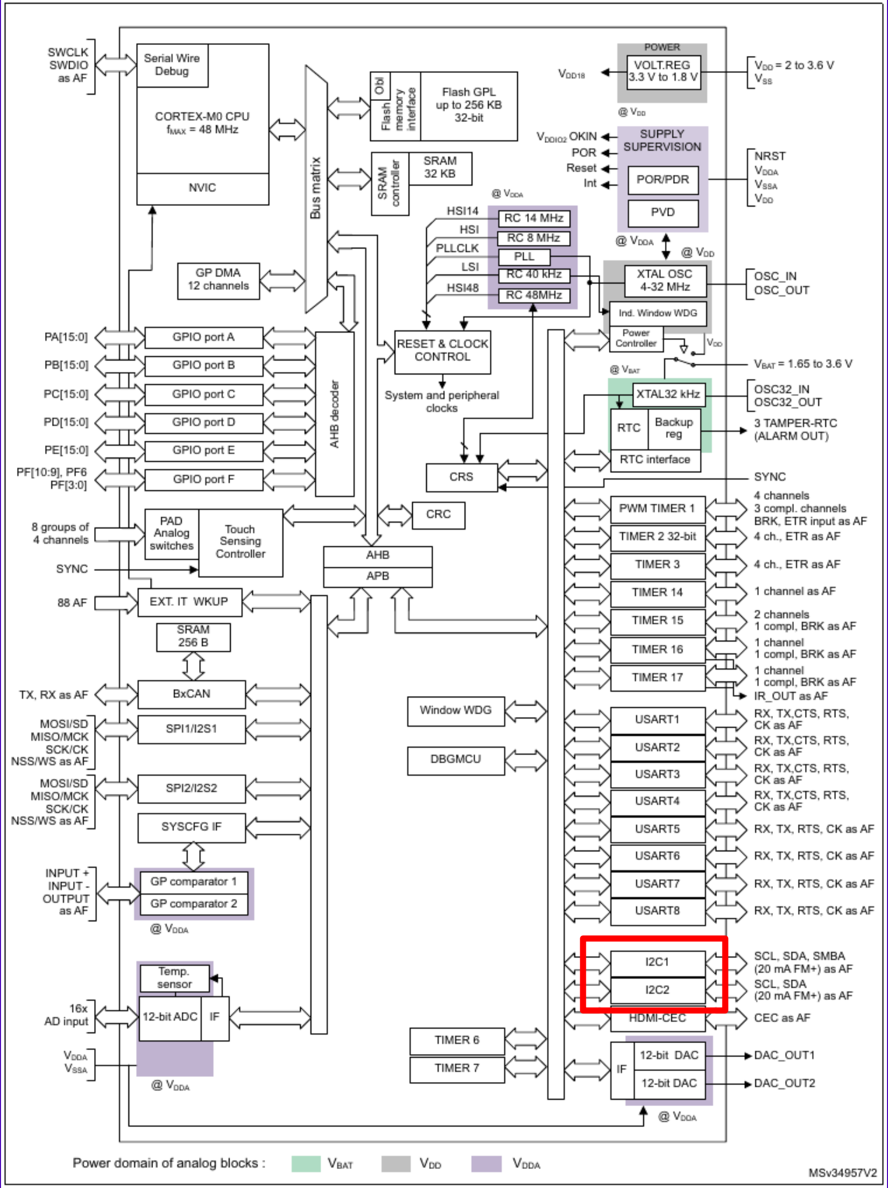
Inter-Integrated Circuit (I^2C) Interface

ECE 362

https://engineering.purdue.edu/ece362/

Reading Assignment

- Textbook, Chapter 22, Serial Communication Protocols, pp. 527 – 598
  - It’s a long chapter.
  - 22.2 (546 – 567) is about I^2C. A good intro. Read and understand.
  - We’ll next look at Section 22.1, UART, pp. 527–545.
  - Don’t worry so much about the USB section.
    - Read that only if you’re curious.
    - Not much we can do with that.
    - Other books are better for understanding USB.
- Family Reference Manual Appendix A.14

