Serial Communication: UART, SPI, and I2C

Computer Science & Engineering Department
Arizona State University
Tempe, AZ 85287

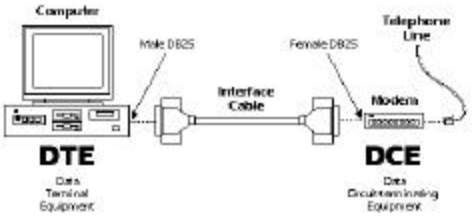
Dr. Yann-Hang Lee
yhlee@asu.edu

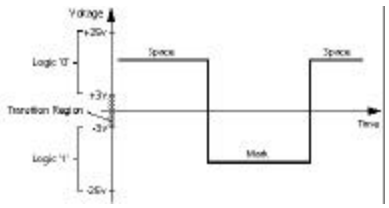
Asynchronous Serial Communication -- UART

- **Universal asynchronous receiver/transmitter**
- **Transmit bits in a single channel**
  - simplex (one way)
  - half-duplex (one direction at a time)
  - full-duplex (two way)
- **A sequence of bits – packet or character**
  - ASCII code – 7 bits for 128 characters (alphabet, numerical, and control)
  - fixed length or variable length
  - Start, stop, and parity bits

EIA RS232

- Connection and signal characteristics
- Data terminal equipment and data communication equipment
- Logic '1' (marking) – -3v to -25v with respect to signal ground
- Logic “0” (spacing) – +3v to +25v
- Not assigned –between -3v and +3v (a transition region)





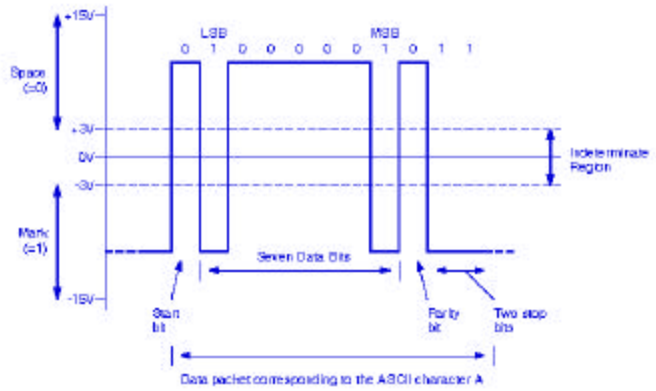
RS232 (continued)

DTE FEP DB25		DCE MOD DB25
1 FG	----------	1 FG
2 TX	---------->	2 TX
3 RX	<----------	3 RX
4 RTS	---------->	4 RTS
5 CTS	<----------	5 CTS
6 DSR	<----------	6 DSR
7 SG	----------	7 SG
8 DCD	<----------	8 DCD
20 DTR	---------->	20 DTR

- Flow control (handshaking) signals to avoid buffer overflow or lock-up.
- RTS – to prepare the DCE device for accepting transmission
- CTS -- to inform the DTE device that transmission may begin
- DCD: data carrier detected
- DSR: DCE ready
- SG: system ground
- DTR: DTE ready

