ARM AMBA

13103518 - CESAR EDUARDO CORREA
16200639 - DANIEL DE SOUZA BAULÉ
16100751 - VINÍCIUS SCHWINDEN BERKENBROCK

ARM AMBA (Advanced Microcontroller Bus Architecture)

- An open standard, on-chip interconnect specification for the connection and management of functional blocks in a System-on-Chip (SoC).
- Facilitates right-first-time development of multi-processor designs with large numbers of controllers and peripherals.
- Promotes design re-use by defining common interface standards for SoC modules

ARM.com

ARM AMBA
(Advanced Microcontroller Bus Architecture)

- An architecture that is widely used in system-on-chip designs, which are found on chip buses.
- The AMBA specification standard is used for designing high-level embedded microcontrollers.
- AMBA's major objective is to provide technology independence and to encourage modular system design.
- It strongly encourages the development of reusable peripheral devices while minimizing silicon infrastructure.

techopedia.com

ARM AMBA (Advanced Microcontroller Bus Architecture)

- It’s the interface(s) everyone uses to bolt blocks together in their chip.

Advanced System Bus	ASB	Now obsolete, so don’t worry about this one!
Advanced Peripheral Bus	APB	Simple, easy, for your peripherals
Advanced High-Performance Bus	AHB	Now used a lot in Cortex-M designs
Advanced eXtensible Interface	AXI	The most widespread, now up to AXI4
Advanced Trace Bus	ATB	For moving trace data around the chip, see CoreSight
AXI Coherency Extensions	ACE	Used in big.LITTLE systems for smartphones, tablets, etc.
Coherent Hub Interface	CHI	The highest performance, used in networks and servers

Ben Walshe

ARM AMBA
(Advanced Microcontroller Bus Architecture)

- It’s the interface(s) everyone uses to bolt blocks together in their chip.

~~Advanced System Bus~~	~~ASB~~	~~Now obsolete, so don’t worry about this one!~~
Advanced Peripheral Bus	APB	Simple, easy, for your peripherals
Advanced High-Performance Bus	AHB	Now used a lot in Cortex-M designs
~~Advanced eXtensible Interface~~	~~AXI~~	~~The most widespread, now up to AXI4~~
Advanced Trace Bus	ATB	For moving trace data around the chip, see CoreSight
~~AXI Coherency Extensions~~	~~ACE~~	~~Used in big.LITTLE systems for smartphones, tablets, etc.~~
~~Coherent Hub Interface~~	~~CHI~~	~~The highest performance, used in networks and servers~~

Ben Walshe

ARM AMBA (Advanced Microcontroller Bus Architecture)

- It's the interface(s) everyone uses to bolt blocks together in their chip.

Advanced Peripheral Bus	APB	Simple, easy, for your peripherals
Advanced High-Performance Bus	AHB	Now used a lot in Cortex-M designs
Advanced Trace Bus	ATB	For moving trace data around the chip, see CoreSight

Ben Walshe

ARM AMBA

A BRIEF HISTORY

ARM AMBA

- 1995 – ARM received EU funding;
- 1996 – ARM introduces the Advanced Microcontroller Bus Architecture as an open architecture;
  - It facilitates development of multiprocessor designs with large numbers of controllers and peripherals
- Since its inception, the scope of AMBA has, despite its name, gone far beyond microcontroller devices.
- Today, AMBA is widely used on a range of ASIC and SoC parts including applications processors which are typically found in modern portable mobile devices like smartphones.

ARM AMBA

- An important aspect of a SoC is not only which components or blocks it houses, but also how they interconnect.
- AMBA served as a solution for how the blocks would interface with each other.
- It soon became the 'de facto' standard interface for anyone which to bring a controller or a peripheral IP block to market.

