Embedded System Development

LISHA/UFSC

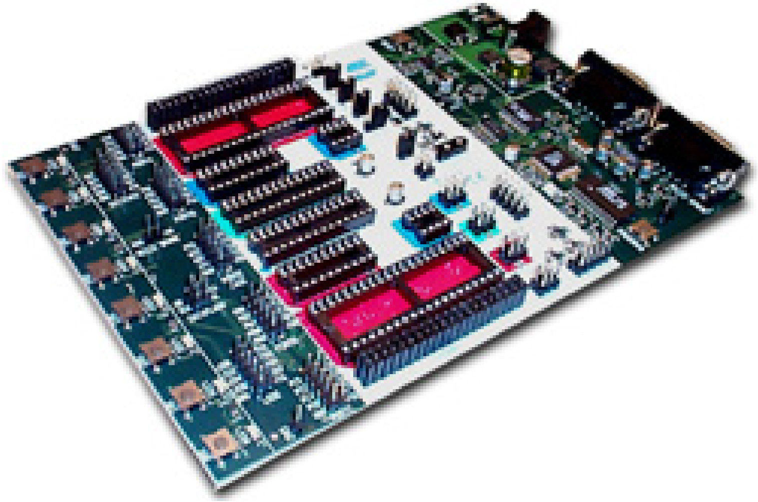
Prof. Dr. Antônio Augusto Fröhlich
Fauze Valério Polpeta
Lucas Francisco Wanner

http://www.lisha.ufsc.br/~guto

March 2009

Hardware Support Tools

- Proto-boards
- Development kits
  - Microcontroller sockets
  - Input and output
    - Leds and buttons
  - Flash writer
  - JTAG



Microprocessors Laboratory

Software Support Tools

- Emulators
  - Easy of development (debugging, live inspections, etc)
- Cross-compilers
  - Develop on a host (e.g. PC with Windows or Linux)
  - Compile for a target ES
- Monitors
  - Upload firmware
  - Integrity checks
- Cross-debuggers
  - Agent on a target ES
  - Debugger on a host (e.g. GDB)

Microprocessors Laboratory

Embedded System Design

- Embedded system design involves both software and hardware elements
  - Software/hardware co-design
- Typical characteristics of embedded systems
  - Single-functioned
    - Executes a single application program, repeatedly
  - Tightly-constrained
    - Low cost, low power, small, fast, etc ...
  - Reactive and real-time
    - Continually reacts to changes in the system’s environment
    - Must compute certain results in real-time without delay

Application-Orientation

- An embedded system exists to perform the tasks specified by a single (set of) application program (s)
- Application requirements guide the development process
  - defining a software system architecture
  - that is build upon a hardware system architecture

Microprocessors Laboratory

Software/Hardware Co-Design

- The design of software and hardware for embedded systems is usually carried out as a single process
- Procedure
  1. Specify application requirements (functional, temporal, etc)
  2. Look for an adequate software architecture that satisfy application requirements
  3. Look for a (the minimal) hardware architecture that is able to support the software architecture defined
  4. Repeat steps 2 and 3 until a compromise is set outputting the design documents necessary to support the implementation of both software and hardware

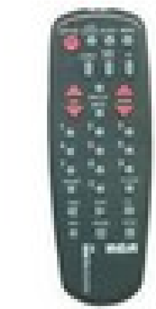
Real-Time Constraints

- Real-time systems

  "A real-time system is a system whose correctness includes its response time as well as its functional correctness".

  (Locke, 2000)

- How to ensure correctness and determinism?
  - Formal modeling
  - Testing and benchmarking
- Classifications
  - Soft real-time
  - Hard real-time



Industry Metrics

- Unit cost
  - the monetary cost of manufacturing each copy of the system
- Non-recurring engineering cost (NRE)
  - the one-time monetary cost of designing the system
- Physical size
- Performance
- Power consumption
- Flexibility
  - the ability to change the functionality of the system without incurring heavy NRE cost

More Industry Metrics

- Time-to-prototype
  - the time needed to build a working version of the system
- Time-to-market
  - the time required to develop a system to the point that it can be released and sold to customers
- Maintainability
  - the ability to modify the system after its initial release
- and of course
  - Correctness
  - Safety
  - etc

Typical Embedded Software Architectures

- Not to to be forgotten
  - The simplest architecture that satisfy the requirements of the application is the best
- Cyclic executive
  - Round-robin
  - Round-robin with interrupts
  - Function queue scheduling
- Real-time operating system

Round-Robin

- Poll all I/O devices in a loop, eventual activating the corresponding tasks
- Example: digital voltmeter
  1. check position of scale button
  2. check status of hold button
  3. read the voltage from A/D converter
  4. perform scale conversions
  5. if !hold then update display
  6. goto 1

Round-Robin Pseudo Code

```
int main(void)
{
  while(true) {
    if( // I/O Device A needs service ) { // Task A
      // Handle I/O from Device A
      // Perform Task A duties
    }
    if( // I/O Device B needs service ) { // Task B
      // Handle I/O from Device B
      // Perform Task B duties
    }
    ...
    if( // I/O Device Z needs service ) { // Task Z
      // Handle I/O from Device Z
      // Perform Task Z duties
    }
  }
}
```

