

I²C bus (Inter-Integrated Circuit)

- ▶ Designed for low-cost, medium data rate applications.
(Phillips Semiconductor, 1980s)
- ▶ **Tutorial:** <http://www.esacademy.com/faq/i2c/>
- ▶ **Characteristics:**
 - ▶ serial, byte-oriented;
 - ▶ multiple-master;
 - ▶ fixed-priority arbitration;
 - ▶ moderate speeds:
 - ▶ standard mode: 100Kbits/s
 - ▶ fast mode: 400Kbits/s
 - ▶ high speed mode: 3.4 Mbits/s
- ▶ Many microcontrollers come with built-in I²C controllers.

Serial Buses Information Page: <http://www.epanorama.net/links/serialbus.html>



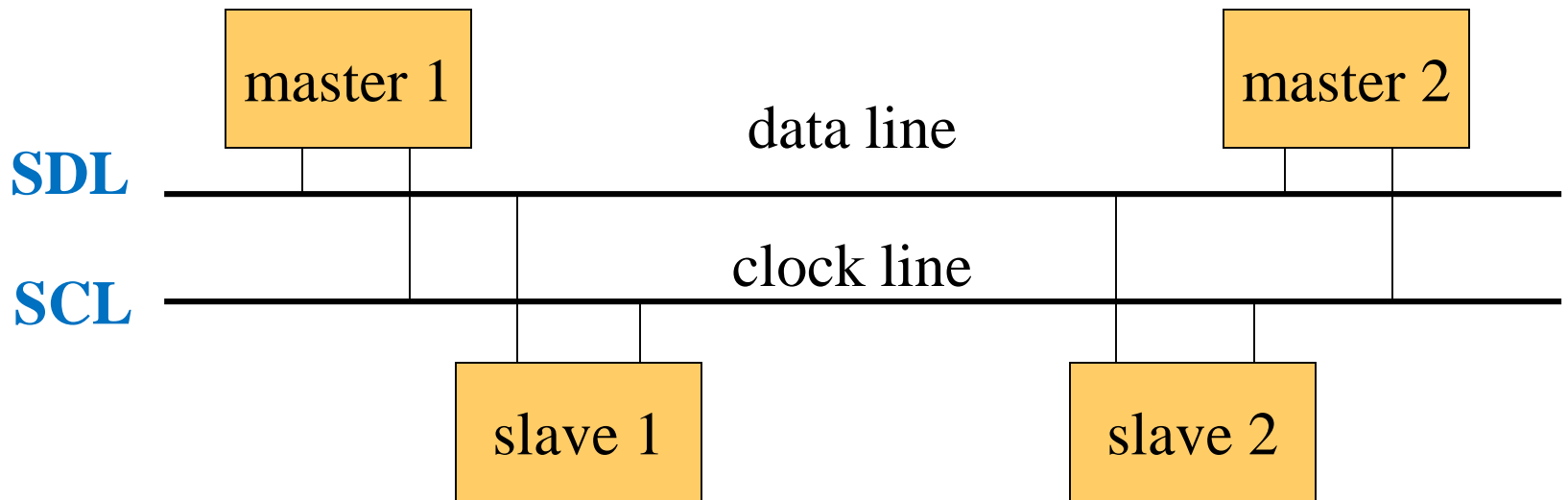
I²C data link layer

- ▶ Every device has an address
 - ▶ Set by device and/or system designer.
 - ▶ 7 bits in standard (10 bits in extension).
 - ▶ Bit 8 of address signals read (1) or write (0).
- ▶ General call address (0000000) for broadcast.
- ▶ Bus transaction = series of one-byte transmissions
 - ▶ Master sends slave address followed by data to or from slave.
 - ▶ Good for “data-push” programming.