ARM AMBA

- AMBA
  - ASB - Advanced System Bus;
  - APB - Advanced Peripheral Bus;
- AMBA 2
  - AHB - High-performance Bus;
- AMBA 3 (2003)
  - AXI - Advanced eXtensible Interface;
  - ATB - Advanced Trace Bus

ARM AMBA

- AMBA 4 (2010)
  - AXI4;
  - ACE - AXI Coherency Extensions;
- AMBA 5 (2013)
  - CHI - Coherent Hub Interface

AMBA 3 APB

ADVANCED PERIPHERAL BUS

AMBA3 APB

- It provides a low-cost interface that is optimized for minimal power consumption and reduced interface complexity.
- The APB interfaces to any peripherals that are low-bandwidth and do not require the high performance of a pipelined bus interface.
- All signal transitions are only related to the rising edge of the clock to enable the integration of APB peripherals easily into any design flow.
- Every transfer takes at least two cycles.

AMBA3 APB

- The APB can interface with:
  - AMBA Advanced High-performance Bus Lite (AHB-Lite);
  - AMBA Advanced Extensible Interface (AXI);
- You can use it to provide access to the programmable control registers of peripheral devices.

AMBA3 APB – Changes from previous specification

- A ready signal, PREADY, to extend an APB transfer.
- An error signal, PSLVERR, to indicate the failure of a transfer.

AMBA 3 APB signals

Signal	Source	Description
PCLK	Clock source	Clock. The rising edge of **PCLK** times all transfers on the APB.
PRESETn	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.

AMBA 3 APB signals

Signal	Source	Description
PADDR	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
PSELx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a **PSELx** signal for each slave.
PENABLE	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when **PWRITE** is HIGH. This bus can be up to 32 bits wide.

AMBA 3 APB signals

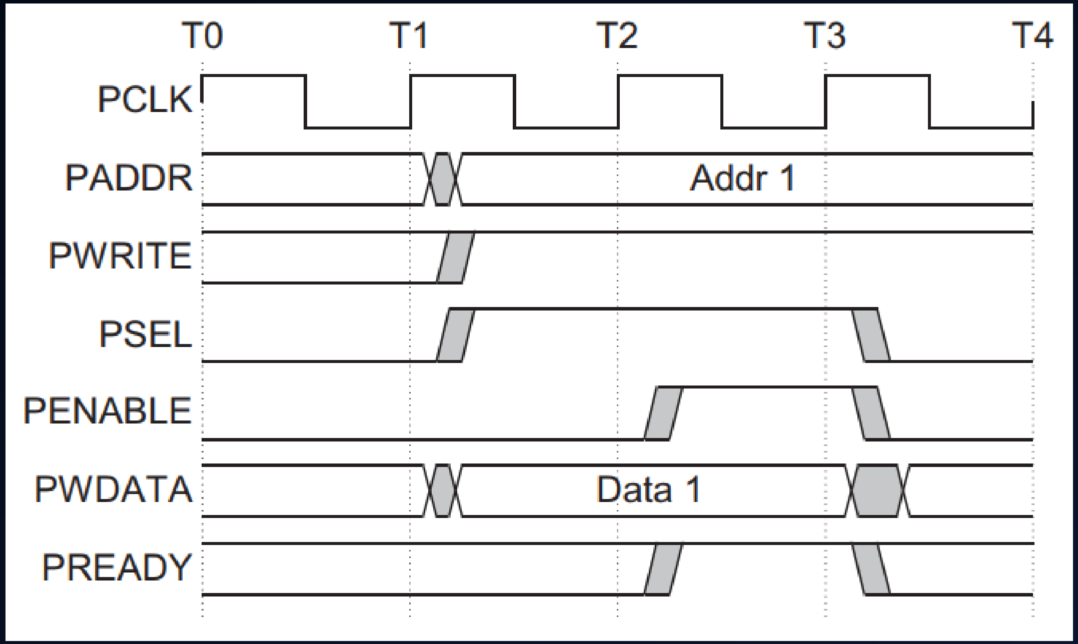
Signal	Source	Description
PREADY	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
PRDATA	Slave interface	Read Data. The selected slave drives this bus during read cycles when **PWRITE** is LOW. This bus can be up to 32-bits wide.
PSLVERR	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the **PSLVERR** pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

AMBA3 APB – Write Transfer

- With no wait states.
- With wait states.