Reasoning about Round-Robin

- Pros
  - Simplicity
    - No interrupts and no shared data
- Cons
  - Maximum waiting time for a device is the loop length
    - Device Z waits the time to handle devices A through Y
    - Loss of interactivity
      - Limited overcome: `A, B, **Z**, C, D, **Z**, E, F, **Z**, ...`
  - Delays in the handling of any device can compromise the servicing of other devices and even the correctness of the whole system

Round-Robin with Interrupts

- Urgent events generate interrupts
  - High-priority tasks are handled inside interrupt service routines (ISR)
  - Low-priority tasks are handled as round-robin tasks implemented on the main routine
- Shared data pitfalls
  - Interaction between tasks and ISRs is handled via shared variables
  - Race conditions must be prevented by proper synchronization of critical sections

Example for Round-Robin with Interrupts

- Full-duplex network bridge
  - ISRs
    - Receive port A incoming packets into a private ring buffer (A -> B)
    - Receive port B incoming packets into a private ring buffer (B -> A)
  - main
    1. if port A is free for sending and there are packets on the corresponding ring buffer then forward a packet
    2. if port B is free for sending and there are packets on the corresponding ring buffer then forward a packet
    3. goto 1

Round-Robin with Interrupts Pseudo Code

```
bool Task_X, Task_Y = false;

void Handle_A(void)
  __attribute__((interrupt))
{
  // Service interrupts from
    I/O Device A
  Task_X = true;
}

void Handle_B(void)
  __attribute__((interrupt))
{
  // Service interrupts from
    I/O Device B
  Task_X = true;
  Task_Y = true;
}
```

```
int main(void)
{
  while(true) {
    if(Task_X) { // Task X
      // Perform Task X
     duties
      Task_X = false;
    }

    if(Task_Y) { // Task Y
      // Perform Task Y
     duties
      Task_Y = false;
    }
  }
}
```

Race Conditions in Round-Robin with Interrupts

```
bool Task_X = false;

void Handle_A(void) __attribute__((interrupt)) {
  // Service interrupts from I/O Device A
  Task_X = true;
}
...

int main(void)
{
  while(true) {
    if(Task_X) { // Task X
      // Perform Task X duties
      Task_X = false;
    }
    ...
  }
}
```

```
load r1, 1
load r2, &Task_X
store (r2), r1
```

interrupt

```
load r3, 0
load r4, &Task_X
store (r4), r3
```

Reasoning about Round-Robin with Interrupts

- Pros
  - Ability to handle urgent events due the different priority of ISRs and round-robin tasks
- Cons
  - Round-robin tasks have all the same priority
    - some code gets shifted into ISRs
    - ISRs tend to grow, thus generating delays
  - Responsiveness of round-robin tasks depends on asynchronous external events
    - Worst case: all tasks + variable ISRs
  - Race conditions
    - demand synchronization

Function Queue Scheduling

- ISRs add function pointers to a scheduling queue
  - Priority scheme defined by queue's elements order
    - high-priority tasks enqueued at the head
    - low-priority tasks enqueued at the tail
    - or explicit priority assignments
- Loop in the main function activates the task at the head of the scheduling queue
- Example: surveillance system
  - ISRs for each kind of sensor (in priority order) trigger alarms
  - tasks for handling different levels of alarms

Microprocessors Laboratory

Function Queue Scheduling Pseudo Code

```
Queue Ready;

void Handle_A(void)
  __attribute__((interrupt))
{
  // Service interrupts from
    I/O Device A
  enqueue(Ready, &Task_X);
}

void Handle_B(void)
  __attribute__((interrupt))
{
  // Service interrupts from
    I/O Device B
  enqueue(Ready, &Task_X);
  enqueue(Ready, &Task_Y);
}
```

```
int main(void)
{
  while(true)
    if(!queue_empty(Ready))
      dequeue(Ready)();
}

void Task_X(void) {
  // Perform Task X duties
}

void Task_Y(void) {
  // Perform Task Y duties
}
```

