NC STATE UNIVERSITY

Device Interfacing with Linux: I^2C

version 1.0

PROTOCOL COMPARISON

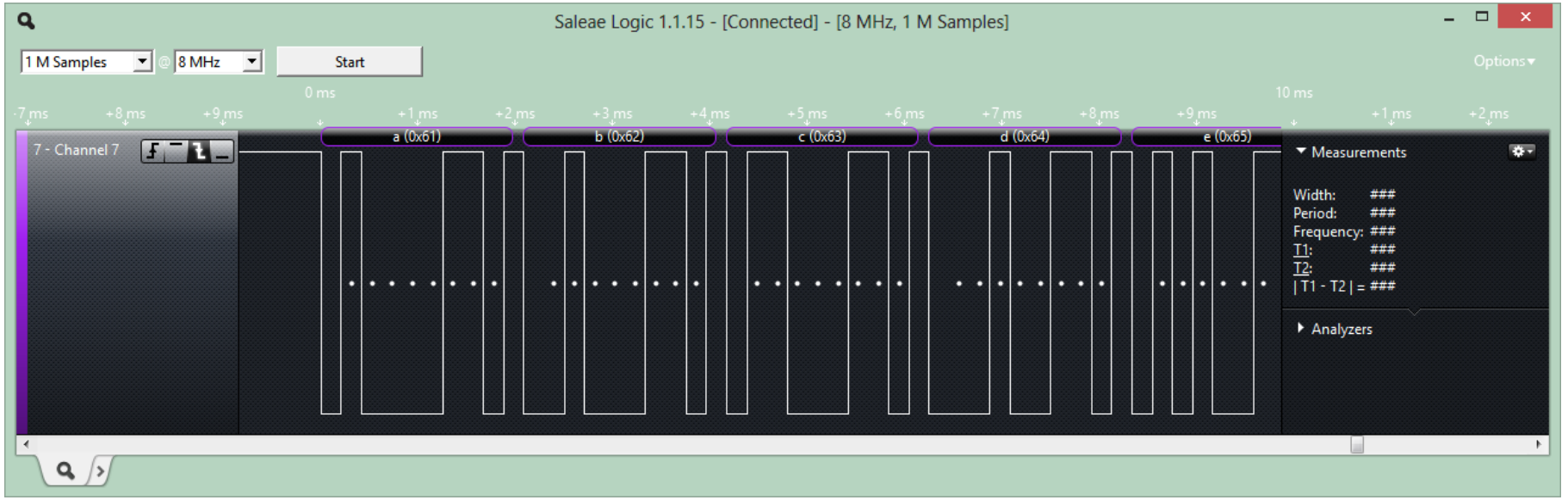
Factors to Consider

- How fast can the data get through?
  - Depends on raw bit rate, protocol overhead in packet
- How many hardware signals do we need?
  - May need clock line, chip select lines, etc.
- How do we connect multiple devices (topology)?
  - Dedicated link and hardware per device - point-to-point
  - One bus for master transmit/Servant receive, one bus for Servant transmit/master receive
  - All transmitters and receivers connected to same bus – multi-point
- How do we address a target device?
  - Discrete hardware signal (chip select line)
  - Address embedded in packet, decoded internally by receiver
- How do these factors change as we add more devices?

Protocol Comparison

Protocol	Speed	Device Addressing	Signals Req. for Bidirectional Communication with N devices
UART (Point to Point)	Fast – Tens of Mbit/s	None	2*N (TxD, RxD)
UART (Multi-drop)	Fast – Tens of Mbit/s	Added by user in software	2 (TxD, RxD)
SPI	Fast – Tens of Mbit/s	Hardware chip select signal per device	3+N for SCLK, MOSI, MISO, and one SS per device
I^2C	Moderate – 100 kbit/s, 400 kbit/s, 1 Mbit/s, 3.4 Mbit/s. Packet overhead.	In packet	2: SCL, SDA

Tools for Serial Communications Development



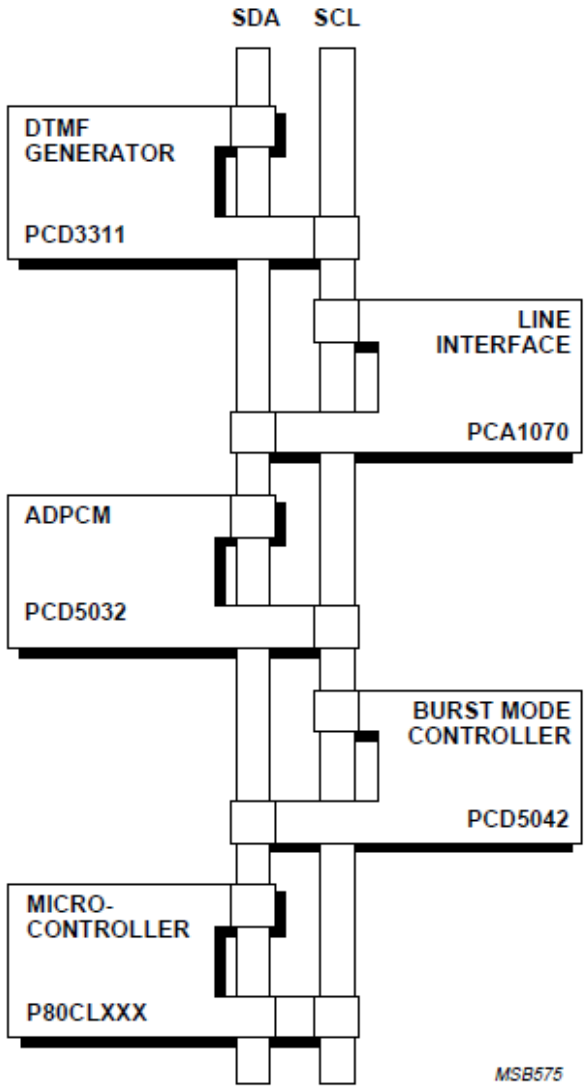
- Tedious and slow to debug serial protocols with just an oscilloscope
- Instead use a logic analyzer to decode bus traffic
- Worth its weight in gold!

- Hardware
  - Digilent Analog Discover 2, Digital Discovery
  - Saleae Logic
  - Other logic analyzers
- Software (FOSS)
  - Sigrok/Pulseview

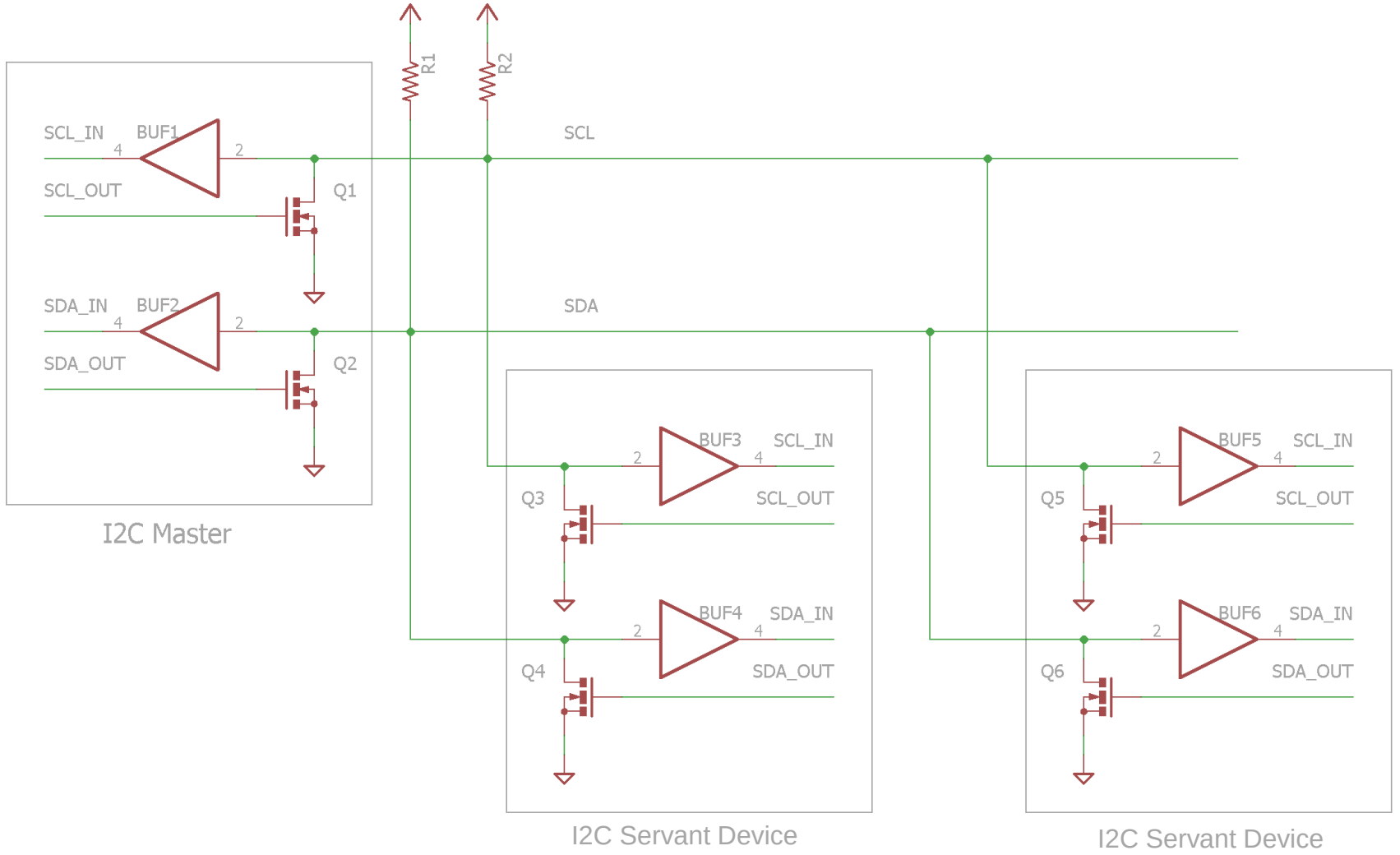
I^2C COMMUNICATIONS

I²C Bus Overview

- "Inter-Integrated Circuit" bus
- Multiple devices connected by a shared serial bus
- Bus is typically controlled by master, servant devices respond when addressed
- I²C bus has two signal lines
  - SCL: Serial clock
  - SDA: Serial data
- Full details available in "The I²C-bus Specification"

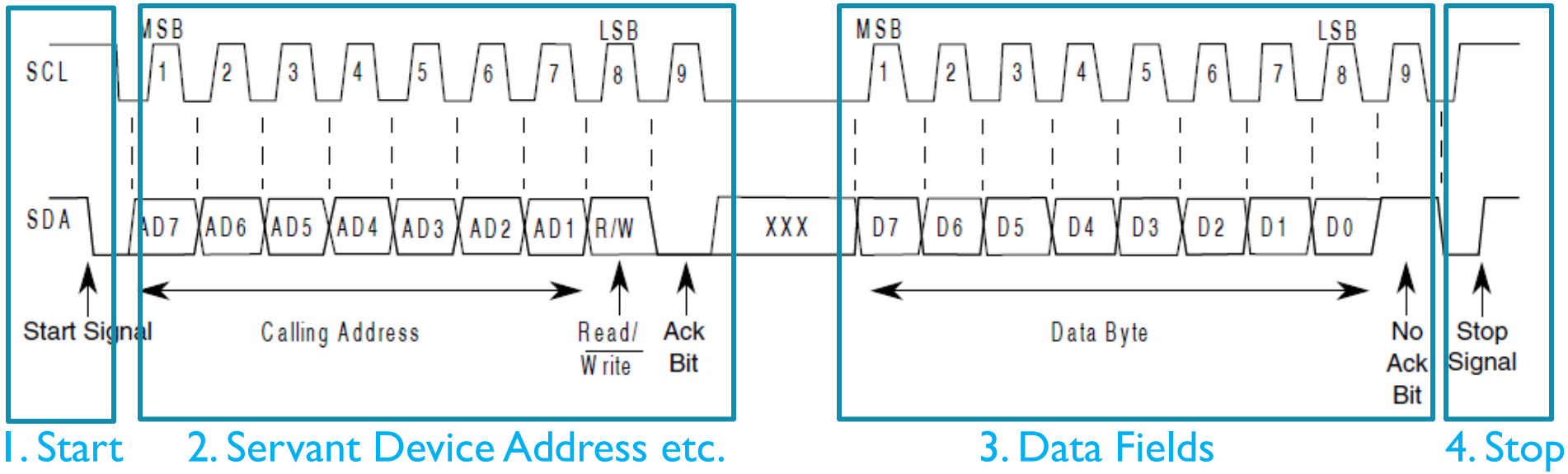


I²C Bus Connections



- Resistors pull up lines to V_{DD}
- Open-drain transistors pull lines down to ground
- Master generates SCL clock signal
  - Can range up to 400 kHz, 1 MHz, or more