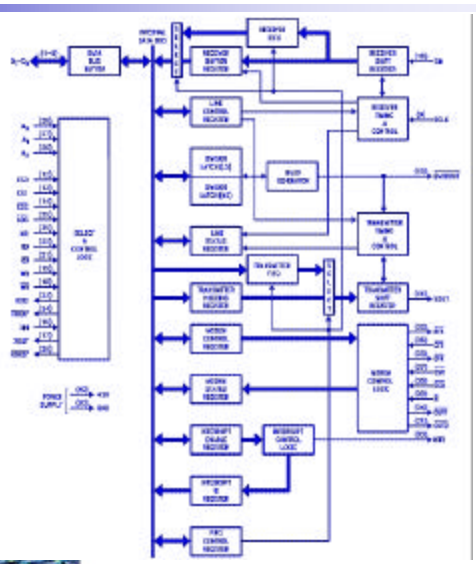
Signal Format for ASCII Character

- data, start, stop, and (even or odd) parity bits



Basic Operations

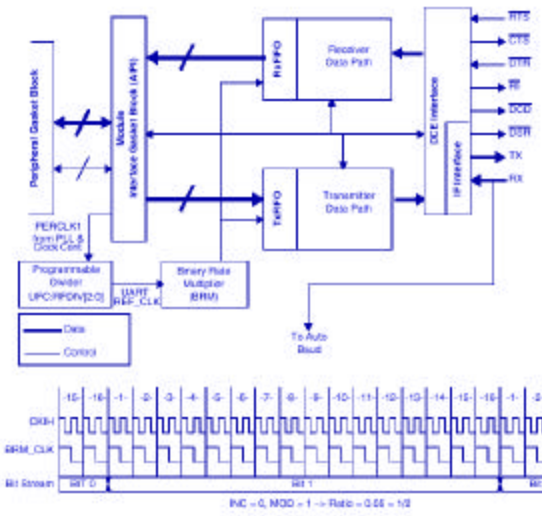
- Parallel / serial conversion
- Buffering (FIFO)
- Clock synchronization and Data sampling
- Baud rate generator
- Parity generator and checking
- Example: 16550



Dragonball MX1 UART

- **Standard RS-232 type Non Return to Zero (NRZ) encoding format and IrDA-compatible infrared modes.**
- **Supports up to 1 Mbps data rate.**
- **Full 8-wire serial DCE interface[1] for Modem Flow Control**
  - Hardware Flow control support for RTS, and CTS signals
  - Software Flow control support for DSR, DCD, and RI signals
  - RTS and DTR edge detect interrupt (edge selectable)
- **Voting logic for improved noise immunity (16X oversampling)**
- **Auto Baud Rate detection**
- **RTS, IrDA asynchronous WAKE, and Receive asynchronous WAKE interrupts will wake MCU from STOP mode**
- **2 DMA requests (transmit FIFO DMA request and Receive FIFO DMA request)**
- **Escape Character Sequence Detection**

Dragonball MX1 UART



Interrupts of Dragonball MX1 UART

- Automatic Baud Rate Detection
- Idle Condition Detected
- Escape Sequence
- BREAK
- Parity and frame Errors
- WAKE
  - set at the detection of a start bit by the receiver
- Asynchronous IR WAKE
  - a pulse is detected on the UART_RX pin
- Asynchronous WAKE
  - a falling edge is detected on the RXD pin
- Data Terminal Ready
- Request to Send
- RTS Delta
- Receive Status
- Receiver Overrun
- Receive Data Ready
  - RxFIFO is filled at a defined level
- Receiver DMA Ready
- Serial Infrared
- Transmitter DMA Ready
- Transmitter FIFO Empty
- Transmitter ready
  - the transmitter has one or more slots available in the TxFIFO
- Transmit Complete

Programming of MX1 UART

```
#define _reg_PORTA_GIUS     (*((volatile unsigned long *)(IO_ADDRESS(0x21C020))))
#define _reg_PORTA_GPR      (*((volatile unsigned long *)(IO_ADDRESS(0x21C038))))
#define _reg_UART1_UCR1     (*((volatile unsigned long *)(IO_ADDRESS(0x206080))))
#define _reg_UART1_UCR2     (*((volatile unsigned long *)(IO_ADDRESS(0x206084))))
#define _reg_UART1_UCR4     (*((volatile unsigned long *)(IO_ADDRESS(0x20608C))))
#define _reg_UART1_UFCR     (*((volatile unsigned long *)(IO_ADDRESS(0x206090))))
#define _reg_UART1_UBIR     (*((volatile unsigned long *)(IO_ADDRESS(0x2060A4))))
#define _reg_UART1_UBMR     (*((volatile unsigned long *)(IO_ADDRESS(0x2060A8))))
#define _reg_UART1_USR2     (*((volatile unsigned long *)(IO_ADDRESS(0x206098))))
#define _reg_UART2_UTXD     (*((volatile unsigned long *)(IO_ADDRESS(0x207040))))
#define TXDC                0x00000008
```

```
_reg_PORTB_GIUS &= 0x0FFFFFFF;
_reg_PORTB_GPR &= 0x0FFFFFFF;
```

What are these assignments?

```
_reg_UART1_UCR1 = 0x5;
_reg_UART1_UCR2 = 0x4027;
_reg_UART1_UCR4 = 1;
_reg_UART1_UFCR = 0x00000A81;
_reg_UART1_UBIR = 0x0000000F;
_reg_UART1_UBMR = 0x00000049;
```