Reasoning about Function Queue Scheduling

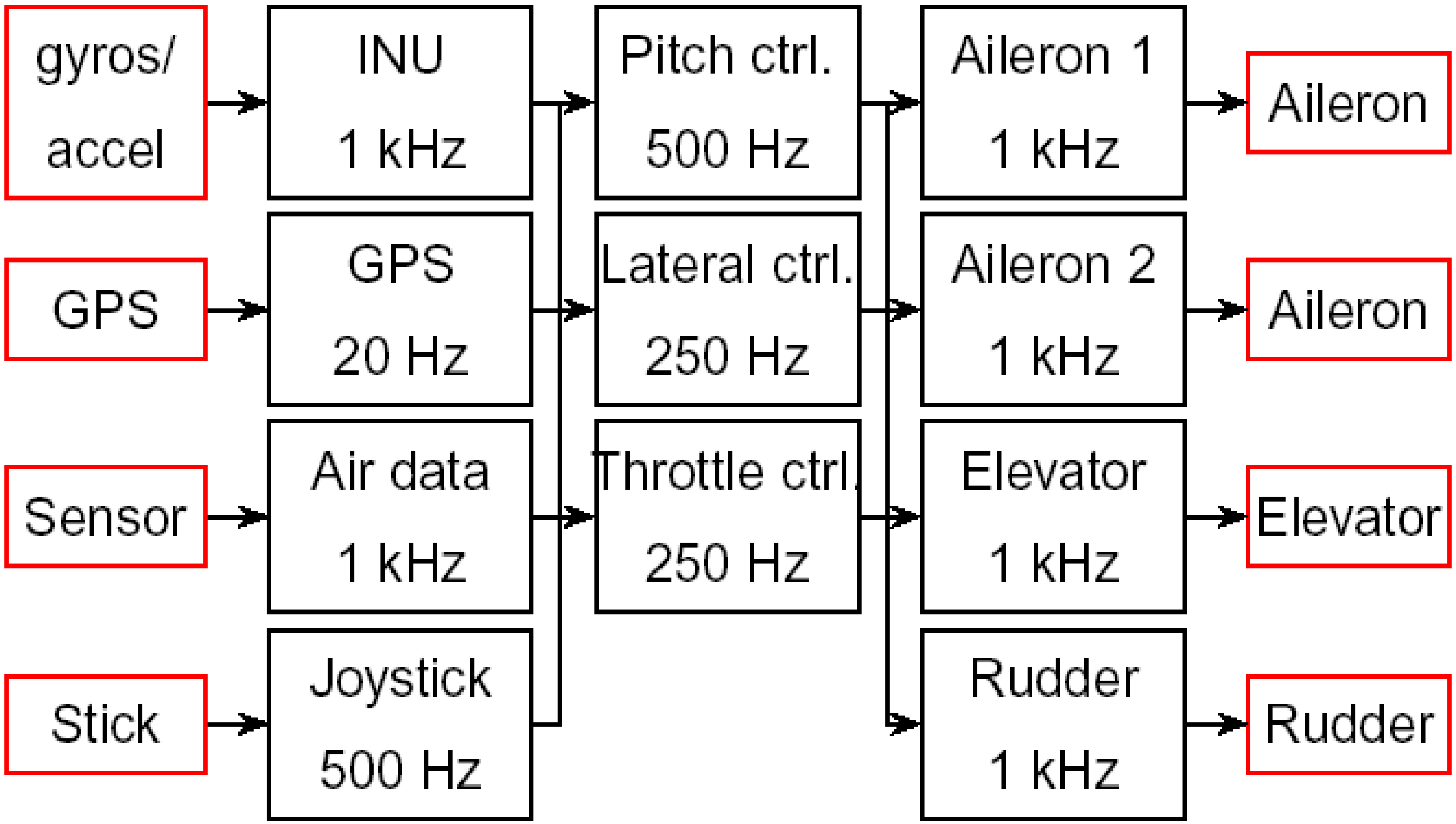
- Pros
  - Ability to define a sophisticated priority scheme
- Cons
  - Longer task code functions can affect system response time
  - A higher-priority task must wait for the current task to release the processor
    - Worst case: time of the longest task
    - Limited overcome: break long tasks in pieces (can be complicated!)

Real-Time Operating System

- Tasks abstracted as processes (threads)
  - Preemptive priority scheduling implemented by the OS
- Interaction between ISR and processes via signals
  - No race conditions
- Some level of hardware abstraction
  - Typical devices: UART, keys, display, etc
  - Sensors and actuators

Example for Real-Time Operating System

- Air plane fly-by-wire system



Real-Time Operating System Pseudo Code

```
Signal Signal_A, Signal_B;

void Handle_A(void)
  __attribute__((interrupt))
{
  // Service interrupts from
    I/O Device A
  signal(Signal_A);
}

void Handle_B(void)
  __attribute__((interrupt))
{
  // Service interrupts from
    I/O Device B
  signal(Signal_A);
  signal(Signal_B);
}
```

```
int main(void)
{
  thr_x = thread_new(Task_X,
                     Pri_X);
  thr_y = thread_new(Task_Y,
                     Pri_Y);

  catch(Signal_A, thr_x);
  catch(Signal_B, thr_y);
}

void Task_X(void) {
  while(true) {
    // Perform Task X duties
    sleep();
  }
}

void Task_Y(void) { ...
```

Microprocessors Laboratory

Microprocessors Laboratory

Reasoning about Real-Time OS

- Pros
  - Improve application development
    - raises the level of abstraction
    - enable software reuse
  - Improve predictability
- Cons
  - The OS itself consumes resources (processing time, memory, etc)

    "... once you decide to use an RTOS, your best design is often the one that uses it least." [Simon:2003]
  - Complex to develop
    - Overcome: you'll probably find one to buy that fulfills your requirements!

Embedded System Programming

- Combination of
  - Paradigms
    - Structured
    - Object-oriented
  - Languages
    - Assembly, C, C++
    - Ada? Eifel?
  - Tools
    - Preprocessors
    - Assemblers
    - Compilers
    - Linkers

Preprocessors

- Preprocessors do mostly simple textual substitution of program fragments
  - Unaware of programming language syntax and semantics
- CPP: the C Preprocessor
  - Directives are indicated by lines starting with `#`
  - Directives to
    - Include other files (`#include`)
    - Define macros and symbolic constant (`#define`)
    - Conditionally compile program fragments (`#ifdef`)

Microprocessors Laboratory

Assembly Language

- Assembly language

  “A symbolic representation of the machine language of a specific processor. “

  (Foldoc)
- Assembler
  - Converts assembly language into machine code
- Is assembly still used?
  - Assembly programming is costly and error-prone
  - But a few low-level tasks cannot be expressed in high-level languages
  - And compilers still need code generators ...

