مدينة زويل للعلوم والتكنولوجيا
Zewail City of Science and Technology

COMMUNICATION AND INFORMATION ENGINEERING

CIE 314
Embedded Systems Fundamentals

Lecture #8
Multi-Tasking Design Methodology

Instructor:
Dr. Ahmad El-Banna

Agenda

Multi-Tasking Design Methodology

- Polling-Interrupts -RTOS

Software Design Issues

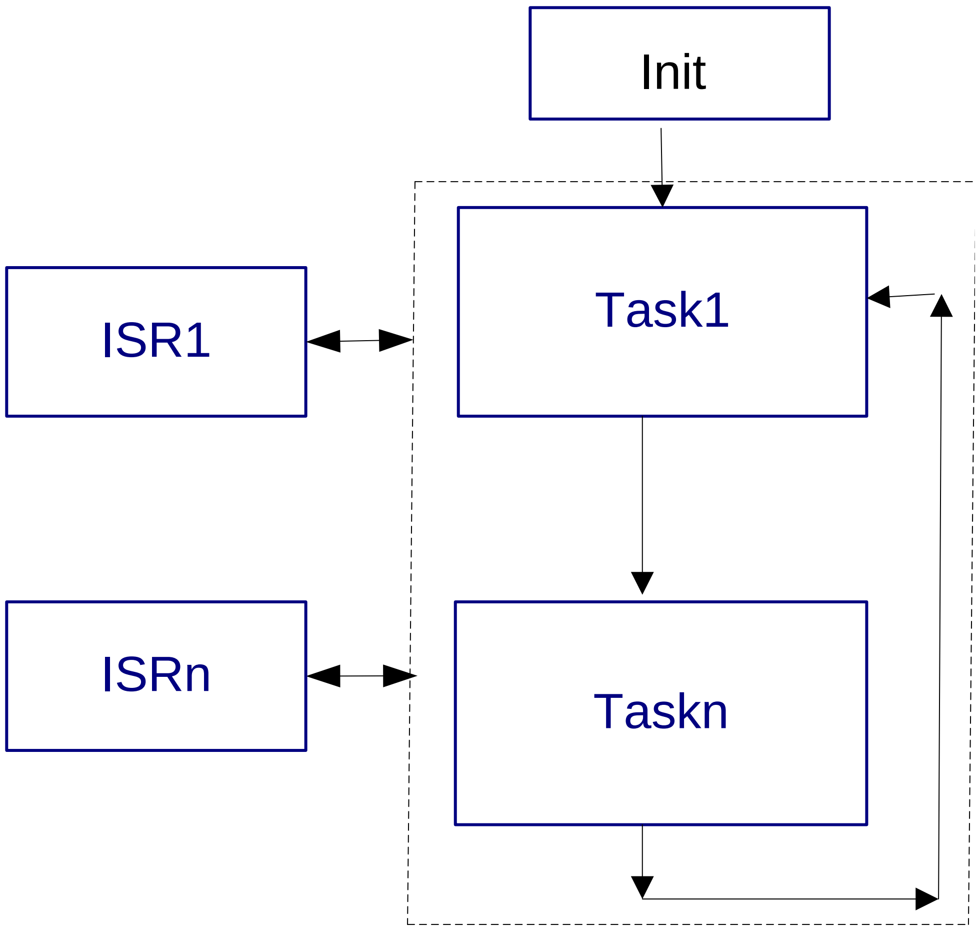
- Task Interactions - Resource Sharing

Design Tips

Multi-Tasking Design Methodology

- Almost all real time embedded systems are real time reactive which must react external events or internal timer events within an expected time limit.
- After the system time requirement analysis and system modeling we can start the software design which will provide a guideline for software coding.
- One of the classical modeling patterns for real time embedded system is a simple explicit loop controlled state Chart for soft real-time operating systems.

Simple Loop Architecture



Contd.

- The shortcoming is when the task k_i is waiting for an unavailable resource the task k_{i+1} can not precede and it will let some other tasks fail to meet the response deadline requirement.
- There is no priority preference among the tasks.
- The advantage is its simplicity and no RTOS support is needed.
- There are many different ways to schedule and design a multi-tasking real time system due to the system complexities and time constraint requirements.
- You can write a task scheduler on your own using polling external events or using external and internal timer interrupts, or use a commercial RTOS.

