CAN II

ECE 3710

A clear conscience is usually the sign of a bad memory.

- Steven Wright

CAN

data is broadcast and devices
decide if they need to respond

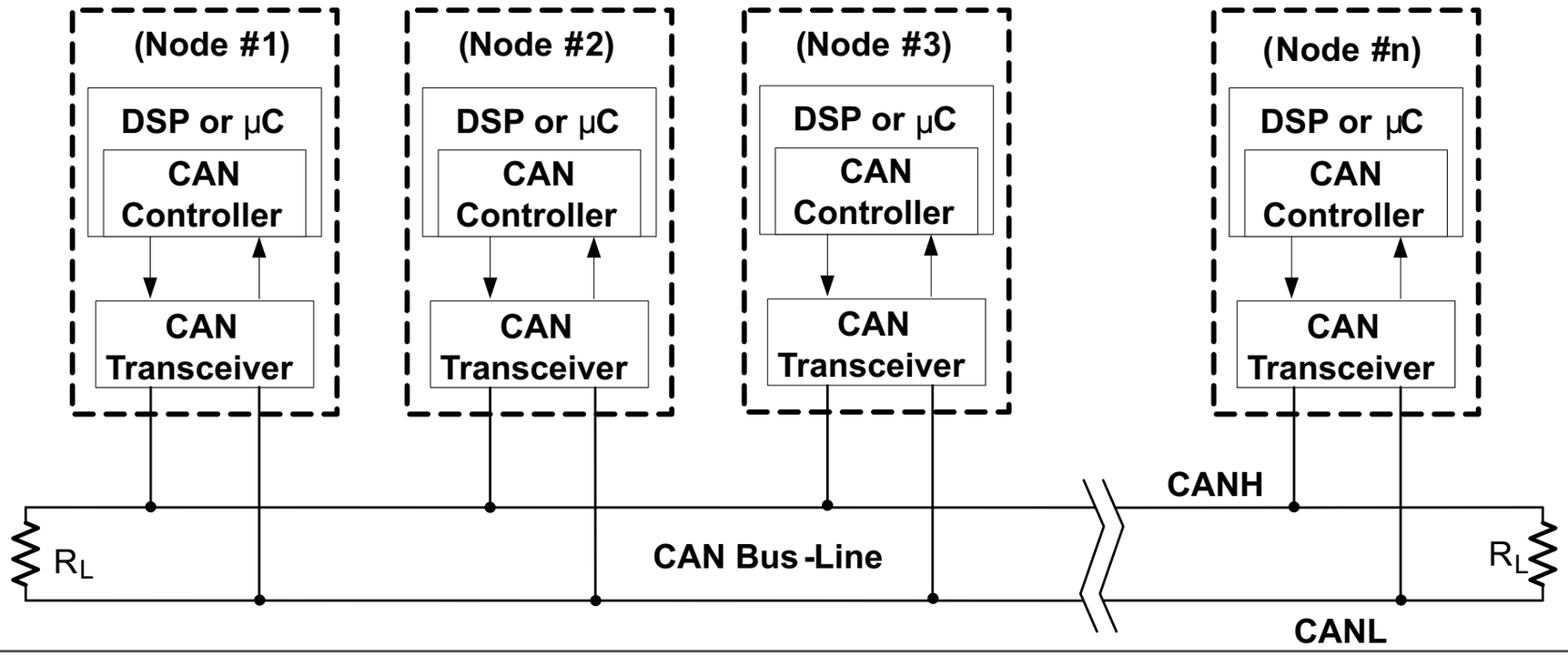
features:

1. no addresses
2. priorities

more important data
sent first

3. multiple access w/o central authority

each device monitors line, stops transmitting
if higher priority data



two or one wire

(if short,
no termination)

CAN data frame

(three others)

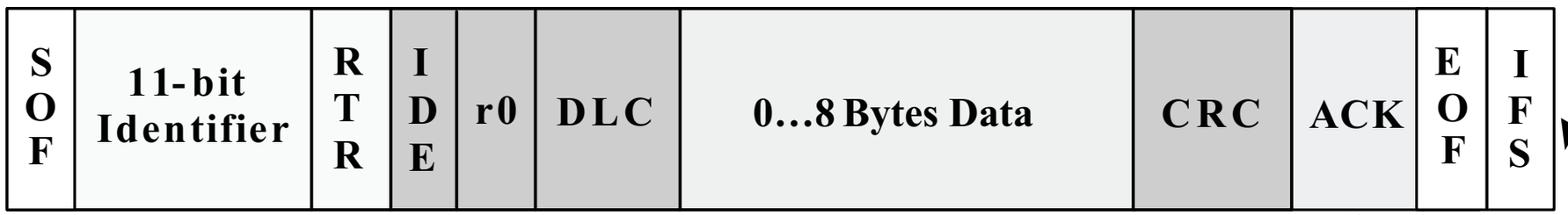
1. what is data about

2. priority

(lower is higher priority)

standard or extended frame

checksum for error detection



start of frame

request data from another device

unused bit

how many bytes of data follow

ack: next slide

end of frame

delay until next frame

NXP 1778 CAN

overview:

1. multiple TX buffers with priorities
2. interrupts on RX/TX
3. acceptance filters

to TX:

1. provide ID
2. data
3. number of bytes to TX

NXP 1778 CAN: acceptance filters

problem:

1. receive every message sent
2. messages cause interrupts
3. don't care about every message
   (only certain IDs)
4. unnecessary interrupts

solution

acceptance filter:

if ID isn't in list, then no interrupt triggered
(as though message wasn't received)

ex: NXP 1778 CAN

each time message w/certain ID received, increment data byte and resend

```
SIGNAL void addCAN (void)  {
  // get things started by sending 0x00 with ID=0xAD
  CAN1ID = 0xAD;                 // CAN message ID
  CAN1L  = 1;                    // message length 1 byte
  CAN1B0 = 0x00;                 // message data byte 0
  CAN1IN = 1;                    // send CAN message with 11-bit ID

  // wait for response
  while (1)  {
    swatch(0.1);
    if(CAN1OUT == 1)  {
      printf("\nReceived msg");
      if(CAN1ID==0xAD)  {
        CAN1ID = 0xAD;           // CAN message ID
        CAN1L  = 1;              // message length 1 bytes
        CAN1B0 = CAN1B0 + 1;     // message data byte 0
        CAN1IN = 1;              // send CAN message with 11-bit ID
      }
    }
  }
}
```

keil CAN interface

(virtual registers)

1. send out 0x00 w/ID=0xAD
2. add one to response and resend w/ID=0xAD

ex: NXP 1778 CAN

```
void CANInit()
{
  // 1. power-on CAN1
  PCONP[0x1] = PCONP[0x1] | 0x20;

  // 4. pin config: CAN functionality, otherwise defaults
  IOCON_P0_21[0x0] = 0x4; //0x4=0b100
  IOCON_P0_22[0x0] = 0x4; //0x4=0b100

  // 5. enable interrupt for RX
  CAN1[0x10] = 0x1; //CAN1 interrupt enable
  ISER0[0x3] = ISER0[0x3] | 0x2; //NVIC interrupt enable for CAN

  // 6. setup acceptance filters
  // 6a. set bypass mode so we can setup filters
  AFC[0x0] = 0x3;
  // 6a. table start and stop addresses (one message ID accepted)
  AFC[0x4] = 0x0; //enable standard, individual
  AFC[0x14] = 0x4; //end of filter addr: bug in simulator,
                   //should be left shifted by two
  AFC[0x8] = AFC[0x14]; //disable standard,group
  AFC[0xC] = AFC[0x14]; //disable extended,individual
  AFC[0x10] = AFC[0x14]; //disable extended,group

  // 6b. define filters
  AF[0x0] = 0xAD;
  // 6c. enable filters and rx
  AFC[0x0] = 0x0;

  // 7. enable CAN1
  CAN1[0x0] = 0x0;
}
```

logic behind steps is the same as your board/TI board

ex: NXP 1778 CAN

```
void CAN_IRQHandler()
{
  // ack IRQ (by releasing receiver buffer)
  CAN1[0x4] = CAN1[0x4] | 0x4;

  // get message bytes and id
  ID = CAN1[0x24];
  MSG1 = CAN1[0x28];

  // based on ID, send response
  if(ID==0xAD)
    {
        CAN1[0x32] = 0x1; //want to tx one byte (set DLC)
        CAN1[0x34] = 0xAD; //ID
        CAN1[0x38] = MSG1+1; //message
        CAN1[0x4] = CAN1[0x4] | 0x21; //transmit buffer one
    }
}
```

let peripheral do most of the work

DMA

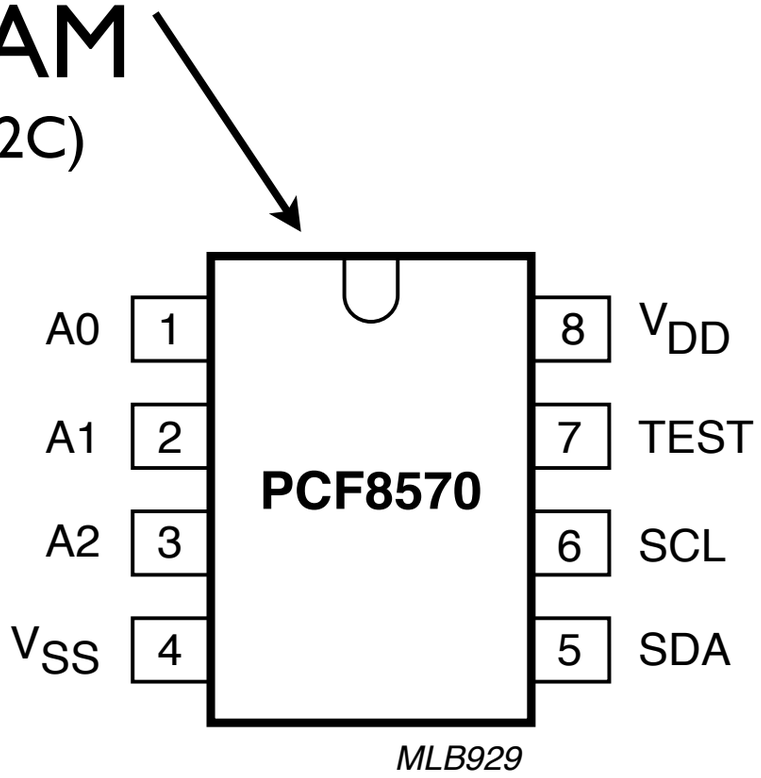
ECE 3710

The early bird gets the worm, but the second mouse gets the cheese.