Example of Assembly Program

```
/* Routine to blink the leds on the Atmel STK 500 AVR
  kit */
#define DDRB   0x17  /* I/O PORT B data direction
  register (0 -> in, 1 -> out) */
#define PORTB  0x18  /* I/O PORT B data register */

    .text
.global blink_leds
blink_leds:
    ldi   r20, 0xff   ; set PORTB to output
    out   DDRB, r20

    ldi   r21, 0x00   ; all leds ON
    out   PORTB, r21
    rcall delay       ; cause some delay
    ldi   r22, 0xff   ; all leds OFF
    out   PORTB, r22
    rcall delay       ; cause some delay
```

The C Programming Language

- Designed by Ritchie at Bell Labs in the early 70's
  - As a system programming language for UNIX
  - Embedded system industry standard (ANSI C)
- The “portable assembly language”
  - Allows for low-level access to the hardware mostly like assembly does
  - Can be easily compiled for different architectures
- The “high-level programing language”
  - As high-level as the high-level programming languages of its time
  - No longer suitable for most application development

Microprocessors Laboratory

Example of C Program

```c
#ifndef N
#define N 10
#endif
#define MAX(a,b) \
  ((a) > (b) ? (a) : (b))

int main() {
  int i;
  int result = 5;


  for(i = 0; i < N; i++)
    result = MAX(i,
  result);

  return result;
}
```

preprocessing

cpp / cc -E

```c
int main() {
  int i;
  int result = 5;

  for(i = 0; i < 10; i++)
    result = ((i) >
  (result)
                  ?(i):
  (result));

  return result;
}
```

Microprocessors Laboratory

Mixing C and Assembly (GCC)

- Why to embed assembly in a C program?
  - To gain low-level access to the machine in order to provide a hardware interface for high-level software constructs
- When the compiler encounters assembly fragment in the input program, it simply copies them directly to the output

```c
int main() {
  asm("nop");
  return 0;
}
```

compiling → gcc -S

```
...
main:
    ...
    nop
    ...
```

Example of C with inline Assembly

- Hitachi H8/300 save-context routine

```
void H8_Context_save() {
    asm("orc   #0x80, ccr   \n"
        "mov.w r6, @(14,r0) \n" /* r6 */
        "stc   ccr, r6l     \n"
        "mov.w r6, @(12,r0) \n" /* cc */
        "mov.w r5, @(0,r0)  \n" /* r5 */
        "mov.w r4, @(2,r0)  \n" /* r4 */
        "mov.w r3, @(4,r0)  \n" /* r3 */
        "mov.w r2, @(6,r0)  \n" /* r2 */
        "mov.w r1, @(8,r0)  \n" /* r1 */
        "mov.w r0, @(10,r0) \n" /* r0 */
        "mov.w @(0,r7), r6  \n"
        "mov.w r6, @(16,r0) \n" /* pc */
        "mov.w @(14,r0) ,r6 \n" /* Restoring r6 */
        "andc #0x7F, ccr    \n"
        "rts                \n");
}
```

GCC Extended Assembly

- `asm` statements with operands that are C expressions
- Basic format

```
asm("assembler template"
    : output operands                /* optional */
    : input operands                 /* optional */
    : list of clobbered registers /* optional */
);
```

Microprocessors Laboratory

GCC Extended Assembly

- Assembler template
  - The set of assembly instructions that will be inserted in the C program
  - Operands corresponding to C expressions are represented by “%n” in the asm statement, with “n” being the order in which they appear in the statement
  - Example (IA-32)

```
int a = 10, b;
asm("movl %1, %0;"
    :"=r"(b) /* output operands */
    :"r"(a)  /* input operands */
    :        /* clobbered register */
);
```

GCC Extended Assembly

- Operands
  - Preceded by a constraint

    `r` operand in a general purpose register

    `m` operand in memory (any supported addressing mode)

    `o` operand in memory, address must be offsetable

    `i` operand is an immediate (integer constant)

    `...` many others, including architecture-specific ones
  - Input operand constraints
    - Are met before issuing the instructions in the `asm` statement
  - Output operand constraints (begin with “=”)
    - Are met after issuing the instructions in the `asm` statement
- Example (AVR8)

```
asm("" : : "z"(mem_prog) : );
```

GCC Extended Assembly

- Clobber list
  - Some instructions can clobber (overwrite) registers and memory locations
  - By listing them, we inform the compiler that they will be modified and their original values should no longer be trusted
  - Example (IA-32)

```
int a = 10, b;
asm("movl %1, %%eax; movl %%eax, %0;"
    :"=r"(b)  /* output operands */
    :"r"(a)   /* input operands */
    :"%eax"   /* clobbered register */
);
```

GCC Extended Assembly

- Volatile assembly
  - When the assembly statement must be inserted exactly where it was placed
  - When a memory region accessed by the assembly statement was not listed in the input or output operands
  - Example (IA-32)

```
int a=10;
asm __volatile__ ("movl %0, 0xfefa;"
                  :            /* output operands */
                  :"r"(a)      /* input operands */
                  :            /* clobbered register */
);
```

The C++ Programming Language

- Designed by Stroustrup at Bell Labs in the early 80's
  - As a multiparadigm programming language
  - Superset of C (a C program a valid C++ program)
  - Strongly typed
  - Supports object-oriented programming (classes , inheritance, polymorphism, etc)
  - Supports generative programming techniques
- Embedded software != applicative software
  - Rational use of late binding (polymorphism, dynamic casts, etc)
  - Extended use of static metaprogramming
  - Always take a look at the assembly produced