Polling

- The simplest looping approach is to have all functional blocks including the event polling functions in a simple infinite loop like a Round-Robin loop.

```
main()
{
  while(1)
  {
     function1();
     function2();
     function3();
  }
}
```

- Here the function1 and function2 may check the external data every 50ms.
- The function3 may store the collected data and make some decisions and take actions based on the collected data.

Contd.

- The question is how to control the timing?
- Without a timer control interrupt due to various reasons such as not enough ports and interrupts available, you can design a time_delay function.

```
void time_delay(unsigned int delay)
{ unsigned int i,j;
  for(i=0; i<=delay; i++)
 {
 for (j=0; j<=100; j++);
  }
}
```

- A function call of *time_dealy(1)* will produce approximate 1 ms for 12 MHz of 8051.
- You can estimate it in this way: The 8051 runs at 1MIPS, the inner loop has 10 assembly machine instructions (by View -> Disassemble window in μvision) and 100 iterations takes about 100x10 μs = 1 ms.

Contd.

- Assume that all function execution time are very short and can be ignored.
- You can insert a time_deay(50) at the end of each cycle to make program poll the I/O ports every 50 ms.

```
main()
{
  while(1)
  {
     function1();
     function2();
     function3();
     time_delay(50);
  }
}
```

- Here we ignore the execution time of all functions.
- If the total execution time of these three functions is 10ms then we can adjust the time delay to 40 ms.
- Of course, in reality you don't see this implementation very often because the time control is not accurate and it is not appropriate for any hard real time systems.
- For very simple application with limited timers and interrupt ports, you can still use this design style

Interrupts

- A popular design pattern for a simple real time system is a division of a background program and several foreground interrupt service functions.
- For example, an application has a time critical job which needs to run every 10ms and several other soft time constrained functions such as interface updating, data transferring, and data notification.
- **Foreground:**

```
void critical_control interrupt INTERRUPT_TIMER_1_OVERFLOW
{
// This ISR is called every 10 ms by timer1
}
```

- **Background:**

```
main()
{  while(1) {
   function1();
  function2();
  function3();   } }
```

Contd.

- This simple background loop with foreground interrupt service routine pattern works fine as long as the ISR itself is short and runs quick.
- However, this pattern is difficult to scale to a large complex system. E.g., the critical time control ISR function itself needs to wait for some data to be available, or to look up a large table, or to perform complex data transformation and computation.
- In this situation, the ISR itself may take more than 10ms and will miss the time deadline and also breach the time requirements for other tasks.

Flag Control

- An alternative solution is to have a flag control variable to mark the interrupt time status and to split the ISR into several sub states as follows.

```
int timerFlag;    // global data needs a protection
void isr_1 interrupt INTERRUPT_TIMER_1_OVERFLOW
{
    timerFlag = 1;
    // This ISR is called every 10 ms by timer1 set
}
int states ;

critical-control()
{
    static states next_state = 0;
    switch(next_state)
    {
        case 0:
           process1();
           next_state=1;
           break;
        case 1:
           process2();
           next_state=2;
           break;
        case 2:
           process3();
           next_state=0;
           break;
    }
}
```

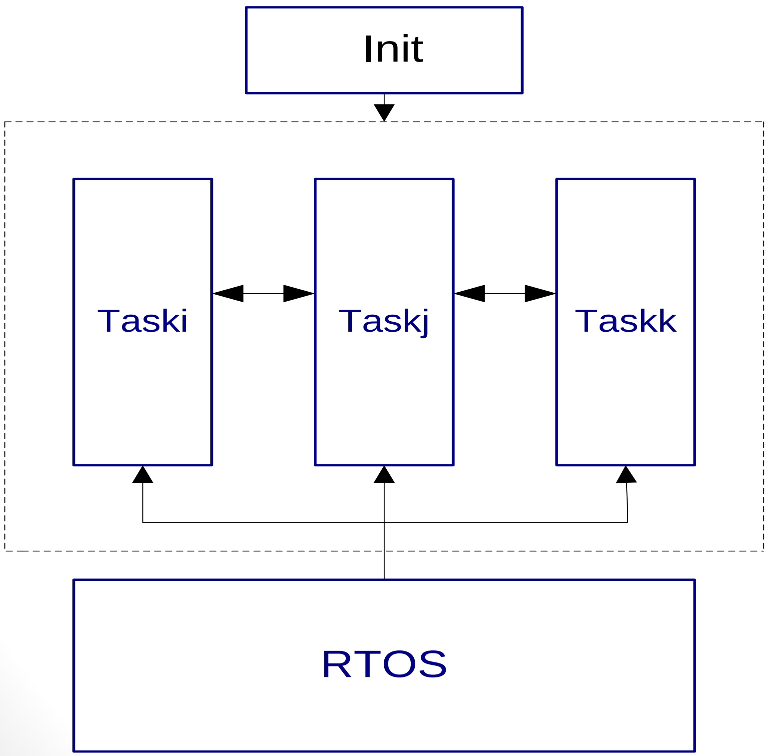
```
main()
{
    Init();
    while(1)
    {
        if (timerFlag)
        {
            critical_control();
            timerFlag=0;
         }
         function1();
         function2();
         function3();
     }
 }
```

Contd.