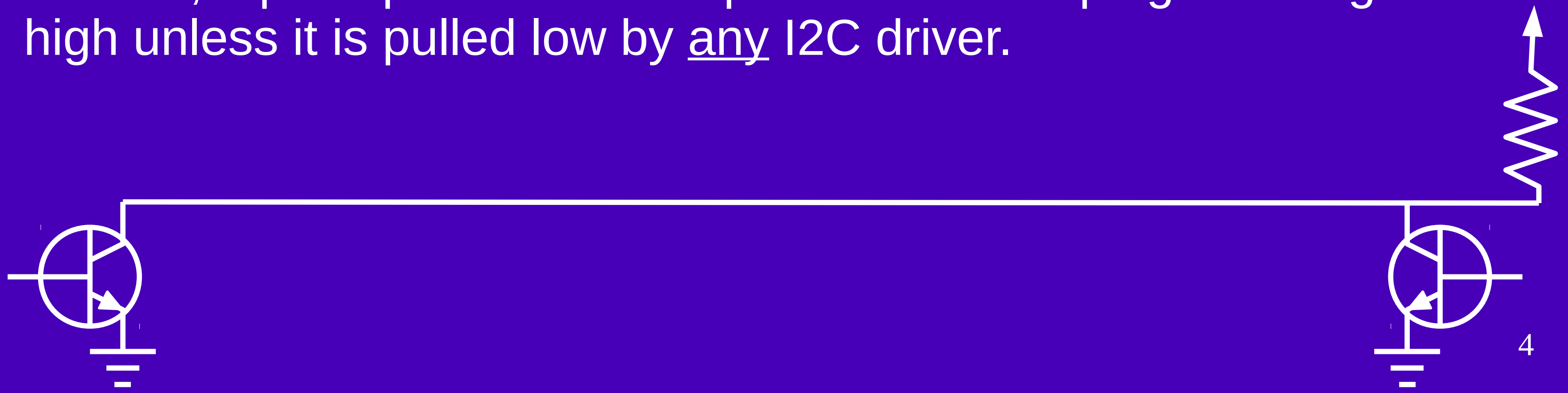


I2C Peripheral

- Two independent "channels".
  - Convert between an internal parallel word and an external *serial* stream.
  - Synchronous clock pulse for each bit.
  - Only "data" (SDA) and "clock" (SCL) signals.
  - No "slave select".
    - Multi-master bus.
    - All I2C endpoints send and receive on *the same wires*.
    - All I2C endpoints have an *address*.

How can everything share the same wires?

- Normal digital logic outputs are connected to push-pull drivers.
- I2C signals are connected to "open-drain" drivers.
  - They can not pull up for a logic high.
  - Instead, a pull-up resistor is responsible for keeping each signal high unless it is pulled low by <u>any</u> I2C driver.



The physical layer

- Two wires (SCL & SDA), plus shared ground.
- Voltages: 5v, 3.3v, 1.8v, … etc
- You may use multiple devices that share a connection with different voltages by using bidirectional level shifters.
  - This has an impact on performance.

Baud rate

- 100 kbit/s (standard mode)
- 400 kbit/s (fast mode)
- 1 mbit/s (fast mode+ (fm+))
- 10 kbit/s (low speed mode)
- 3.4 mbit/s (high speed mode) [not popular]
- Speed also limited by
  - Bus capacitance (typically 400pF)
  - Strength of the pull-up resistor.
  - Length of the network.

Contrast to SPI, which can easily run at multiple megabits per second.

Slowest we can clock a 48MHz STM32's SPI is ~187 kHz.

- Effective data rate is less than half of clock rate due to addressing, acknowledgements, etc.

I2C is a *synchronous* protocol

- A clock pulse accompanies each data bit.
- A clock signal must be delivered to each data recipient.

- By comparison, an *asynchronous* protocol would require only a data line.

I2C is Multi-Master

- Any I2C device could transition between master and slave.
  - Normally, a CPU is a master, and a peripheral is a slave.
  - There could easily be multiple CPUs on a single I2C bus.
- Devices can be receivers, transmitters, or both:
  - Temperature sensor: only transmits
  - LCD display: only receives
  - Flash memory IC: receives and transmits

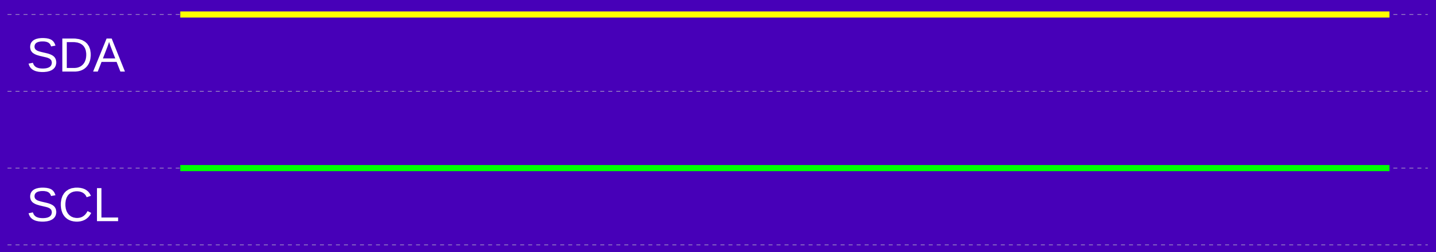
Why use I2C if it’s slower than SPI?

- It’s convenient to easily connect multiple devices using only two wires in total.
- If speed is not critical for an application, it doesn’t matter that it’s slower than SPI.
  - For example: reading multiple temperature sensors

What do the signals look like?