Example of C++ Program

```
struct AVR8 {
    Reg8 r0;
    // ...
    union {
        struct{
            Reg8 r30;
            Reg8 r31;
        };
     Reg16 z;
    };
};

struct AT90S: public AVR8
{
    static const unsigned
  short RAM_SIZE = 0x0200;

    IOReg8 reserved_00;
    // ...
    IOReg8 sreg;

    char ram[RAM_SIZE];
};

int main()
{
    AT90S * at90s = reinte
  rpret_cast<AT90S *>(0);

    at90s->ddrb = 0xff;
    while(true) {
     at90s->portb = 0;
     delay();
     at90s->portb = 0xff;
     delay();
    }

    return 0;
}
```

Microprocessors Laboratory

Mixing C++ and C

- C++ and C use different linkage and symbol generation conventions
  - C++ does name mangling
    - Symbols corresponding to member functions embed parameter types
- In order to call C functions from C++

```
extern “C” { /* C function prototypes */ }
```

- In order to call C++ functions from C
  - one has to know the mangled function names

Linking

- Linkage
  - The process of collecting relocatable object files into a executable
- Styles of linking
  - Static linking, dynamic linking, runtime linking
- Linker scripts

```
SECTIONS {
.text 0x8000:   {
        *(.text)
        *(.rodata)
        *(.strings)
         _etext = .
        }  > ram
.data : {
        *(.data)
        *(.tiny)
         _edata = .
        }  > ram
}
```

Microprocessors Laboratory

Microprocessors Laboratory

Embedded Software Debugging

- Debugging

  "Debugging is the process of locating and fixing errors (known as bugs), in a computer program or hardware device"

  (PIE Software Inc.)

- Strategies of debugging
  - Leds
  - Display
  - Serial
  - GDB Client
  - JTAG

Embedded Software Debugging

- GDB Client
  - GDB provides a "remote target" debugging capability across a serial port or network connection
    - A small program running on the target hardware helps GDB carry out requests to monitor and control the application being debugged

Microprocessors Laboratory

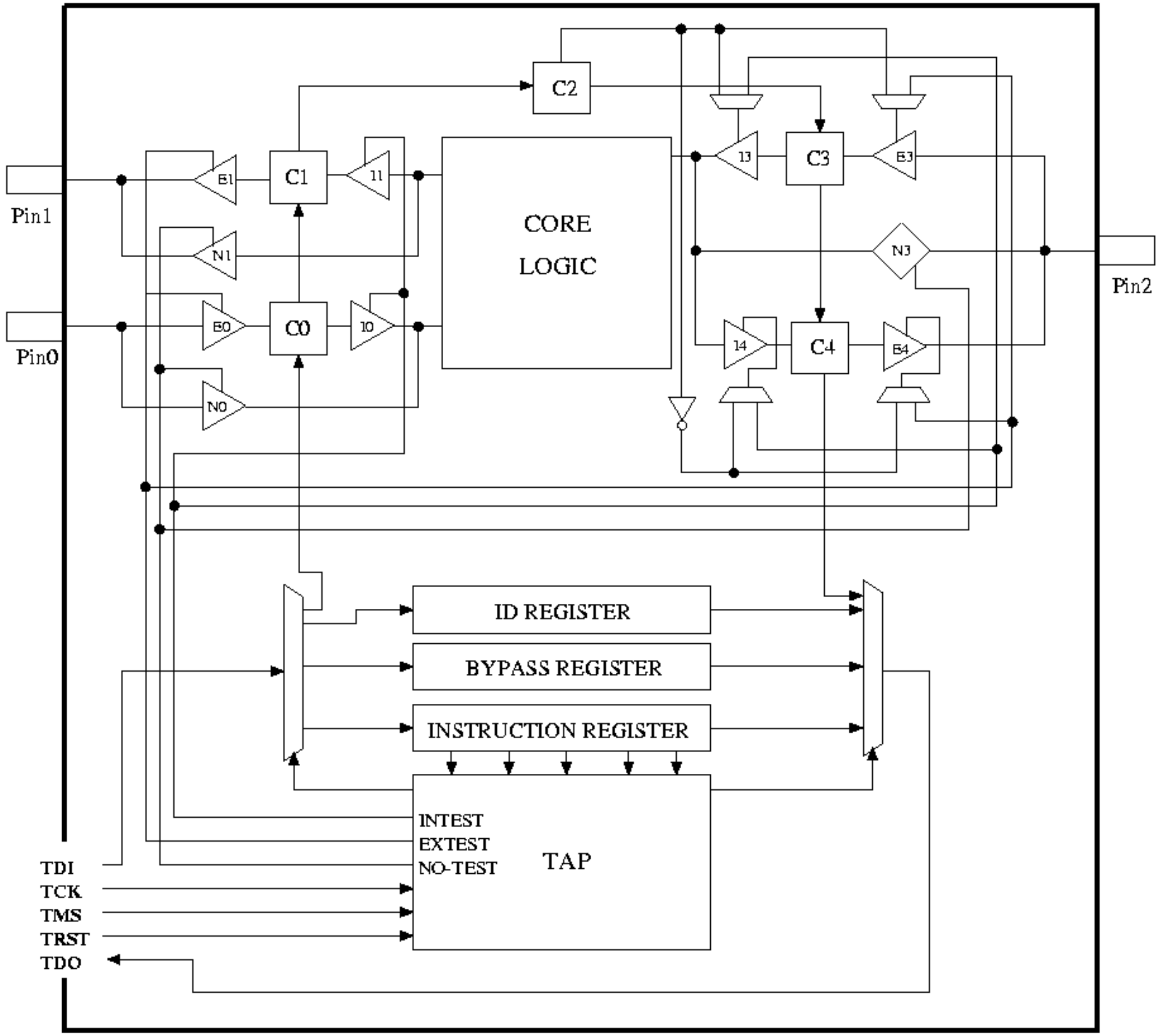
Joint Test Access Group (JTAG)

