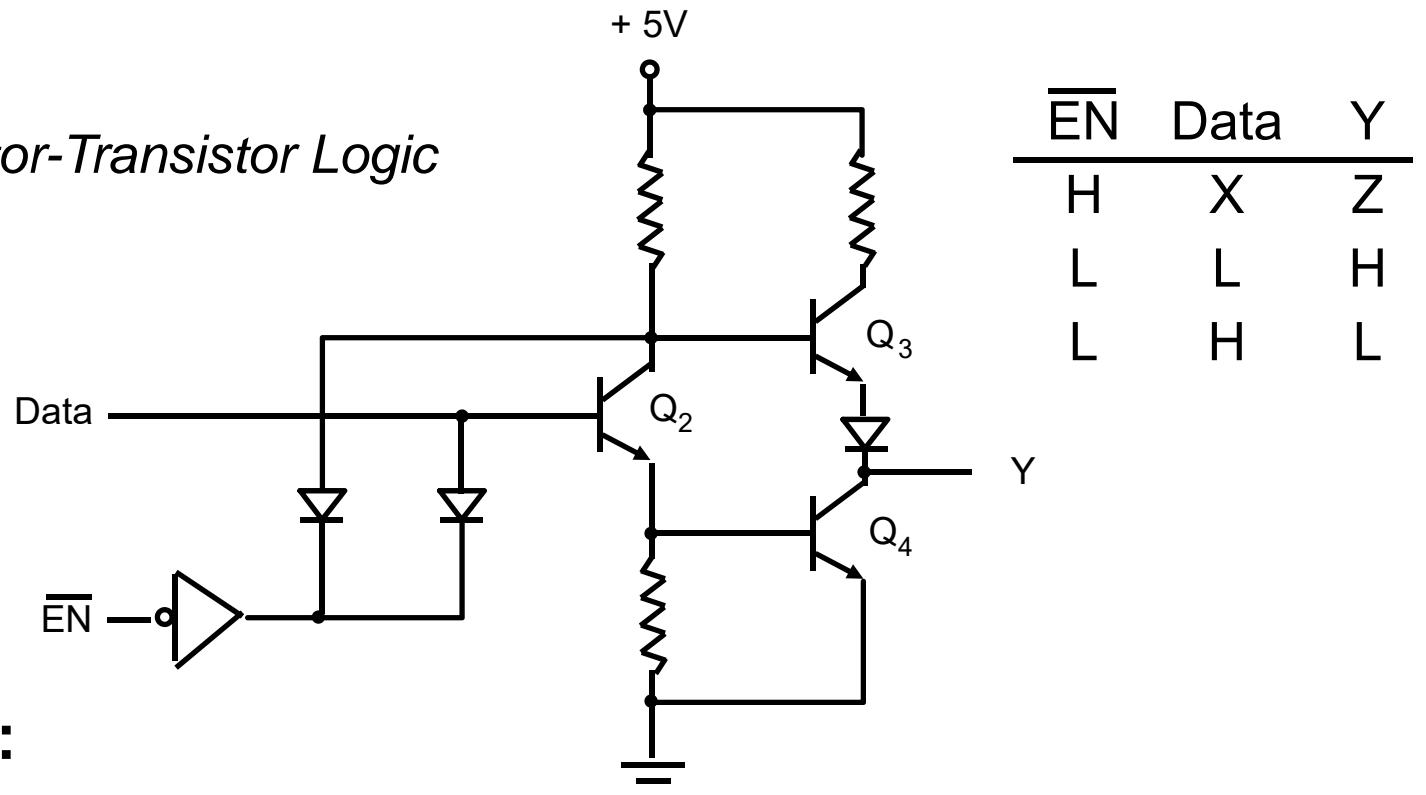
Logic Symbol



- Hysteresis is used to improve the rejection of noise in the input signals.
- When there is hysteresis, the switching threshold of an increasing input signal is greater than the switching threshold for a decreasing input signal.
- Once the output has changed state going in one direction, it is harder for the input to cause the output to change back in the opposite direction.
- Schmitt trigger inputs are widely used in bus transceivers.

TTL Output With Tri-state Enable

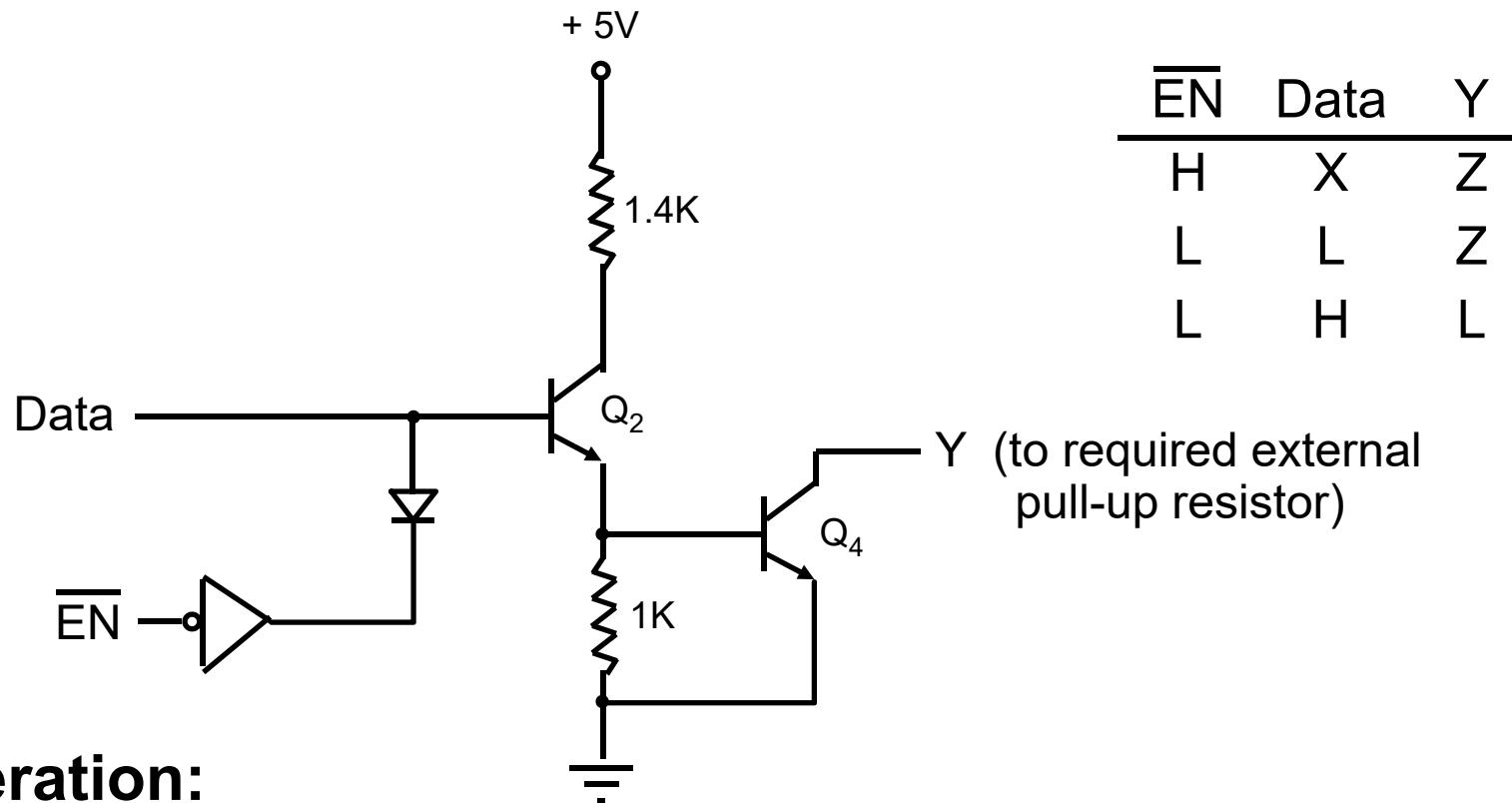
TTL = *Transistor-Transistor Logic*



Operation:

- when \overline{EN} is deasserted, the base-emitter junctions of transistors Q₂ and Q₃ are shut off. The output Y then assumes a high impedance state, Z, which can be viewed as a third state in addition to driving L and H.
- when \overline{EN} is asserted, all three transistors operate unimpeded, and output Y becomes a buffered (and inverted) copy of the data input signal.