I²C Message Format



- Message is made of
  - Signals: Start, Stop, Repeated Start
  - Bytes
  - Acknowledgement bits
- Message-oriented data transfer with four parts
  1. Start condition
  2. Servant Address transmission
     - Address
     - Command (read or write)
     - Acknowledgement by receiver
  3. Data fields
     - Data byte
     - Acknowledgement by receiver
  4. Stop condition

I²C Addressing: Devices and Registers

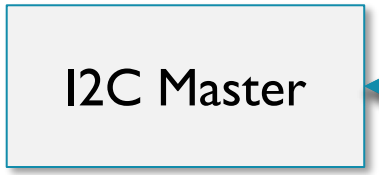
- Device addressing
  - Each device has a seven-bit address
  - Can support up to 2^7=128 different devices on same bus
  - Different types of device have different default addresses
  - Some devices support alternate address via config. pin

	Device Adx		Data	…	…	Data	

- Register addressing
  - Some devices have multiple addressable registers within device (e.g. control, status, data)
  - First byte of data is interpreted as ***register*** address

	Device Adx		Reg Adx	Data	…	Data	

  - Example: First seven registers of MMA8451 I2C accelerometer



0x1e
HMC5883L Magnetometer

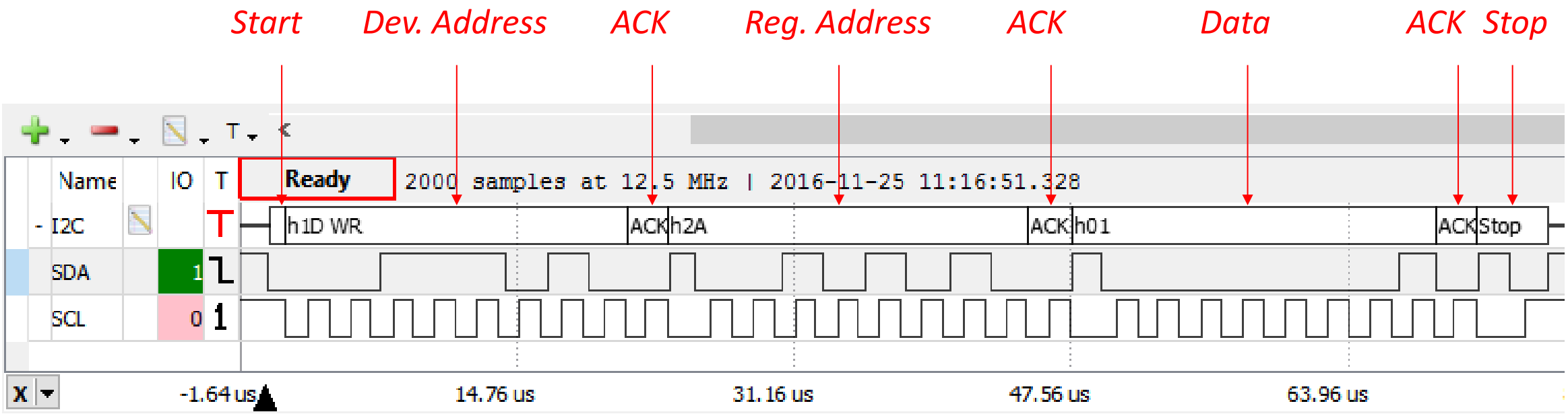
Address Location	Name
00	Configuration Register A
01	Configuration Register B
02	Mode Register
03	Data Output X MSB Register
04	Data Output X LSB Register
05	Data Output Z MSB Register
06	Data Output Z LSB Register
07	Data Output Y MSB Register
08	Data Output Y LSB Register
09	Status Register
10	Identification Register A
11	Identification Register B
12	Identification Register C

0x1c
MMA8451 Accelerometer

Name	Type	Register Address	Comment
STATUS/F_STATUS(1)(2)	R	0x00	FMODE = 0, real time status FMODE > 0, FIFO status
OUT_X_MSB(1)(2)	R	0x01	[7:0] are 8 MSBs of 14-bit sample. Root pointer to XYZ FIFO data.
OUT_X_LSB(1)(2)	R	0x02	[7:2] are 6 LSBs of 14-bit real-time sample
OUT_Y_MSB(1)(2)	R	0x03	[7:0] are 8 MSBs of 14-bit real-time sample
OUT_Y_LSB(1)(2)	R	0x04	[7:2] are 6 LSBs of 14-bit real-time sample
OUT_Z_MSB(1)(2)	R	0x05	[7:0] are 8 MSBs of 14-bit real-time sample
OUT_Z_LSB(1)(2)	R	0x06	[7:2] are 6 LSBs of 14-bit real-time sample

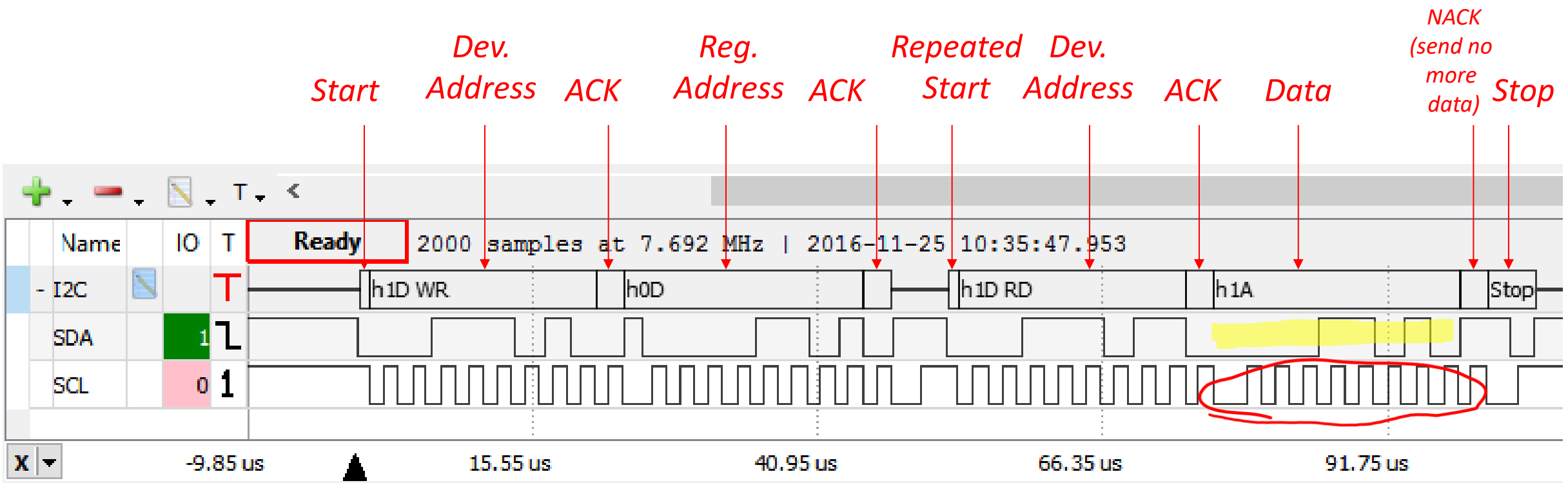
I2C Master Writing One Byte to Servant Device Register

Master Tx	Start	Device Address	W		Register Address		Data		Stop
Serv. Dev. Tx				ACK		ACK		ACK	



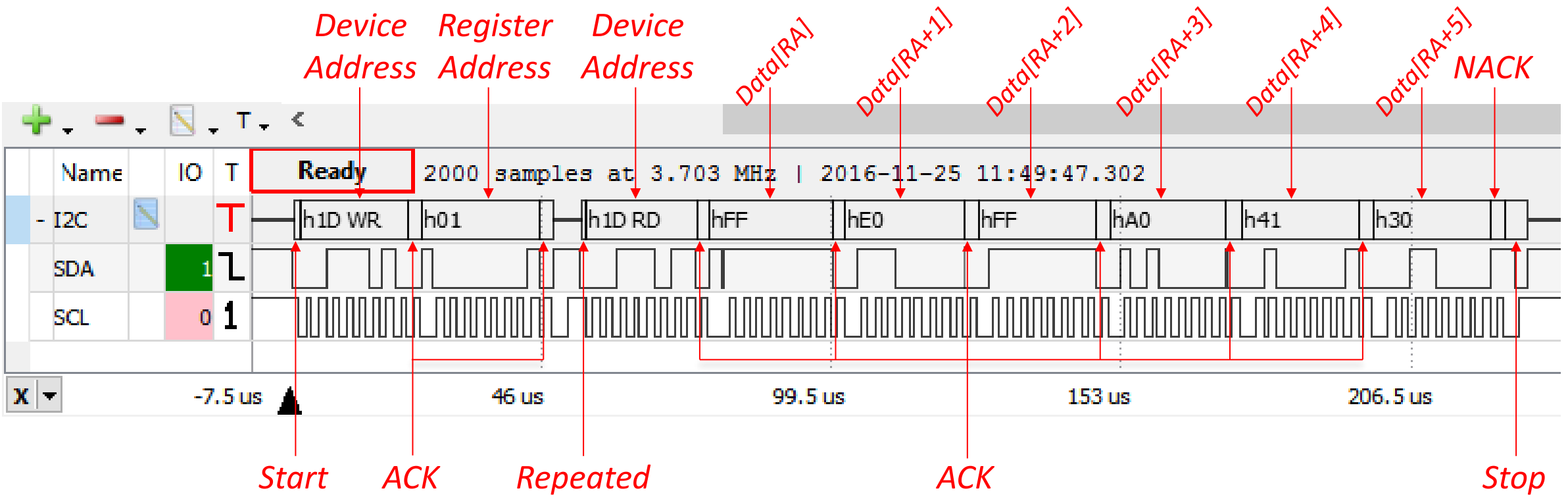
I2C Master Reading One Byte from Servant Device Register

Master Tx	Start	Device Address	W(0)		Register Address		Start	Device Address	R(1)				NACK(1)	Stop
Servant Tx				ACK(0)		ACK(0)				ACK(0)	Data			



I2C Master Reading Multiple Bytes from Servant Register(s)