- Formed in 1985 to develop a method to test populated circuit boards after manufacture
- Defined a test and programming interface for digital IC's used by over 200 electronic companies
  - Large shift register through the entire IC where each bit (ports, RAM, register etc.) can be accessed like a conveyor belt in a parcel distribution
- First processor released with JTAG in 1990
  - Intel 80486

JTAG: Interface

- Interface
  - Test Data Input (TDI)
  - Test Mode Select (TMS)
  - Clock (TCK)
  - Reset (TRST)
  - Test Data Output (TDO)

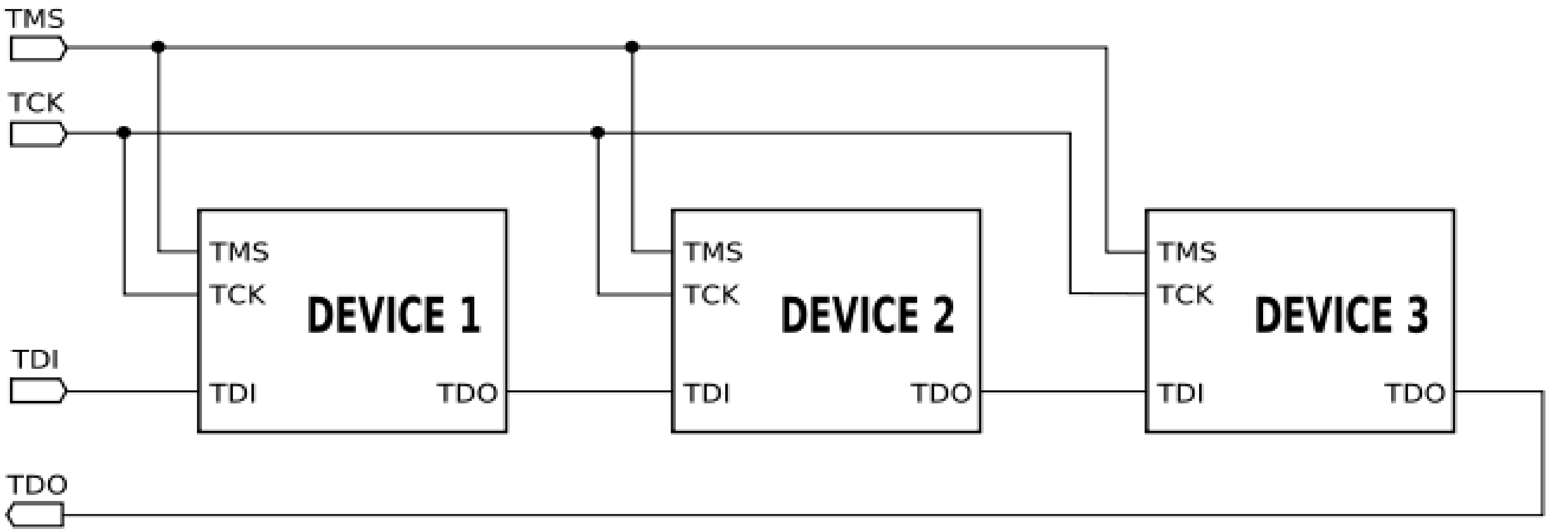
JTAG: Boundary Scan