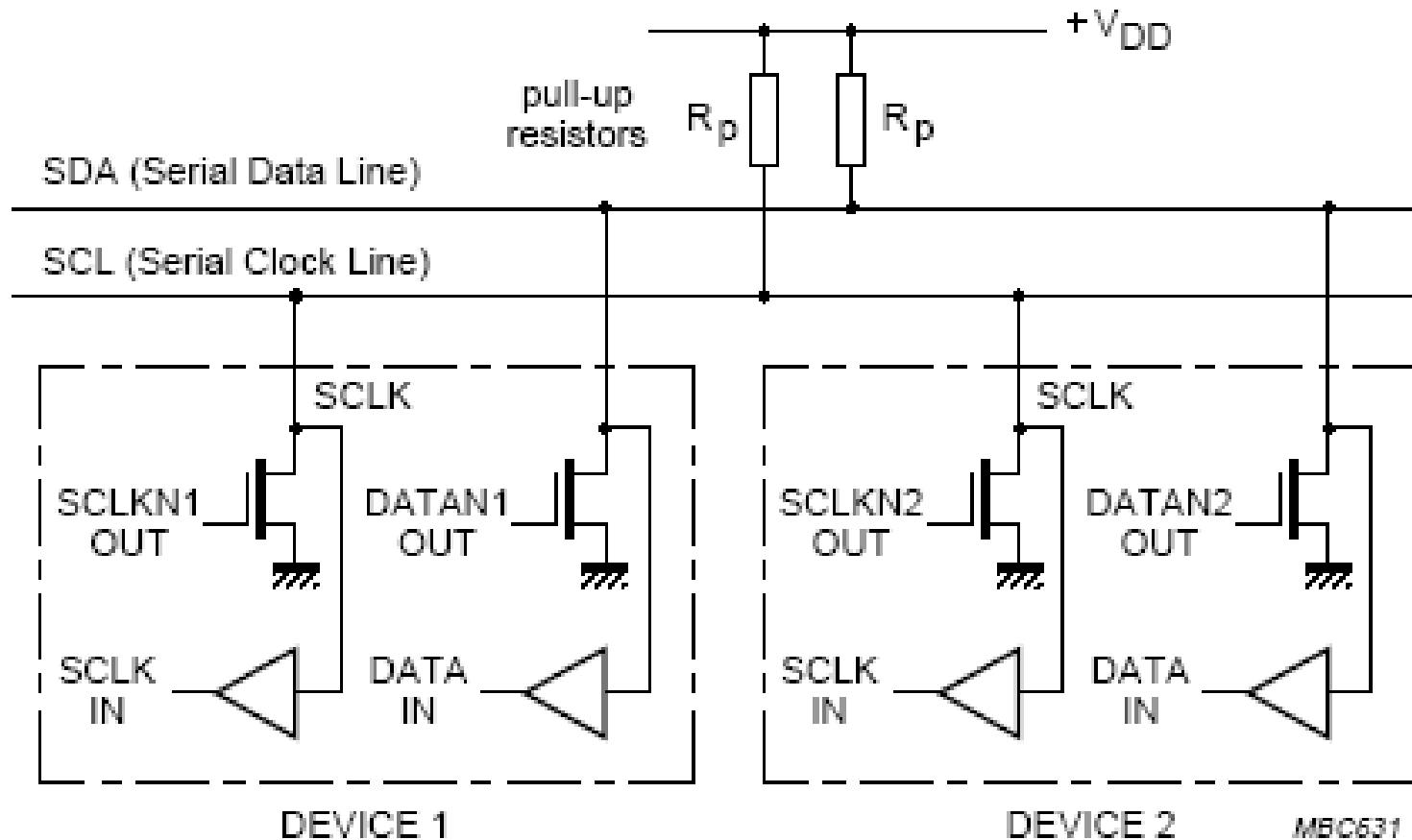
I²C physical layer

- Uses only two wires (plus ground)



I²C electrical interface (standard & fast speeds)

- ▶ Open collector/drain drivers (default state high)
- ▶ No global master for clock



I²C signaling

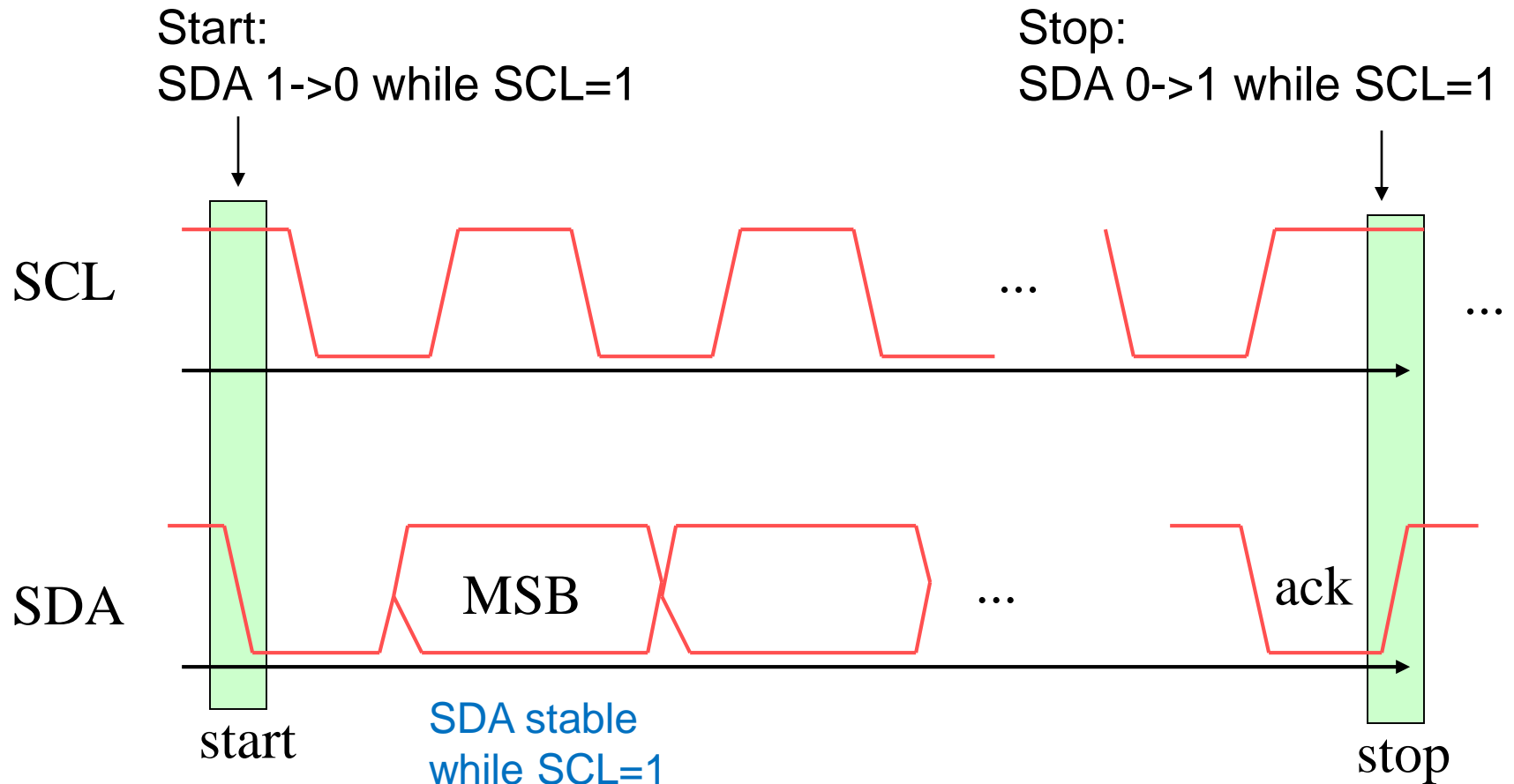
- ▶ Bus = “wired-AND” configuration
 - ▶ Open collector/drain drivers on SDA & SCL
 - ▶ Resistor pulls bus up to logic 1.
 - ▶ Any sender can pull the bus down to 0, even if other senders are trying to drive the bus to 1.
 - ▶ Sender “releases” SDA by disabling its driver, allowing SDA to be pulled up to logic 1
- ▶ Data on SDA must be stable while SCL high
 - ▶ Data on SDA is sampled while SCL is high
 - ▶ SDA may change only while SCL low

Exceptions:

- ▶ SDA 1->0 while SCL=1 signals START condition
- ▶ SDA 0->1 while SCL=1 signals STOP condition