- Steven Wright

LM3S1968 burst TX example

256x8-bit RAM (I2C)



MLB929

to read memory:

1. specify address (master TX)
2. get byte(s) (master RX)

to write memory:

1. specify address (master TX)
2. send byte(s) (master TX)

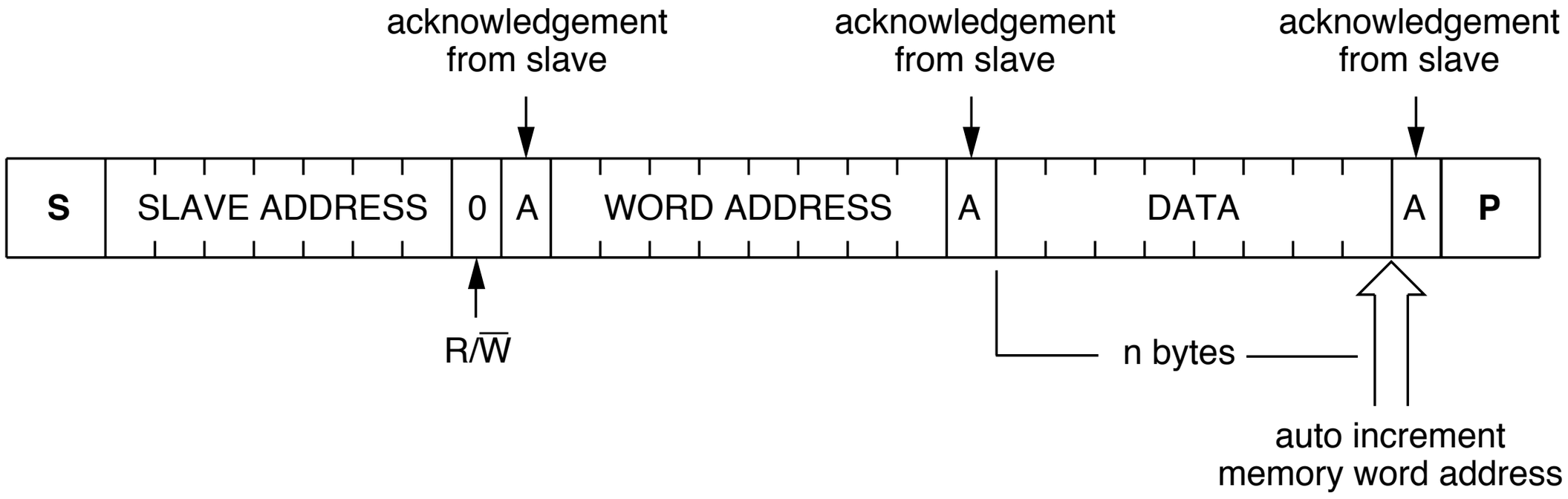
multiple RX/TX need to read/write to device

LM3S1968 burst TX example

requires burst mode

(master doesn't give up line
or issue STOP/START)

write (master TX/TX):



last addr specified + 1
remembered by device

(remembers current addr for a special no-addr read)

STM32F103xx I2C

unlike TI, no built-in TX/RX modes (have to handle all protocol aspects ourselves)

```
I2C_MEM_WR
  ; this routine performs write procedure for
  ;  writing to PCF8570-style RAM
  ; R9: src of bytes to write
  ; R10: # bytes to write
  ; R11: destination of byte
  ; R12: slave addr

  ; 0. save return addr
  push {LR}

  ; 1. send start bit
  ldr R1,=I2C1
  mov R0,#0x101
  str R0,[R1,#0x0]
  ; wait until start bit has been sent
  bl I2C1_POLL_SB

  ; 2. send slave addr and direction
  ;    (lsb=0 for write)
  lsl R0,R12,#0x1
  str R0,[R1,#0x10]
  ; wait until addr has been sent
  bl I2C1_POLL_ADDR
  ; to clear ADDR bit, we read SR1
  ldr R0,[R1,#0x18]
```

```
I2C1_POLL_SB
  ; R0: tmp
  ; R1: base addr of I2C1
  ldr R0,[R1,#0x14] ;status register
  ands R0,#0x1 ;(Z=1 if result is zero)
  beq I2C1_POLL_SB ;(branch if Z=1)
  bx LR
```

```
I2C1_POLL_ADDR
  ; R0: tmp
  ; R1: base addr of I2C1
  ldr R0,[R1,#0x14] ;status register
  ands R0,#0x2 ;(Z=1 if result is zero)
  beq I2C1_POLL_ADDR ;(branch if Z=1)
  bx LR
```