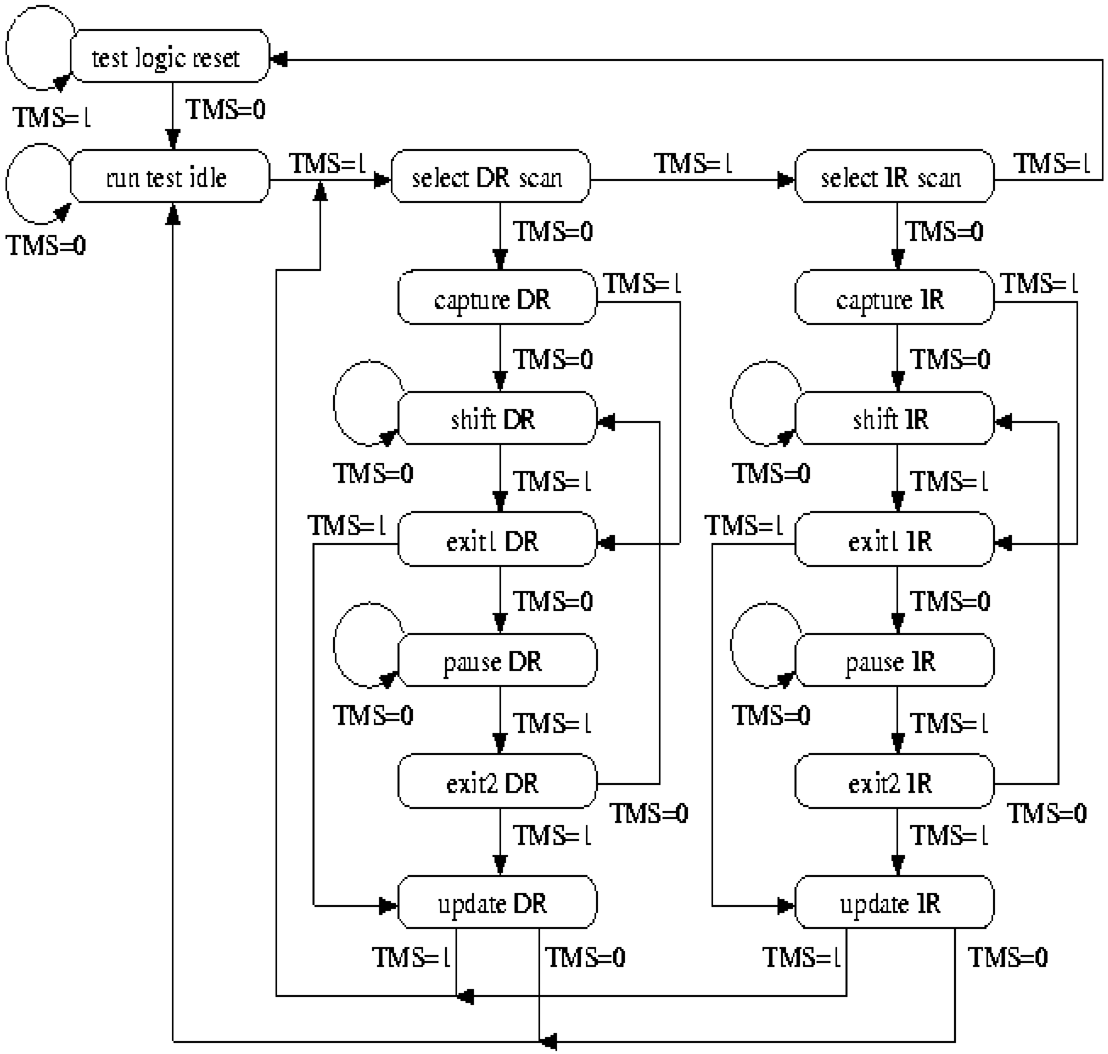
JTAG Boundary Scan Interface Architecture

```
Pin(0,2): Input
Pin(1,2): Output
C(0-4): BSR
I(0-4): Internal
E(0-4): External
N(0-4): Normal
IDR: Hardwired
BR: 1-clk delay
IR: TAP instr
```

JTAG: Daisy Chain



JTAG: TAP State Machine



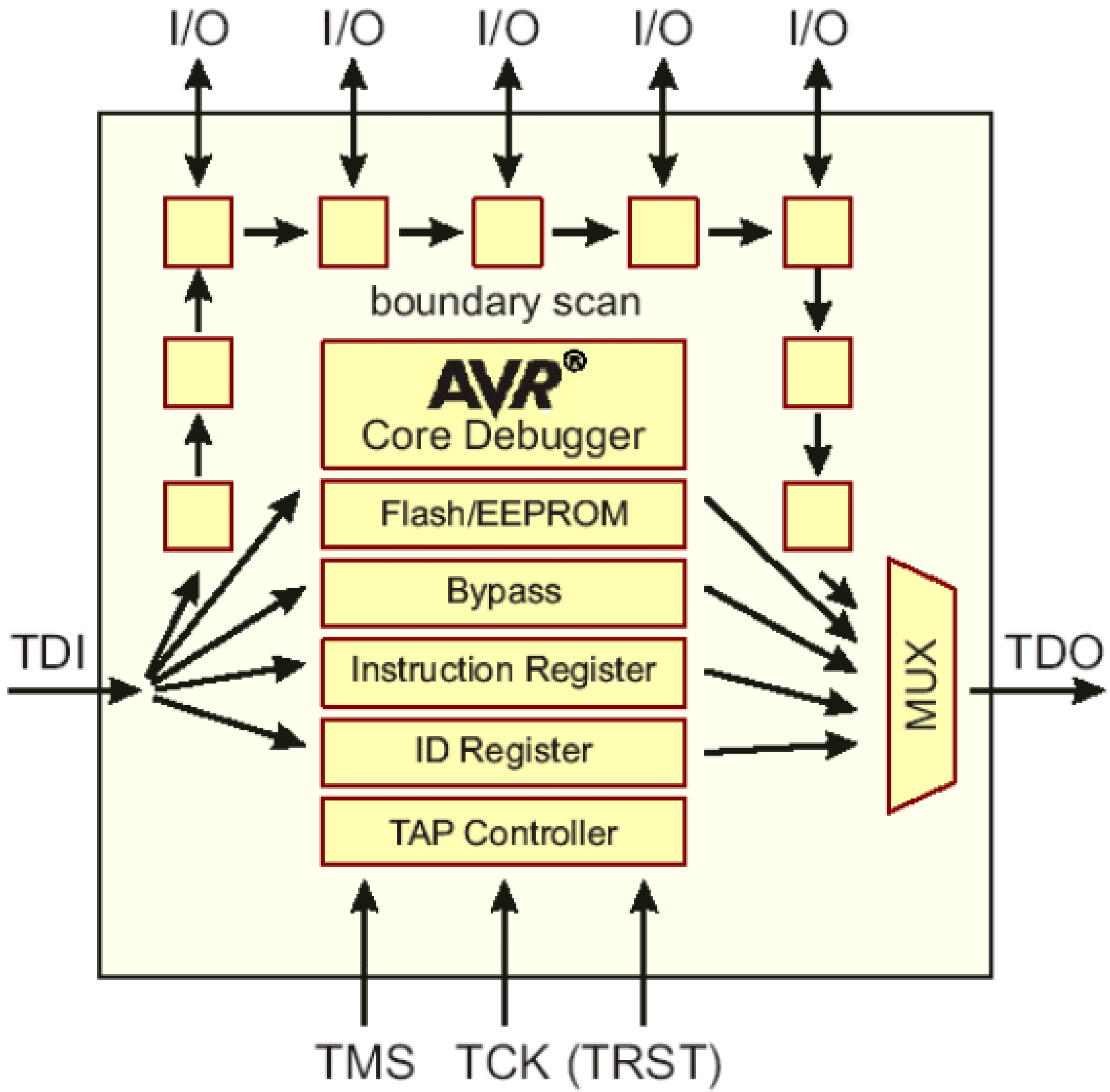
JTAG Test Access Port (TAP) controller state transition diagram

```
*-> test logic reset
--> run test idle
--> select DR scan
--> select IR scan
--> capture IR
--> shift IR --> ... n
  times ... --> shift
  IR
--> exit1 IR
--> update IR
--> run test idle ->*
```