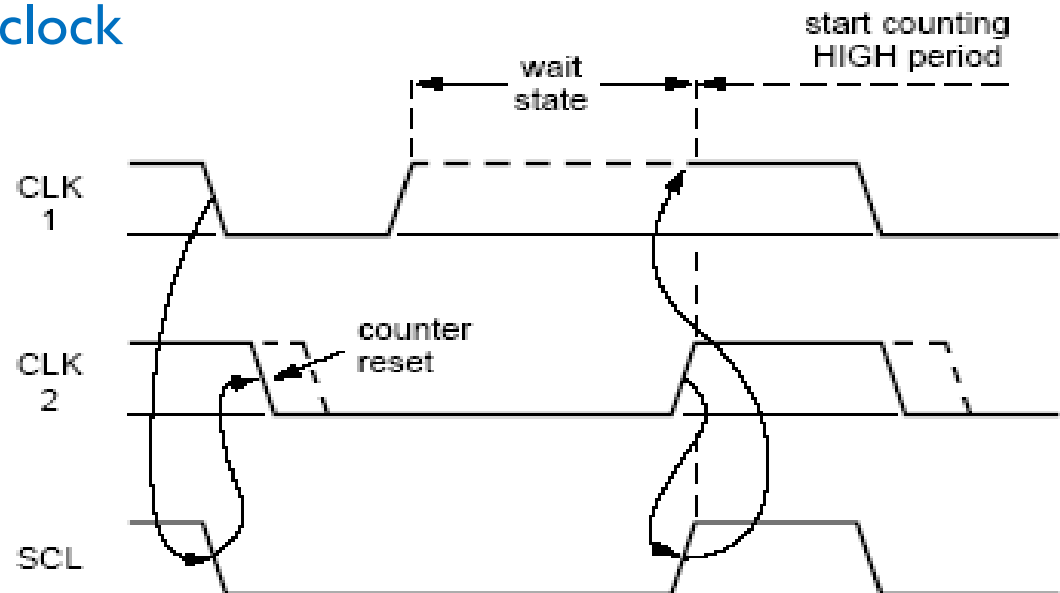


I²C data format



Clock synchronization

- ▶ Master generates its own clock on SCL during data xfer
- ▶ Clock synchronization uses wired-AND
 - ▶ Driving low pulls SCL low, resetting all clock counters
 - ▶ SCL remains low while any driver pulls it low
 - ▶ SCL low time = slowest clock (others in wait states)
 - ▶ First device to finish high state pulls SCL low



Source: I2C Specification

MBC632

Four I2C device operating modes

- ▶ **Master-sender**
 - ▶ Module issues **START** and **ADDRESS**, and then transmits data to the addressed slave device
- ▶ **Master-receiver**
 - ▶ Module issues **START** and **ADDRESS**, and receives data from the addressed slave device
- ▶ **Slave-sender**
 - ▶ Another master issues **START** and the **ADDRESS** of this module, which then sends data to the master
- ▶ **Slave-receiver**
 - ▶ Another master issues **START** and the **ADDRESS** of this module, which then receives data from the master.

Some devices only support slave modes – sensors, memories, etc.