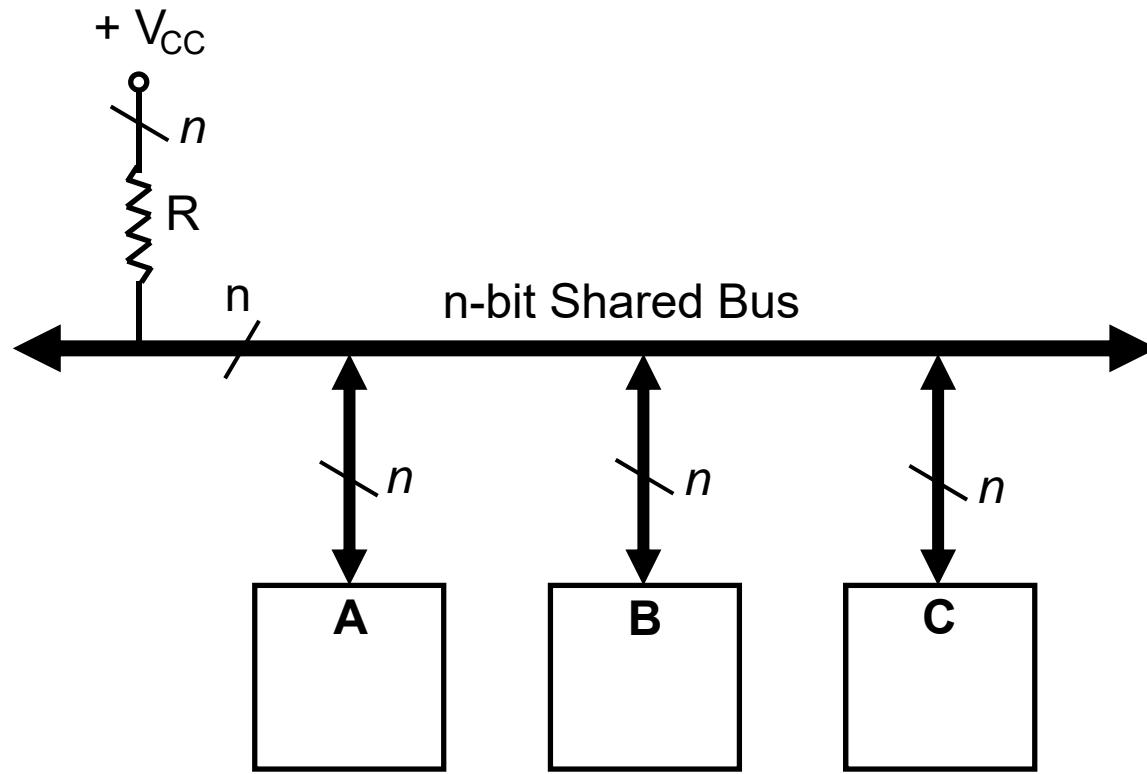
TTL Output with Open-Collector Output



Operation:

- an external pull-up resistor must be connected to Y in order to provide valid high output signals, in addition to the driven low signals.
- enable signal \overline{EN} allows Q₂, and hence also Q₄, to be turned off.

Wired-AND Bus



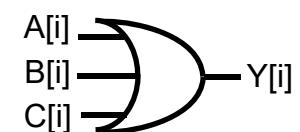
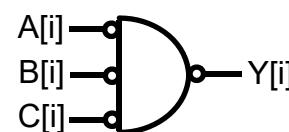
Note: n pull-up resistors R are required to ensure a high (H) signal on each of the n bus lines when none of output drivers connected to the line is driving a low (L). This creates the AND logic operation.

Wired-AND Bus with Inverting Drivers

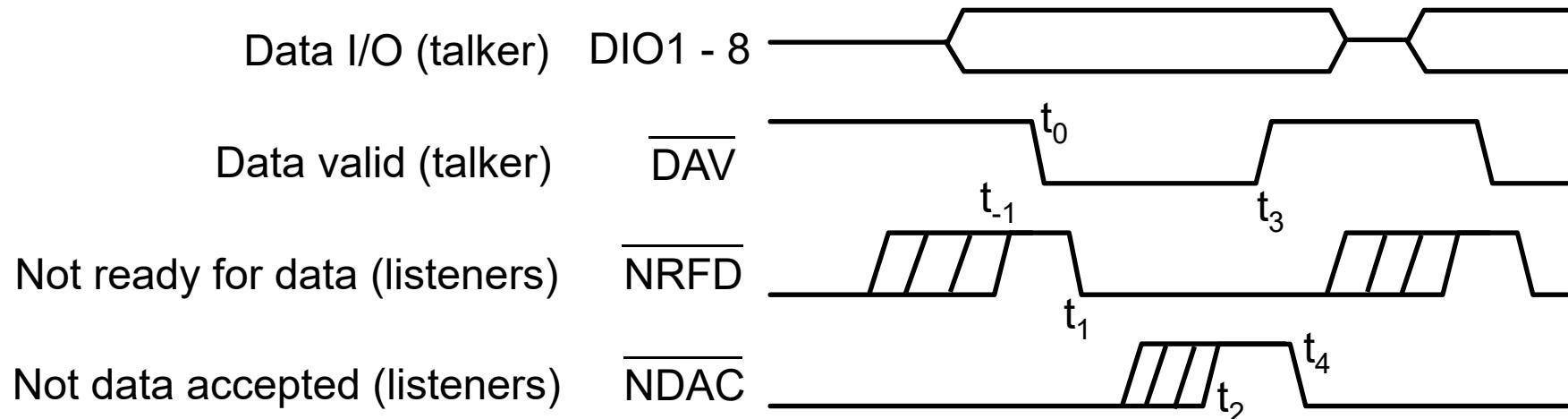
$A[i]$	$B[i]$	$C[i]$	$\overline{A[i]}$	$\overline{B[i]}$	$\overline{C[i]}$	$\overline{Y[i]}$	$Y[i]$
0	0	0	Z	Z	Z	H	0
0	0	1	Z	Z	L	L	1
0	1	0	Z	L	Z	L	1
0	1	1	Z	L	L	L	1
1	0	0	L	Z	Z	L	1
1	0	1	L	Z	L	L	1
1	1	0	L	L	Z	L	1
1	1	1	L	L	L	L	1

$$\overline{Y[i]} = \text{AND}(\overline{A[i]}, \overline{B[i]}, \overline{C[i]})$$

$$Y[i] = \text{OR}(A[i], B[i], C[i])$$



GPIB 3-Way Handshake Signals



t_{-1} : All acceptors become ready for the next byte.
NRFD goes high with the slowest acceptor.

t_0 : Source validates data (\overline{DAV} asserted)

t_1 : Fastest acceptor asserts NRFD to indicate that it is no longer ready to accept any more data.

t_2 : NDAC goes inactive when the slowest acceptor indicates that it has accepted the data.

t_3 : DAV goes inactive to indicate that the data bus no longer carries valid data.

t_4 : Fastest acceptor asserts NDAC in preparation for the next data transfer cycle.

GPIB Interface Management Lines

Mnemonic	Name	Function
ATN	Attention	Asserted low when the data bus carries a command (rather than data).
IFC	Interface clear	Asserted low to initialize the GPIB system to the idle state.
SRQ	Service request	Asserted low to alert the controller to the need to transfer data.
REM	Remote enable	Asserted low to cause devices to go into remote, GPIB-controlled mode.
EOI	End or identify	Indicates last data byte transfer; also used to obtain 8 status bits in parallel.

GPIB Commands

- When the ATN (attention) signal is asserted low, then the bus is in “command mode” as opposed to “data transfer mode”.
- While in command mode, all GPIB-connected devices must read and examine the data bytes, and then interpret them as commands.
- There are four classes of GPIB commands:
 - 1) **Address Commands:** used to select talkers and listeners
 - 2) **Universal Commands:** heard by all devices
 - 2.1) **Uniline Commands:** signalled using one bus line
 - Attention (ATN): interpret the data as a command
 - Interface Clear (IFC): initialize all devices to idle state
 - Remote Enable (REN): enable remote program control
 - Service Request (SRQ): alert controller from other device
 - 2.2) **Multiline Commands:** data bus code specifies the command
 - Untalk (UNT, \$5F): unaddress the current talker
 - Unlisten (UNL, \$37): unaddress all current listeners

GPIB Commands (cont'd)

2.2) Multiline Commands (cont'd):

Device Clear (DCL, \$14): reset all devices on the bus

Local Lockout (LLO, \$11): lock out front panel of all devices

Serial Poll Enable (SPE, \$18): request status from listeners

Serial Poll Disable (SPD, \$19): return to data transfer mode

Parallel Poll Unconfigure (PPU, \$15): turn off parallel poll

3) Addressed Commands: Devices must have been addressed previously by the controller. Command type specified on data bus.

Group Execute Trigger (GET, \$08): trigger all current listeners

Selected Device Clear (SDC, \$04): reset one device

Go to local (GTL, \$01): re-enable front panel controls

Parallel Poll Configure (PPC, \$04): set up device for parallel poll

4) Secondary Commands:

Extended (two-byte) talker and listener addresses

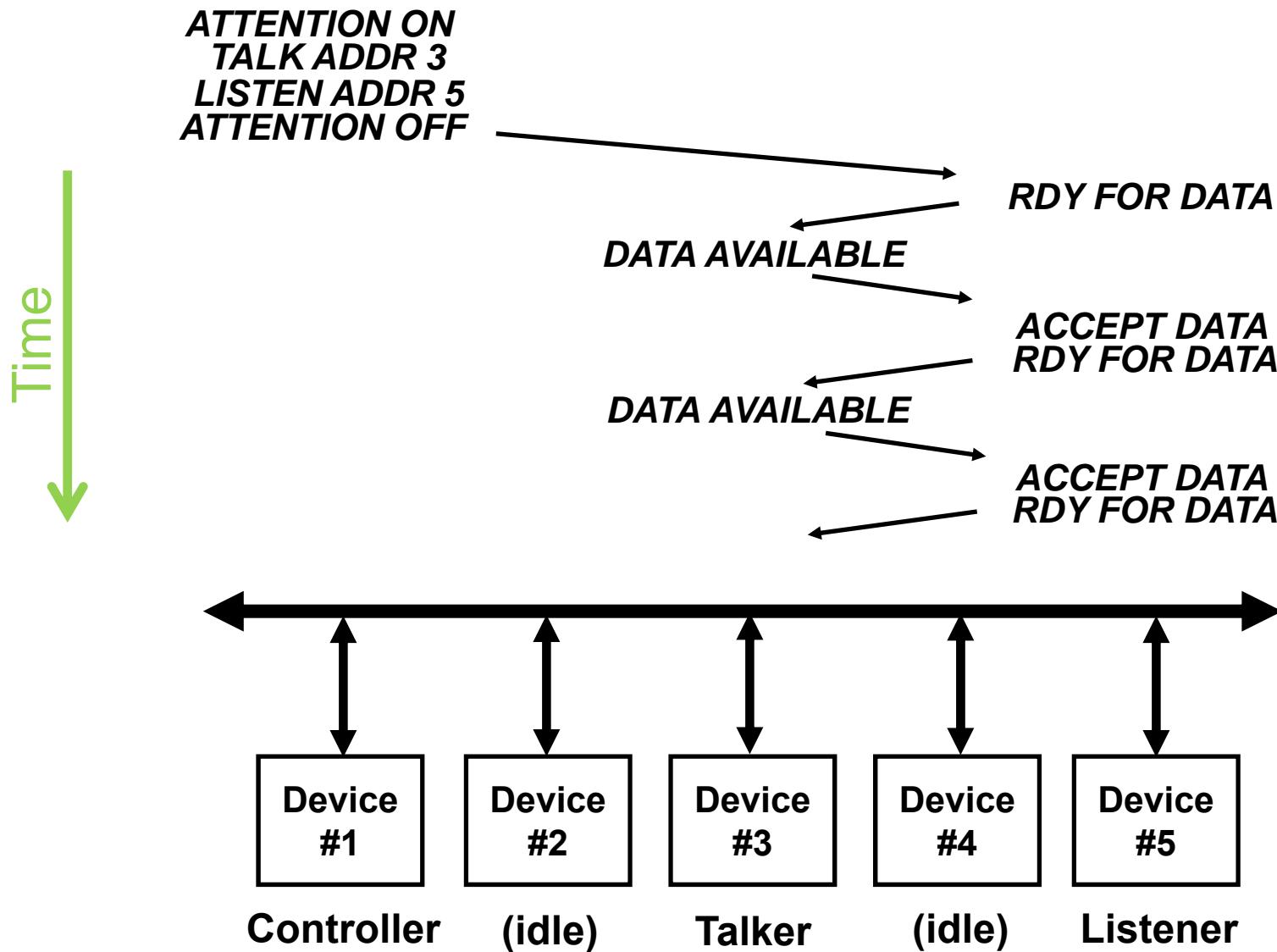
Parallel Poll Enable (PPE, \$60-6F): poll multiple devices in parallel

Parallel Poll Disable (PPD, \$70): disable parallel poll feature

Four Possible GPIB Device States

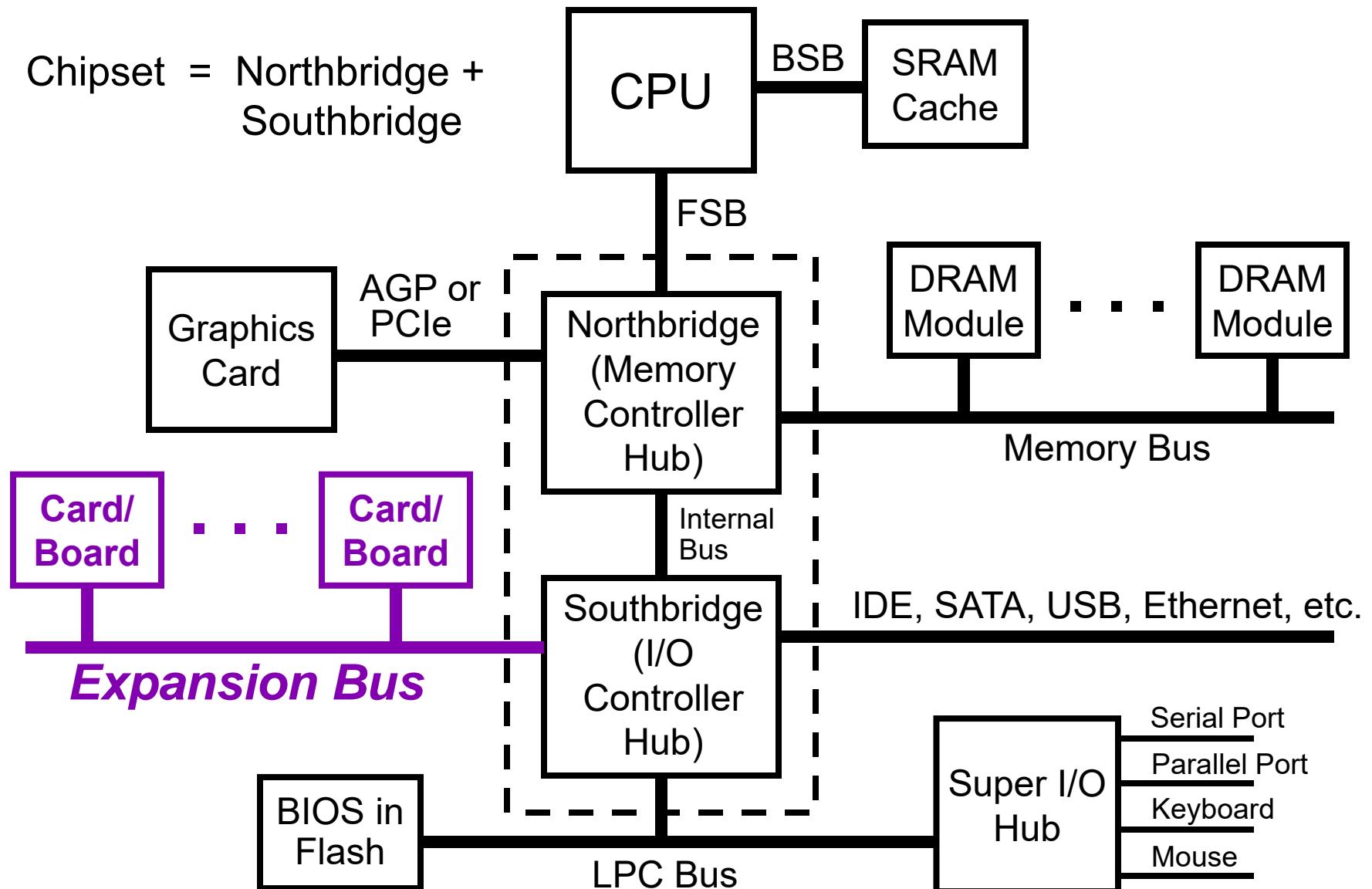
- 1) **Idle:** Not currently participating in the bus transaction
- 2) **Controller:** A device capable of specifying one talker and up to 14 listeners for a data transfer over the bus. There may be multiple controllers on the GPIB, but only one controller can be the active Master Controller.
- 3) **Listener:** Able to receive data over the bus (after being addressed previously by the controller).
- 4) **Talker:** Able to transmit data over the bus (after being addressed previously).

Typical GPIB Bus Transaction



PC Expansion Busses

Chipset = Northbridge + Southbridge



Advantages of Standard Expansion Busses

- 1) Few or no new boards to design when enhancing the PC.
- 2) Multiple vendors from whom to buy "off-the-shelf" boards.
- 3) Board-level communications already specified.
- 4) No new bus specifications to develop.
- 5) Interface ICs are available for designing custom boards.
- 6) Reduced design time and debugging time.
- 7) Faster time to market => economic survival and success

Main disadvantages:

- 1) Standard boards tend to be loaded up with features that may not be useful in the given application or product.
- 2) Standard boards can be expensive, possibly unprofitable.

Historical PC Expansion Busses

Industry Standard Architecture (ISA) Bus, 1981 (first IBM-only, then open)

- 16-bit data transfers at 8 MHz

Microchannel Architecture (MCA) Bus, 1987 (IBM standard)

- 32-bit data transfers at 10 MHz

Enhanced ISA (EISA) Bus, 1988 (open standard)

- replacement for the ISA
- 32-bit transfers at 8.33 MHz

Video Electronics Standards Association (VESA) Local Bus, 1989

- allows up to two cards to be attached directly to a 33 MHz CPU-memory local bus
- Intended to provide sufficient bandwidth for high-performance graphics subsystems. Limited usefulness as a general expansion bus.