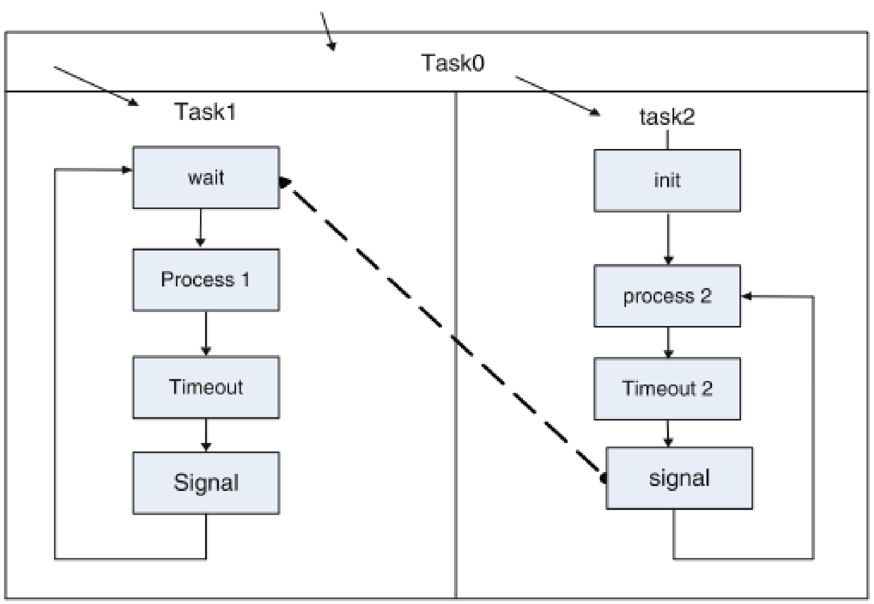
- A rule of thumb for the timer interval is always to make the interval shorter enough to ensure the critical functions get serviced at desired frequency.
- For a large and complex real time system with more than dozen concurrent tasks, you need to use RTOS to make priority-based schedule for multi-tasking jobs.

RTOS

- RTOS makes complex hard real time embedded software design much easier.
- The links between tasks can represent the synchronization signals, exchange data, or even a timeout notification between tasks.



Parallel Architecture



A simple state Chart diagram

Contd.

- RTOS is a background program which controls and schedules executions and communications of multiple time constrained tasks, schedules resource sharing, and distributes the concerns among tasks.
- There are a variety of commercial RTOS available for various microcontrollers such as POSIX(Portable Operating System Interface for Computing Environments) and CMX-RTX.
- RTOS is widely used in complex hard real time embedded software.
- Here is a simple pseudo example of RTX51 for 8051 microcontroller for you to get first touch to the RTOS.

Implementation

```
#include <reg51.h>
#include <rtx51.h>

void task1 (void) _task_ 1
        //task1 is assigned a priority 0 by default
        //4 priority levels: 0-3 in RTX51
{
   while(1)
   {
     os_wait(K_SIG,0,0);// wait in blocked state
                        // for signal to be activated
                        // to a ready state

       proc1();
       os_wait(K_TMO,30,0); // wait for timeout
       os_send_signal(2);   // send signal to task 2
     }
   }
```