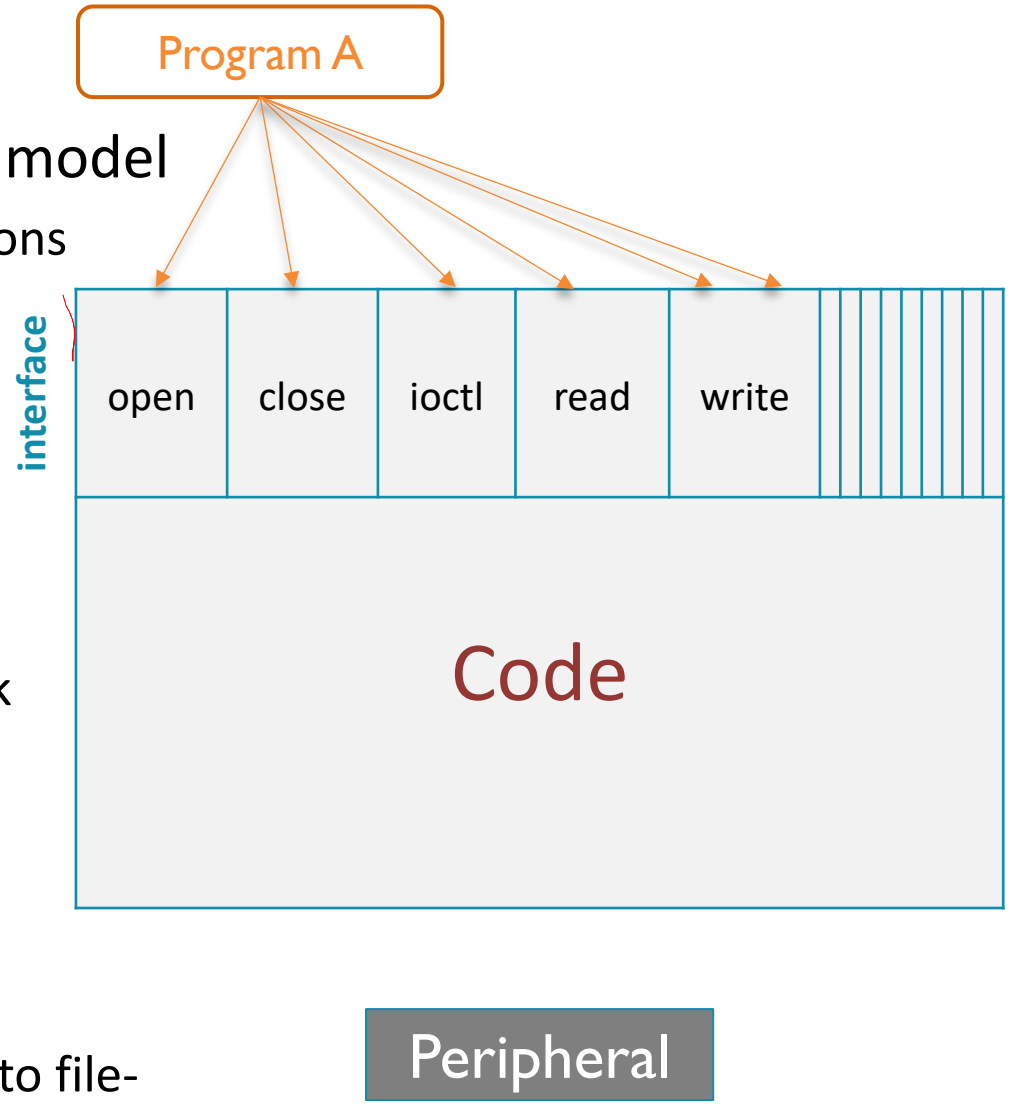
Master Tx	Start	Device Adx	W		Register Adx		Start	Device Adx	R(0)		ACK		ACK		ACK		ACK		ACK		NACK(1)	Stop
Servant Tx				ACK		ACK				ACK	Data		Data		Data		Data		Data			



PROGRAMMED ACCESS TO I^2C DEVICES

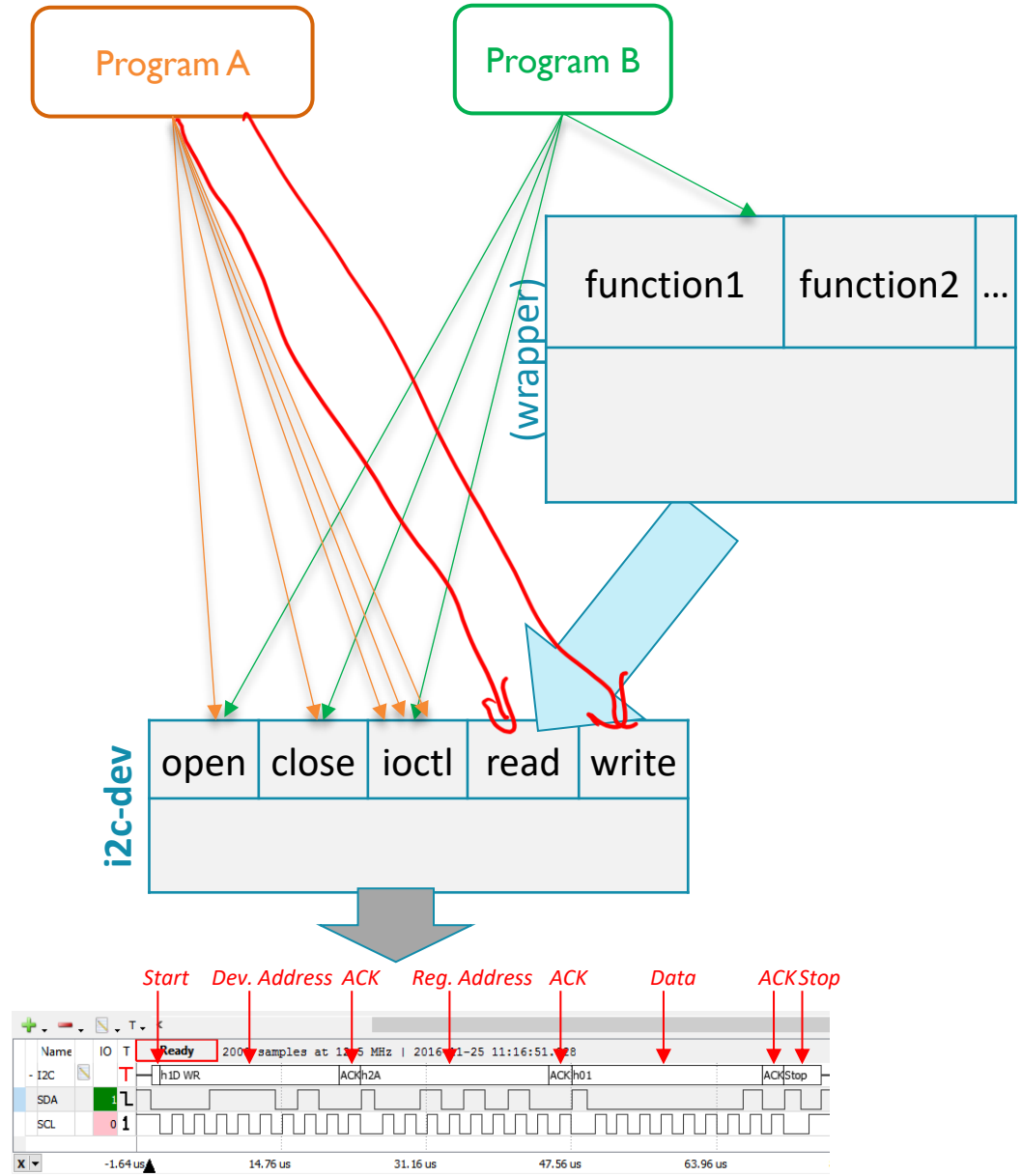
Background: Some Useful Device Driver Concepts

- Device driver interface follows “everything’s a file” model
  - Contains file_operations structure with pointers to functions implementing operations
- File-like operations
  - Open, close device
  - Read up to N bytes, write up to N bytes
  - Positioning: llseek
  - Poll device to find out on whether operations would block
  - Flush: block until all pending device operations finish
  - mmap: map device memory to process’s address space
  - Many others possible as well. Asynch. read/write, etc.
- Other
  - ioctl: generic device-specific commands which don’t fit into file-like operations



I²C Interfacing in C

- **i2c-dev** device driver module
  - Provides user-space I²C access
  - Table of contents: https://www.kernel.org/doc/html/latest/i2c/index.html
  - Nice overview: https://www.kernel.org/doc/html/latest/i2c/dev-interface.html
  - /usr/include/linux/i2c-dev.h
  - Load module i2c-dev to use device interface from userspace (link with libi2c via `-li2c`)
- i2c-dev has complex interface
  - User-space code may be simplified by using a wrapper with higher-level interface
  - Examples: libi2c, twiddler, **Exploring RPi** Chapter 8 I2CDevice



Overview of Using Kernel's I²C Device Driver

- Include headers `<linux/i2c-dev.h>`
- Open I²C adaptor (bus controller)
- Communicate with I²C device using `ioctl I2C_RDWR` calls
  - `ioctl`: I/O control interface for features which don't fit into [open|read|write|lseek|close] template
  - Example ioctl call:
    - i2c_file indicates I2C adapter to use
    - I2C_RDWR indicates operation
    - packets contains operation parameters
    - ret indicates success (>=0) or failure (<0)
- Close I²C adaptor

```
#include <linux/i2c-dev.h>

// open i2c adaptor as file
i2c_file = open("/dev/i2c-1", O_RDWR);
if (i2c_file < 0) {
        // handle error
}

// configure packets for I2C transaction
packets. … = …;

// have driver perform the transaction
ret = ioctl(i2c_file, I2C_RDWR, &packets);
if (ret < 0) {
        // handle error
}

// If read, access received data in packets
… = packets. …

// Close i2c adaptor
close(i2c_file);
```

I²C ioctl Commands

- General
  - I2C_SLAVE
    - Specify address for servant device. Is persistent until next ioctl with this command.
  - I2C_RDWR
    - Perform combined read/write transfer (repeated start)
    - Takes * to i2c_rdwr_ioctl_data, which includes Servant device address, register address, data
  - I2C_TENBIT
    - Selects seven or ten bit I²C addresses (if supported)
  - I2C_FUNCS
    - Returns list of I2C adapter features (“functionalities”)
    - https://www.kernel.org/doc/html/next/i2c/functionality.html
- SMBus only
  - I2C_PEC
    - Enables or disables SMBus Packet Error Checking
  - I2C_SMBUS
    - If available, use i2c_smbus_* functions

Discussion: Why not use lseek, read and write?

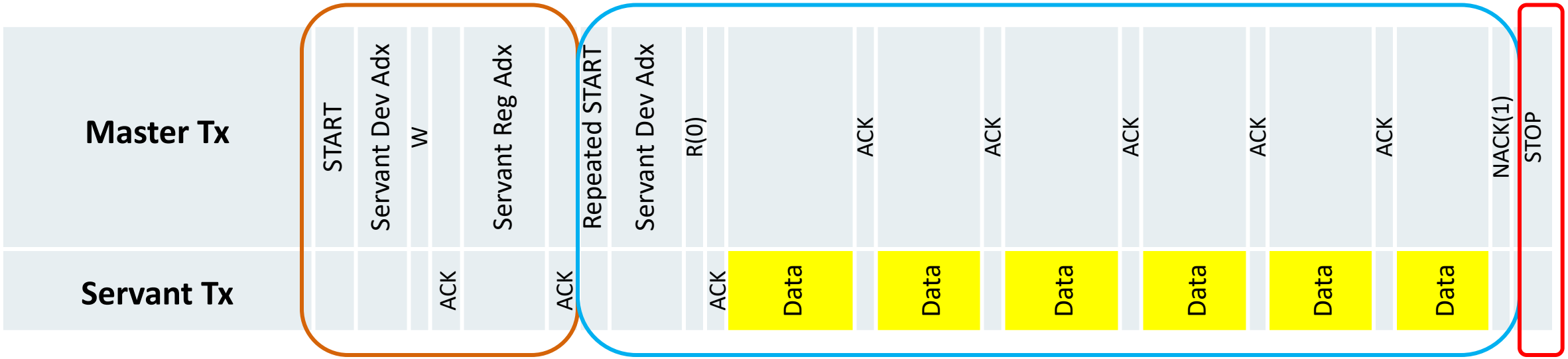
- lseek: set file pointer at specified location
- Reference
  - Write activities by master
    - Send device address + write command
    - Send register number
    - Send data byte(s)
  - Read activities by master
    - Send device address + write command
    - Send register number
    - Send repeated start
    - Send device address + read command
    - Receive data byte(s)

Writing I2C Device Registers

Master Tx	Start	Servant Dev Adx	W		Servant Reg Adx		Data		Data		Stop
Servant Tx				ACK		ACK		ACK		ACK	

- Writing to device register(s) is easy
  - Master sends START, sends device address, write command, (starting) register address, all data, then sends STOP
  - Servant just acknowledges each byte received

Reading I2C Device Registers: Transaction Segments



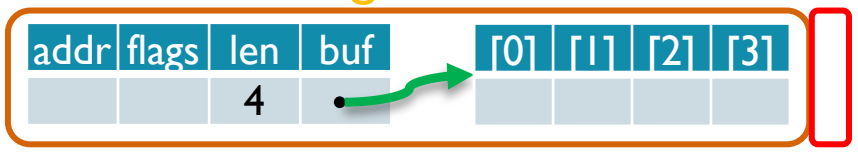
- Reading from a device register requires two I2C transaction segments
- Segment 1:
  - Master sends START, Servant device address, write command and Servant register address. Servant acknowledges each byte received.