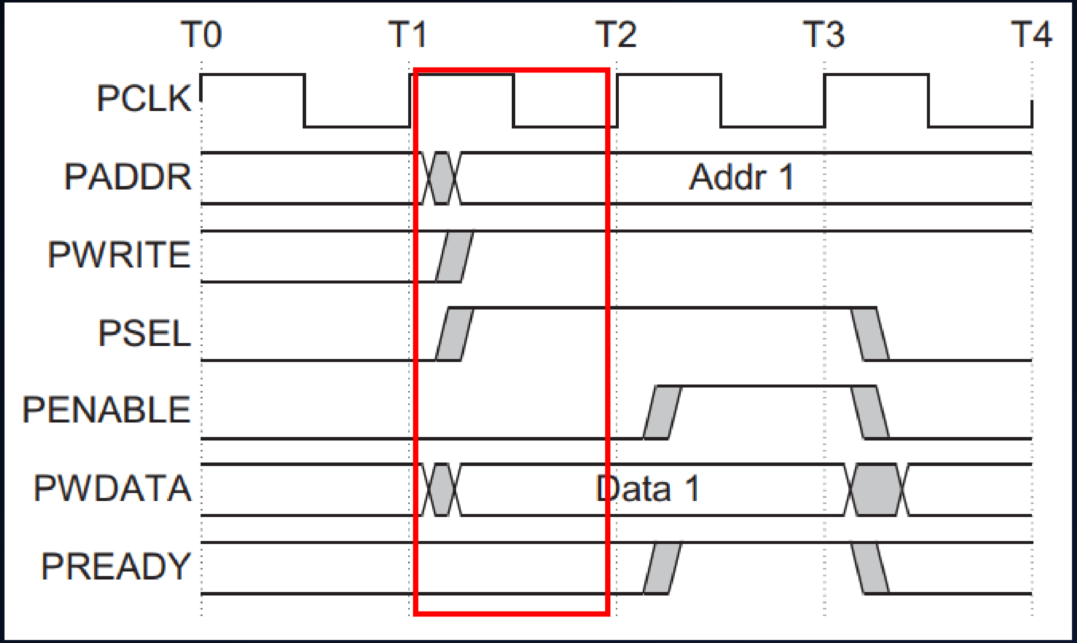
AMBA3 APB – Write Transfer with no wait states

- Signals:
  - PCLK
  - PADDR
  - PWRITE
  - PSEL
  - PENABLE
  - PWDATA
  - PREADY



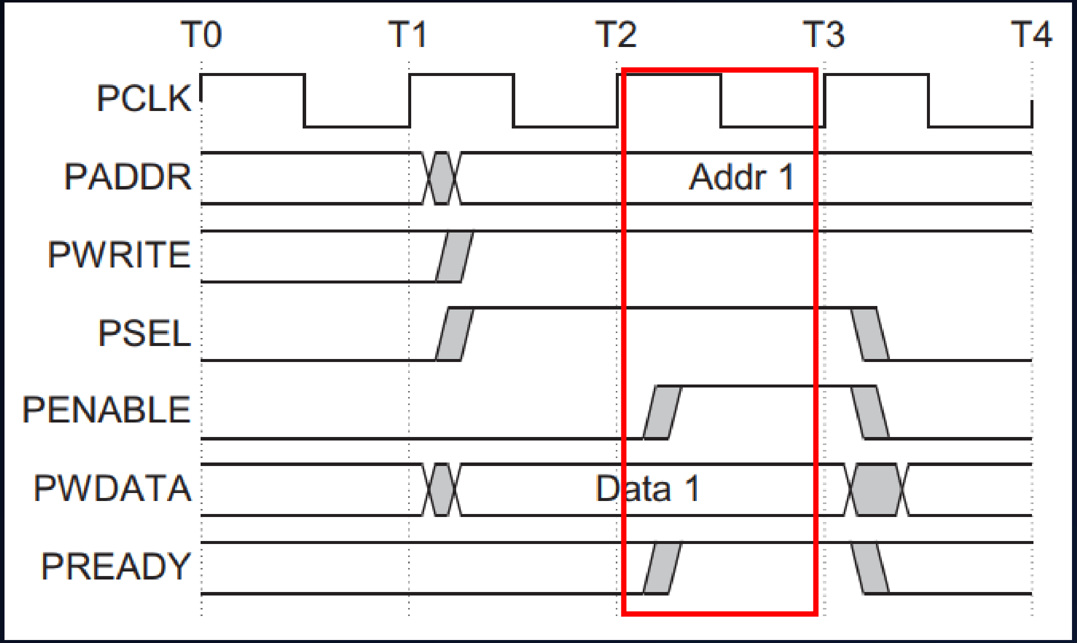
AMBA3 APB – Write Transfer with no wait states