Receive a string

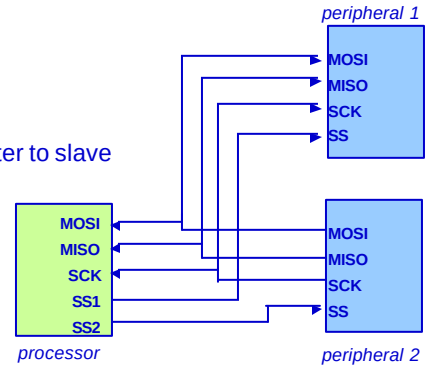
```
static int mx1uart_console_read(struct console *co, const char *s, unsigned long count)
    {
        char *w;
        int c;
        unsigned long rx;

        c = 0;
        w = (char *)s;

        rx = _reg_UART1_URXD;
        _reg_UART2_UCR4 |= DREN;
        do
        {
            c++;
            *w++ = (char)rx;
            if (c == count)
                break;
        }while((rx = _reg_UART1_URXD) & CHARRDY);
        return c;       //return the count
    }
```


Transmit a string

```
static void mx1uart_console_write(struct console *co, const char *s, unsigned long count)
{
    int i;
    _reg_UART2_UCR4 &= ~DREN;
    /*  disable interrupts  */
    if(_reg_UART1_UCR1 &  UCR1_TRDYEN ||_reg_UART1_UCR1 & UCR1_TXMPTYEN)
            _reg_UART1_UCR1 &= ~(UCR1_TRDYEN | UCR1_TXMPTYEN);
    for (i = 0; i < count; i++)
    {
            do
            {               ;
            }while(!(_reg_UART1_USR2 & TXDC));
            _reg_UART1_UTXD = (unsigned long)s[i];
    }
    /* Finally, wait for transmitter to become empty and restore the TCR            */
    do
    {                       ;
    } while (!(_reg_UART1_USR2 & TXDC));
    /*  reenable interrupts  */
    _reg_UART1_UCR1 |= (UCR1_TRDYEN | UCR1_TXMPTYEN);
}
```

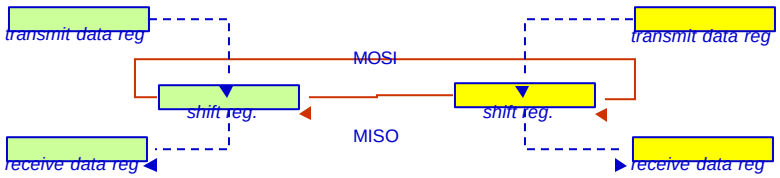
Serial Peripheral Interface (SPI)

- **A duplex, synchronous, serial communication between CPU and peripheral devices**
  - Master mode and slave mode
  - Bi-directional mode
  - Synchronous serial clock
- **Signals:**
  - MOSI: master out slave in
  - MISO: master in slave out
  - SS: select signal from master to slave
  - SCK: serial clock



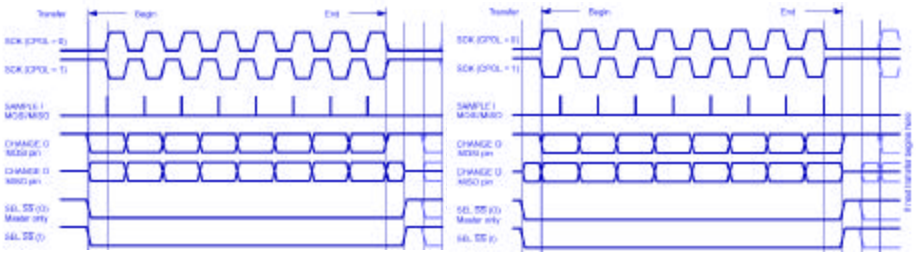
SPI Operation

- **Data registers in the master and the slave form a distributed register.**
- **When a data transfer operation is performed, this distributed register is serially shifted by the SCK clock from the master**
- **Can shift in burst mode**