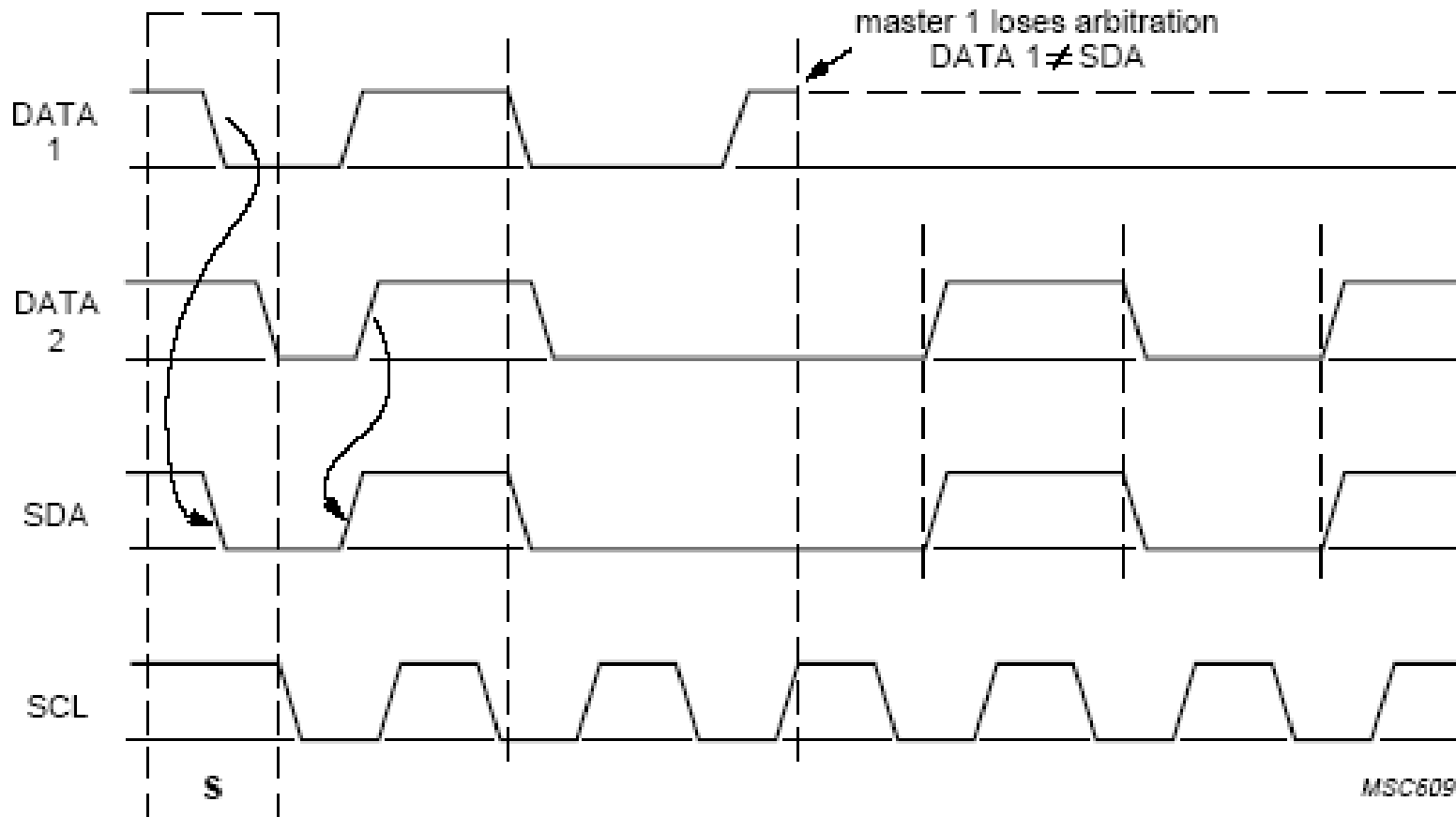


I²C bus arbitration

- ▶ Master may start sending if bus free
 - ▶ 2 or more may generate START at same time
- ▶ Sender listens while sending.
 - ▶ Test SDA while SCL high
- ▶ Sender stops transmitting if arbitration lost
 - ▶ Transmit 1 and hear 0 on SDA.
- ▶ Arbitration continues through address & ack bits, and then data & ack bits if necessary



Arbitration example



Source: I2C Specification

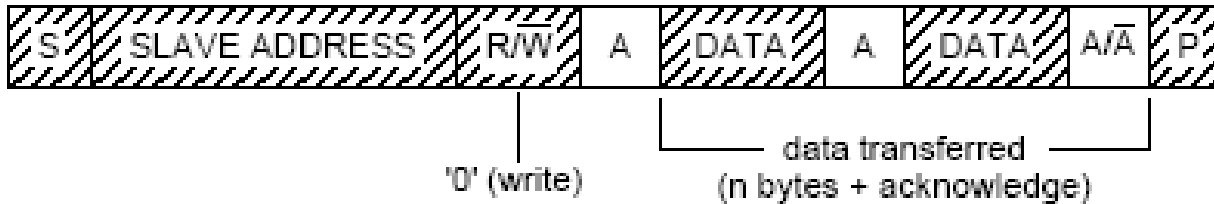
Data transfer

- ▶ Send 8-bit byte (MSB first)
- ▶ Each byte followed by acknowledge bit
 - ▶ master releases SDA line (high) during ack clock
 - ▶ slave must pull SDA low for proper acknowledge
 - ▶ if SDA left high, master may stop or repeat start
 - ▶ if master is receiving from slave, slave releases SDA to allow master to pull SDA low for ack
- ▶ Slave can hold SCL low to force wait time between bytes



Basic data formats

Master transmitting data to slave



 from master to slave

 from slave to master

MBC505

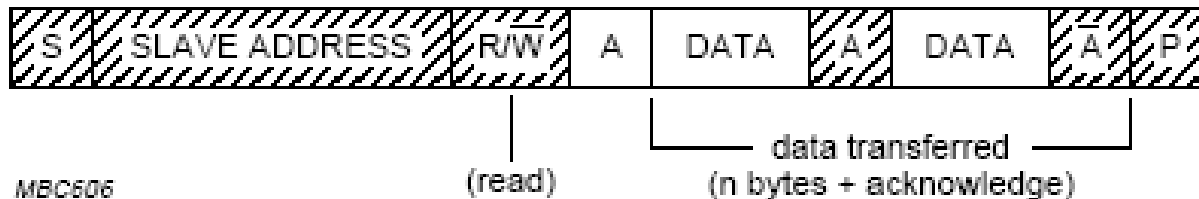
A = acknowledge (SDA LOW)

\bar{A} = not acknowledge (SDA HIGH)

S = START condition

P = STOP condition

Master receiving data from slave



MBC506

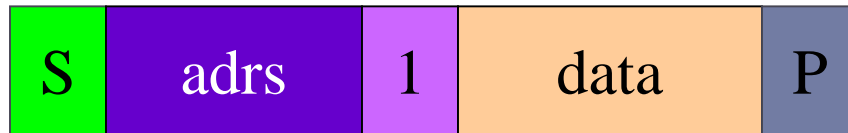


I²C transmissions (ACKs not shown)