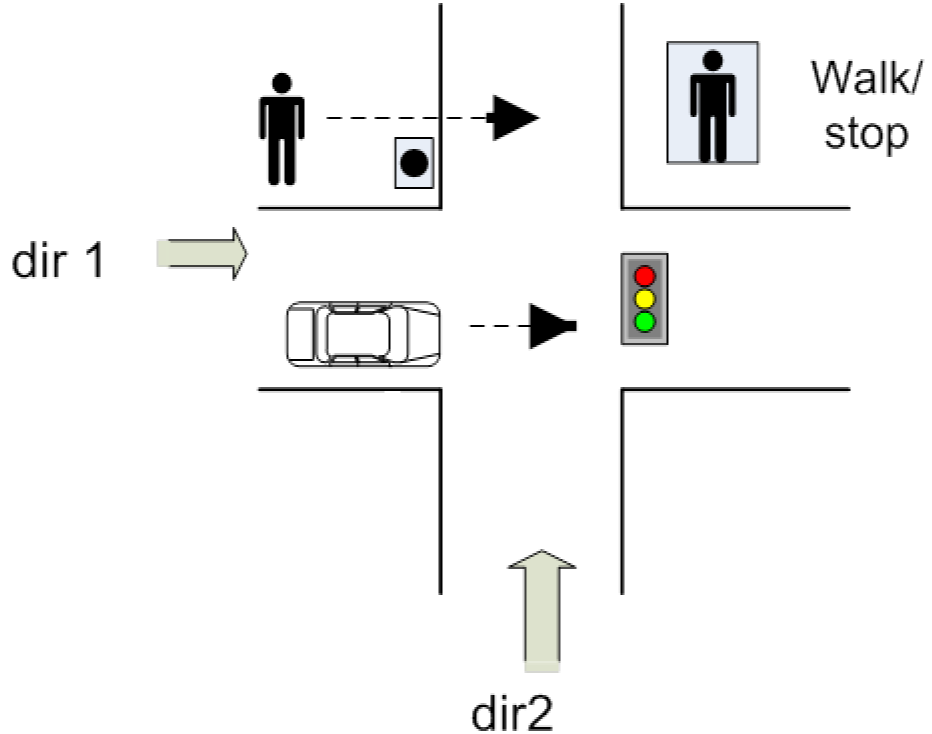
```
void task2 (void) _task_ 2   //Task 2
{
    while(1)
    {
        os_wait(K_SIG,0,0);
        proc2();
        os_wait(K_TMO,30,0);
        os_send_signal(1);   //send a signal to task1
    }
}


void start_task (void) _task_ 0 _priority_ 1
                //task 0 with higher priority 1
{
    system_init();
    os_create_task(1);      //make task1 ready
    os_create_task(2);      //make task2 ready
    os_delete_task(0);      //make task0 itself sleep
}


void main (void)
{
    os_start_system (0);        //start up task0
}
```

Assignment

For the shown typical Traffic Light Control System,

1. Sketch a FSM for that system
2. Assume execution times and deadlines for the tasks that you indicated and sketch a time frame using RMS scheduling.

(Write any assumption that you used)



DESIGN TIPS

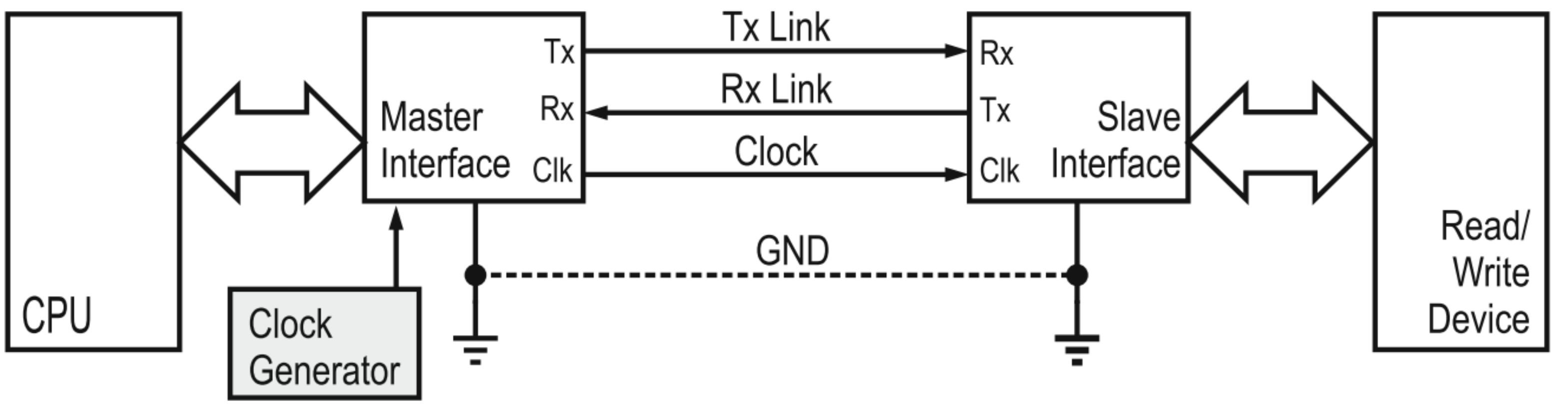


Fig. 9.20 A synchronous, full-duplex serial channel denoting the transmitted clock