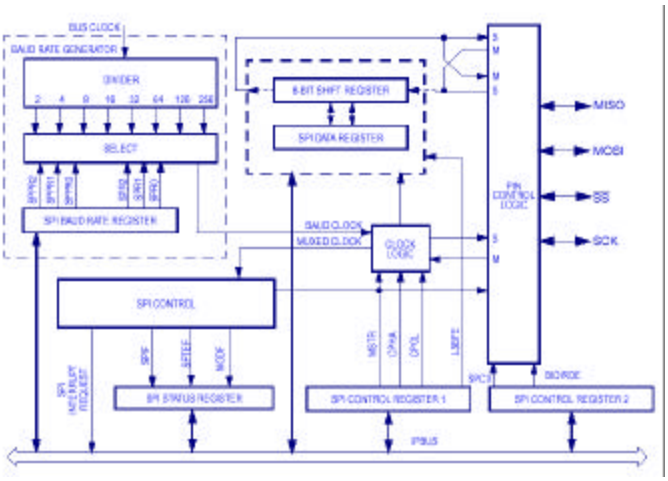


CPOL and CPHA (Polarity and Phase)

- CPHA=0 – the first edge on the SCK line is used to clock the first data bit (the first bit of the data must be ready when selected)
- CPHA=1 – if required, the first SCK edge before the first data bit becomes available at the data out pin

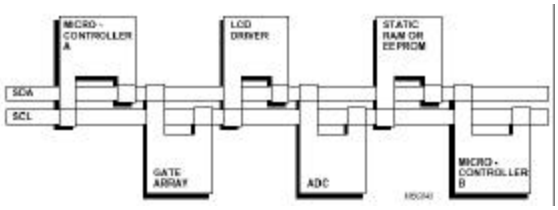


SPI Block Diagram



I2C protocol – Background

- Inter-Integrated Circuit Protocol
- I2C is a low-bandwidth, short-distance, two-wire interface for communication amongst ICs and peripherals
- Originally developed by Philips for TV circuits
- Currently, many devices are available that have hardware interfaces compatible with this protocol
- Figure shows example connection of devices based on this protocol



I2C Features

- Only two bus lines are required
  - The SDA(for data) and SCL(for clock)
- Each device connected to the bus is software addressable
  - Devices can be 7-bit or 10-bit addressed
- Simple master-slave relation amongst devices (either can be receiver or transmitter)
- Multi-bus master collision detection and arbitration is supported
- Serial, 8-bit oriented, bi-directional data transfer can be achieved up to100 kbit/s (and up to 3.4Mbit/s in the high speed mode)

I2C Benefits

- **Designer**
  - I2C bus compatible devices allow a system design to rapidly progress directly from a functional block diagram to a prototype
  - ICs can be added and removed from the system without affecting other ICs on the bus (this clipping and unclipping of devices allows easy modification and upgradation)
  - Easy fault diagnosis
- **Manufacturer**
  - Simple 2-wire serial communication
  - Reduces interconnection on the ICs (lesser pins) and hence cheaper PCBs
  - Completely integrated I2C-bus protocol eliminates the need for address decoders
  - Multi-master capability of the I2C-bus allows rapid testing and alignment of end user equipment via external connections to an assembly line

When to choose the I2C

- **8-bit oriented digital control applications**
- **A system that usually consists of at least one micro-controller and other peripheral devices such as memories**
- **The inter-connection cost of the devices in the system must be minimized**
- **Does not require high-speed data transfer (typical control functions do not require high-speed data exchange)**

The I2C-Bus

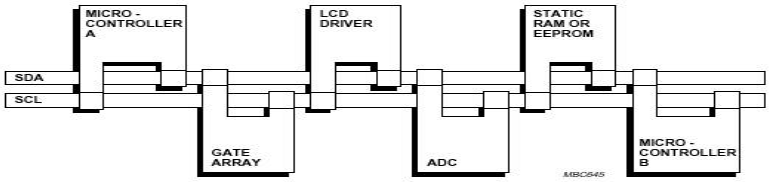
- Supports any fabrication (NMOS, CMOS or bi-polar)
- Two wires SDA(serial data) and SCL(serial clock) carry information
- Each device is recognized by a unique address and can operate as either receiver or transmitter
- Devices have master-slave relation
- I2C bus is a multi-master bus
- Generation of clock signals in the I2C bus is always the responsibility of the master devices