Data structures used for I^2C are a little more complex. This enables actions common to both segments to share data and code

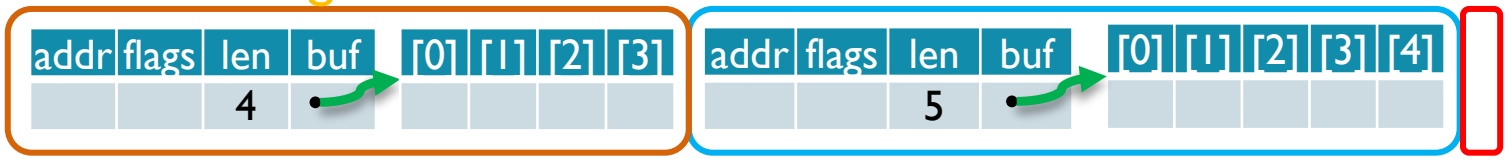
- Segment 2:
  - Master sends START (without an intervening STOP, which makes it a repeated start), sends Servant device address, read command. Servant acknowledges each byte received.
  - Servant sends first data byte
  - Master uses acknowledgement to control if Servant sends another data byte
    - If Master sends ACK, Servant sends another byte
    - If Master sends NACK, it doesn't want more data from Servant. Master sends STOP and ends transaction

Data Structure for Transaction Segments: i2c_msg

Write: One segment + STOP



Read:Two segments + STOP

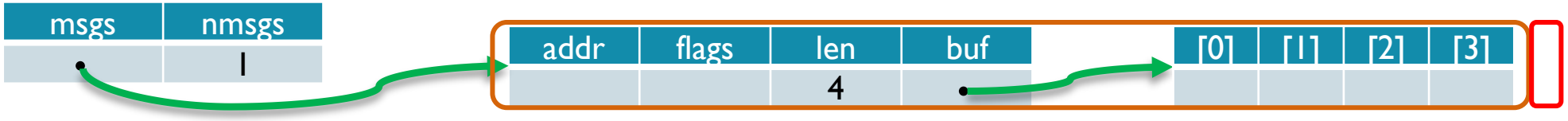


- Common actions for transaction segments
  - Send START condition
  - Send address
  - Send command (read or write)
  - Transfer N bytes of data to/from buffer
- "I2C transaction segment" description
  - Described by i2c_msg, defined in linux/i2c.h
  - Address of Servant device is 7 or 10 bits
  - Flags for message – indicate functionality.
    - I2C_M_RD: read. Absence implies write.
  - Length of data in buffer
  - Pointer to data buffer

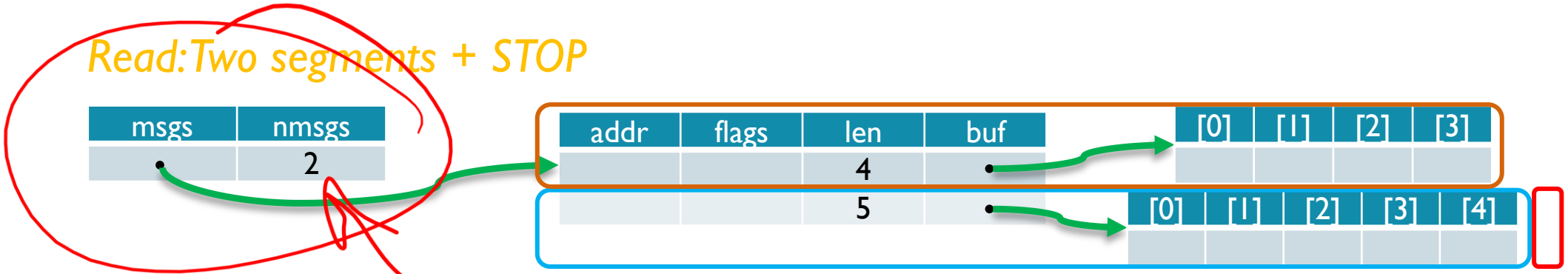
```
struct i2c_msg {
        __u16 addr;
        __u16 flags;
        __u16 len;              /* msg length
        __u8 *buf;              /* pointer to msg data
};
```

Data Structure for Transaction: i2c_rdwr_ioctl_data

Write: One segment + STOP



Read: Two segments + STOP

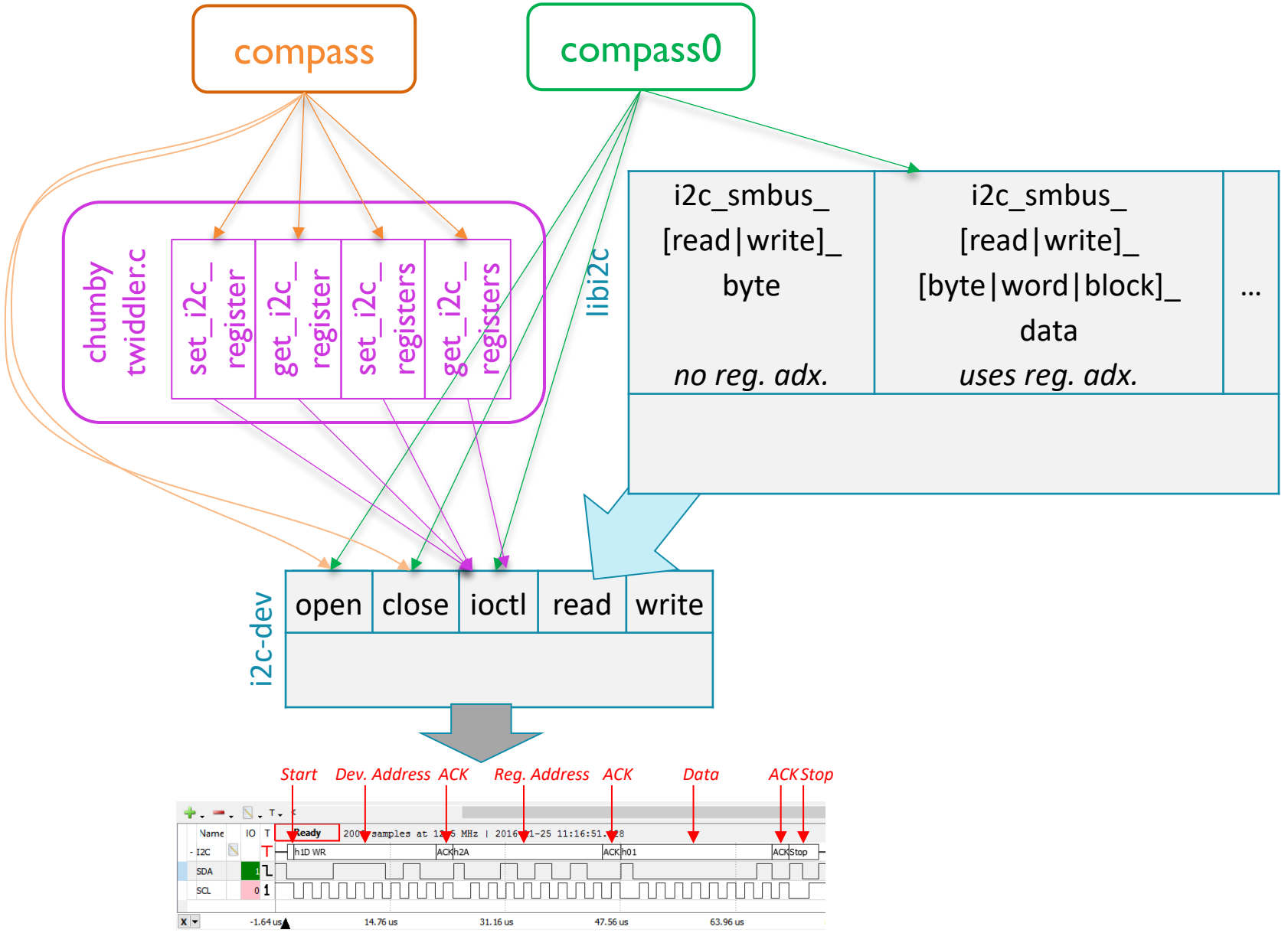


- Data structure defined in linux/i2c-dev.h
  - Pointer to array of i2c_msgs
  - Number of i2c_msgs

```
/* This is the structure as used in the I2C_RDWR ioctl call */
struct i2c_rdwr_ioctl_data {
        struct i2c_msg __user *msgs;    /* pointers to i2c_msgs */
        __u32 nmsgs;                    /* number of i2c_msgs */
};
```

Interface Encapsulation

- Encapsulate common code to assemble and manipulate data structures
- Useful to encapsulate into cleaner interface
- Compass example
  - Uses functions derived from chumby twiddler.c because they were documented better
  - twiddler.c was derived from …?

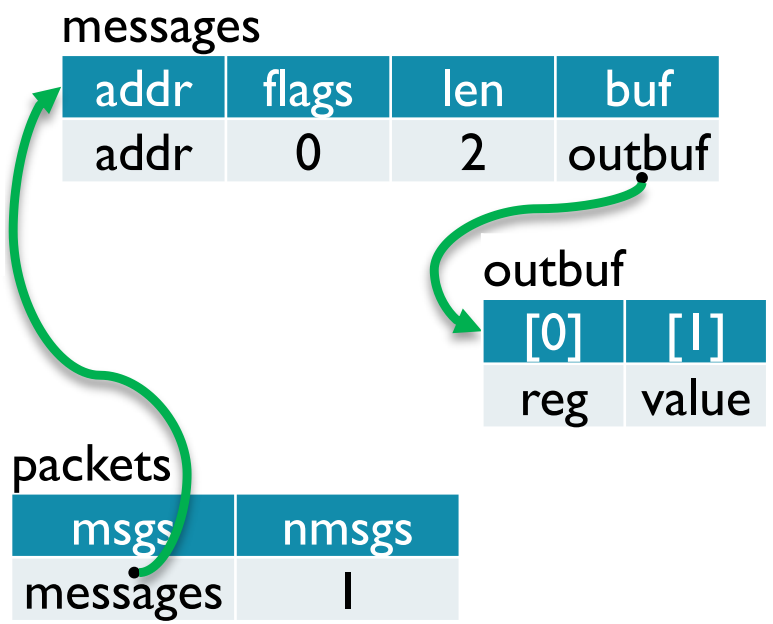


Write: set_i2c_register(file, addr, reg, value)

- 1. Prepare **messages[0]** entry to describe segment (write)
  - Device address, flags, data buffer, data length
- 2. Load up **outbuf** with data to send
  - First byte is register number
  - Second byte is data value
- 3. Prepare **packets** with list of segments ("messages") to send
- 4. Call ioctl, passing arguments…
  - File: which i2c bus
  - Operation: I2C_RDWR
  - Pointer to packets

```
unsigned char outbuf[2];
struct i2c_rdwr_ioctl_data packets;
struct i2c_msg messages[1];

messages[0].addr  = addr;
messages[0].flags = 0;
messages[0].len   = sizeof(outbuf);
messages[0].buf   = outbuf;


outbuf[0] = reg;

outbuf[1] = value;

packets.msgs  = messages;
packets.nmsgs = 1;

if(ioctl(file, I2C_RDWR, &packets) < 0) {
  perror("Unable to send data");
  return 1;
}
```