Peripheral Component Interconnect (PCI) Bus, 1993 (open std., led by Intel)

- high-performance, CPU-independent bus
- 32- or 64- bit transfers at 33 or 66 MHz
- intended to entirely replace ISA, MCA, EISA, and VESA

I/O Requirements in a Personal Computer

<i>Peripheral Type</i>	<i>Technology</i>	<i>Bus Bandwidth</i>
Graphics	24-bit colour	30 Mbytes / s
Local Area Network	FDDI (optical fibre)	12 Mbytes / s
Disk Drive	SCSI 1.0	10 Mbytes / s
Disk Drive	USB 2.0 (serial)	480 Mbits / s (60 Mbytes / s)
Full-Motion Video with Audio	1024 x 768 @ 30 frames/sec & 16-bit sound	67+ Mbytes / s

I/O Capabilities of PC Peripheral Busses

<i>Standard</i>	<i>Width</i>	<i>Max. Slots</i>	<i>Clock</i>	<i>Bandwidth</i>
ISA	8/16 bits	6	8 MHz	16 Mbytes/s
EISA	8/16/32 bits	8	8.33 MHz	33 Mbytes/s
VESA LB	32 bits	3	25-50 MHz	100-200 Mbytes/s
PCI	32/64 bits	5	33 MHz	132 Mbytes/ s (peak) 120 Mbytes / s (sustained)

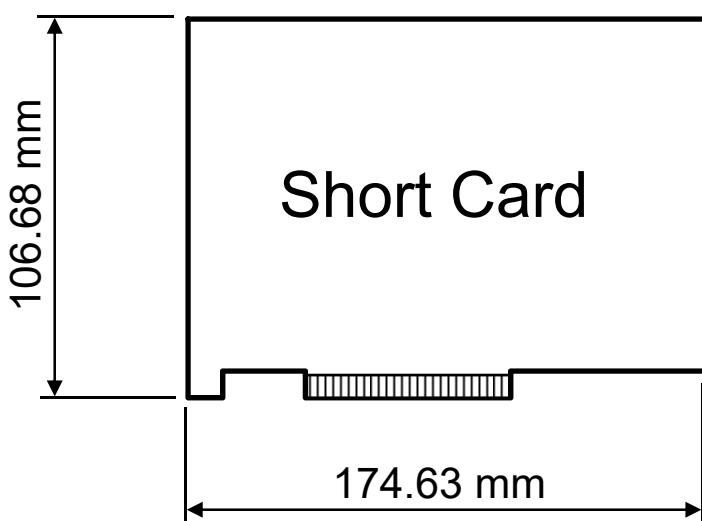
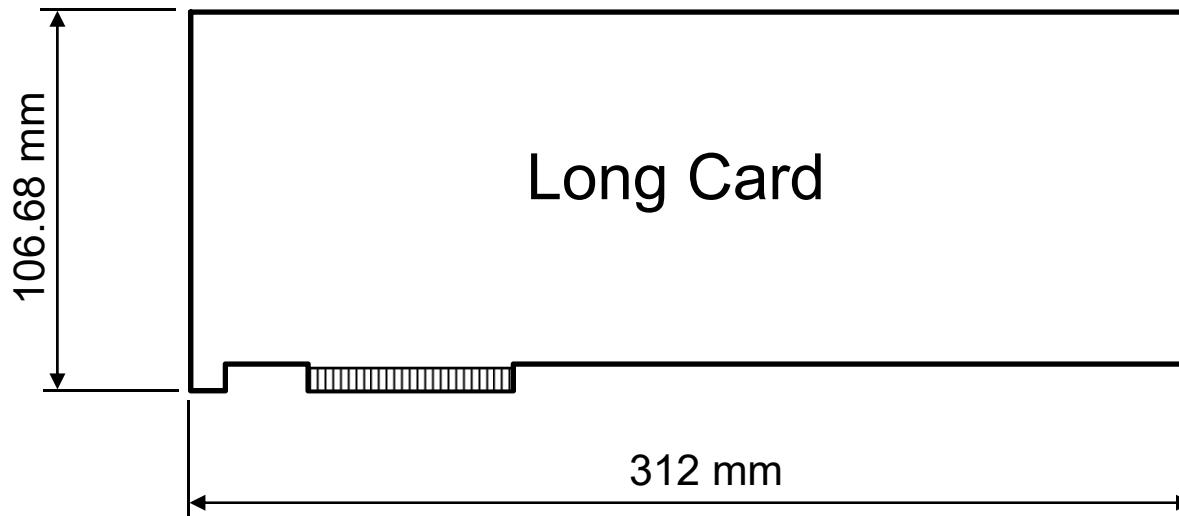
The Peripheral Component Interconnect (PCI) Bus

- Developed at Intel Corp. in the early 1990s.
- Increasing demand for high-performance peripherals, e.g. graphics cards, Ethernet LAN, SCSI, USB 2.0, Firewire
- The existing ISA and EISA standard PC busses were too slow.
- The high-speed VESA local bus (VLB) was a short-term solution that could not be extended physically beyond one or two slots.
- The Personal Computer industry (software and hardware) required a new high-performance bus for the Intel 80486 and the more powerful Pentium-class superscalar microprocessors.
- Needed support for both 5-V and 3.3-V boards.
- Needed support for power management features.
- Needed upgrade path from 32-bit boards to 64-bit boards.
- Needed support for autoconfiguration of I/O boards, so-called “plug-and-play” capability.

PCI Background

- a new open standard initially proposed by Intel Corp.
- in early 1990s support grew for the PCI standard for use in many major microcomputer platforms
- the PCI Special Interest Group, an alliance of many key manufacturers, now controls the PCI standard and guides its continuing evolution
- the PCI bus is essentially a multiplexed version of the 80486 bus, with control mechanisms modified and extended to provide burst I/O bus transactions
- the PCI specification provides limited backwards compatibility with the ISA and EISA standards (older ISA and EISA cards can be used)
- the PCI specification includes mechanical, electrical, and limited low-level software aspects

PCI Mechanical Specifications



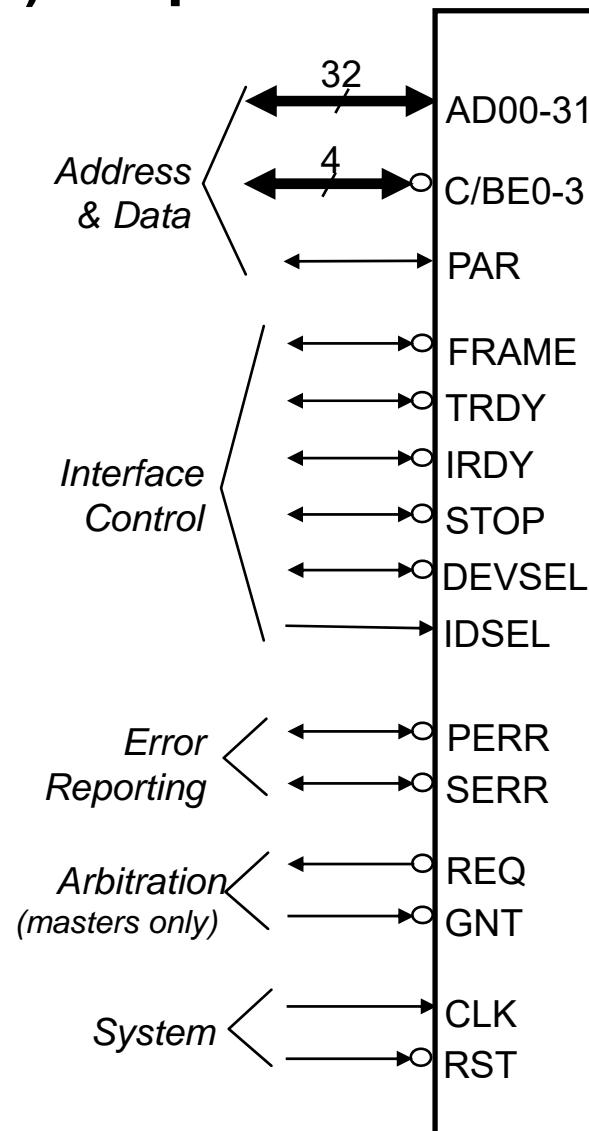
- 32-bit data bus uses a 124-pin connector (only 49 signals).
- 64-bit data bus uses a 188-pin connector.

Electrical Specifications of PCI

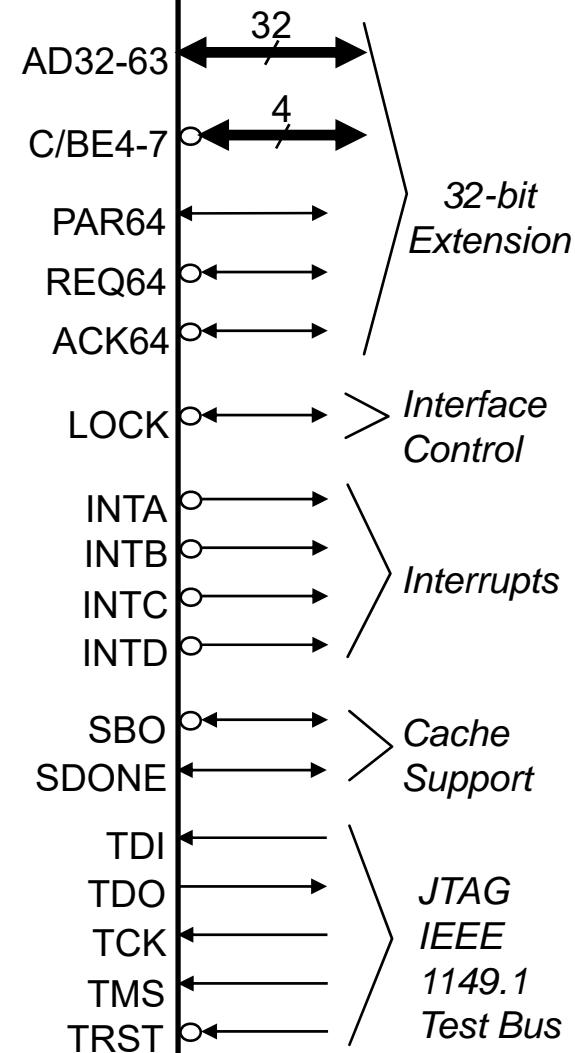
- Most previous busses use “Incident Wave Switching” where the bus drivers produce full-height logic signals in one step.
- PCI devices use “Reflected Wave Switching”
 - The bus drivers produce half-height logic signals, saving power.
 - Reflections off far end of the bus are used to double the logic signals to their full height. Source output resistance stops further reflections.
- PCI devices employ CMOS drivers at TTL levels (for compatibility).
- The bus supply voltage is either 5 V (TTL) or 3.3 V (more modern).
- PCI clock ranges in frequency from DC to 33 MHz. The clock is typically, but not necessarily, derived from the CPU clock.
- Up to 10 PCI “loads” can be driven at 33 MHz using PCI chip sets.
- The original PCI chip sets could only drive three fully loaded PCI slots.
- The signals are surrounded by a large number of power and ground connections to provide shielding and to minimize radiation.