- Setup Phase:
  - The write transfer starts with the address, write data, write signal and select signal all changing after the rising edge of the clock.



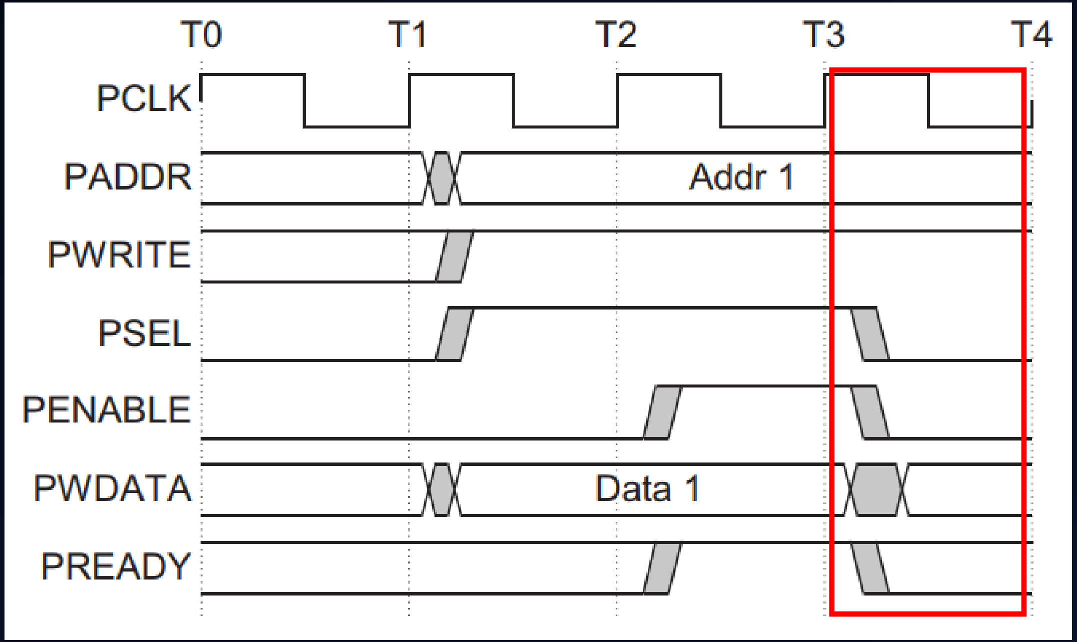
AMBA3 APB – Write Transfer with no wait states

- Access phase:
  - After the following clock edge the enable signal is asserted, PENABLE, and this indicates that the Access phase is taking place.
  - The address, data and control signals all remain valid throughout the Access phase.
  - The transfer completes at the end of this cycle.