Write: One segment + STOP

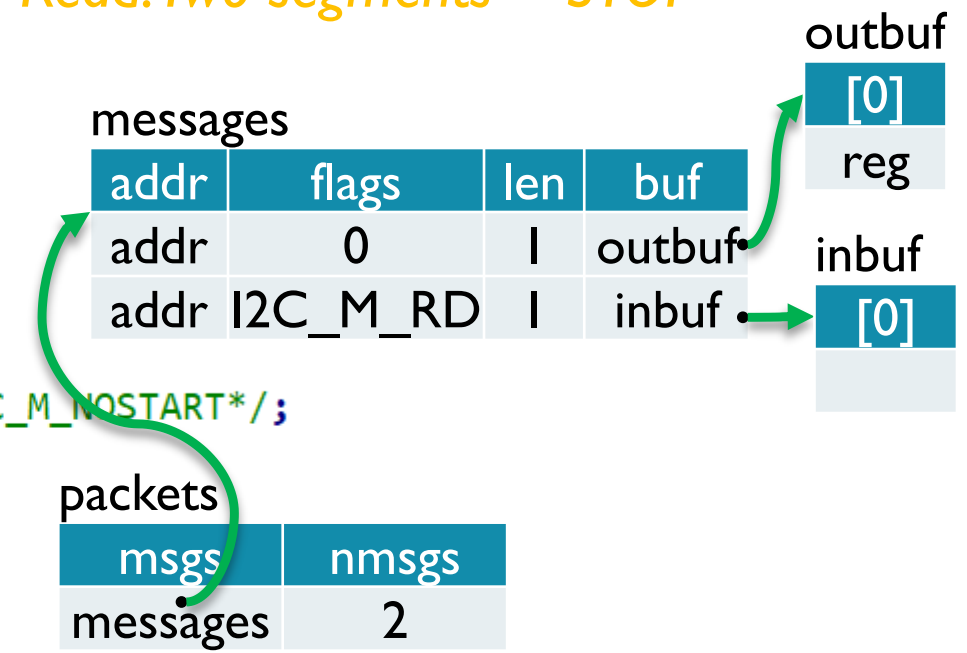


Read: get_i2c_register(file, addr, reg, *value)

- 1. Prepare **outbuf** with **reg**
- 2. Prepare **messages[0]** entry to describe segment (write)
- 3. Prepare **messages[1]** entry to describe segment (read)
- 4. Prepare **packets** with list of segments to send
- 5. Call ioctl, passing arguments
- 6. If successful, copy result from inbuf to *val

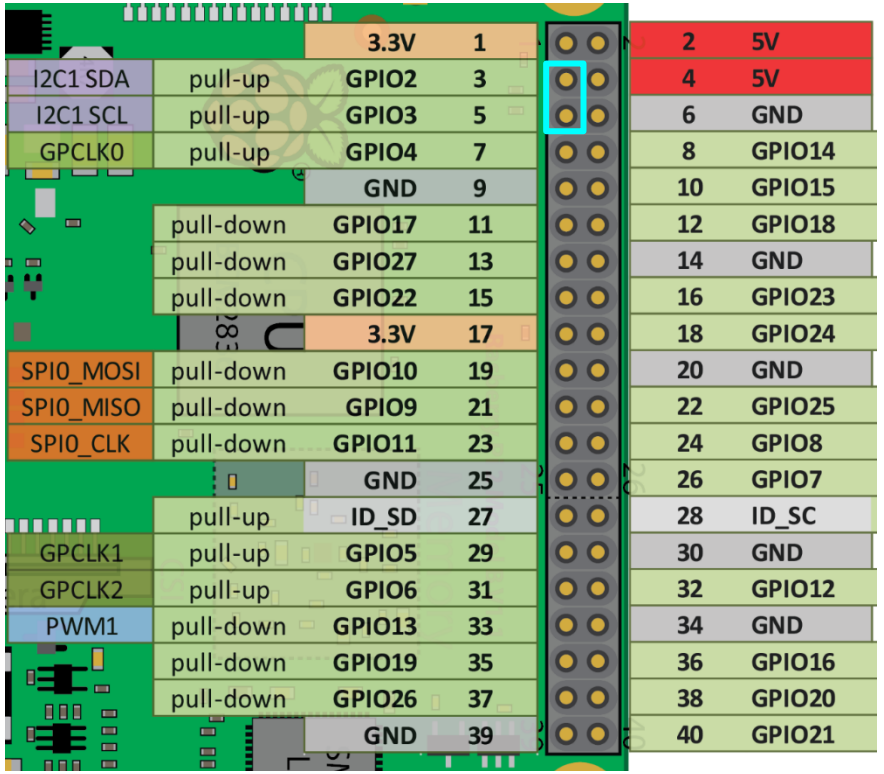
```
unsigned char inbuf, outbuf;
struct i2c_rdwr_ioctl_data packets;
struct i2c_msg messages[2];

outbuf = reg;
messages[0].addr  = addr;
messages[0].flags = 0;
messages[0].len   = sizeof(outbuf);
messages[0].buf   = &outbuf;

messages[1].addr  = addr;
messages[1].flags = I2C_M_RD/* | I2C_M_NOSTART*/;
messages[1].len   = sizeof(inbuf);
messages[1].buf   = &inbuf;

packets.msgs      = messages;
packets.nmsgs     = 2;

if(ioctl(file, I2C_RDWR, &packets) < 0) {
    perror("Unable to send data");
    return 1;
}
*val = inbuf;
```

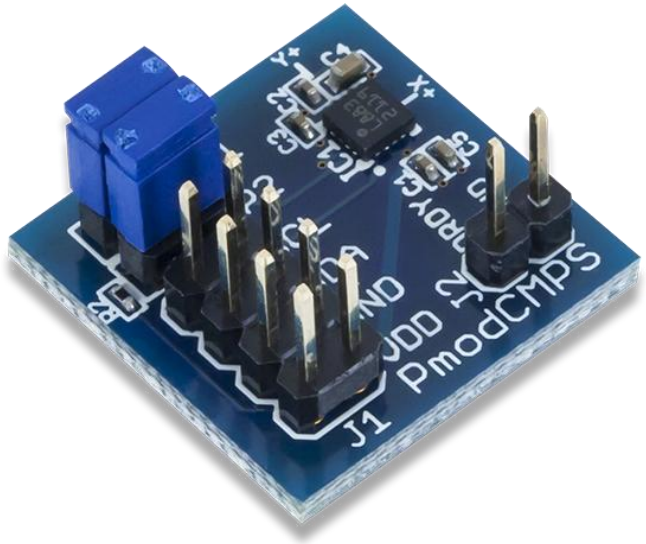
Read:Two segments + STOP



Interfacing with a Three-Axis Magnetometer

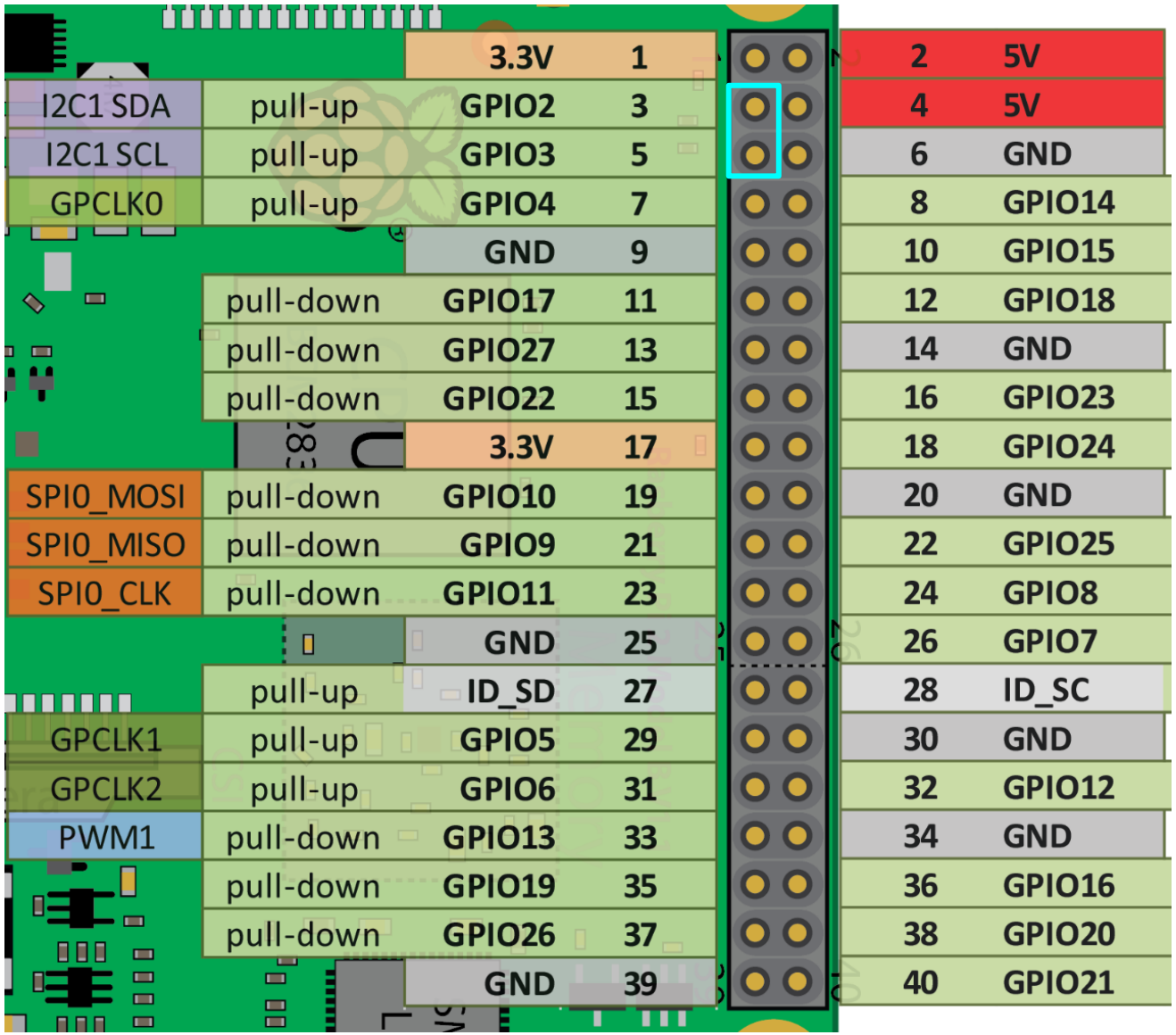
- Use to determine heading of vehicle, wind direction, robot arm direction, etc.
- Digilent PModCMPS
  - http://store.digilentinc.com/pmodcmps-3-axis-digital-compass/
  - (discontinued, replaced by PModCMPS2)
- Uses Honeywell HMC5883L
- Details
  - Connect to RPi
    - GND to pin 9
    - VCC to pin 1
    - SCL to pin 5
    - SDA to pin 3
  - I2C device address is 0x1e
  - Can access control registers with i2cget and i2cset