STM32F103xx I2C

```
  ; 3. tx addr of data
  str R11,[R1,#0x10]
  ; wait for data to finish transmitting
  bl I2C1_POLL_TXE

  ; 4. tx data
  mov R2,#0 ;our counter
TX
  cmp R2,R10
  beq Stop ;have sent R10 bytes
  ldr R0,[R9,R2] ;get next byte to send
  str R0,[R1,#0x10] ;tx byte
  ; wait for data to finish transmitting
  bl I2C1_POLL_TXE
  add R2,#1 ;increment counter
  b TX

  ; 5. now send stop bit
Stop
  mov R0,#0x201
  str R0,[R1,#0x0]

  ; 6. we're done: restore LR and return to main
  pop {LR}
  bx LR
```

TxE flag set on ACK

```
I2C1_POLL_TXE
  ; R0: tmp
  ; R1: base addr of I2C1
  ldr R0,[R1,#0x14] ;status register
  ands R0,#0x80 ;(Z=1 if result is zero
  beq I2C1_POLL_TXE ;(branch if Z=1)
  bx LR
```

Tx until all bytes sent

```
; R9: src of bytes to write
; R10: # bytes to write
; R11: destination of byte
; R12: slave addr
```

```
; 4. tx data
  mov R2,#0 ;our counter
TX
  cmp R2,R10
  beq Stop ;have sent R10 bytes
  ldr R0,[R9,R2] ;get next byte to send
  str R0,[R1,#0x10] ;tx byte
  ; wait for data to finish transmitting
  bl I2C1_POLL_TXE
  add R2,#1 ;increment counter
  b TX
```

steps:

1. get data from memory
2. store data in memory

inefficient

(lowers throughput)

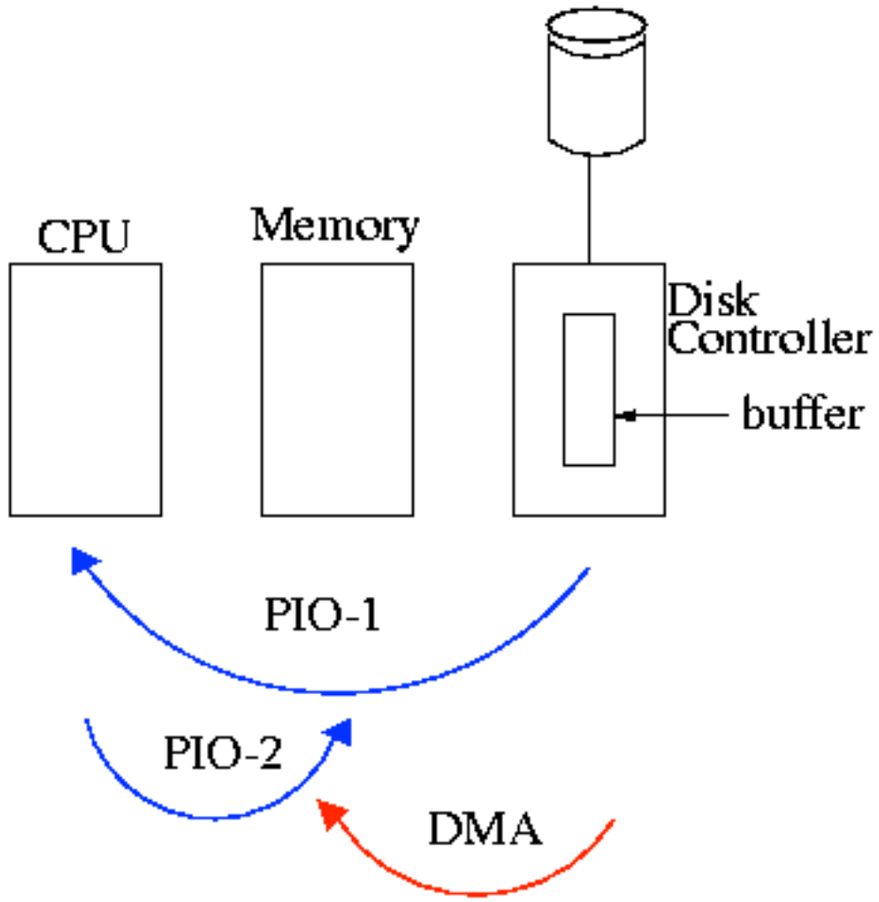
additional overhead: counter and check

peripherals waiting
on data:

given:

1. many devices
2. I/O device nearly as fast as bus

direct memory access (DMA)



Programmed I/O vs
Direct Memory Access

removes:

1. processor from
2. unnecessary steps

(memory->cpu->memory)

why have proc handle moving data around?

a simple task:

1. *src*
2. *dst*
3. *n*

direct memory access (DMA)