JTAG: Operation

- Instruction and Data Register -> IR(IR path)
- Value -> DR (DR path)(N times)
- Public Instructions
  - BYPASS
  - IDCODE
  - EXTEST
  - INTEST
- Private

Case Study: AVR JTAG



Microprocessors Laboratory

Case Study: AVR JTAG

- The flash and EEPROM memory in the AVR can be programmed in-system
- Each peripheral unit of the controller can be easily accessed, tested and debugged
- The CPU can be stopped or single-stepped

Embedded Software Testing

- Testing

  "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."

  (IEEE Std 610.12-1990)

  "A disciplined process that consists of evaluating the application (including its components) behavior, performance, and robustness - usually against expected criteria"

  (Vincent Encontre – IBM)

Embedded Software Testing

- Why testing
  - To check if product meets functional and performance targets
  - To ensure safety and regulatory compliance
  - To ensure that production standards are met
- Errors in embedded software are critical
  - We cannot restart an embedded system using “ctrl +alt+del”
- If we don't test the system, the user will do it

Embedded Software Testing

- In 1979 an AT&T software bug knocked out all long-distance phone service to Greece

```
switch(MessageType) {
      case INCOMING_MESSAGE
         if (RemoteSwitch == NOT_IN_SERVICE) {
            if (LocalBuffer == EMPTY )
              SendInServiceMsg(3B);
            else
              break; /*Bad News!*/
   }
   ProcessIncomingMessage(); /*Statement skipped*/
   break;
//...
}
```

Embedded Software Testing

- Challenges in testing embedded systems
  - Coexistence of various implementation paradigms
  - Lack of clear design models
  - A wide range of deployment architectures
  - Limited direct interfaces
  - Limited processing resources and spare memory
  - Physical restrictions
  - Many test tools don’t support embedded testing

Types of Testing

- Black box versus White box
  - Black box testing assumes no knowledge of the internal structure or design of the product
  - White box testing has detailed knowledge of internal structure and design
- Conformance versus Benchmarking
  - Conformance testing checks that product meets its specifications
  - Benchmarking records/characterizes the level of performance, capability, capacity, etc. that the product has

Microprocessors Laboratory

Types of Testing

- Qualification versus Regression
  - Qualification testing checks that the product first meets a required objective
  - Regression testing checks that it continues to meet that objective after some change has been made to the product
- Structured versus Ad-hoc testing
  - Structured testing defines the precise details related to the test before execution
    - This usually follows a defined process from specifications through to documented tests
  - Ad-hoc testing uses a tester's experience to direct the testing activities

Types of Testing

- Controlled versus Live environment testing
  - Controlled environments allow the operations and behavior of a target product to be exercised in controlled and measured way
    - This allows greater repeatability of testing
  - Live (in-service) testing in a full operational environment will always be needed to some degree (e.g. Beta-test, acceptance test)
    - Inevitably, live testing is always less controlled and predictable

Strategies of Testing

- Test scaffolds
  - A software that provides the same entry points as does the hardware-dependent code on the target system, and it calls the same functions in the hardware-independent code
  - The host system is a much friendlier environment for testing than the target
- Instruction set simulators
  - Programs that run on host and mimic the target microprocessor and memory
  - Help to determine response and throughput and to test your startup code

Strategies of Testing

- Assert macro
  - The assert macro tests assumptions in the code and forces the running program to stop immediately if one of those assumptions is false
- Laboratory tools
  - Multimeters, oscilloscopes, logic analyzers, etc.
- Monitors
  - A combination if software and hardware to give you standard debugging capabilities (leds, displays ...)