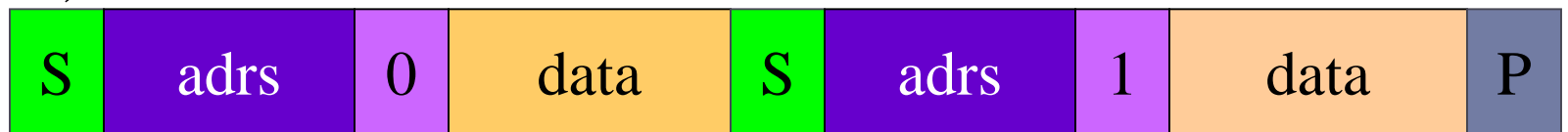
multi-byte write



read from slave



write, then read



↑
Re-start without giving up the bus

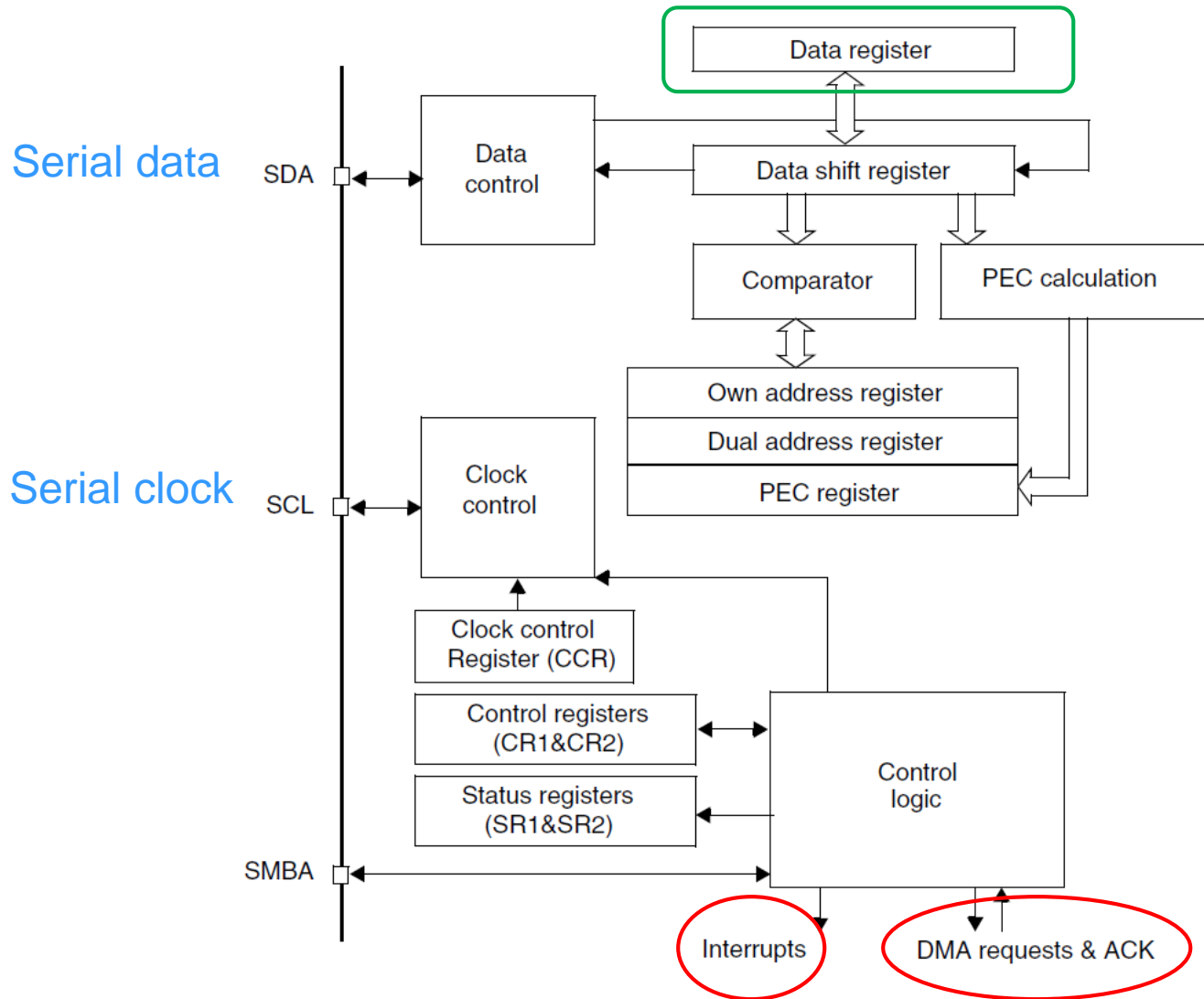


STM32 I²C Module (3 in STM32F407)

- ▶ **Standard I²C compliant bus interface.**
 - ▶ All I²C bus-specific sequencing, protocol, arbitration, timing
 - ▶ 7-bit and 10-bit addressing
 - ▶ Standard ($\leq 100\text{KHz}$) or Fast ($\leq 400\text{KHz}$) speed modes
 - ▶ Multi-master capability – use as master or slave
- ▶ **Also supports standards:**
 - ▶ SMBus (System Management Bus)
 - ▶ PMBus (Power Management Bus)
- ▶ **DMA support – between memory and data register**
- ▶ **2 interrupt vectors – data transfer complete and errors**



STM32 I²C Module

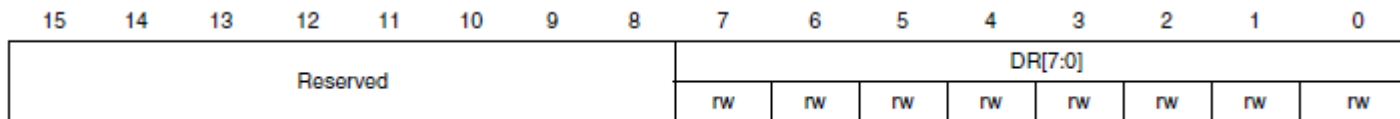


STM32 I²C registers

I2C_DR – I²C data register

byte to be transmitted (start on DR write)

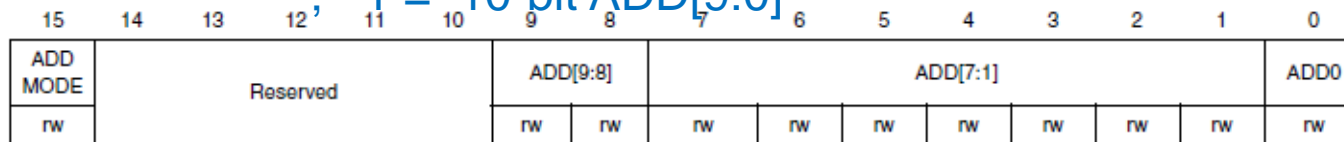
byte received (RxNE=1)



I2C_OAR1 – I²C own address register 1

ADDMODE 0 = 7-bit ADD[7:1]

; 1 = 10-bit ADD[9:0]



(A second “own address” is also supported)



STM32 I²C – control register 1

I2C_CR1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWRST	Res.	ALERT	PEC	POS	ACK	STOP	START	NO STRETCH	ENGC	ENPEC	ENARP	SMB TYPE	Res.	SMBUS	PE
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw

- PE** Peripheral function Enable (1 enables the I²C module)
- STOP** Generate after current byte xfer or after start condition sent
- START** Master: repeated start generation, Slave: release bus after byte xfer
- ACK** ACK to be returned after byte received
- POS** If ACK bit = 1: return ACK after current byte (0) or next byte (1)
-
- SWRST** Software reset (or in reset state)
- NOSTRETCH** Enable/disable clock stretch in slave mode when ADDR or BRG flag set, until reset by software
- ENGC** Enable “general call” (ACK address 0x00)
- SMBUS** 0 for I²C mode; 1 for SMBus mode

(other bits for packet error checking (PEC) or SMBus setup)



SRM32 I²C – control register 2

I2C_CR2

12	11	10	9	8	7	6	5	4	3	2	1	0	
LAST	DMA EN	ITBUF EN	ITEVT EN	ITERR EN	Reserved			FREQ[5:0]					
rw	rw	rw	rw	rw				rw	rw	rw	rw	rw	rw

FREQ[5:0] = peripheral clock frequency (in MHz)
allowed values [2MHz ... 42MHz]

DMA Control:

LAST: 1 = next DMA EOT is the last transfer

DMAEN: 1 = DMA requests when TxE=1 or RxNE=1

Interrupt Control (interrupt generation events on next slide)

ITBUFEN: 1 = TxE/RxNE event generates Event interrupt

ITEVTEN: 1 = Event interrupt enabled

ITERREN: 1 = Error interrupt enabled



STM32 I²C interrupts

Interrupt event	Event flag	Enable control bit
Start bit sent (Master)	SB	ITEVFEN
Address sent (Master) or Address matched (Slave)	ADDR	
10-bit header sent (Master)	ADD10	
Stop received (Slave)	STOPF	
Data byte transfer finished	BTF	
Receive buffer not empty	RxNE	ITEVFEN and ITBUFEN
Transmit buffer empty	TxE	
Bus error	BERR	ITERREN
Arbitration loss (Master)	ARLO	
Acknowledge failure	AF	
Overrun/Underrun	OVR	
PEC error	PECERR	
Timeout/Tlow error	TIMEOUT	
SMBus Alert	SMBALERT	



STM32 I²C – status register 1 (of two)

I2C_SR1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMB ALERT	TIME OUT	Res.	PEC ERR	OVR	AF	ARLO	BERR	TxE	RxNE	Res.	STOPF	ADD10	BTF	ADDR	SB
rc_w0	rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	r	r		r	r	r	r	r

ADDR: Master: 1= address sent

Slave: 1= received address matched OAR register or gen call

SB: Master: 1= Start generated (clear by reading SR1 & DR)

TxE: 1= transmitter buffer (DR) empty (can send a new byte)

RxNE: 1= receiver buffer (DR) not empty (byte has been received)

BTF: 1= data byte transfer finished successfully

RxNE=1 & DR not read yet; TxE=1 & DR not written yet

ARLO: 1= arbitration lost detected (this device lost to another)

STOPF: 1= slave detected stop condition after ACK

OVR: 1= DR register overrun/underrun (data lost)

AF: 1= ACK failure (no ACK returned)

BERR: 1= bus error (misplaced Start/Stop condition)

ADD10: 1= master sent 1st byte of 10-bit address



STM32 I²C – status register 2

I2C_SR2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PEC[7:0]								DUALF	SMB HOST	SMBDE FAULT	GEN CALL	Res.	TRA	BUSY	MSL
r	r	r	r	r	r	r	r	r	r	r	r		r	r	r

BUSY: 1= communication ongoing on the bus (cleared by Stop)

MSL: 0= slave mode (default)
1= master mode (START has been sent)

TRA: From R/W address bit:
1= data bytes to be TRAnsmitted
0= data bytes to be received

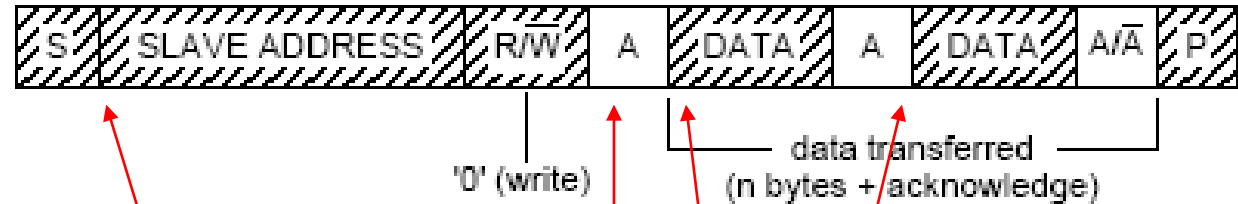
DUALF: Received address matches OAR1 (0) or OAR2 (1)

GENCALL: General call address (0x00) received when ENARP=1

(Other bits for PEC or SMBus)



STM32 I2C bus “events” (from flags)

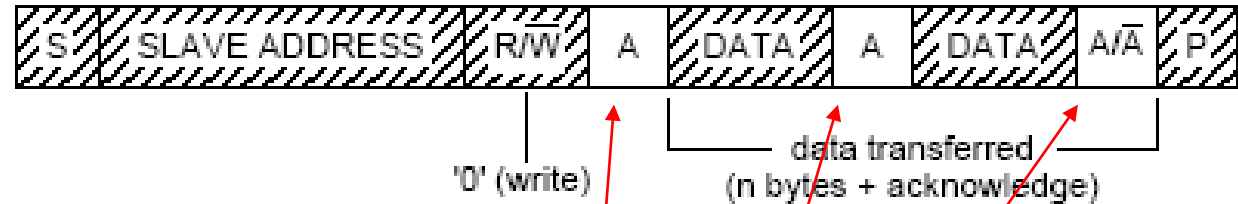


▶ Master modes:

- ▶ EV5: Start bit sent
 - ▶ **BUSY – MSL – SB**
- ▶ EV6: Slave acknowledged address
 - ▶ **BUSY – MSL – ADDR**
- ▶ EV8: DR ready for new byte to transmit
 - ▶ **BUSY – MSL – TXE** (transmit buffer empty)
- ▶ EV9: new byte received in the DR
 - ▶ **BUSY – MSL – RXNE** (receive buffer not empty)



STM32 I2C bus “events” (from flags)



▶ Slave modes:

- ▶ EV1: Own address received, data to be received from master
 - ▶ **BUSY – ADDR** (MSL=0, TRA=0)
- ▶ EV1: Own address received, data to be sent to master
 - ▶ **BUSY – ADDR – TRA** (MSL=0)
- ▶ EV2: Slave byte received
 - ▶ **BUSY – RNXE** (receive buffer not empty)
- ▶ EV3: Slave byte transmitted
 - ▶ **BUSY – TRA - TXE** (transmit buffer empty)
 - ▶ **BUSY – TRA – TXE - BTF** (transmit buffer empty and byte transfer finished)



I²C clock control register

I2C_CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
F/S	DUTY	Reserved			CCR[11:0]											
rw	rw				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

F/S 0=standard mode ($\leq 100\text{KHz}$), 1=fast mode ($\leq 400\text{KHz}$)

Standard Mode:

$$T_{\text{high}} = T_{\text{low}} = \text{CCR} * T_{\text{PCLK1}}$$

Fast Mode, DUTY = 0

$$T_{\text{high}} = T_{\text{low}} = \text{CCR} * T_{\text{PCLK1}}$$

Fast Mode, DUTY = 1 (to reach 400KHz)

$$T_{\text{high}} = 9 * \text{CCR} * T_{\text{PCLK1}}$$

$$T_{\text{low}} = 16 * \text{CCR} * T_{\text{PCLK1}}$$

Ex: To generate 100KHz SCL in standard mode.