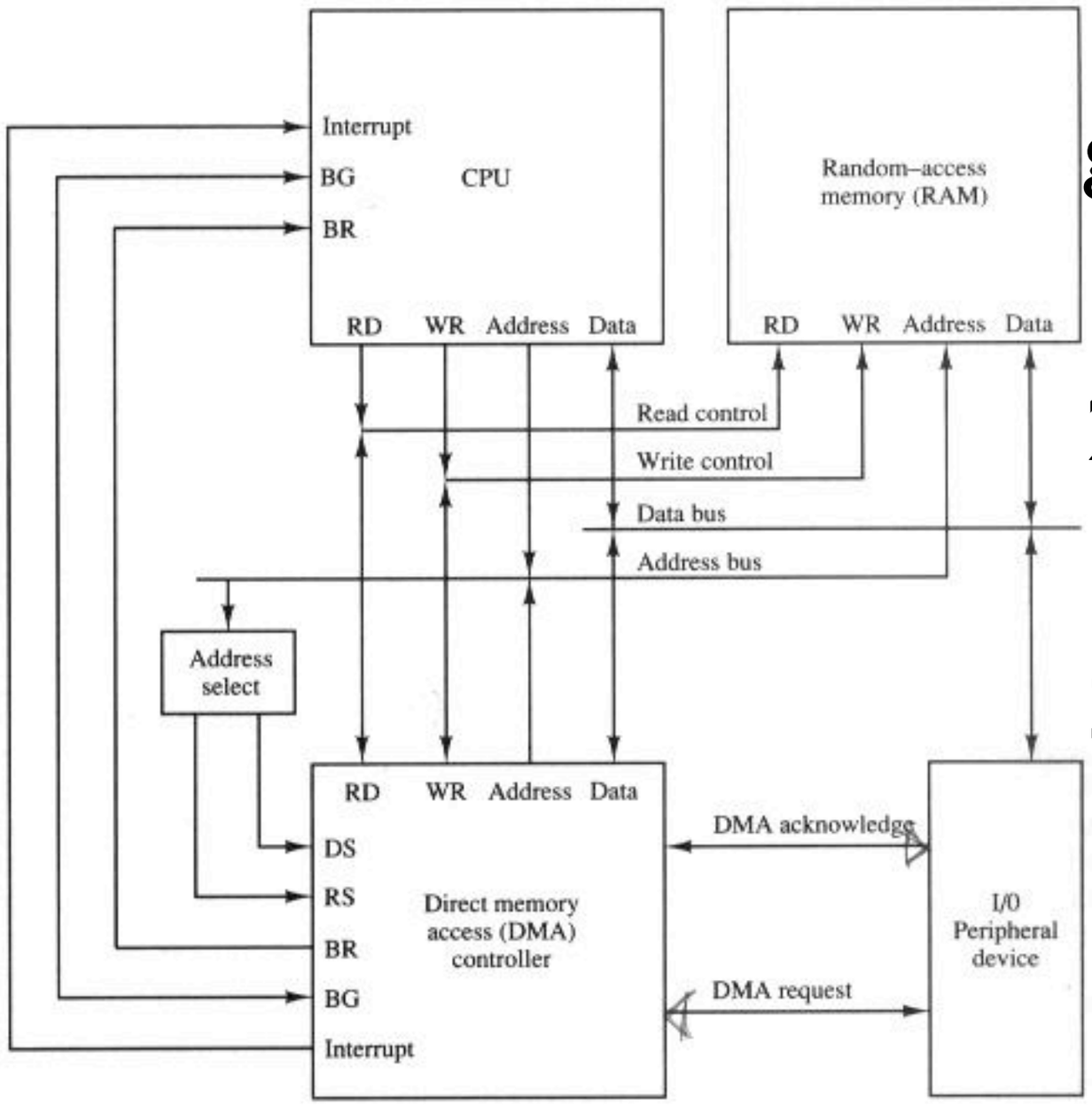


Figure 11-18 DMA transfer in a computer system.