- An idle I2C bus:

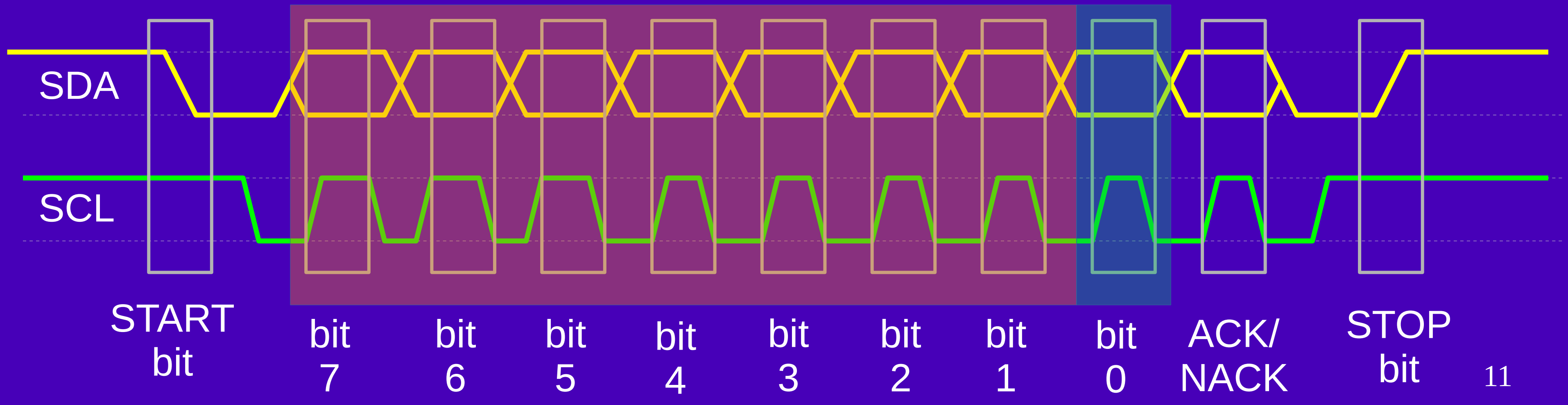
  (SDA and SCL high. Nothing pulling low.)



What communication looks like

- An I2C device starts a transaction with a START (S) bit.
- Sends a 7-bit address. (10 bit addr? Not common.)
- Sends a 1-bit intent 0: write, 1: read.
- Listens for an ACK/NACK (sent by receiver).
- Sends a STOP (P) bit.

START and STOP bits are the only times that SDA changes when SCL is held high.



ACK/NACK

- low = ACK, high = NACK
- If no device on the I2C bus will respond to the particular address that was sent, then nothing will acknowledge.
  - If nothing acknowledges, there is nothing to pull the line low. This indicates failure.
- A long transmission ends with a NACK.
  - This indicates termination of a multi-byte transaction.

All data is sent MSB first

- Both addresses and data are sent most-significant-bit first.
- I2C protocol makes no definitions for the contents of the data fields.
  - Whatever the devices agree on.

Data can follow the ACK.

- When a master device writes a command and single data byte to a slave device, it looks like this:

S	slave addr	R/W	A	command	A	data	A	P
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	8 bits	1 bit	1 bit
	(address)	(0)	(0)	(command)	(0)	(data)	(0)	

Many transactions with one stop bit

- A master reading a byte from a slave:

S	slave addr	R/W	A	command	A	
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	(request)
	(address)	(0)	(0)	(command)	(0)	

S	slave addr	R/W	A	data	N	P	
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	1 bit	(reply)
	(same address)	(1)	(0)	(data)	(1)		

Who sends this NACK?

Exploring an I2C device

- Consider a device such as an EEPROM (24AA32AF)
  - 32kbit (4 kbyte) serial EEPROM.
  - How can we find the datasheet?
  - How can we interpret the datasheet?
  - How should we connect it?
  - What are the protocols to access it?
  - How should we program the STM32 to access it?

Finding datasheets

- Go to major electronics supplier website. (Digikey, Newark, Jameco)
- Type in part name.
- Refine search.
- Click on the PDF icon.

Interpreting the Datasheet

- Main things:
  - Features, Pinout
  - DC, AC Characteristics, timing (2.5 – 5.5V)
  - Pin descriptions. (A0...A2. WP. SDA resistor)
  - Functional description.
    - 24XX32A does not generate <u>any</u> ack bits if an internal programming cycle is in progress.
  - Device addressing.
  - Write and read protocols.

Device Addressing

- I2C peripherals often have fixed I2C addresses.
  - This one is 0x50 (binary 1010000).
  - The lower 3 bits of the 24AA32A's address are configurable by wiring voltages to three pins on the device.
- Could put 8 of these EEPROMS on the same bus by giving each one an address 0x50, 0x51, 0x52, …, 0x57.
- Some devices have an entirely configurable I2C addr.
  - Such as an STM32.

Write and Read Protocols

- Section 6 of datasheet explains write.
  - Control byte, high address, low address, data, …
  - Immediately after write, the EEPROM stops ACKing until the write cycle is complete.
- Section 8 of datasheet explains read.
  - Control byte, data. (Current address read)
  - Control byte, high address, low address, START, control byte, data, … (random read and sequential read)

Could send zero bytes of data.

A good question:

- Every I2C slave device on the same bus must have a unique address.
- Why doesn’t the master device need an address?

How to set up STM32?