If **FREQR** = 08, $T_{\text{PCLK1}} = 125\text{ns}$

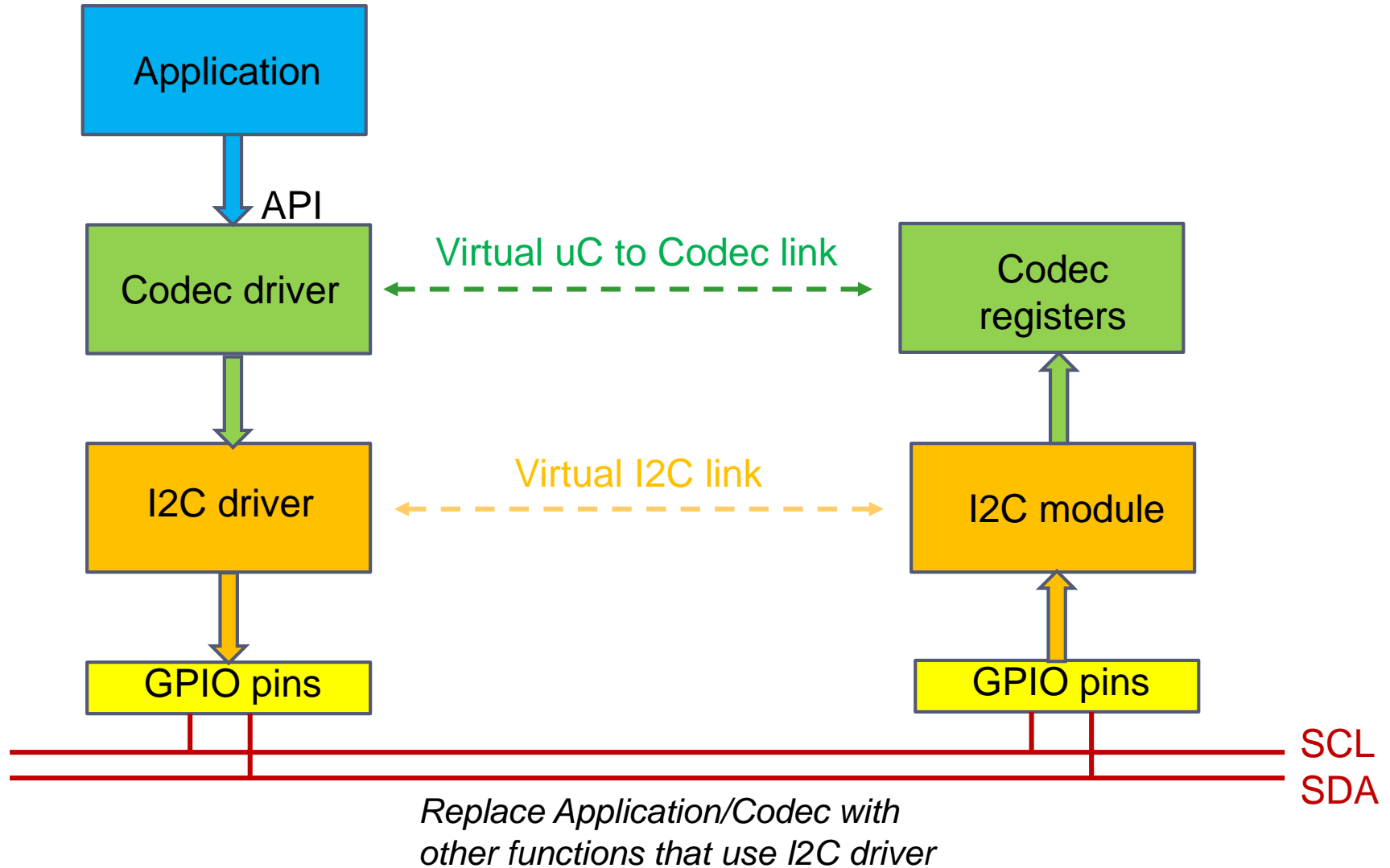
Set CCR = 40 (0x28)

$$T_{\text{high}} = T_{\text{low}} = 40 * 125\text{ns} = 5000\text{ns}$$

FREQR in CR2



Hierarchical/modular software design



STM32 I2C peripheral driver functions

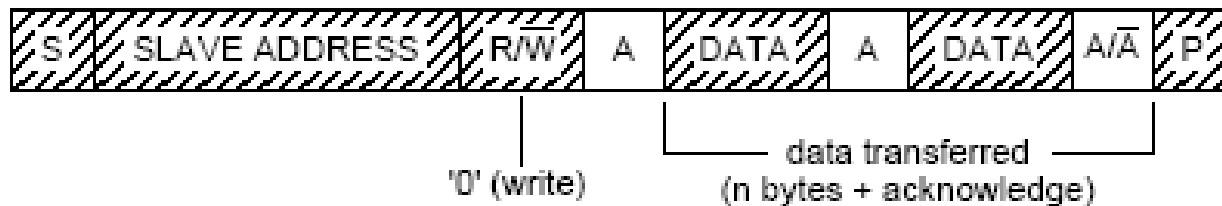
- ▶ **Configure control registers, etc.**
 - ▶ **I2C_Init()** – initialize control registers, clock, etc.
 - ▶ **I2C_Cmd()** – enable the I2C module
 - ▶ Other functions to set/clear individual control bits
- ▶ **Bus management functions**
 - ▶ **I2C_GenerateStart()** – signal START on the bus
 - ▶ **I2C_Send7bitAddress()** – send slave address
 - ▶ **I2C_GenerateStop()** - signal STOP on the bus
- ▶ **Data transfer functions**
 - ▶ **I2C_SendData()** – send one byte to DR
 - ▶ **I2C_ReceiveData()** – get one byte from DR
- ▶ **Bus monitoring functions**
 - ▶ **I2C_CheckEvent()** – test status flags for a bus “event”
 - ▶ **I2C_GetFlagStatus()** – test one flag in status register