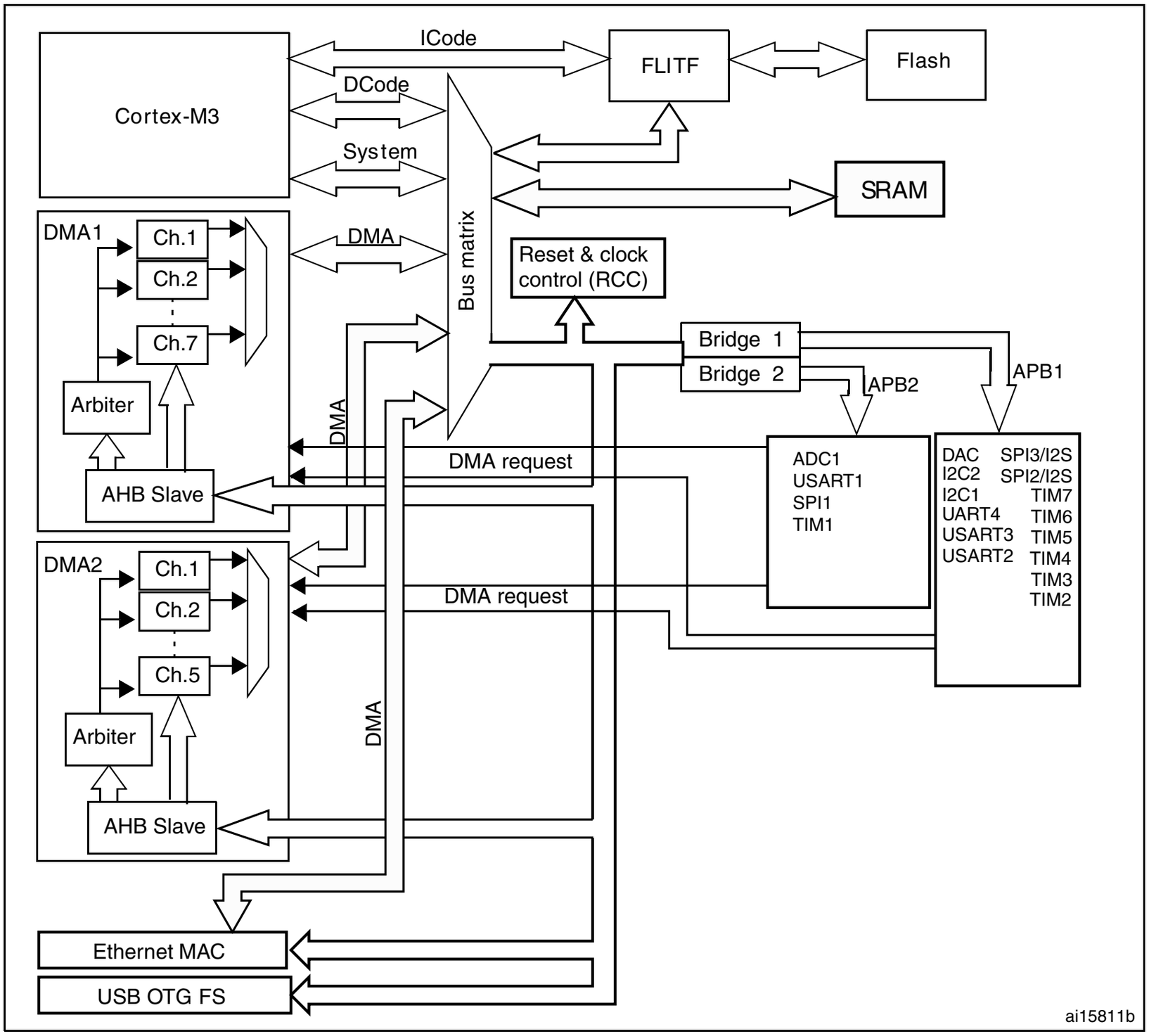
BG: bus grant
BR: bus request
DS: device select
RS: register select

generic steps:

1. CPU makes DMA request
   (src, dst, n)
2. DMA controller moves data around
   (may prevent CPU from access to data/addr bus)
3. controller notifies CPU that data transfer is complete
   (polling or interrupt)

src/dst: memory or peripherals

STM32F103xx DMA Controller



Q:

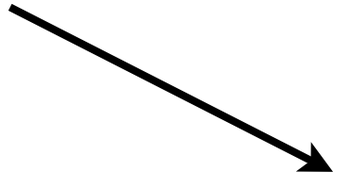
1. how many DMA requests
2. which peripherals

STM32F103xx DMA Controller

1. up-to twelve requests w/priorities
2. requests: hardware (from device)
or software (your code)
3. src/dst: mem2per, per2mem,
mem2mem (use in conjunction w/external memory mapping), and per2per
4: auto-increment of src/dst address (based on data size[8,16,32])
or circular (put new data in same place)
5. interrupts (or polling): transfer complete/half-complete and
error