PCI Signals

47 (+2) Required Pins



Optional Pins



PCI Bus Signals

CLK	Input	System Clock (DC to 33 MHz)
RST	Input	System Reset
AD00-31	Tri-state	Multiplexed Address & Data
C/BE0-3	Tri-state	Multiplexed Bus Command and Byte Enables
PAR	Tri-state	Parity (even) across AD & C/BE
FRAME	Tri-state	Cycle Frame (valid cycle in Progress) Asserted by the Initiator at start of cycle, and de-asserted just before the last data transfer
DEVSEL	Tri-state	Device Select: asserted low by the Target.
IRDY	Tri-state	Initiator Ready for the next data transfer read: master prepared to accept new data write: valid data is present on the AD bus
TRDY	Tri-state	Target Ready for the next data transfer read: valid data is present on the AD bus write: slave is ready to accept new data

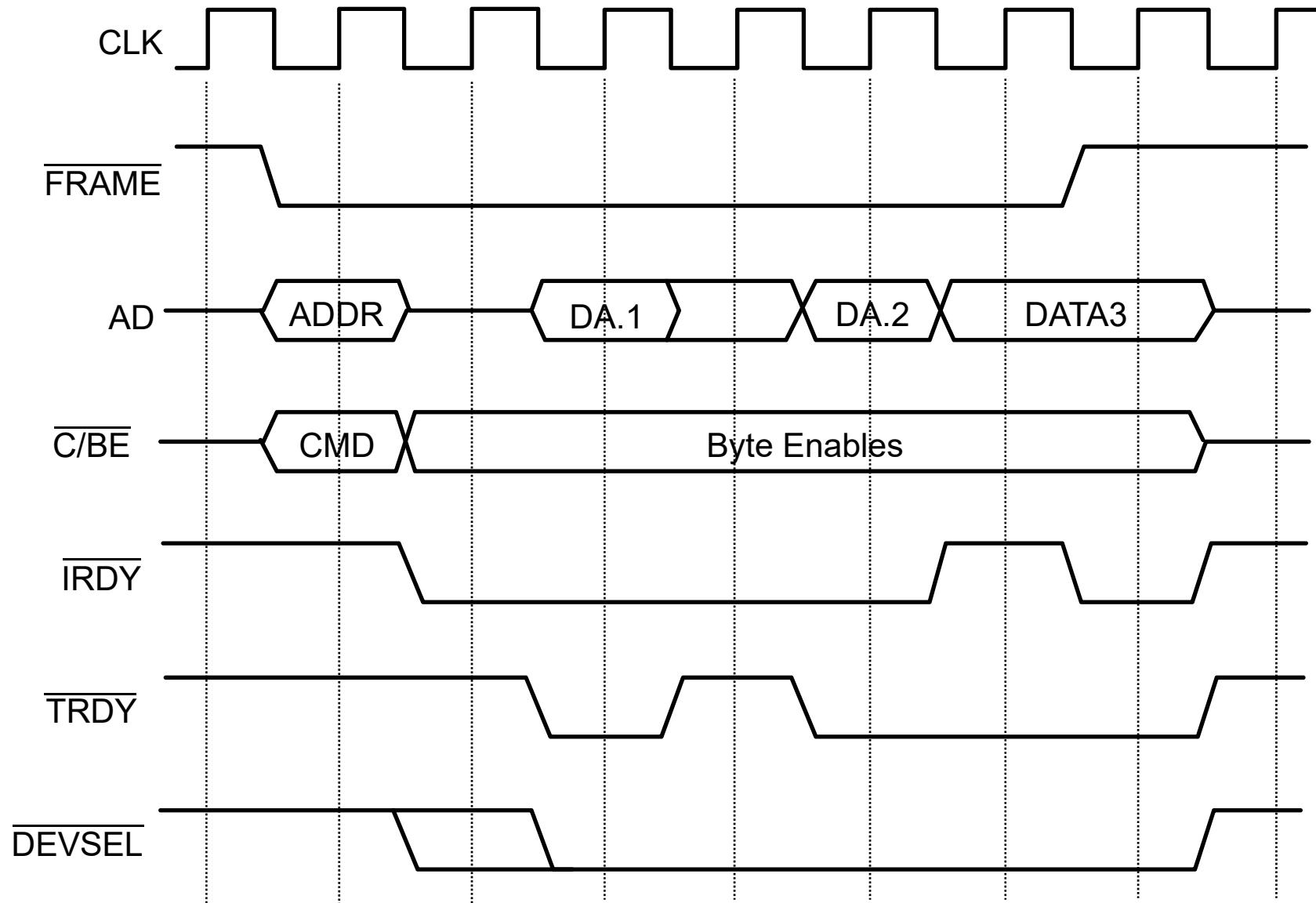
PCI Bus Signals (cont'd)

LOCK	Tri-state	Lock
IDSEL	Input	Initialization Device Select (select during autoconfig.)
STOP	Tri-state	Stop (asserted by target)
REQ	Tri-state	Request
GNTR	Tri-state	Grant
PERR	Tri-state	Parity Error
SERR	Open Drain	System Error
INTA	Open Drain	Interrupt A
INTB	Open Drain	Interrupt B
INTC	Open Drain	Interrupt C
INTD	Open Drain	Interrupt D
SBO	Input/Output	Snoop Backoff
SDONE	Input/Output	Snoop Done

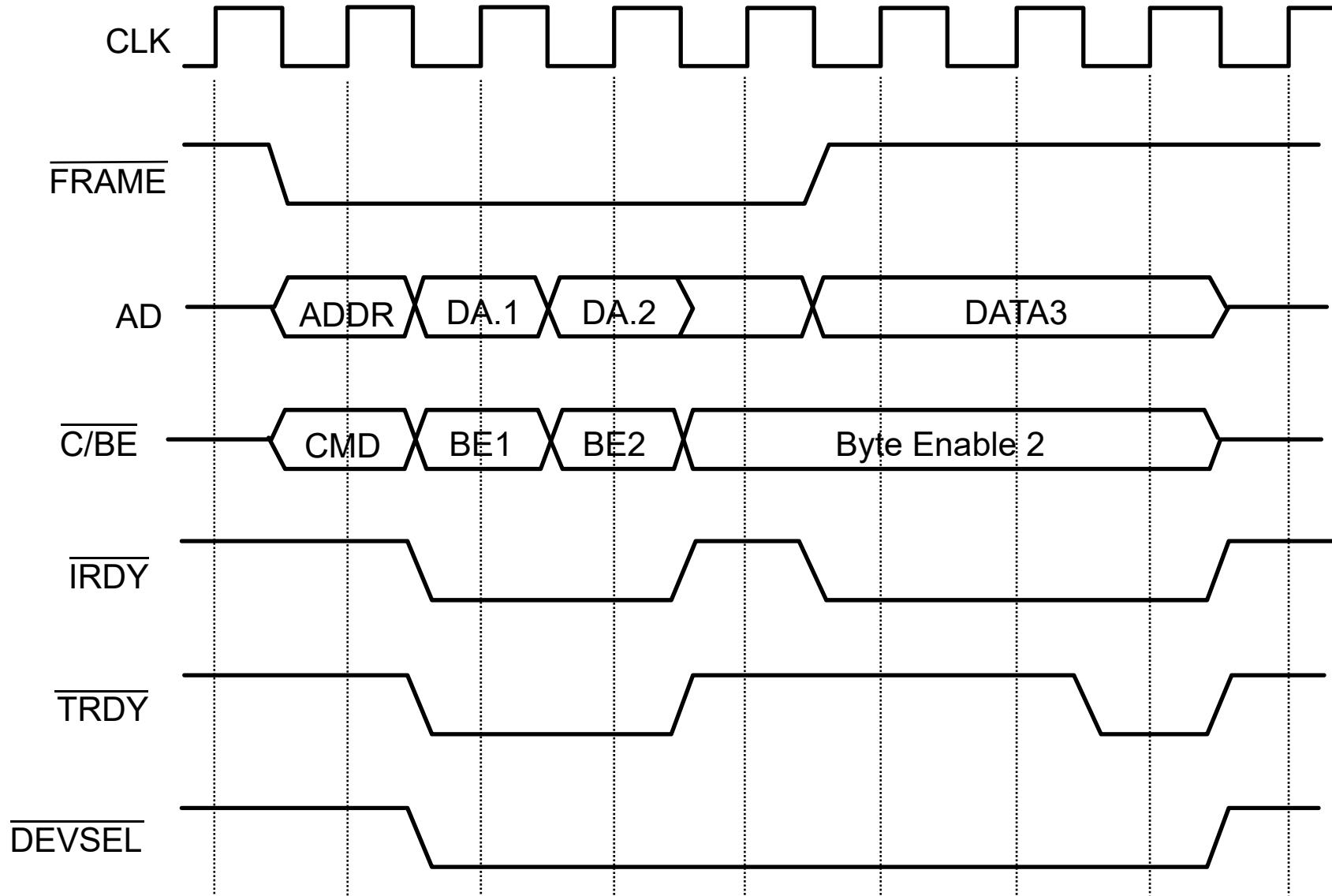
Bus Command and Byte Enables

C/BE3-0	Command Type
0000	Interrupt Acknowledge
0001	Special Cycle
0010	I / O Read
0011	I / O Write
0100	Reserved
0101	Reserved
0110	Memory Read
0111	Memory Write
1000	Reserved
1001	Reserved
1010	Configuration Read
1011	Configuration Write
1100	Memory Read Multiple
1101	Dual Address Cycle
1110	Memory Read Line
1111	Memory Write and Invalidate

Basic PCI Read Operation



Basic PCI Write Operation



PCI Device Configuration

Previous Practice:

- when a new I/O card is installed on an ISA bus one must typically:
 - set up correct IRQs (jumper settings?)
 - set up DMA channels (base addresses?)