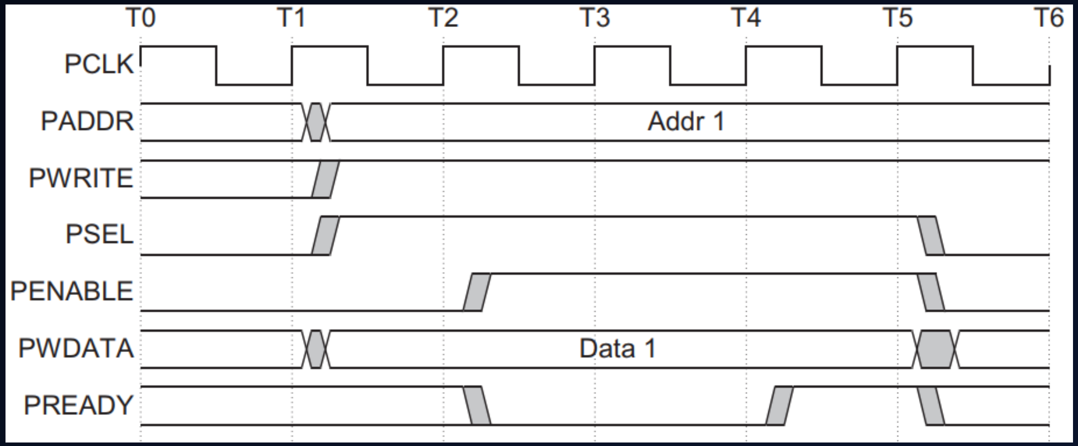
AMBA3 APB – Write Transfer with no wait states

- Final phase:
  - The enable signal, PENABLE, is deasserted at the end of the transfer.
  - The select signal, PSELx, also goes LOW unless the transfer is to be followed immediately by another transfer to the same peripheral.



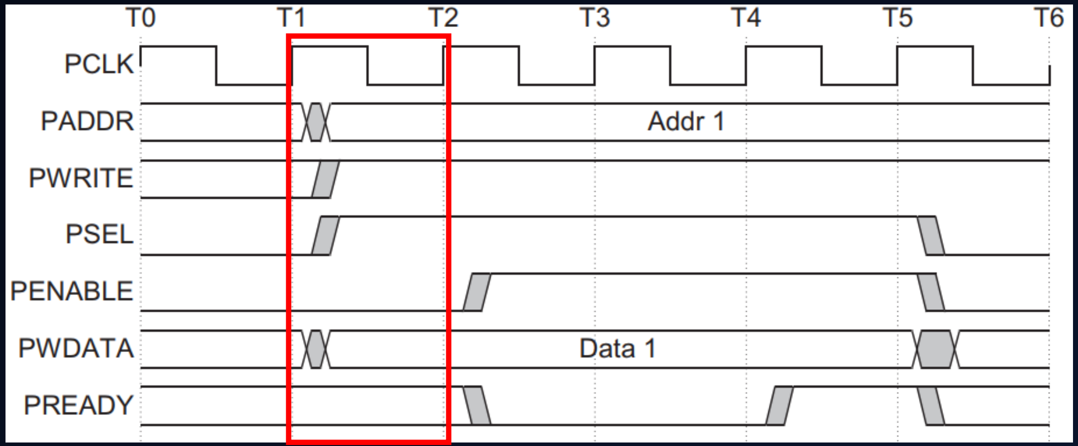
AMBA3 APB – Write Transfer with wait states

- Registers:
  - PCLK
  - PADDR
  - PWRITE
  - PSEL
  - PENABLE
  - PWDATA
  - PREADY



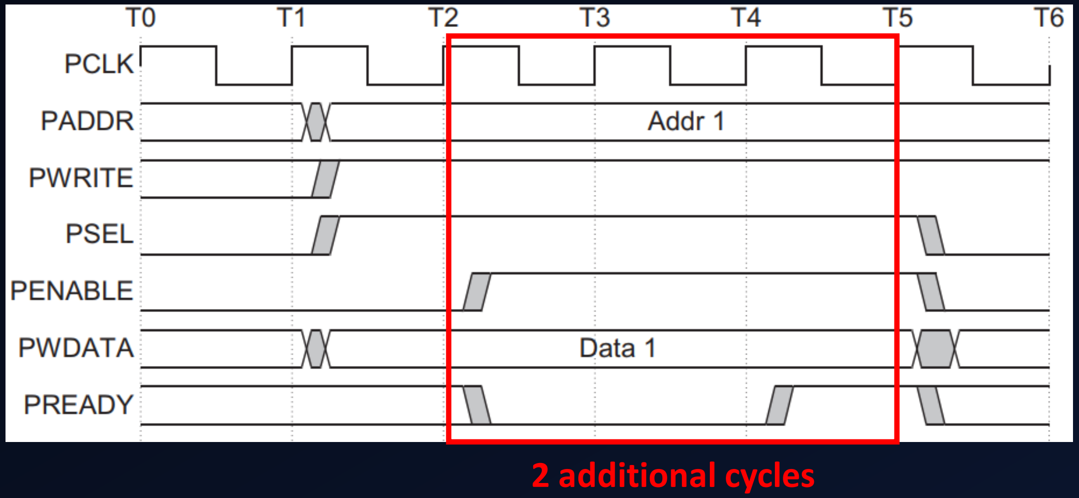
AMBA3 APB – Write Transfer with wait states