Key terminology

- Transmitter: The device which sends data to the bus
- Receiver: The device which receives data from the bus
- Master: The device which initiates a transfer, generates clock signals and terminates a transfer
- Slave: The device addressed by a master

- Multi-master: More than one master can attempt to control the bus at the same time without corrupting the message
- Arbitration: Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted
- Synchronization: Procedure to synchronize the clock signals of two or more devices

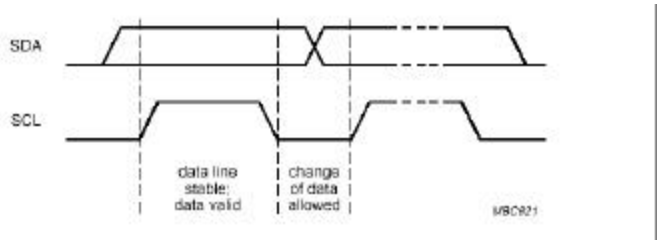
Key Steps: Overview



- **Suppose Micro-controller A wants to send info to micro-controller B**
  1. A (master) addresses B(slave)
  2. A (master-transmitter) sends data to B(slave-receiver)
  3. A terminates the transfer
- **Suppose Micro-controller A wants to receive info from micro-controller B**
  1. A (master) addresses B(slave)
  2. A (master-receiver) receives data from B(slave-transmitter)
  3. A terminates the transfer

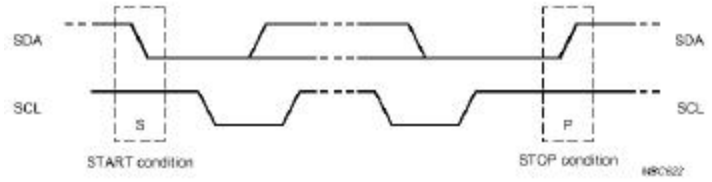
I2C Bit-Transfer

- **One clock pulse is generated for each data bit that is transferred**
- **Data Validity**
  - The data on the SDA line must be stable during the HIGH(1) period of the clock. The data line(SDA) can change data only when the clock signal (SCL) is LOW(0)



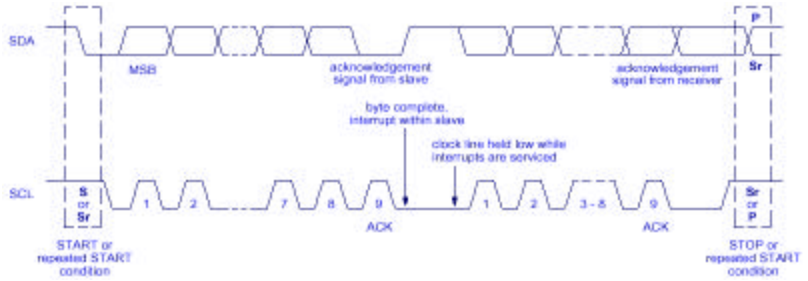
I2C START/STOP Conditions

- START condition: Signals begin of transfer (occupies the bus)
  - A HIGH to LOW transition on the SDA line while the SCL is HIGH
- STOP condition: Signals end of transfer (releases the bus)
  - A LOW to HIGH transition on the SDA line while the SCL is HIGH
- Both these are always generated by the Master
- Repeated START condition is allowed
  - Repeated start is used for changing the slave, or changing the direction of data transfer (Send/Receive) for the same slave



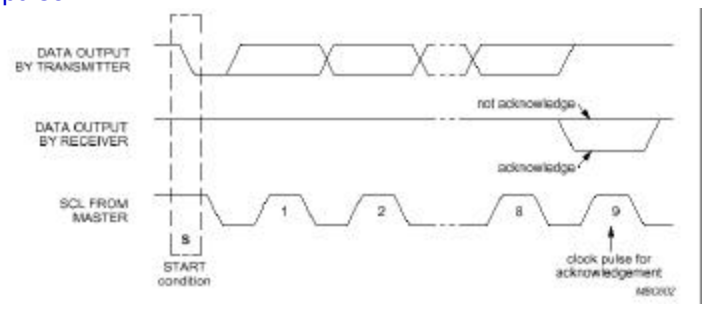
I2C Data Transfer

- Every byte on the SDA line must be 8-bits long
- Each byte must be followed by an acknowledgement from the receiver
- Data byte is transferred bit-wise with the MSB as the first bit sent
- A slave can force the master to wait by holding the clock line SCL LOW