- Textbook and FRM have decent examples.
  - With several mistakes or omissions.
- Textbook section 22.2 shows lots of calcuations for timing parameters. Whew.
  - Use table 83, page 642 of FRM for parameters for an 8MHz clock. (Uses HSI clock by default.)
- Textbook Example 22-11 shows clock enable.
  - Symbols for STM32 are different.

RCC clock & I2C1 setup

```c
//===========================================================================
// Initialize I2C1 to 400 kHz
void i2c_init(void) {
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    GPIOB->MODER |= 2<<(2*6) | 2<<(2*7);
    GPIOB->AFR[0] |= 1<<(4*6) | 1<<(4*7);

    RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;
    //RCC->CFGR3 |= RCC_CFGR3_I2C1SW;   // to set for 48MHz sysclk
                                        // default is 8MHz "HSI" clk
    // I2C CR1 Config
    I2C1->CR1 &= ~I2C_CR1_PE;           // Disable to perform reset.
    I2C1->CR1 &= ~I2C_CR1_ANFOFF;       // 0: Analog noise filter on.
    I2C1->CR1 &= ~I2C_CR1_ERRIE;        // Error interrupt disable
    I2C1->CR1 &= ~I2C_CR1_NOSTRETCH;    // Enable clock stretching

    // From table 83. p642 of FRM.  Set for 400 kHz with 8MHz clock.
    I2C1->TIMINGR = 0;
    I2C1->TIMINGR &= ~I2C_TIMINGR_PRESC;// Clear prescaler
    I2C1->TIMINGR |= 0 << 28;           // Set prescaler to 0
    I2C1->TIMINGR |= 3 << 20;           // SCLDEL
    I2C1->TIMINGR |= 1 << 16;           // SDADEL
    I2C1->TIMINGR |= 3 << 8;            // SCLH
    I2C1->TIMINGR |= 9 << 0;            // SCLL

    // I2C "Own address" 1 register (I2C_OAR1)
    I2C1->OAR1 &= ~I2C_OAR1_OA1EN;       // Disable own address 1
    I2C1->OAR1 =   I2C_OAR1_OA1EN | 0x2;// Set 7-bit own address 1
    I2C1->OAR2 &= ~I2C_OAR2_OA2EN;      // Disable own address 2

    I2C1->CR2 &= ~I2C_CR2_ADD10;        // 0 = 7-bit mode; 1 = 10-bit
    I2C1->CR2 |= I2C_CR2_AUTOEND;       // Enable the auto end
    I2C1->CR2 |= I2C_CR2_NACK;          // For slave mode: set NACK

    I2C1->CR1 |= I2C_CR1_PE;            // Enable I2C1
}
```

See I2C I/O reg description on FRM page 662.

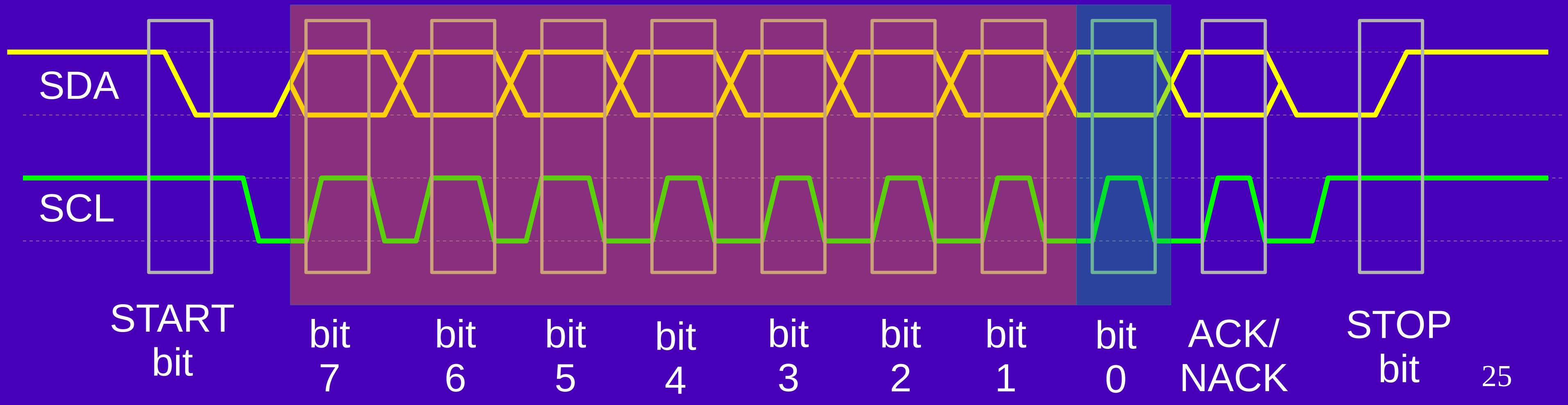
Why must it be so difficult?

- A prescaler divides down the selected clock. (PRESC)
- You must configure the data setup time.
  - Time between data transition and next SCL posedge.
  - This is the "delay" of SCL (SCLDEL).
- You must configure the data hold time.
  - Time between SCL negedge and data transition.
  - This is the "delay" of SDA (SDADEL).
- To set up a 400kHz clock, you’re not just configuring a 400kHz square wave. Instead:
  - you must configure the clock’s <u>minimum</u> low time (SCLL)
  - you must configure the clock’s <u>minimum</u> high time (SCLH)… Why? Clock stretching!