- Setup Phase:
  - The write transfer starts with the address, write data, write signal and select signal all changing after the rising edge of the clock.
  - Same as with no wait states.



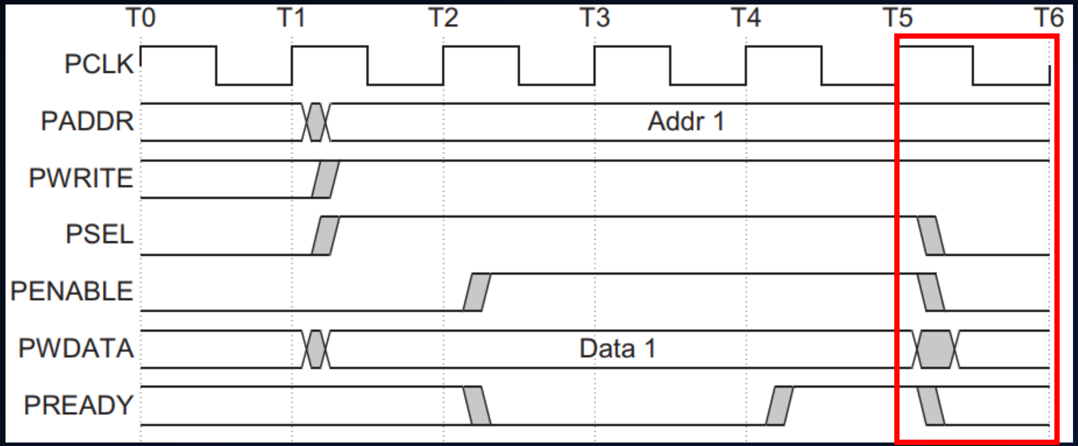
AMBA3 APB – Write Transfer with wait states

- Access phase:
  - During an Access phase, when PENABLE is HIGH, the transfer can be extended by driving PREADY LOW.
  - Remain unchanged:
    - PADDR
    - PWRITE
    - PSEL
    - PENABLE
    - PWDATA



AMBA3 APB – Write Transfer with wait states

- Final phase:
  - The enable signal, PENABLE, is deasserted at the end of the transfer.
  - The select signal, PSELx, also goes LOW unless the transfer is to be followed immediately by another transfer to the same peripheral.