Acknowledgement Scheme

- The acknowledge-related clock-pulse is generated by the master
- The transmitter (master or slave) releases the SDA line i.e. SDA is HIGH for the ACK clock pulse
- The receiver must pull-down the SDA line during the acknowledge clock pulse (stable LOW) during the HIGH period of the clock pulse

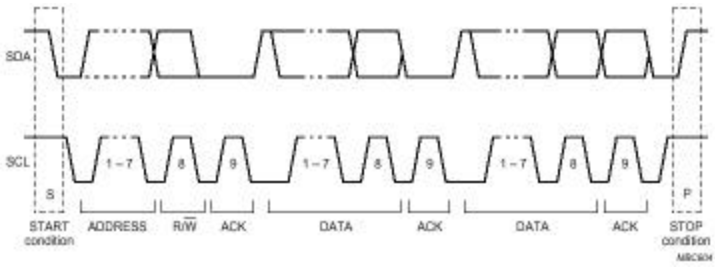


Acknowledgement Scheme

- The receiver is obliged to generate an acknowledge after each byte received
- When a slave does not acknowledge slave address (when busy), it leaves the data line HIGH. The master then generates either STOP or attempts repeated START
- If a slave-receiver does ack the slave address, but some time later during the transfer can not receive more data (this is done by leaving SDA HIGH during the ack pulse), then the master either generates STOP or attempts repeated START
- If a master-receiver is involved in a transfer, it must signal the end of data to the slave-transmitter by not generating an ack on the last byte that was clocked out of the slave. The slave-transmitter must release the data line to allow the master to generate a STOP or repeated START condition

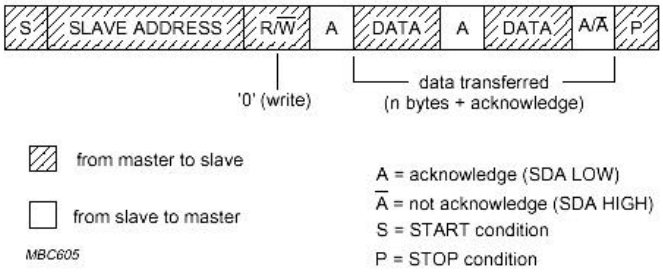
Data Transfer With 7-Bit Device Address

- After START condition (S), a slave address(7-bit) is sent.
- A read/write (R/W') direction is then sent(8th bit)
- Data transfer occurs, and then always terminated by STOP condition. However, repeated START conditions can occur.



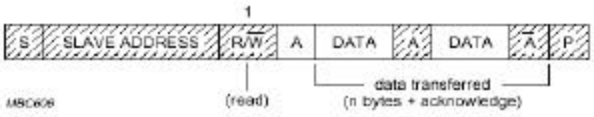
Master-Transmitter to Slave-Receiver Data Transfer

- **In this, the transmission direction never changes. The set-up and transfer is straight-forward**



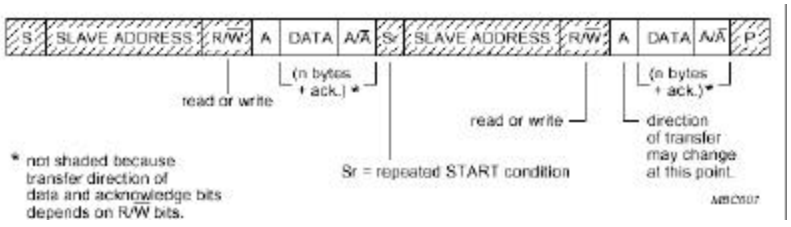
Master-Receiver and Slave-Transmitter Data Transfer

- Master initiates the data transfer by generating the START condition followed by the start byte (with read/write bit set to 1 i.e. read mode)
- After the first ack from the slave, the direction of data changes and the master becomes receiver and slave transmitter.
- The STOP condition is still generated by the master (master sends not-ACK before generating the STOP)



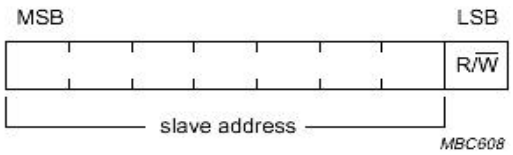
Read and Write in the Same Data Transfer

- Change in direction of data transfer can happen by the master generating another START condition (called the repeated START condition) with the slave address repeated
- If the master was a receiver prior to the change, then the master sends a not-ack (A') before the repeated START condition



7-Bit Addressing

- The addressing info is contained in the first byte after the STA RT condition and it determines which slave will be selected by the master
- The first 7 bits contain the address and LSB contains the direction of transfer(R/W' : 0 = write;1= read)
- When an address is sent, each device compares the first seven bits and considers itself addressed.