Pinouts

- RPi 3



- RPi 4

Function	Mode	Pin Numbers		Mode	Function
3V3	3.3V	1	2	5V	5V
SPI3 MOSI/SDA3	GPIO2	3	4	5V	5V
SPI3 SCLK/SCL3	GPIO3	5	6	GND	GND
SPI4 CE0 N/SDA 3	GPIO4	7	8	GPIO14	TXD1/SPI5 MOSI
GND	GND	9	10	GPIO15	RXD1/SPI5 SCLK
	GPIO17	11	12	GPIO18	SPI6 CEO N
SPI6 CE1 N	GPIO27	13	14	GND	GND
SDA6	GPIO22	15	16	GPIO23	SCL6
3V3	3.3V	17	18	GPIO24	SPI3 CE1 N
SDA5	GPIO10	19	20	GND	GND
RXD4/SCL4	GPIO9	21	22	GPIO25	SPI4 CE1 N
SCL5	GPIO11	23	24	GPIO8	SDA4/TXD4
GND	GND	25	26	GPIO7	SCL4/SPI4 SCLK
SPI3 CE0 N/TXD2/SDA6	ID_SD	27	28	ID_SC	SPI3 MISO/SCL6/RXD2
SPI4 MISO/RXD3/SCL3	GPIO5	29	30	GND	GND
SPI4 MOSI/SDA4	GPIO6	31	32	GPIO12	SDA5/SPI5 CEO N/TXD5
SPI5 MISO/RXD5/SCL5	GPIO13	33	34	GND	GND
SPI6 MISO	GPIO19	35	36	GPIO16	SPI1 CE2 N
SPI5 CE1 N	GPIO26	37	38	GPIO20	SPI6 MOSI
GND	GND	39	40	GPIO21	SPI6 SCLK

I2C
UART
SPI

Setting Up I²C on RPi

- Enable I²C
  - May be disabled by default
  - To enable
    - Using raspi-config ("Interfaces"), or
    - Edit /boot/config.txt to include `dtparam=i2c_arm=on`
- Module i2c_dev provides access via /dev/…

- Get i2c-tools package for useful tools
  - Install with **sudo apt install i2c-tools**
  - Command line programs: i2cdetect, i2cdump, i2cget, i2cset
  - Detailed coverage in ERP, pp.318-325

- Setting communication speed: See Exploring RPi

Registers for HMC5883L

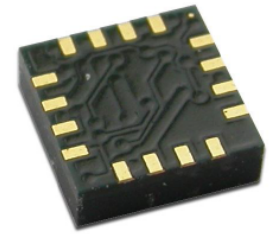
Register	Name	Access
00	Configuration Register A	Read/Write
01	Configuration Register B	Read/Write
02	Mode Register	Read/Write
03	Data Output X MSB Register	Read
04	Data Output X LSB Register	Read
05	Data Output Z MSB Register	Read
06	Data Output Z LSB Register	Read
07	Data Output Y MSB Register	Read
08	Data Output Y LSB Register	Read
09	Status Register	Read
10	Identification Register A	Read
11	Identification Register B	Read
12	Identification Register C	Read

3-Axis Digital Compass IC HMC5883L

Honeywell

Advanced Information

The Honeywell HMC5883L is a surface-mount, multi-chip module designed for low-field magnetic sensing with a digital interface for applications such as low-cost compassing and magnetometry. The HMC5883L includes our state-of-the-art, high-resolution HMC118X series magneto-resistive sensors plus an ASIC containing amplification, automatic degaussing strap drivers, offset cancellation, and a 12-bit ADC that enables 1° to 2° compass heading accuracy. The I^2C serial bus allows for easy interface. The HMC5883L is a 3.0x3.0x0.9mm surface mount 16-pin leadless chip carrier (LCC). Applications for the HMC5883L include Mobile Phones, Netbooks, Consumer Electronics, Auto Navigation Systems, and Personal Navigation Devices.



The HMC5883L utilizes Honeywell's Anisotropic Magnetoresistive (AMR) technology that provides advantages over other magnetic sensor technologies. These anisotropic, directional sensors feature precision in-axis sensitivity and linearity. These sensors' solid-state construction with very low cross-axis sensitivity is designed to measure both the direction and the magnitude of Earth's magnetic fields, from milli-gauss to 8 gauss. Honeywell's Magnetic Sensors are among the most sensitive and reliable low-field sensors in the industry.

FEATURES	BENEFITS
3-Axis Magnetoresistive Sensors and ASIC in a 3.0x3.0x0.9mm LCC Surface Mount Package	Small Size for Highly Integrated Products. Just Add a Micro-Controller Interface, Plus Two External SMT Capacitors Designed for High Volume, Cost Sensitive OEM Designs Easy to Assemble & Compatible with High Speed SMT Assembly
12-Bit ADC Coupled with Low Noise AMR Sensors Achieves 2 milli-gauss Field Resolution in ±8 Gauss Fields	Enables 1° to 2° Degree Compass Heading Accuracy
Built-In Self Test	Enables Low-Cost Functionality Test after Assembly in Production
Low Voltage Operations (2.16 to 3.6V) and Low Power Consumption (100 µA)	Compatible for Battery Powered Applications
Built-In Strap Drive Circuits	Set/Reset and Offset Strap Drivers for Degaussing, Self Test, and Offset Compensation
I^2C Digital Interface	Popular Two-Wire Serial Data Interface for Consumer Electronics
Lead Free Package Construction	RoHS Compliance
Wide Magnetic Field Range (+/-8 Oe)	Sensors Can Be Used in Strong Magnetic Field Environments with a 1° to 2° Degree Compass Heading Accuracy
Software and Algorithm Support Available	Compassing Heading, Hard Iron, Soft Iron, and Auto Calibration Libraries Available
Fast 160 Hz Maximum Output Rate	Enables Pedestrian Navigation and LBS Applications

Example Program: BES/I2C/Compass

- Uses open, close, ioctl and [get|set]_i2c_register[s]
- Basic operations
  - Initialization of I2C, compass
  - Reading data
  - Heading calculation
  - Statistics

Compass Program

- Try to open i2c file and check for errors
- Configure compass

```
if ((i2c_file = open("/dev/i2c-1", O_RDWR)) < 0) {
   perror("Unable to open i2c controller/file");
   exit(1);
}

address = COMPASS_DEV_ADDR;
if (configure_compass(i2c_file, address)) {
   printf("Unable to configure compass\n");
   return 1;
}
```

configure_compass(file, address)

- Read ID registers (“H43”)
- Test for ID string
- Write to configuration and mode registers

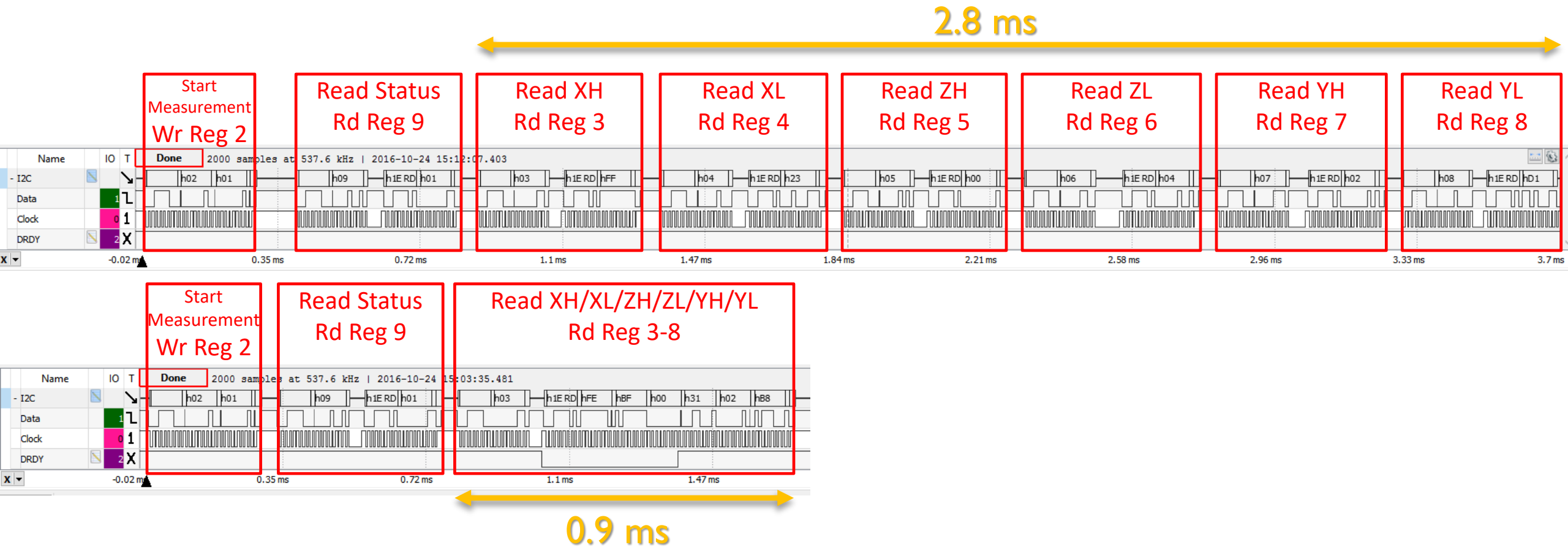
```
uint8_t compass_ID[3];

if (get_i2c_register(file, address, ID_REG_A, &(compass_ID[0])))
  return 1;
if (get_i2c_register(file, address, ID_REG_B, &(compass_ID[1])))
  return 1;
if (get_i2c_register(file, address, ID_REG_C, &(compass_ID[2])))
  return 1;
if (strncmp((char *) compass_ID, "H43", 3)) {
  printf("Compass not found. Expected H43, got %s.\n", compass_ID);
  return 1;
} else {
  printf("Compass found.\n");
}

// 4 averages, 30 Hz, normal measurement. 0101 0100
if (set_i2c_register(file, address, CONFIG_A_REG, 0x54))
  return 1;
// Gain
if (set_i2c_register(file, address, CONFIG_B_REG, 0x00))
  return 1;
// Operation mode
if (set_i2c_register(file, address, MODE_REG, IDLE_MODE))
  return 1;
```

Loop: Read Data

- Start measurements
- Poll status for completion
  - HMC-specific register and feature
- Read mag. vector data with six or one reads

```
// Start measurement
set_i2c_register(i2c_file, address, MODE_REG, SINGLE_MEASUREMENT_MODE);
// Await data ready
do {
  get_i2c_register(i2c_file, address, STATUS_REG, &status);
} while (!(status & 1));

#if READ_BYTES_INDIVIDUALLY // Read each field strength byte individually
for (i=0; i<6; i++) {
  get_i2c_register(i2c_file, address, XH_REG+i, &(data[i]));
  #if PRINT_RAW_BYTES
  printf("%02x ", data[i]);
  #endif
}
#else // read all field strength bytes in one transaction
get_i2c_registers(i2c_file, address, XH_REG, 6, data);
#if PRINT_RAW_BYTES
for (i=0; i<6; i++) {
  printf("%02x ", data[i]);
}
#endif // PRINT_RAW_BYTES
#endif // Not READ_BYTES_INDIVIDUALLY
```