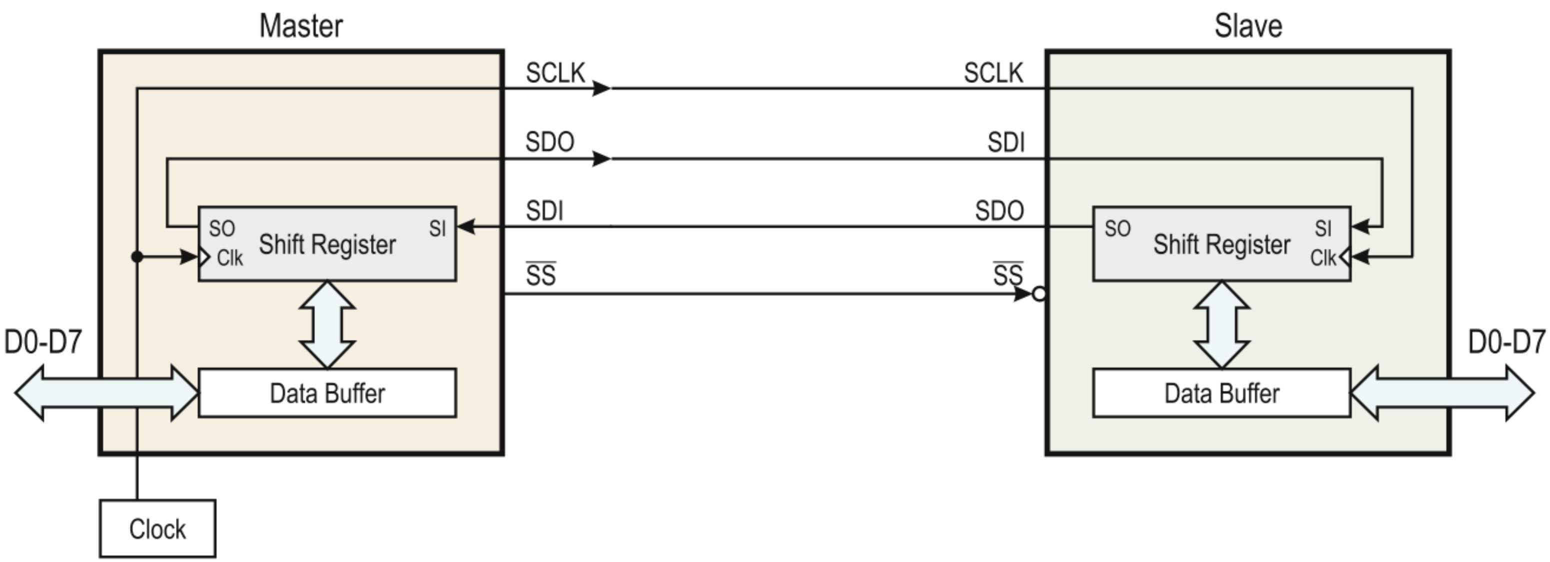


Fig. 9.21 SPI synchronous bus for a point-to-point connection

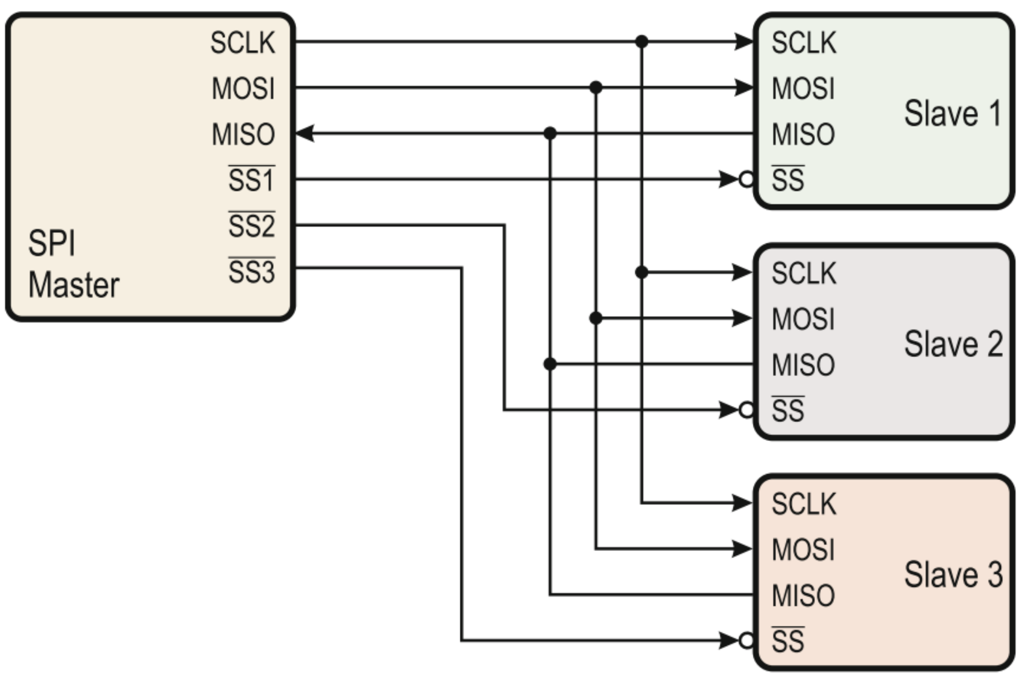


Fig. 9.22 SPI single-master, multi-slave connection