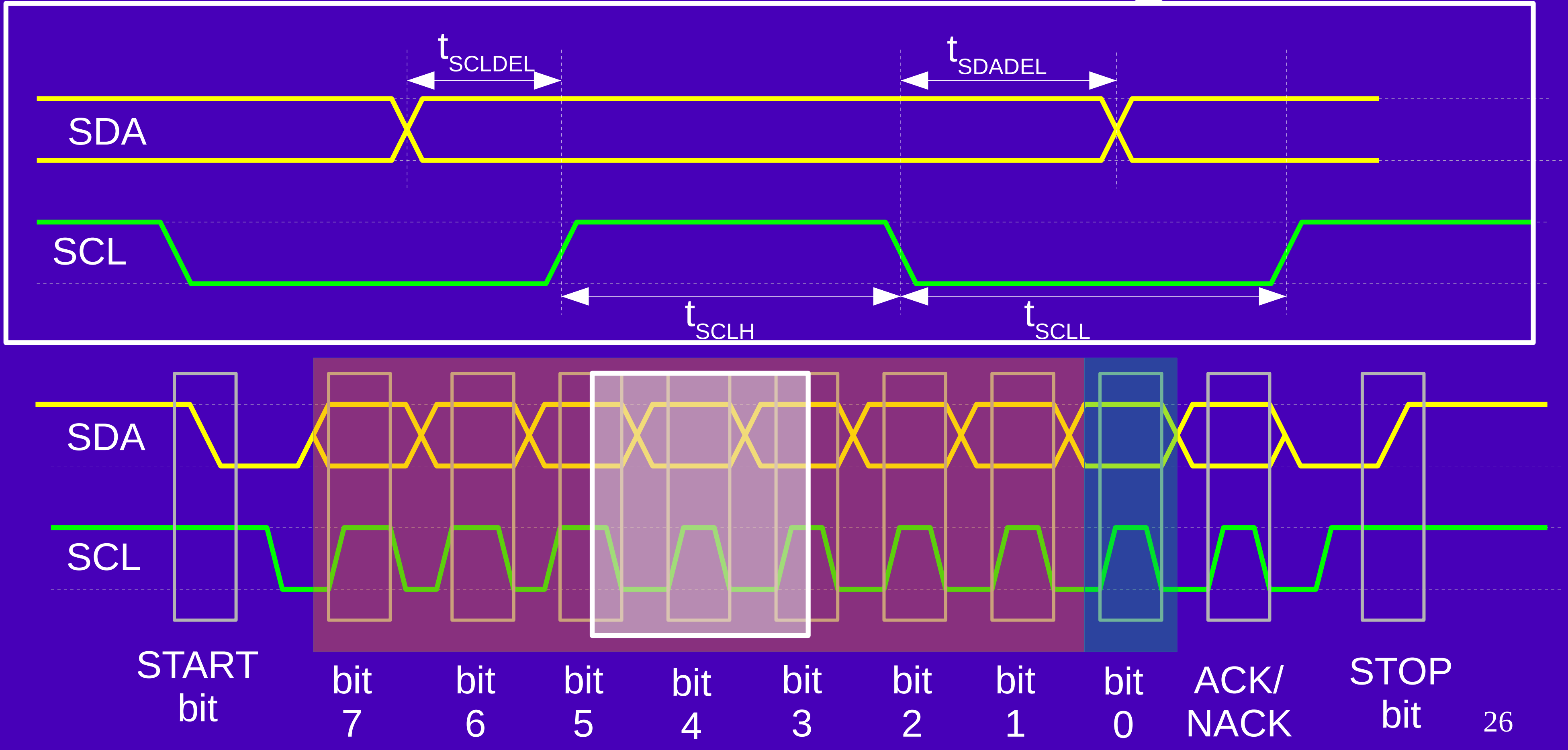
Recall what the signals look like

- An I2C device starts a transaction with a START (S) bit.
- Sends a 7-bit address. (10 bit addr? Not common.)
- Sends a 1-bit intent 0: write, 1: read.
- Listens for an ACK/NACK (sent by receiver).
- Sends a STOP (P) bit.

START and STOP bits are the only times that SDA changes when SCL is held high.