PCI Practice (“Plug and Play”)

- all PCI devices have a set of configuration registers that are examined upon power up
- configuration software (probably part of the BIOS) detects the presence of all PCI devices
- the configuration space of each PCI device is accessed, and then unique memory and I/O regions are assigned to each device
- to add a new PCI device:
 - (1) power down the system
 - (2) **plug** in the new cards
 - (3) power up the system
 - (4) use / **play** with the new system

PCI Configuration Space

- each PCI card must have 64 mandatory configuration registers (PCI config. space header)
- there may be up to a maximum of 256 registers

\$00	Device ID		Vendor ID	
\$04	Status Register		Command Register	
\$08	Class Code		Revision ID	
\$0C	BIST	Header Type	Latency Timer	Cache Line Size
\$10	Base Address 0			
\$14	Base Address 1			
\$18	Base Address 2			
\$1C	Base Address 3			
\$20	Base Address 4			
\$24	Base Address 5			
\$28	Cardbus CIS Pointer			
\$2C	Subsystem ID		Subsystem Vendor ID	
\$30	Expansion ROM Base Address			
\$34	Reserved			
\$38	Reserved			
\$3C	Max. Latency	Min.-Grant	Interrupt Pin	Interrupt Line

PCI Class Code Base Classes

Base Class	Meaning
\$00	Early device with no class code
\$01	Mass storage controller
\$02	Network Controller
\$03	Display Controller
\$04	Multimedia device
\$05	Memory controller
\$06	Bridge device
\$07	Simple communications controller
\$08	Base system peripheral
\$09	Input device
\$0A	Docking station
\$0B	Processor
\$0C	Serial bus controller
\$0D - \$FE	Reserved
\$FF	Device does not fit any defined class

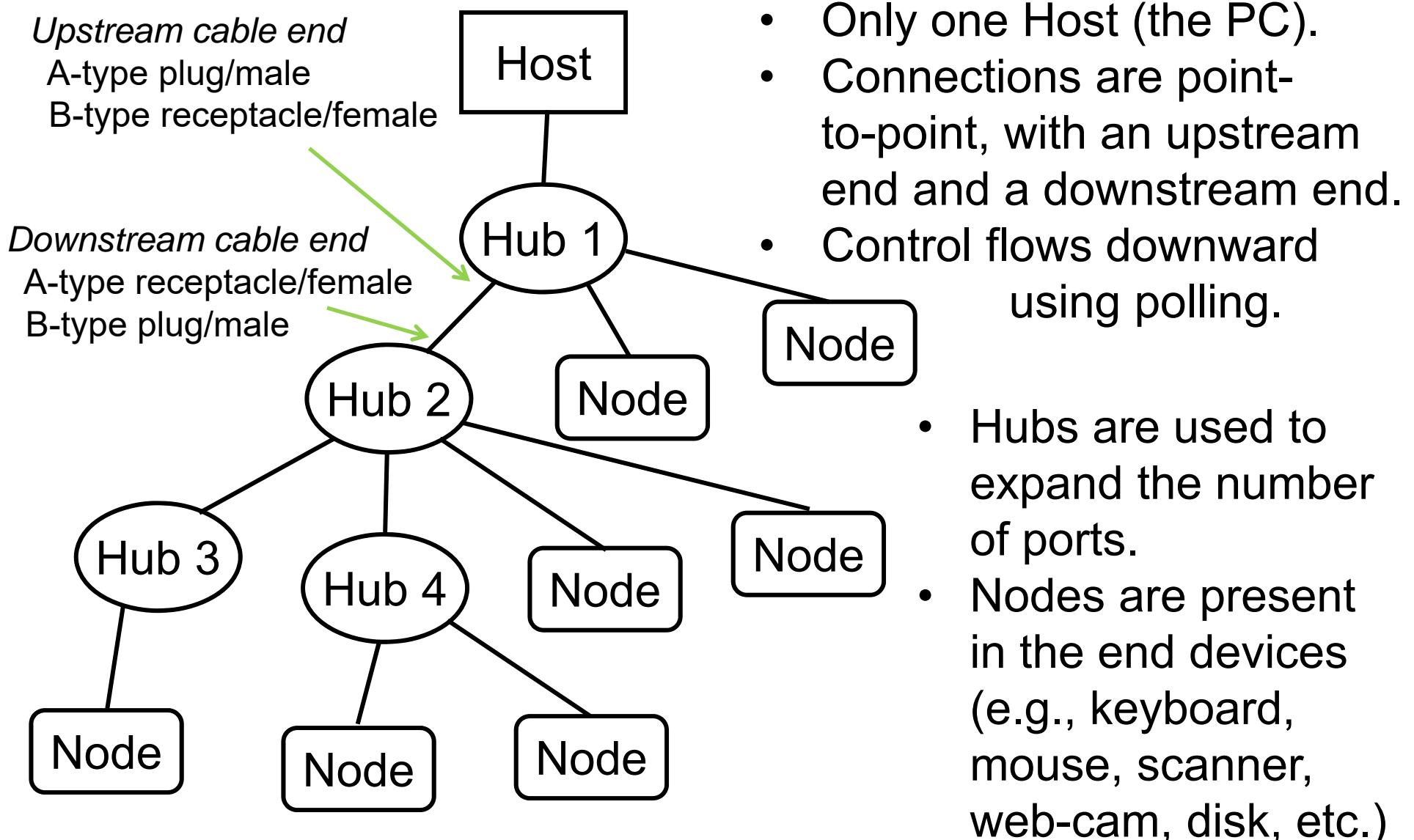
Motivations for Universal Serial Bus (USB)

- Want to simplify the interfacing of peripherals to the PC.
- Automatic configuration of peripherals: just plug in.
- Serial connection to reduce cost and avoid bulky wiring:
 USB link has only power + ground + data+ + data-
- Inclusion of power connections to power up small devices
 Power pins designed to permit “hot plugging”
 Large devices may need separate power connection
- Multiple versions for multiple different data rates:
 low: 1.5 Mbit / sec for keyboard, mice, joystick, etc.
 med.: 12 Mbit / sec for scanners, audio devices, etc.
 high: 480 Mbit / sec for hard drives, video, etc.
- Implementation of simple point-to-point connections.
- Need a standard with broad industry-wide support:
 Intel has strongly promoted USB since the mid-1990s.

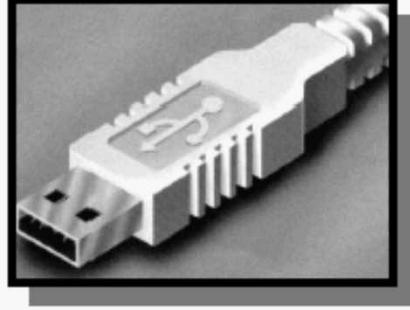
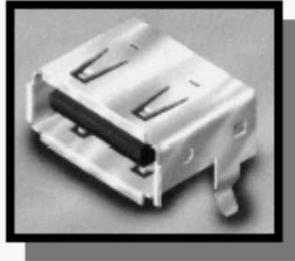
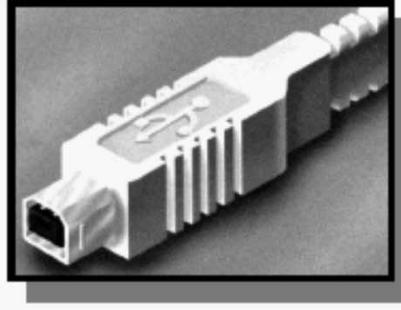
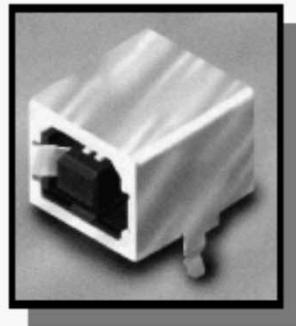
A Brief History of USB

- USB 1.0 released in January 1996
 - 4 wires (ground, power, differential data pair)
 - Only the 1.5 Mbits/s rate was supported (e.g., keyboard, mouse)
- USB 1.1 released in September 1998
 - This updated standard corrected several flaws in USB 1.0.
- USB 2.0 released in April 2000
 - Medium (12 Mbit/s) and high speed (480 Mbit/s) added.
 - Backward compatible with USB 1.1.
 - Medium and high speed operation requires shielded cable.
- USB 3.0 released in November 2008
 - SuperSpeed mode (3.2 Gbit/s) added.
 - Adds a shield and two more differential data pairs.
 - Data signalling uses advanced techniques similar to PCIe.
 - New power modes added (idle, sleep, suspend).