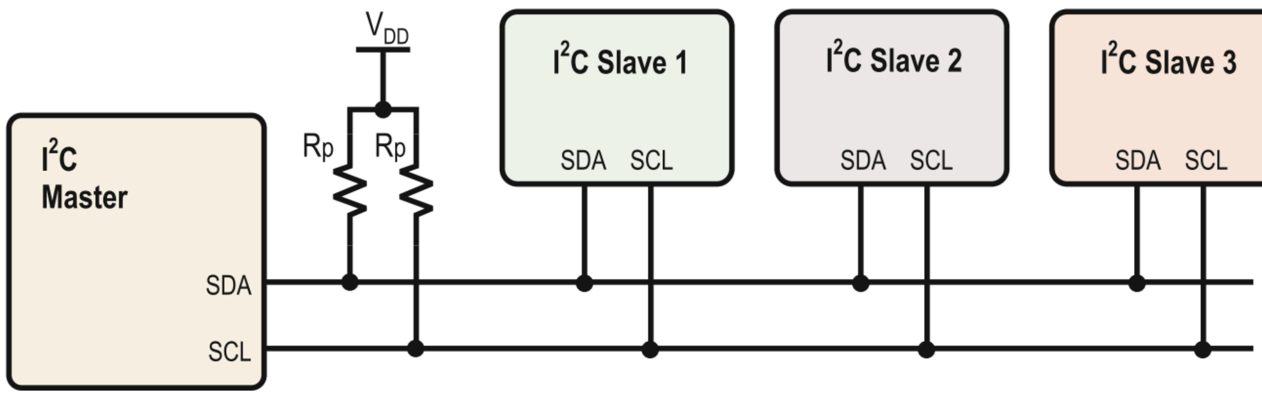


Fig. 9.23 Topology of an I^2C bus connection

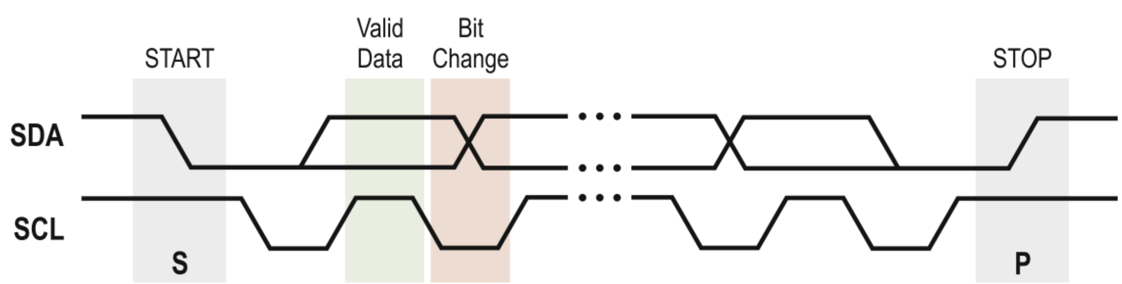


Fig. 9.25 Timing diagram of I^2C signals denoting the start, transfer, and stop conditions