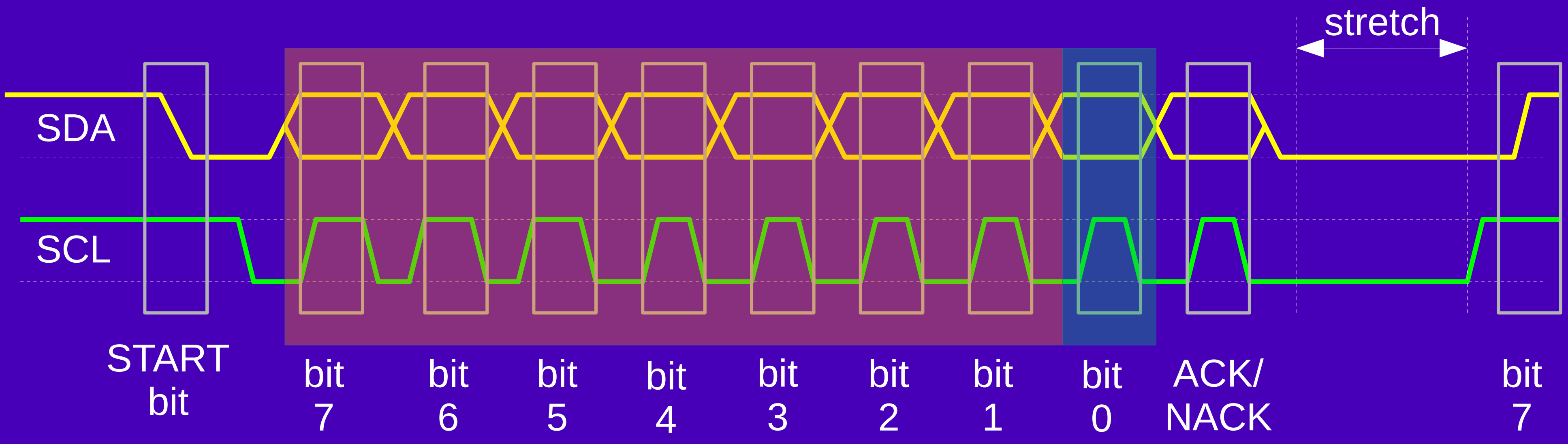
SDA and SCL timing



Clock stretching

- Both the data and clock lines of an I2C bus are open-drain.
- When clock stretching is enabled (is not disabled), any device on the I2C bus can lengthen the low time of a clock.
- The master drives the clock, but a slave device may not be able to keep up.
- Any slave device may lengthen the clock <u>only</u> after the ACK bit and before the MSB of the next byte.

Clock stretch example



Writing and Reading Data

- See the textbook:
  - Ex. 22-13: I2C_Start()
  - Ex. 22-14: I2C_Stop()
  - Ex. 22-15: I2C_WaitLineIdle()
  - Ex. 22-16: I2C_SendData()
  - Ex. 22-17: I2C_ReceiveData()

These are almost usable “as is.”

Example Program

```c
int main(void)
{
    i2c_init();

    while(1) {
        i2c_waitidle();
        i2c_start(0x50, 0, 0);
        int x=0;
        while((I2C1->ISR & I2C_ISR_TC) == 0 &&
               (I2C1->ISR & I2C_ISR_STOPF) == 0 &&
               (I2C1->ISR & I2C_ISR_NACKF) == 0)
            x++;  // Wait until TC flag is set
        if (I2C1->ISR & I2C_ISR_NACKF)
            I2C1->ICR |= I2C_ICR_NACKCF;
        if (I2C1->ISR & I2C_ISR_STOPF)
            I2C1->ICR |= I2C_ICR_STOPCF;
        else
            i2c_stop();
        nano_wait(1000000);
    }
}
```

Try this.
Put the oscilloscope on it.

More meaningful example

```
int main(void)
{
    init_lcd();
    display1("___");
    i2c_init();

    char addr1[] = "\0\0Hello, World!";
    i2c_senddata(0x50, addr1, sizeof addr1);

    while(1) {
        if (i2c_senddata(0x50, addr1, 2) < 0) {
            I2C1->ICR |= I2C_ICR_NACKCF;
            I2C1->ICR |= I2C_ICR_STOPCF;
        } else
            break;
    }

    while(1) {
        uint8_t addr[] = {0,0};
        i2c_senddata(0x50, addr, sizeof addr);
        char line[32];
        i2c_recvdata(0x50, line, sizeof line);
        display1(line);
        for(;;);
    }
}
```

Write a string to I2C EEPROM.
Memory address 0x000

Try a 2-byte mem address update.
Wait for an ACK instead of NACK.

Try a 2-byte mem address update.

Read 32 bytes from addr 0x000.

Important I/O Registers

- I2Cx_TIMINGR: set up the clock rate and setup/hold values
- I2Cx_CR1: configure channel
- I2Cx_CR2: set up operations
  - START, STOP
- I2Cx_TXDR/RXDR: data registers
- I2Cx_ISR: read status
  - NACKF, STOPF
- I2Cx_ICR: clear status