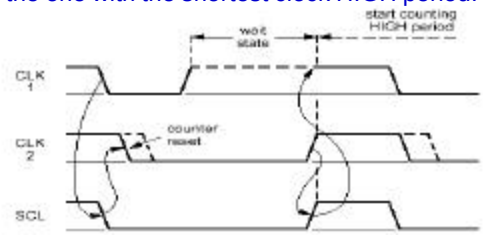


7-Bit Addressing

- **A slave address can be made up of a fixed and a programmable part.**
  - The I2C-bus committee coordinates the allocation of I2C device hardware addressing.
  - However, since many devices can be identical in the system, each device will have a software address as well. The number of address bits reserved for software depends on the device.
- **A 'general call' addressing is also available, to address all the slaves at the same time.**

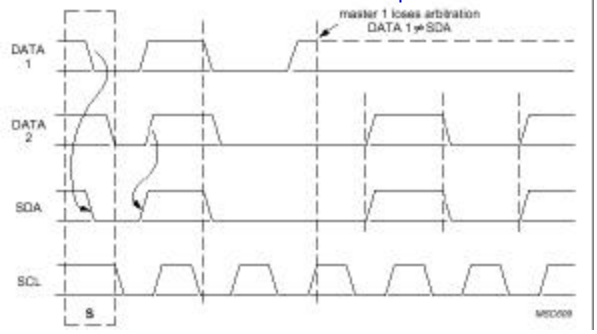
Multi-Master Clock Synchronization

- In the I2C bus, clock synchronization is performed using the wired-AND connection of the interfaces to SCL line.
  - If at least one master clock goes from HIGH to LOW, then the SCL is held LOW irrespective of the other masters' clock.
  - The SCL line goes to a HIGH state only when all the master clocks are in HIGH (this SCL HIGH is triggered by the last master clock that entered into its HIGH state).
- The synchronized clock is generated with its LOW period determined by the device with the longest clock LOW period and its HIGH period determined by the one with the shortest clock HIGH period.



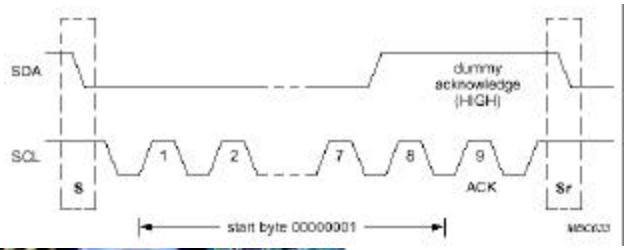
Multi-Master Arbitration Using the Clock Syn.

- If more than one device is capable of being a master, then an arbitration mechanism is needed to choose the master that takes control of the bus
- Arbitration takes place on the SDA, while the SCL is at the HIGH line,
  - the master which transmits a HIGH level,
  - another master is transmitting LOW level will switch off its DATA output stage because the level on the bus does not correspond to its own level.



Software Interface to I2C Bus

- The earlier slides assume that the micro-controllers have an on-chip hardware interface to the I2C bus. However, it is also possible to use a software interface
- In this case, the software constantly monitors the bus. Hence, these controllers are inherently slower as they have to spend time for polling (Example: The bus has to be polled a minimum of 2 times per clock pulse to detect START/STOP)
- The START process is lengthier also.
  - START ; START byte (00000001);ACK clock pulse; Repeated START



I2C Conclusion

- **Compared to other serial bus protocols like SPI and Microwire**
  - The pin (and connection) requirements are the least in I2C
  - The noise immunity is higher for I2C
  - There is a feedback to the the transmitter (Ack signal) for conveying the success of the transfer
  - I2C now has fast and high speed modes of operation