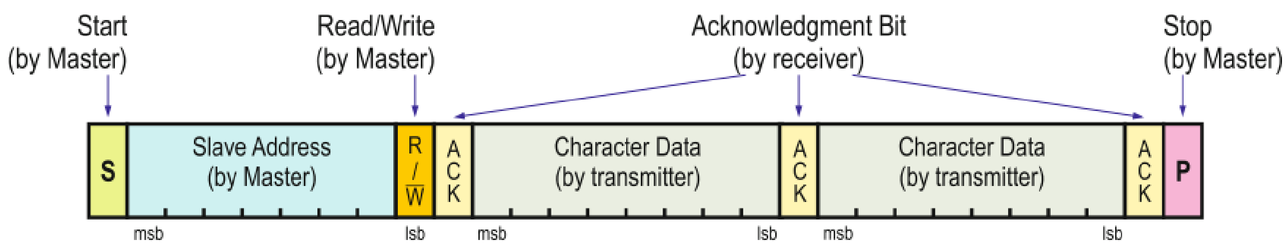


Fig. 9.26 Structure of an I^2C message

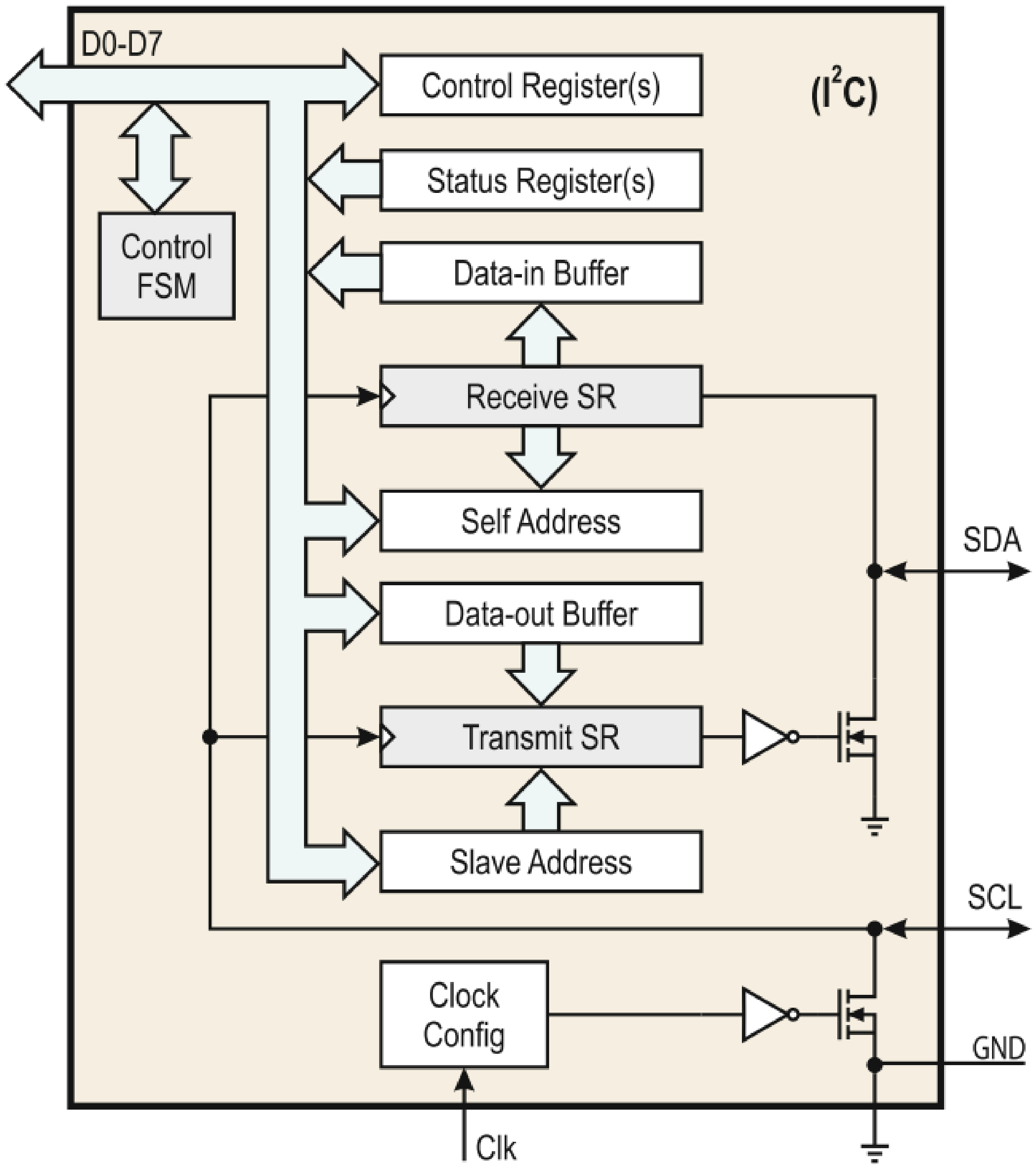


Fig. 9.24 Internal structure of an I^{2}C interface

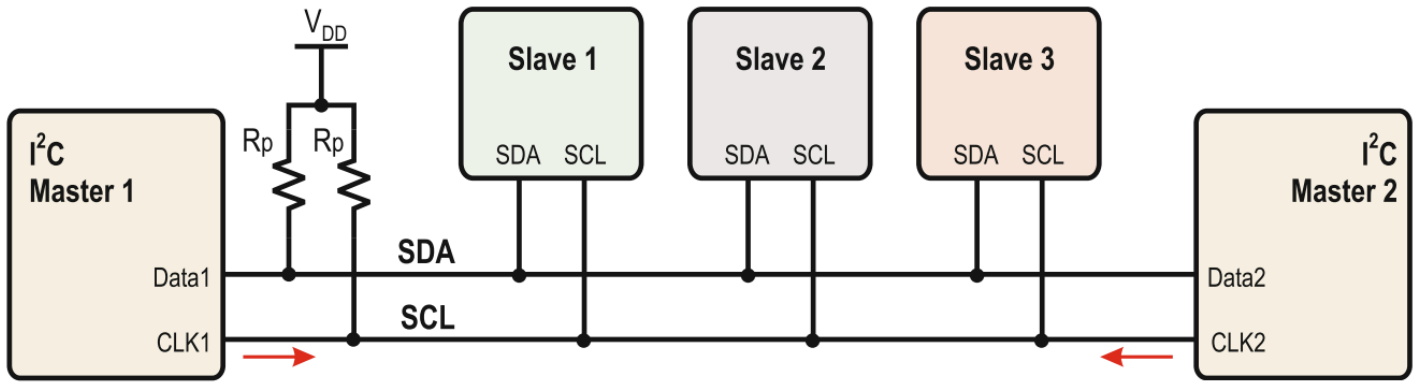
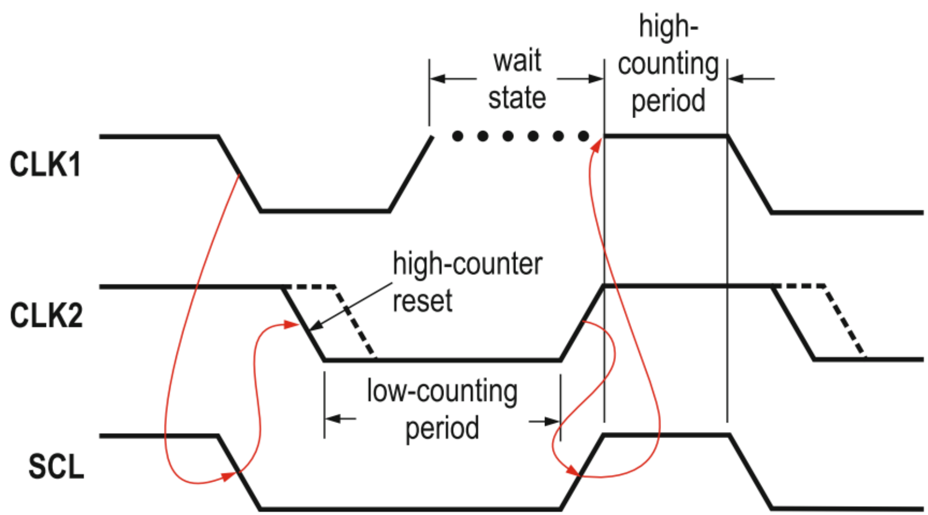


Fig. 9.27 Multimaster I^2C bus configuration

Fig. 9.28 Multimaster signal timing for a clock synchronization process



- For more details, refer to:
  - Chapter 2 at **Embedded Software Development with C**, Springer 2009 by Kai Qian et al.
  - Chapter 9 at **Introduction to Embedded Systems,** Springer 2014 by Manuel Jiménez et al.

- The lecture is available online at:
  - *http://bu.edu.eg/staff/ahmad.elbanna-courses*

- For inquires, send to:
  - ahmad.elbanna@feng.bu.edu.eg