Send start and stop bits

```
void i2c_start(uint32_t devaddr, uint8_t size, uint8_t dir) {
    // dir: 0 = master requests a write transfer
    // dir: 1 = master requests a read transfer
    uint32_t tmpreg = I2C1->CR2;
    tmpreg &= ~(I2C_CR2_SADD | I2C_CR2_NBYTES |
                I2C_CR2_RELOAD | I2C_CR2_AUTOEND |
                I2C_CR2_RD_WRN | I2C_CR2_START | I2C_CR2_STOP);
    if (dir == 1)
        tmpreg |= I2C_CR2_RD_WRN;  // Read from slave
    else
        tmpreg &= ~I2C_CR2_RD_WRN;  // Write to slave
    tmpreg |= ((devaddr<<1) & I2C_CR2_SADD) | ((size << 16) & I2C_CR2_NBYTES);
    tmpreg |= I2C_CR2_START;
    I2C1->CR2 = tmpreg;
}

void i2c_stop(void) {
    if (I2C1->ISR & I2C_ISR_STOPF)
        return;
    // Master: Generate STOP bit after current byte has been transferred.
    I2C1->CR2 |= I2C_CR2_STOP;
    // Wait until STOPF flag is reset
    while( (I2C1->ISR & I2C_ISR_STOPF) == 0);
    I2C1->ICR |= I2C_ICR_STOPCF; // Write  to clear STOPF flag
}

void i2c_waitidle(void) {
    while ( (I2C1->ISR & I2C_ISR_BUSY) == I2C_ISR_BUSY);  // while busy, wait.
}
```

- Start sends the slave addr, and intent, and configures the size.

i2c_senddata()

```c
int8_t i2c_senddata(uint8_t devaddr, void *pdata, uint8_t size) {
    int i;
    if (size <= 0 || pdata == 0) return -1;
    uint8_t *udata = (uint8_t*)pdata;
    i2c_waitidle();
    // Last argument is dir: 0 = sending data to the slave device.
    i2c_start(devaddr, size, 0);

    for(i=0; i<size; i++) {
        // TXIS bit is set by hardware when the TXDR register is empty and the
        // data to be transmitted must be written in the TXDR register.  It is
        // cleared when the next data to be sent is written in the TXDR reg.
        // The TXIS flag is not set when a NACK is received.
        int count = 0;
        while( (I2C1->ISR & I2C_ISR_TXIS) == 0) {
            count += 1;
            if (count > 1000000) return -1;
            if (i2c_checknack()) { i2c_clearnack(); i2c_stop(); return -1; }
        }
        // TXIS is cleared by writing to the TXDR register.
        I2C1->TXDR = udata[i] & I2C_TXDR_TXDATA;
    }
    // Wait until TC flag is set or the NACK flag is set.
    while((I2C1->ISR & I2C_ISR_TC) == 0 && (I2C1->ISR & I2C_ISR_NACKF) == 0);

    if ( (I2C1->ISR & I2C_ISR_NACKF) != 0)
        return -1;
    i2c_stop();
    return 0;
}
```

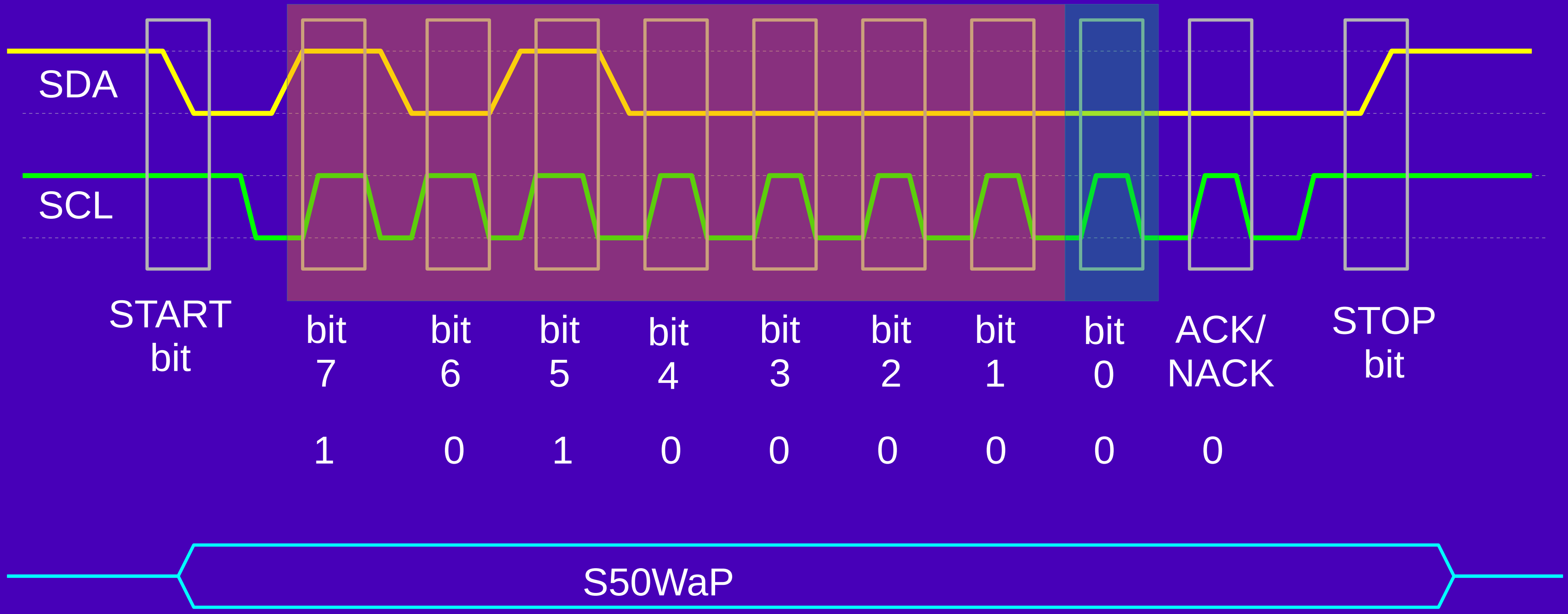
i2c_recvdata()

```
int i2c_recvdata(uint8_t devaddr, void *pdata, uint8_t size) {
    int i;
    if (size <= 0 || pdata == 0) return -1;
    uint8_t *udata = (uint8_t*)pdata;
    i2c_waitidle();
    // Last argument is dir: 1 = receiving data from the slave device.
    i2c_start(devaddr, size, 1);
    for(i=0; i<size; i++) {
        int count = 0;
        while( (I2C1->ISR & I2C_ISR_RXNE) == 0) {
            count += 1;
            if (count > 1000000) return -1;
            if (i2c_checknack()) { i2c_clearnack(); i2c_stop(); return -1; }
        }
        udata[i] = I2C1->RXDR;
    }
    // Wait until TC flag is set or the NACK flag is set.
    while((I2C1->ISR & I2C_ISR_TC) == 0 && (I2C1->ISR & I2C_ISR_NACKF) == 0);
    if ( (I2C1->ISR & I2C_ISR_NACKF) != 0)
        return -1;
    i2c_stop();
    return 0;
}
```