USB Topology



USB Connectors

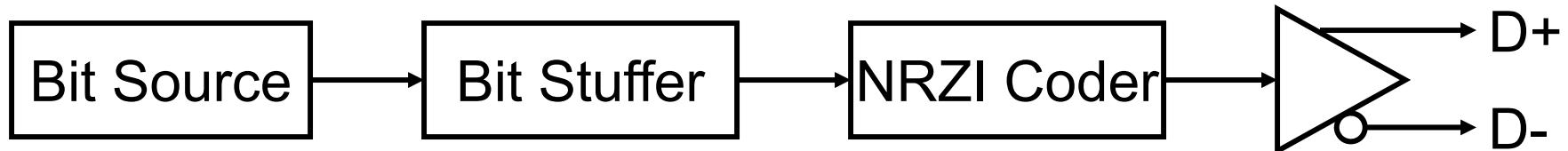
Series "A" Connectors	Series "B" Connectors
<ul style="list-style-type: none">◆ Series "A" plugs are always oriented upstream towards the <i>Host System</i>  <p>"A" Plugs <i>(From the USB Device)</i></p> <p>"A" Receptacles <i>(Downstream Output from the USB Host or Hub)</i></p> 	<ul style="list-style-type: none">◆ Series "B" plugs are always oriented downstream towards the <i>USB Device</i>  <p>"B" Plugs <i>(From the Host System)</i></p> <p>"B" Receptacles <i>(Upstream Input to the USB Device or Hub)</i></p> 

USB Cabling and Signal Levels

- Only four wires are present in a USB 1.1 or 2.0 connection

VBus (power)	Pin #1	Red insulation
D- (negated data)	Pin #2	White insulation
D+ (true data)	Pin #3	Green insulation
GND (power return)	Pin #4	Black insulation
- VBus can supply up to 500 mA at 5 Volts.
- VBus and GND pins are slightly longer than the data pins.
- Low-speed (1.5 Mbit/s) cable is untwisted and unshielded.
- Medium (12 Mbit/s) and high-speed (480 Mbit/s) cable uses twisted data wires, and the entire cable is shielded.
- D- and D+ carry a differential data signal with encoded clock.
logical “1” => $200 \text{ mV} < (\text{D+} \text{ minus } \text{D-}) < 4 \text{ V}$
logical “0” => $-4 \text{ V} < (\text{D+} \text{ minus } \text{D-}) < -200 \text{ mV}$
- The clock is encoded in the data using a combination of “bit-stuffing” and Non Return to Zero Inverted (NRZI) coding

Bit Timing Encoding in the USB Signal



Non Return to Zero Inverted (NRZI) Coder:

- If the incoming bit is “0”, toggle the output bit
- If the incoming bit is “1”, keep the same output bit

Bit Stuffer:

- Default action is to pass the input bits through to the output stream.
- If the previous 6 incoming bits were all “1”s, then insert (i.e. stuff) a “0” into the output stream.
- After a “0” has been inserted, disable further bit stuffing for the next five input bits.

Bit Timing Recovery from the USB Signal



Hybrid:

- Subtracts transmitted signal from the bi-directional signal to get the received signal alone.

NRZI Decoder:

- Produce as the output bit the XNOR of the present input bit with the previous input bit

Bit Destuffer:

- If a block of 6 “1”s is received, delete the next “0”.
- Pass all other bits through to the output unchanged.

Identification of USB Nodes/Devices

- An addressing scheme is used to permit the USB host to keep track of and communicate with USB devices.
- Each USB device is associated with following information:
 - Vendor Identifier (2 bytes)
 - Device Identifier (2 bytes)
 - Version Number (2 bytes)
 - Class Code (3 bytes)
- During initialization, the USB host does the following:
 - Obtains the full address of each USB device
 - Determines required data bandwidth and data types
 - Assigns to each USB a unique 7-bit address

USB Pipes and Frame Formats

- The USB bit-stream must be carefully structured to ensure efficient and flexible use of the available bandwidth.
- The host sets up logical data “pipes” to the USB nodes. Each pipe is allocated some of the total USB bandwidth.
- The data transfer in a pipe is encoded in packets of 3 types:

<i>Token:</i>	SYNC 00000001	Packet ID 01XX10XX	Address AAAAAAA	ENDP EEECCCCC	CRC-5
---------------	------------------	-----------------------	--------------------	------------------	-------

<i>Handshake:</i>	SYNC 00000001	Packet ID 10XX01XX
-------------------	------------------	-----------------------

<i>Data:</i>	SYNC 00000001	Packet ID 00XX11XX	Data DDD...DDD	CRC-16 CCC...CCC
--------------	------------------	-----------------------	-------------------	---------------------

Note: Data packets can be up to 1500 bytes long.

USB Transfer Modes

- Some of the data being communicated over a USB link (e.g., audio and video) must be transferred in according to strict time constraints to avoid unacceptable jitter.
- A high-priority *isochronal mode* is available in USB that guarantees a certain fixed amount of data bandwidth.
- Less time sensitive data is transferred using a lower priority *asynchronous (or bulk) mode*.
- At least 20% of the USB bandwidth is reserved for asynchronous mode so that asynchronous data (e.g. keyboard inputs) is never completely locked out.
- Initialization of USB devices and status queries are supported by a *control mode*.

High-Speed Serial Busses

- Several high-speed bus standards have emerged that adopt synchronous serial transmission, for example:
 - Ethernet offering 10, 100, 1000 & 10000 Mbit/s
 - USB 1.1-3.0 offering 1.5, 12, 480 & 3200 Mbit/s
 - IEEE 1392 (called FireWire by Apple) offering 0.4-3.2 Gbit/s
 - PCIe offering 2.5 to 10 Gbit/s over each of up to 32 "lanes"
- **Advantages** of such serial interfaces?
 - simpler and cheaper cabling
 - smaller and cheaper connectors
 - no problems with skew & crosstalk between parallel signal paths, so data rates can be higher and the receiver is simpler
 - can be extended from wired to optical and wireless media
- **Disadvantages?**
 - usually limited to point-to-point topologies (e.g., hub and spokes, or a fully-connected graph or clique)

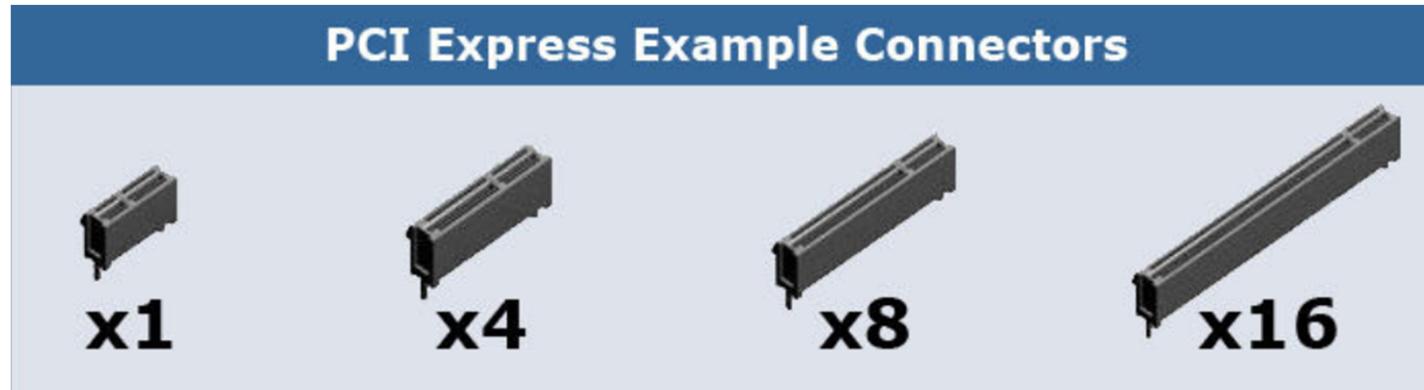
PCI Express (PCIe)

- Introduced in 2004 to replace earlier high-performance expansion bus standards such as AGP, PCI and PCI-X.
- Backed originally by Intel, Dell, HP and IBM. A *PCI Special Interests Group* now maintains the standard.
- Topology uses serial *point-to-point links* rather than a shared parallel bus. Data is encapsulated for serial transmission in packets.
- The wiring and connectors used in PCIe are completely different from those used in PCI.
- PCIe is compatible with respect to PCI at upper protocol layers, so much older driver software can be carried over with no changes.
- A *transaction layer* converts data bytes to packets in the transmit direction, and converts packets to data bytes in the receive direction.