STM32F103xx DMA Controller

memory (32-bits) to 8-bit device: data is truncated



Source port width	Destination port width	Number of data items to transfer (NDT)	Source content: address / data	Transfer operations	Destination content: address / data
8	8	4	@0x0 / B0 @0x1 / B1 @0x2 / B2 @0x3 / B3	1: READ B0[7:0] @0x0 then WRITE B0[7:0] @0x0 2: READ B1[7:0] @0x1 then WRITE B1[7:0] @0x1 3: READ B2[7:0] @0x2 then WRITE B2[7:0] @0x2 4: READ B3[7:0] @0x3 then WRITE B3[7:0] @0x3	@0x0 / B0 @0x1 / B1 @0x2 / B2 @0x3 / B3
8	16	4	@0x0 / B0 @0x1 / B1 @0x2 / B2 @0x3 / B3	1: READ B0[7:0] @0x0 then WRITE 00B0[15:0] @0x0 2: READ B1[7:0] @0x1 then WRITE 00B1[15:0] @0x2 3: READ B3[7:0] @0x2 then WRITE 00B2[15:0] @0x4 4: READ B4[7:0] @0x3 then WRITE 00B3[15:0] @0x6	@0x0 / 00B0 @0x2 / 00B1 @0x4 / 00B2 @0x6 / 00B3
8	32	4	@0x0 / B0 @0x1 / B1 @0x2 / B2 @0x3 / B3	1: READ B0[7:0] @0x0 then WRITE 000000B0[31:0] @0x0 2: READ B1[7:0] @0x1 then WRITE 000000B1[31:0] @0x4 3: READ B3[7:0] @0x2 then WRITE 000000B2[31:0] @0x8 4: READ B4[7:0] @0x3 then WRITE 000000B3[31:0] @0xC	@0x0 / 000000B0 @0x4 / 000000B1 @0x8 / 000000B2 @0xC / 000000B3
16	8	4	@0x0 / B1B0 @0x2 / B3B2 @0x4 / B5B4 @0x6 / B7B6	1: READ B1B0[15:0] @0x0 then WRITE B0[7:0] @0x0 2: READ B3B2[15:0] @0x2 then WRITE B2[7:0] @0x1 3: READ B5B4[15:0] @0x4 then WRITE B4[7:0] @0x2 4: READ B7B6[15:0] @0x6 then WRITE B6[7:0] @0x3	@0x0 / B0 @0x1 / B2 @0x2 / B4 @0x3 / B6
16	16	4	@0x0 / B1B0 @0x2 / B3B2 @0x4 / B5B4 @0x6 / B7B6	1: READ B1B0[15:0] @0x0 then WRITE B1B0[15:0] @0x0 2: READ B3B2[15:0] @0x2 then WRITE B3B2[15:0] @0x2 3: READ B5B4[15:0] @0x4 then WRITE B5B4[15:0] @0x4 4: READ B7B6[15:0] @0x6 then WRITE B7B6[15:0] @0x6	@0x0 / B1B0 @0x2 / B3B2 @0x4 / B5B4 @0x6 / B7B6
16	32	4	@0x0 / B1B0 @0x2 / B3B2 @0x4 / B5B4 @0x6 / B7B6	1: READ B1B0[15:0] @0x0 then WRITE 0000B1B0[31:0] @0x0 2: READ B3B2[15:0] @0x2 then WRITE 0000B3B2[31:0] @0x4 3: READ B5B4[15:0] @0x4 then WRITE 0000B5B4[31:0] @0x8 4: READ B7B6[15:0] @0x6 then WRITE 0000B7B6[31:0] @0xC	@0x0 / 0000B1B0 @0x4 / 0000B3B2 @0x8 / 0000B5B4 @0xC / 0000B7B6
32	8	4	@0x0 / B3B2B1B0 @0x4 / B7B6B5B4 @0x8 / BBBAB9B8 @0xC / BFBEBDBC	1: READ B3B2B1B0[31:0] @0x0 then WRITE B0[7:0] @0x0 2: READ B7B6B5B4[31:0] @0x4 then WRITE B4[7:0] @0x1 3: READ BBBAB9B8[31:0] @0x8 then WRITE B8[7:0] @0x2 4: READ BFBEBDBC[31:0] @0xC then WRITE BC[7:0] @0x3	@0x0 / B0 @0x1 / B4 @0x2 / B8 @0x3 / BC
32	16	4	@0x0 / B3B2B1B0 @0x4 / B7B6B5B4 @0x8 / BBBAB9B8 @0xC / BFBEBDBC	1: READ B3B2B1B0[31:0] @0x0 then WRITE B1B0[7:0] @0x0 2: READ B7B6B5B4[31:0] @0x4 then WRITE B5B4[7:0] @0x1 3: READ BBBAB9B8[31:0] @0x8 then WRITE B9B8[7:0] @0x2 4: READ BFBEBDBC[31:0] @0xC then WRITE BDBC[7:0] @0x3	@0x0 / B1B0 @0x2 / B5B4 @0x4 / B9B8 @0x6 / BDBC
32	32	4	@0x0 / B3B2B1B0 @0x4 / B7B6B5B4 @0x8 / BBBAB9B8 @0xC / BFBEBDBC	1: READ B3B2B1B0[31:0] @0x0 then WRITE B3B2B1B0[31:0] @0x0 2: READ B7B6B5B4[31:0] @0x4 then WRITE B7B6B5B4[31:0] @0x4 3: READ BBBAB9B8[31:0] @0x8 then WRITE BBBAB9B8[31:0] @0x8 4: READ BFBEBDBC[31:0] @0xC then WRITE BFBEBDBC[31:0] @0xC	@0x0 / B3B2B1B0 @0x4 / B7B6B5B4 @0x8 / BBBAB9B8 @0xC / BFBEBDBC

STM32F103xx DMA Controller

configuration (mem2per):

0a. enable peripheral clock

0b. determine channel peripheral is connected to

Table 78, p273

1. set peripheral addr (dst)

CPARx, p279

2. set memory addr (src)

CMARx, p279

3. set n (amount to transfer)

CNDTRx, p278

4--9. set channel priority, mem and per size, mem and per increment, TX direction, interrupts, enable

CCRx, p277--8

have to calc offset for config registers

(0xXX + 20*[chan-1])

DMA config reg offset (1.--3.)

note: peripheral must be configured to use DMA

STM32F103xx I2C+DMA

(enable DMA for I2C)

26.6.2 I²C Control register 2 (I2C_CR2)

Address offset: 0x04
Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			LAST	DMA EN	ITBUF EN	ITEVT EN	ITERR EN	Reserved		FREQ[5:0]					
			rw	rw	rw	rw	rw			rw	rw	rw	rw	rw	rw

Bits 15:13 Reserved, must be kept at reset value

Bit 12 **LAST**: DMA last transfer

0: Next DMA EOT is not the last transfer
1: Next DMA EOT is the last transfer

Note: This bit is used in master receiver mode to permit the generation of a NACK on the last received data.