Benefits of Single Transaction Reading Multiple Registers



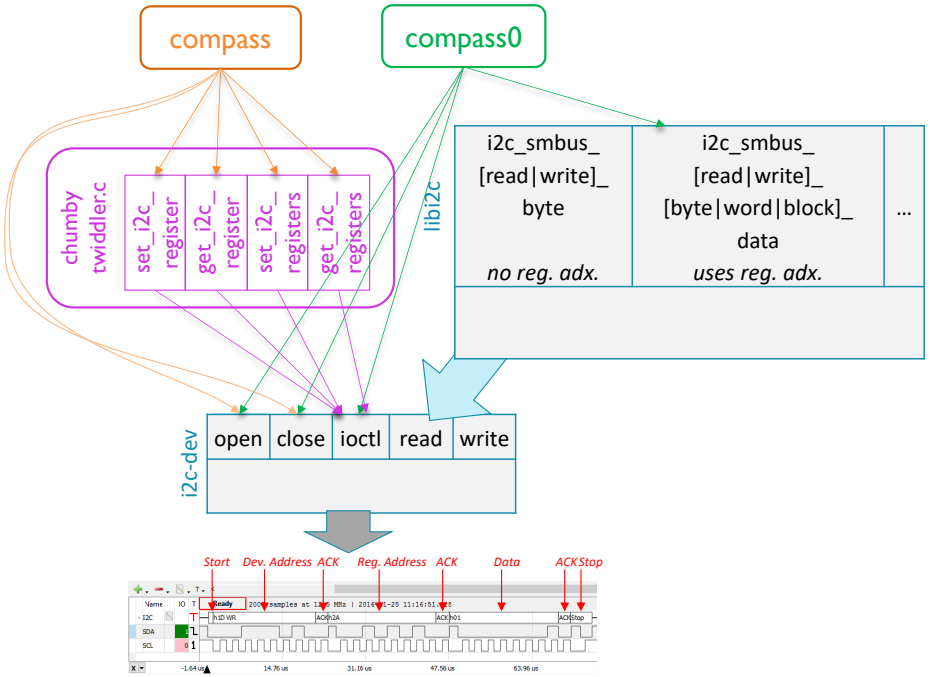
Loop: Use Data

- Convert bytes to field strength components
- Calculate statistics, field strength
- Calculate heading, print data

```
x_val = (((int16_t) data[0]) << 8) + data[1];
z_val = (((int16_t) data[2]) << 8) + data[3];
y_val = (((int16_t) data[4]) << 8) + data[5];

update_limits(x_val, y_val, z_val);
strength = sqrt(x_val*x_val + y_val*y_val + z_val*z_val);

printf("M:(%4d, %4d, %4d) ", x_val, y_val, z_val);
#if PRINT_RANGE
printf("M Range:(%4d~%3d Y: %4d~%3d Z: %4d~%3d) ",
       x_lim.min, x_lim.max, y_lim.min, y_lim.max, z_lim.min, z_lim.max);
#endif
printf(" Hdg: %6.2f Str: %6.2f\r", calc_heading(x_val, y_val), strength);
```

Alternative Implementations

Comparing Alternative Implementations



- ??? (Sean Cross, Chumby)
  - Uses more of kernel interface
    - File I/O to open/close device
    - ioctl I2C_RDWR for read/write
  - Provides functions
    - get_i2c_register, set_i2c_register, get_i2c_registers

- Exploring Raspberry Pi (chp08)
  - Uses part of kernel interface
    - File I/O to open/read/write/close device
    - ioctl to set Servant address
  - Provides base C++ class I2CDevice
    - open, close, readRegister, readRegisters, writeRegister, debugDumpRegisters
  - Derive device-specific classes (e.g. ADXL345) from I2CDevice
- Pan/Tilt Base code from Arducam
  - Bypasses I2C hardware, uses software-implemented I^2C in sccb_bus.c/h
  - Provides functions:
    - wrSensorReg8_8, rdSensorReg8_8
    - Bus operations: sccb_bus_[init, start, write_byte, read_byte, send_ack, send_noack, stop]
  - Uses memory-mapped access to peripherals (gpio, spi, pwm, sys_timer, uart, cm_pwm) in bcm283x_board_driver.c/h
    - Doesn't work with libi2c, need to port application code

??? (chumby) (built on ioctl I2C_RDWR)

- From Sean Cross / Chumby Industries
- Used in Compass demo
- See BES/I2C/compass/twiddler.c

```
int set_i2c_register(int file,
                     unsigned char addr,
                     unsigned char reg,
                     unsigned char value);

int get_i2c_register(int file,
                     unsigned char addr,
                     unsigned char reg,
                     unsigned char *val);

int get_i2c_registers(int file,
                     unsigned char addr,
                     unsigned char first_reg,
                     unsigned char num_regs,
                     unsigned char *val);
```

Example Program: BES/I2C/Compass0

- Basic operations
  - Initialization of I2C, compass
  - Reading data
- Heading calculation
- Offset error
  - Measurement
  - Compensation

read and write?

I2C Interfacing in C

- Try to open the device, and check for errors

```
int file;
int adapter_nr = 2; /* probably dynamically determined */
char filename[20];

snprintf(filename, 19, "/dev/i2c-%d", adapter_nr);
file = open(filename, O_RDWR);
if (file < 0) {
  /* ERROR HANDLING; you can check errno */
  exit(1);
}
```

Limitations of using I2C Read and Write Operations

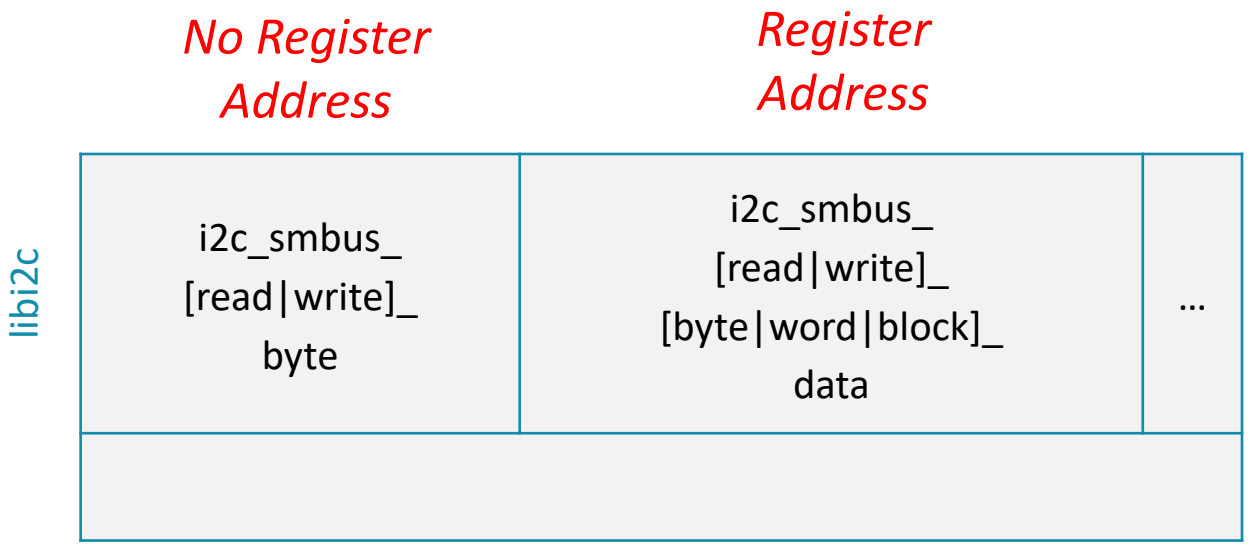
```
int addr = 0x40; /* The I2C device address */
__u8 register = 0x10; __s32 res; char buf[10];
if (ioctl(file, I2C_SLAVE, addr) < 0)
        exit(1);
/*Using I2C Write, equivalent of
 * i2c_smbus_write_word_data(file, reg, 0x6543)
 */
buf[0] = register; // write 0x6543 to register
buf[1] = 0x43; buf[2] = 0x65;
if (write(file, buf, 3) != 3) {
     /* ERROR HANDLING: i2c transaction failed */
}
/* Using I2C Read, equivalent of i2c_smbus_read_byte(file) */
if (read(file, buf, 1) != 1) {
   /* ERROR HANDLING: i2c transaction failed */
} else {
   /* buf[0] contains the read byte */
}
```

Messy and tedious having to handle messages and bytes at this level. And how to do repeated start conditions?

Compass0: smbus functions

SMBus Protocol

- System Management Bus protocol
  - Developed for smart battery systems, system and power management components, etc.
  - Subset of I²C protocol
- Documentation
  - https://www.kernel.org/doc/html/next/i2c/smbus-protocol.html
  - Recommended to use smbus rather than I2C functions to increase compatibility
    - *If you write a driver for some I2C device, please try to use the SMBus commands if at all possible (if the device uses only that subset of the I2C protocol). This makes it possible to use the device driver on both SMBus adapters and I2C adapters (the SMBus command set is automatically translated to I2C on I2C adapters, but plain I2C commands can not be handled at all on most pure SMBus adapters).*
  - Refers to register address as "command" (clashes with I2C read/write command bit)



- Functions defined in smbus.h
  - Data sizes: byte, word (two bytes), block (up to 32 bytes)
  - i2c_smbus_*_byte vs. i2c_smbus_*_data
    - *_byte functions don't use register address
    - *_data functions do use register address
  - Other functions
    - Process call, handle host notify, alert,
  - Other i2c functions
    - _i2c_smbus_[read|write]_i2c_block_data

Compass0 Program

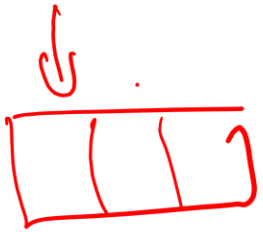
- Uses open/close and calls to libi2c (i2c_smbus_....)
- Open i2c file
- Set device address for upcoming I2C communications

```
if ((i2c_file = open("/dev/i2c-1", O_RDWR)) < 0) {
   perror("Unable to open i2c control file");
   exit(EXIT_FAILURE);
 }

 address = COMPASS_DEV_ADDR;
 if ((ioctl(i2c_file, I2C_SLAVE, address) < 0)) {
   perror("ioctl error\n");
   exit(EXIT_FAILURE);
 }
```

Configure Compass Sensor

- Read ID registers (“H43”)

```
result = i2c_smbus_read_i2c_block_data(file, ID_REG_A,
                                       3, compass_ID);

if (result < 0) {
  perror("read byte data error");
  exit(EXIT_FAILURE);
}
…
```



- Write to configuration and mode registers

```
result = i2c_smbus_write_byte_data(file, CONFIG_A_REG, 0x54);
if (result)
  return 1;
result = i2c_smbus_write_byte_data(file, CONFIG_B_REG, 0x00);
…
result = i2c_smbus_write_byte_data(file, MODE_REG, IDLE_MODE);
…
```

Read and Use Data

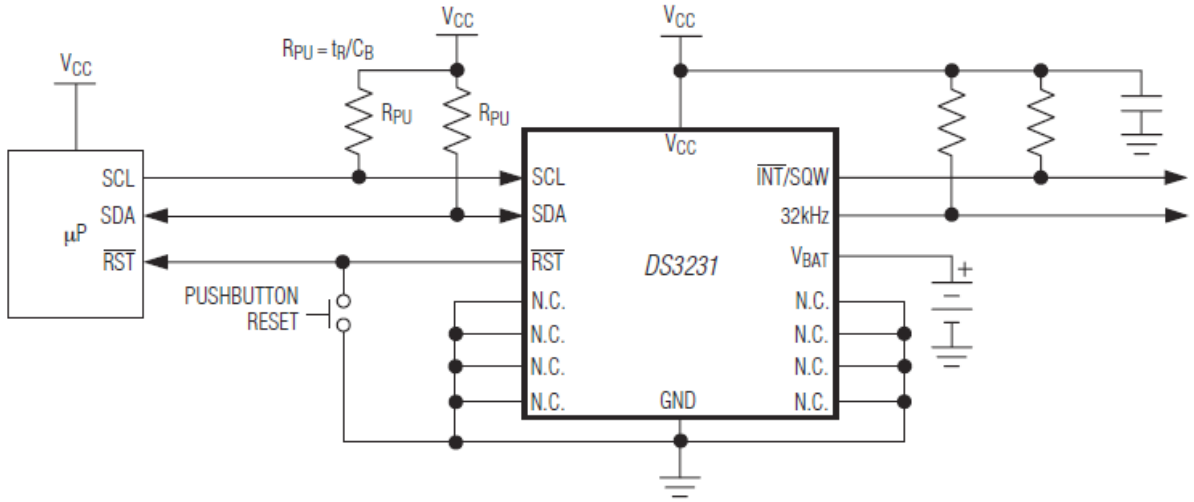
- Start measurements
- Poll status for completion
- Read mag. vector data
- Close i2c file

```
while(run) {
  i2c_smbus_write_byte_data(i2c_file, MODE_REG,
SINGLE_MEASUREMENT_MODE);

  // Await data ready
  do {
    result = i2c_smbus_read_byte_data(i2c_file, STATUS_REG);
    if (result < 0) {
      perror("Error reading status\n");
    } else
      status = result;
  } while (!(status & 1));

  // read all field strength bytes in one transaction
  result = i2c_smbus_read_i2c_block_data(i2c_file, XH_REG,
     6, data);
...
}
close(i2c_file);
```