PCIe Links and Lanes

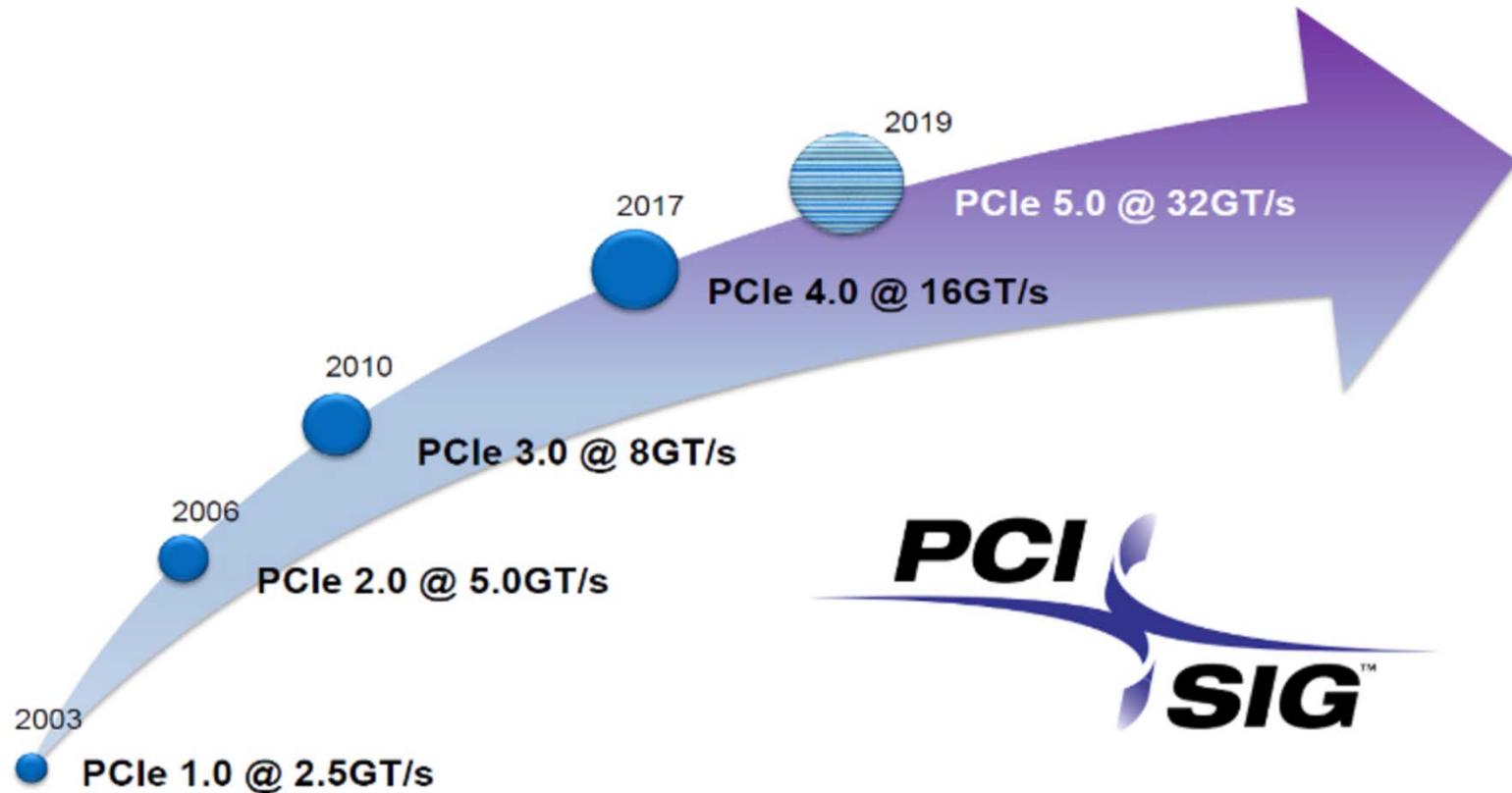
- Two PCIe ports are connected by a point-to-point bidirectional *link*.
- Each link includes 1, 2, 4, 8, 16 or 32 independent bidirectional *lanes*.
- Each lane has a differential pair for carrying the signal in each of the two directions. The connections are thus *duplex*.
- Line codes (e.g., 8b/10b or 128b/130b) are used, and so the raw bit bandwidth (in bit transfers per second per lane) needs to be reduced accordingly to get the actual effective data bandwidth.
- The bandwidth of a PCIe connection can be given as:
 - a) A transfer rate, which includes the overhead bits from the code
 - b) A data rate in bits per second, which omits the overhead bits
 - c) A data rate in Bytes per second, which omits the overhead bits

Standard PCIe Connectors



- The various PCIe standards are both forward and backward compatible with respect to the connectors.
- Also, if an open-ended connector is used and sufficient physical space is provided on the board, a wider-lane PCIe board can be plugged into a narrower PCIe connector. For example, you may be able to plug a x4 board into a x1 connector.

Updated and Enhanced Versions of PCIe



Courtesy of the Peripheral Component Interconnect Special Interest Group (PCI-SIG)

Updated and Enhanced Versions of PCIe

Standard	Year	Clock	Code	Transfer rate	x1 BW	x4 BW	x16 BW
PCIe 1.0a	2003	1.25 GHz	8b/10b	2.5 GT/s	250 MB/s	1 GB/s	4 GB/s
PCIe 2.0	2007	2.5 GHz	8b/10b	5 GT/s	500 MB/s	2 GB/s	8 GB/s
PCIe 3.0	2010	4.0 GHz	128b/130b	8 GT/s	985 MB/s	3.94 GB/s	15.75 GB/s
PCIe 4.0	2017	8.0 GHz	128b/130b	16 GT/s	1.97 GB/s	7.877 GB/s	31.51 GB/s
PCIe 5.0	2019	16.0 GHz	128b/130b	32 GT/s	3.938 GB/s	15.75 GB/s	63.0 GB/s

Note: One Blu-ray™ disk contains 50 GBytes, which can encode about 4 hours of high-definition video, depending on the degree of data compression.

PCIe Protocol Stack

- ***Transaction Layer***
 - implements split transactions (read cmd. first, data later)
 - performs data packetization & depacketization
- ***Data Link Layer***
 - checks packet sequence numbers
 - checks data integrity using a 32-bit cyclic redundancy code (CRC)
 - acknowledges back to transmitter the successful (ACK) or unsuccessful (NAK) reception of each packet
- ***Physical Layer***
 - PCIe 1.0 and 2.0 used the 8b/10b line code
 - PCIe 3.0-5.0 use a more efficient 128b/130b code
 - data bytes are "striped" across multiple lanes

FPGA Accelerator Board with PCIe x16

Photo courtesy of BitWare Inc.



- BitWare S5PE-DS FPGA accelerator board with two Intel Stratix V FPGAs
- Card edge connector supports PCIe Gen1, Gen2 & Gen 3 with x16 lanes

PCI and PCIe Connectors on a Motherboard

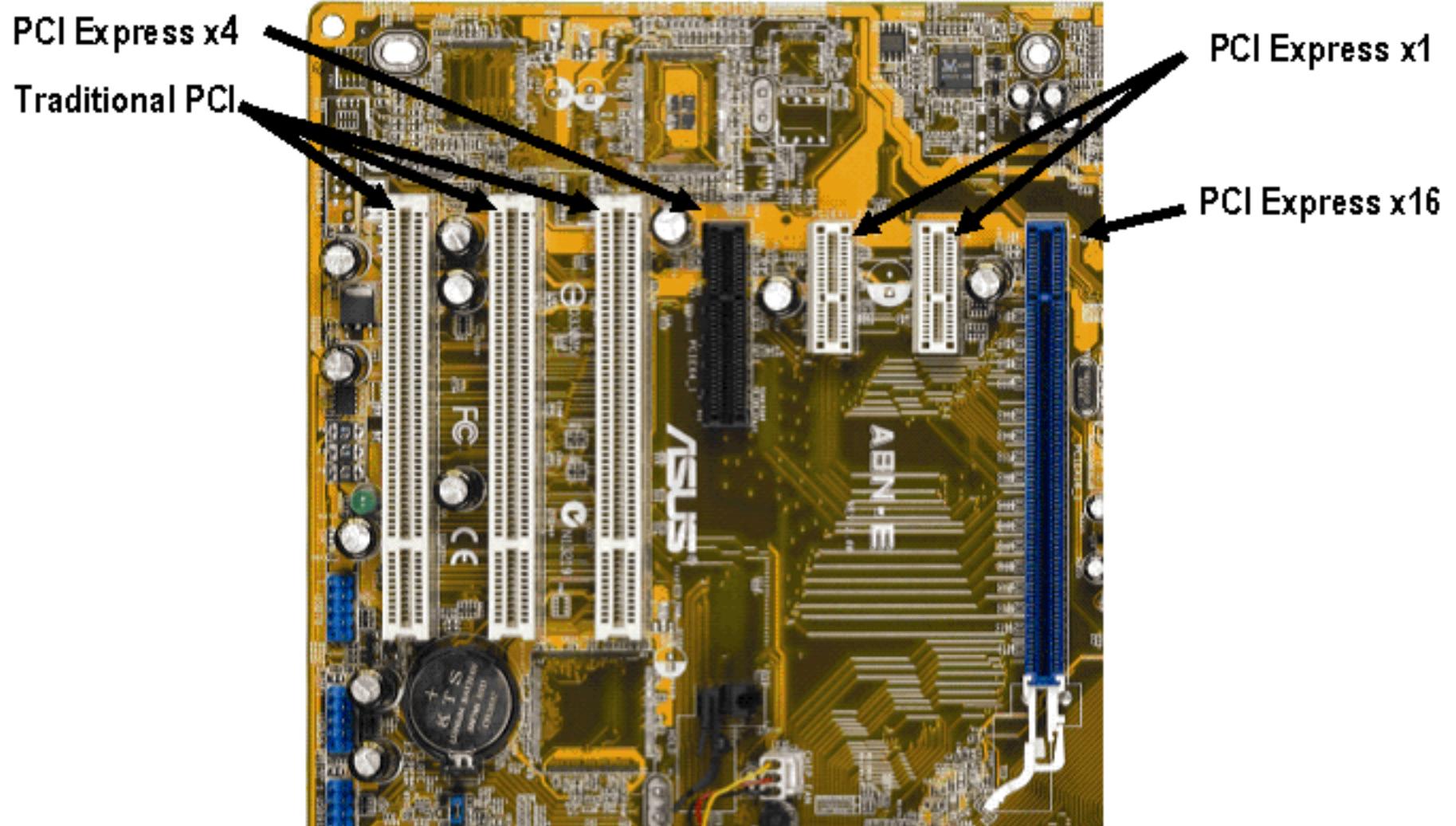


Photo courtesy of AsusTek Computer Inc.