Bit 11 **DMAEN**: DMA requests enable

0: DMA requests disabled
1: DMA request enabled when TxE=1 or RxNE =1

p746

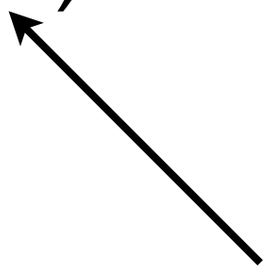
notes:

1. I2C module makes request only on data transfer
2. DMA controller does not handle start,stop, or addr

ex: DMA write to I2C memory

Q:

what is peripheral addr (dst)?



src/n is whatever we want

ex: DMA write to I2C memory

(set DMA dst for I2C)

Q:

what is peripheral addr (dst)?

A:

I2C data register

26.6.5 I²C Data register (I2C_DR)

Address offset: 0x10
Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								DR[7:0]							
								rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:8 Reserved, must be kept at reset value

Bits 7:0 **DR[7:0]** 8-bit data register

Byte received or to be transmitted to the bus.

–Transmitter mode: Byte transmission starts automatically when a byte is written in the DR register. A continuous transmit stream can be maintained if the next data to be transmitted is put in DR once the transmission is started (TxE=1)

–Receiver mode: Received byte is copied into DR (RxNE=1). A continuous transmit stream can be maintained if DR is read before the next data byte is received (RxNE=1).

Note: In slave mode, the address is not copied into DR.

Note: Write collision is not managed (DR can be written if TxE=0).

Note: If an ARLO event occurs on ACK pulse, the received byte is not copied into DR and so cannot be read.

p749

remember: DMA just puts data in a location (up to peripheral to use it)

ex: DMA write to I2C memory

```
; DMA1 init
; 1. enable dma1 clock: bit 0
ldr R1,=SYSCTL
mov R0,#0x1
str R0,[R1,#0x14]

; NOTE: I2C TX uses channel six
; 2. set peripheral address (offset = 0x10+0d20*(6-1)=0x74)
;    dma1 should write to i2c data register
ldr R1,=I2C1
add R0,R1,#0x10 ;i2c1 data register
ldr R1,=DMA1
str R0,[R1,#0x74]

; 3. set the memory address (offset = 0x14+0d20*(6-1)=0x78)
mov R9,#0x0 ;data src
str R9,[R1,#0x78]

; 4. number of bytes to transfer (offset = 0xC+0d20*(6-1)=0x70)
mov R10,#0x4 ;#bytes
str R10,[R1,#0x70]

; 5. channel priority

; 6. configuration (offset=0x8+0d20*(6-1)=0x6C)
;    8-bits for memory and peripheral, memory increment, read from memory, enable channel
mov R0,#0x91 ;0x91=0b10010001
str R0,[R1,#0x6C]

; set i2c options (not handled by dma)
mov R11,#0x0 ;dest of data
mov R12,#0x2A ;slave addr

; TX DATA TO MEMORY
bl I2C_MEM_WR
```

2. where DMA writes to (dst)

3. where DMA reads from (src)

4. how much data DMA reads (n)

ex: DMA write to I2C memory

```
I2C_MEM_WR
  ; this routine performs DMA write procedure for
  ;  writing to PCF8570-style RAM
  ; R11: destination of byte
  ; R12: slave addr

  ; 0. save return addr
  push {LR}

  ; 1. send start bit
  ldr R1,=I2C1
  mov R0,#0x101
  str R0,[R1,#0x0]
  ; wait until start bit has been sent
  bl I2C1_POLL_SB

  ; 2. send slave addr and direction (lsb=0 for write)
  lsl R0,R12,#0x1
  str R0,[R1,#0x10]
  ; wait until addr has been sent
  bl I2C1_POLL_ADDR
  ; to clear ADDR bit, we read SR1
  ldr R0,[R1,#0x18]

  ; 3. tx addr of data
  str R11,[R1,#0x10]
  ; wait for data to finish transmitting
  bl I2C1_POLL_TXE
```

because controller doesn't handle start/stop/addr this remains the same

DMA only enabled after we send start bit and addr

ex: DMA write to I2C memory

```
; 4. enable dma for i2c1 peripheral
mov R0,#0x8
strb R0,[R1,#0x5]
; wait for data to finish transmitting
ldr R3,=DMA1
bl DMA1_POLL_TCIF
; disable dma channel six
mov R0,#0x0
str R0,[R3,#0x6C]

; 5. disable dma for i2c1 peripheral
mov R0,#0
strb R0,[R1,#0x5]

  ; 6. now send stop bit
  ; wait for last dma byte to finish
transmitting
  bl I2C1_POLL_TXE
  mov R0,#0x201
  str R0,[R1,#0x0]

  ; 7. we're done: restore LR and
  ;    return to main
  pop {LR}
  bx LR
```

```
DMA1_POLL_TCIF
  ; R0: tmp
  ; R3: base addr of DMA1
  ldr R0,[R3,#0x2] ;status register
  ands R0,#0x20 ;(Z=1 if result is zero)
  beq DMA1_POLL_TCIF ;(branch if Z=1)
  ; clear dma channel six bits
  mov R0,#0x10
  str R0,[R1,#0x6]
  bx LR
```

ISR & IFCR, p275--6

note: not necessary to disable DMA for I2C device; but only enable DMA channel when we want DMA-controlled transfer