Lab 9

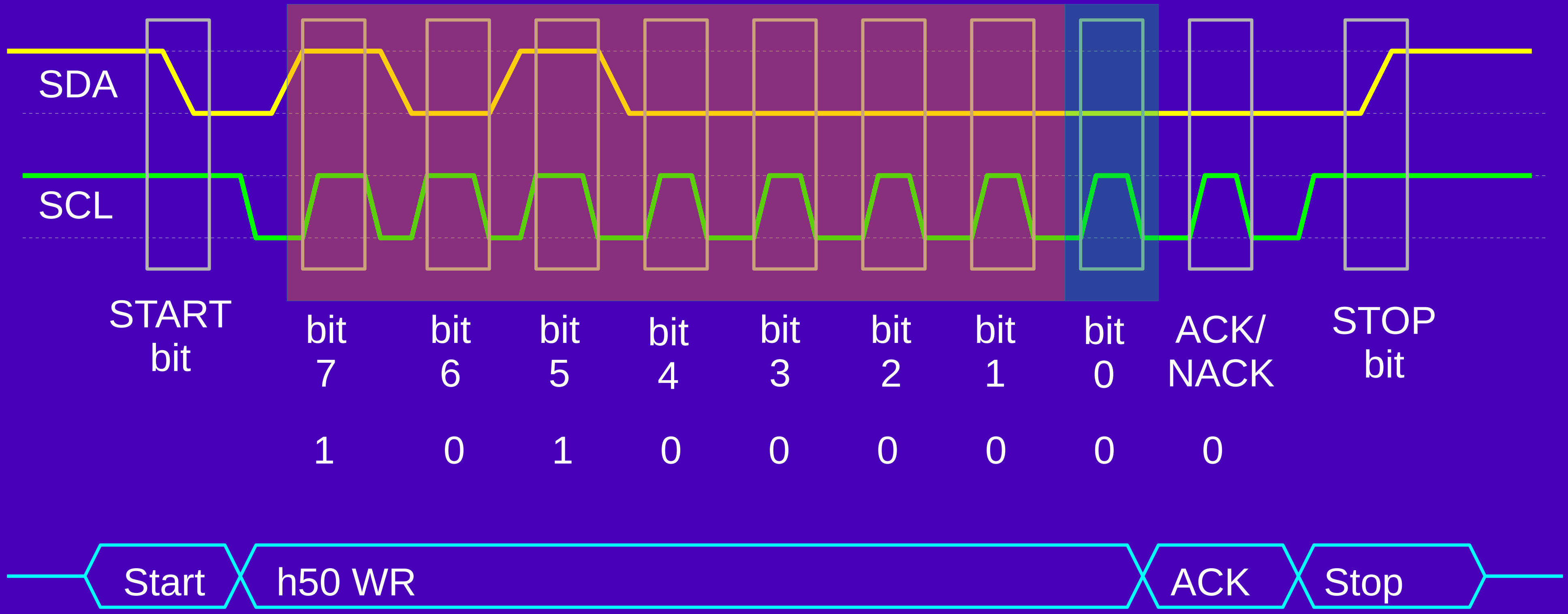
- Using I2C devices (EEPROM and GPIO chip)
- Writing code to read/write the devices
- Looking at signals with the oscilloscope / protocol analyzer

Oscilloscope interpretation



START 0x50 WRITE ACK STOP

AD2 interpretation



Debugging I2C

- When you can’t get an I2C device to work,
  - Put the scope on it and make sure you see the master sending proper waves.
  - If I2C slave device does not ack, try slower rate, double-check the address.
  - Worst case, remove everything else from the bus and try to send commands to the "general call address", 0x00.
    - Any working I2C slave device should respond with an ack.