Old Slides

Interfacing with a Real-Time Clock

- If device loses track of time when powered off, add an external real-time clock (RTC)
- Details
  - Maxim DS3231 IC on Macetech Chronodot v2.1, http://docs.macetech.com/doku.php/chronodot_v2.0
  - I2C device address is 0x68
  - Access control registers with i2cget and i2cset
  - Note that time and calendar registers are in BCD (binary-coded decimal) format



DS3231 Register Map

ADDRESS	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0 LSB	FUNCTION	RANGE
00h	0	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Hours	1–12 + $\overline{AM}$/PM 00–23
03h	0	0	0	0	0	Day			Day	1–7
04h	0	0	10 Date		Date				Date	01–31
05h	Century	0	0	10 Month	Month				Month/ Century	01–12 + Century
06h	10 Year				Year				Year	00–99
07h	A1M1	10 Seconds			Seconds				Alarm 1 Seconds	00–59
08h	A1M2	10 Minutes			Minutes				Alarm 1 Minutes	00–59
09h	A1M3	12/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + $\overline{AM}$/PM 00–23
0Ah	A1M4	DY/$\overline{DT}$	10 Date		Day				Alarm 1 Day	1–7
					Date				Alarm 1 Date	1–31
0Bh	A2M2	10 Minutes			Minutes				Alarm 2 Minutes	00–59
0Ch	A2M3	12/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + $\overline{AM}$/PM 00–23
0Dh	A2M4	DY/$\overline{DT}$	10 Date		Day				Alarm 2 Day	1–7
					Date				Alarm 2 Date	1–31
0Eh	$\overline{EOSC}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE	Control	—
0Fh	OSF	0	0	0	EN32kHz	BSY	A2F	A1F	Control/Status	—
10h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	Aging Offset	—
11h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	MSB of Temp	—
12h	DATA	DATA	0	0	0	0	0	0	LSB of Temp	—

More I2C Devices

- Microchip 24LC32A EEPROM
  - 4 kilobyte memory
  - http://ww1.microchip.com/downloads/en/DeviceDoc/21713M.pdf
- Honeywell HMC6352 Digital Compass Solution
  - I2C interface
  - 2.7 to 5.2V supply range
  - 1 to 20Hz selectable update rate
  - 0.5 degree heading resolution
  - 1 degree repeatability
  - Supply current : 1mA @ 3V
  - http://www.sparkfun.com/datasheets/Components/HMC6352.pdf
  - http://www.sparkfun.com/commerce/product_info.php?products_id=7915

Changing I2C Baud Rate

- **ERP** Chapter 8, p.315 doesn't seem to work

- Do this
  - sudo nano /boot/config.txt
  - Change line dtparam=i2c_arm=on to dtparam=i2c_arm=on,i2c_arm_baudrate=400000
  - Save file and exit
  - sudo reboot

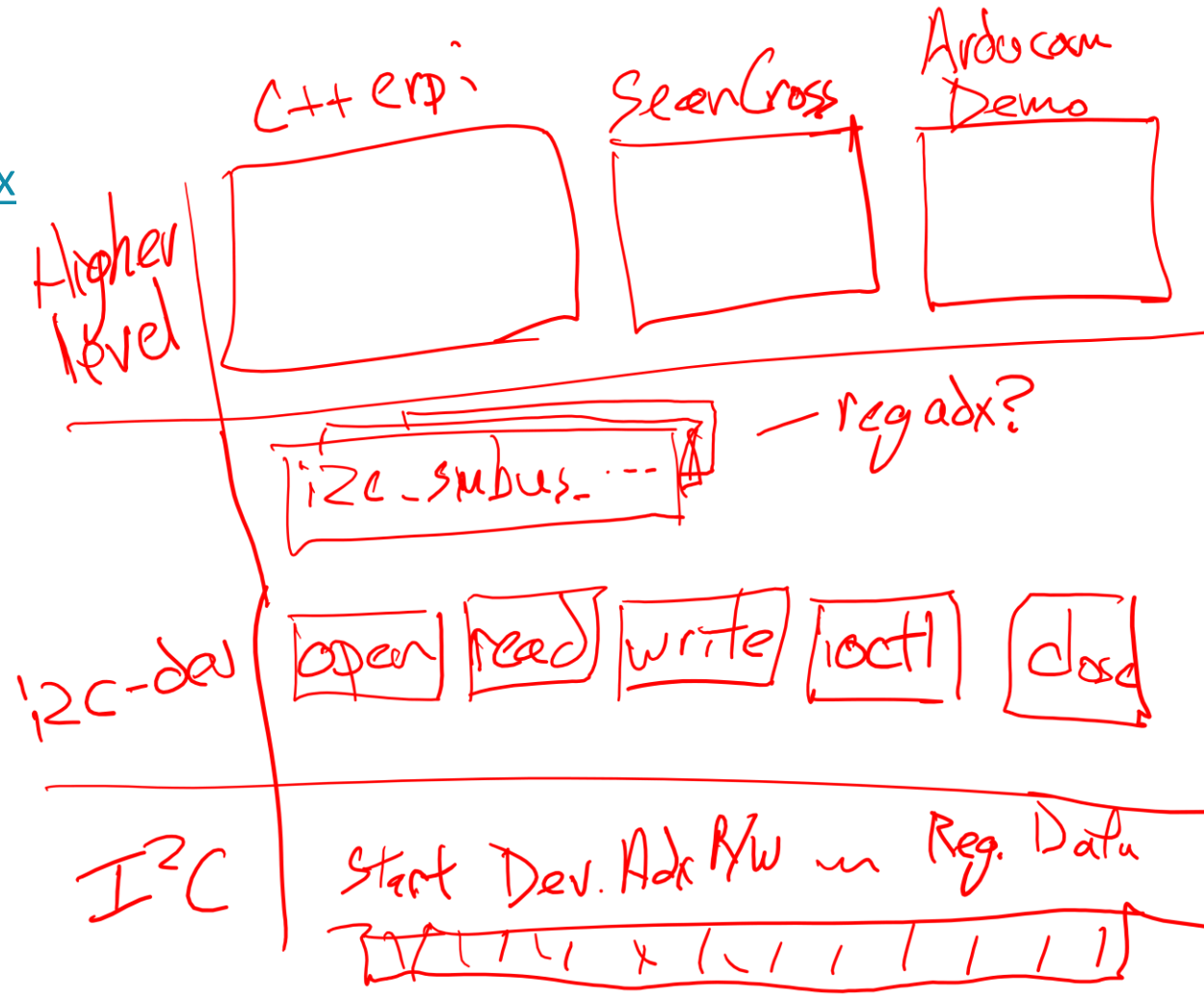
- Maximum I2C baud rate for CPU (and many peripherals) is 400 kHz
  - https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf

Monitoring Communications with Logic Analyzer

- What is the baud rate?
- Is there any significant delay when preparing to send a message?

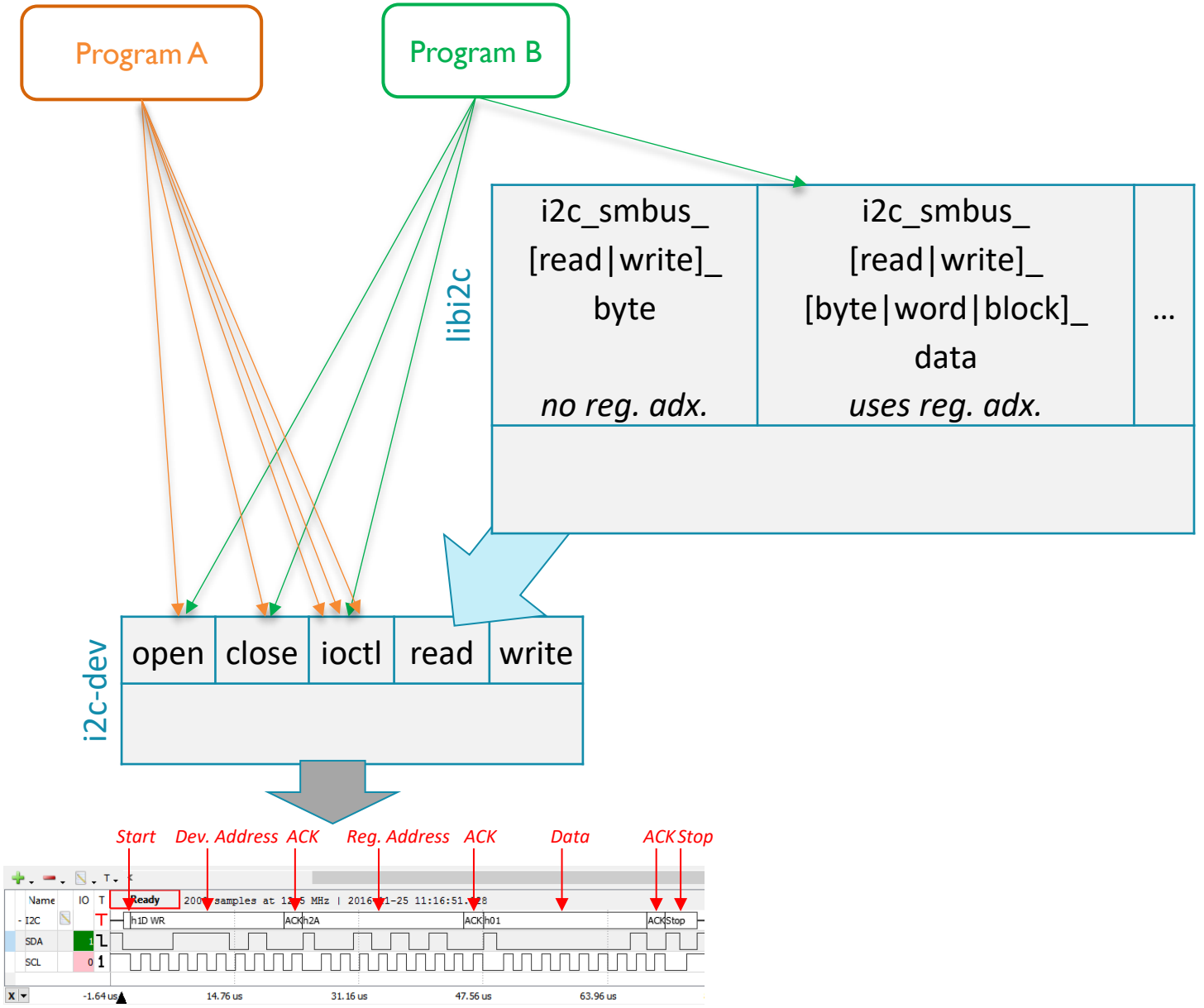
I2C Interfacing in C

- Kernel device driver module i2c-dev provides I2C access
  - Table of contents: https://www.kernel.org/doc/html/latest/i2c/index.html
  - Nice overview: https://www.kernel.org/doc/html/latest/i2c/dev-interface.html
- Load module i2c-dev to use dev interface from userspace
- Code often uses higher-level interface which builds on this interface



I2C Interfacing in C

- Kernel device driver module i2c-dev provides I2C access
  - Table of contents: https://www.kernel.org/doc/html/latest/i2c/index.html
  - Nice overview: https://www.kernel.org/doc/html/latest/i2c/dev-interface.html
  - /usr/include/linux/i2c-dev.h
- Load module i2c-dev to use dev interface from userspace (link with libi2c via `-li2c`)
- Code often uses higher-level interface which builds on these interfaces



I2C Interfacing in C

- Kernel device driver module i2c-dev provides I2C access
  - Table of contents: https://www.kernel.org/doc/html/latest/i2c/index.html
  - Nice overview: https://www.kernel.org/doc/html/latest/i2c/dev-interface.html
  - /usr/include/linux/i2c-dev.h
- Load module i2c-dev to use dev interface from userspace (link with libi2c)
- Code often uses higher-level interface which builds on these interfaces