Typical master-to-slave transfer

Codec_WriteRegister(RegAddr,RegValue)

- ▶ **I2C_GetFlagStatus()** – check flag BUSY=0
- ▶ **I2C_GenerateStart()** – signal START on the bus
- ▶ **I2C_CheckEvent()** – test EV5 flags (start correct)
- ▶ **I2C_Send7bitAddress()** – send slave address
- ▶ **I2C_CheckEvent()** – test EV6 flags (slave address ACK)
- ▶ **I2C_SendData()** – send first byte (register address) to DR
- ▶ **I2C_CheckEvent()** – test EV8 flags (data sending, DR ready for byte)
- ▶ **I2C_SendData()** – send second byte (register value) to DR
- ▶ **I2C_GetFlagStatus()** – check flag BTF=1 (byte transfer finished)
- ▶ **I2C_GenerateStop()** - signal STOP on the bus



Audio Code driver: key functions

- ▶ **Codec_Init()** – all related device/module initialization:
 - ▶ **Codec_GPIO_Init()**
 - ▶ Enable clocks in RCC and all GPIO pins for I2C, I2S, DAC
 - ▶ **Codec_Reset()** – reset the Codec (RESET pin)
 - ▶ **Codec_CtrlInterface_Init()**
 - ▶ Calls I2C_Init() with required parameters
 - ▶ Configure all Codec registers via I2C functions
 - ▶ **Codec_AudioInterfaceInit()**
 - ▶ Initialize DAC and I2S modules
- ▶ **Codec_WriteRegister()** – write value to a code register
- ▶ **Codec_ReadRegister()** – read value from a code register

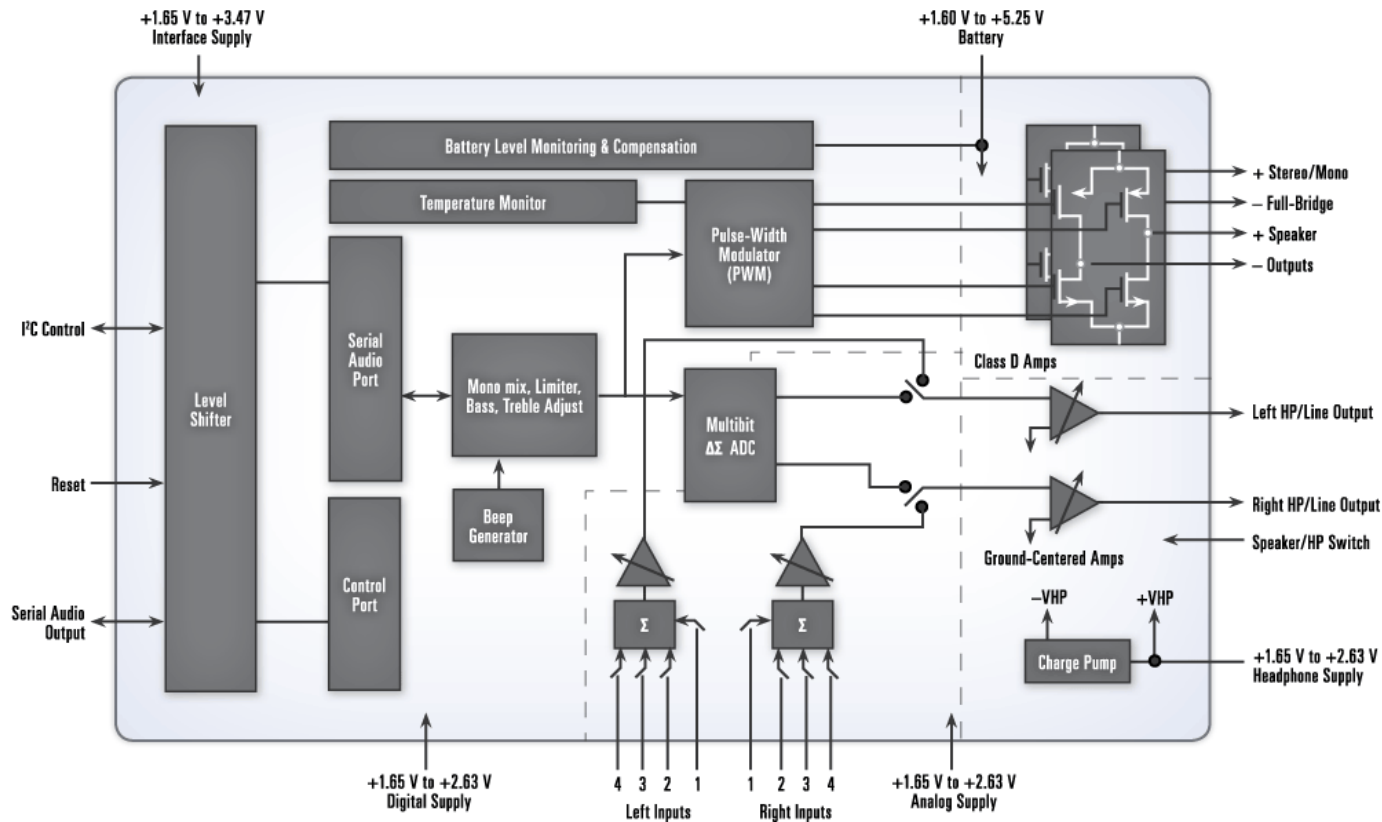


STM32F4-Discovery Software

- ▶ STM32F407VG peripheral drivers added to project from “Pack” (I2C, SPI, DAC, etc.)
 - ▶ *stm32f4xx_i2c.c* => *all I2C control/access functions*
- ▶ Discovery board chip drivers in
 - ▶ `..\stm32f4discovery_fw\Utilities\STM32F4-Discovery\stm32f4_discovery_audio_codec.c`
 - ▶ Initialize and control audio codec chip
 - ▶ Calls functions from I2C, I2S, GPIO, RCC module drivers



Cirrus Logic CS43L22 Portable Audio DAC with Integrated Class D Speaker Driver



Discovery CS43L22 schematic