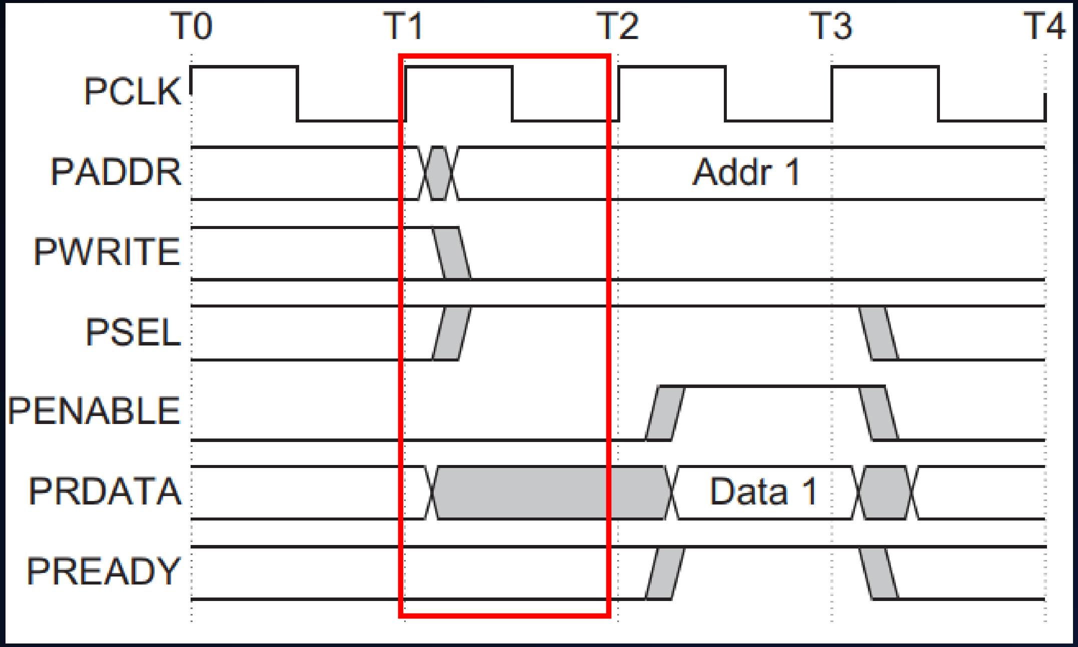


AMBA3 APB – Read Transfer

- With no wait states.
- With wait states.

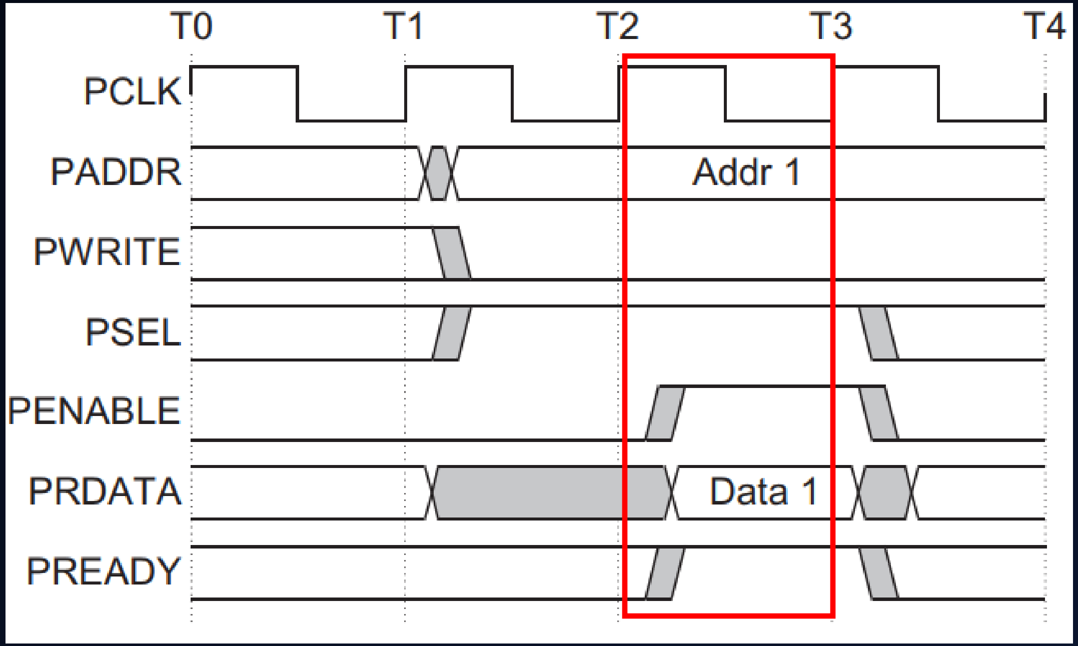
AMBA3 APB – Read Transfer with no wait states

- Setup Phase:
  - The read transfer starts the same as the write transfer: with the address, write signal and select signal all changing after the rising edge of the clock.
  - PWRITE set to LOW.



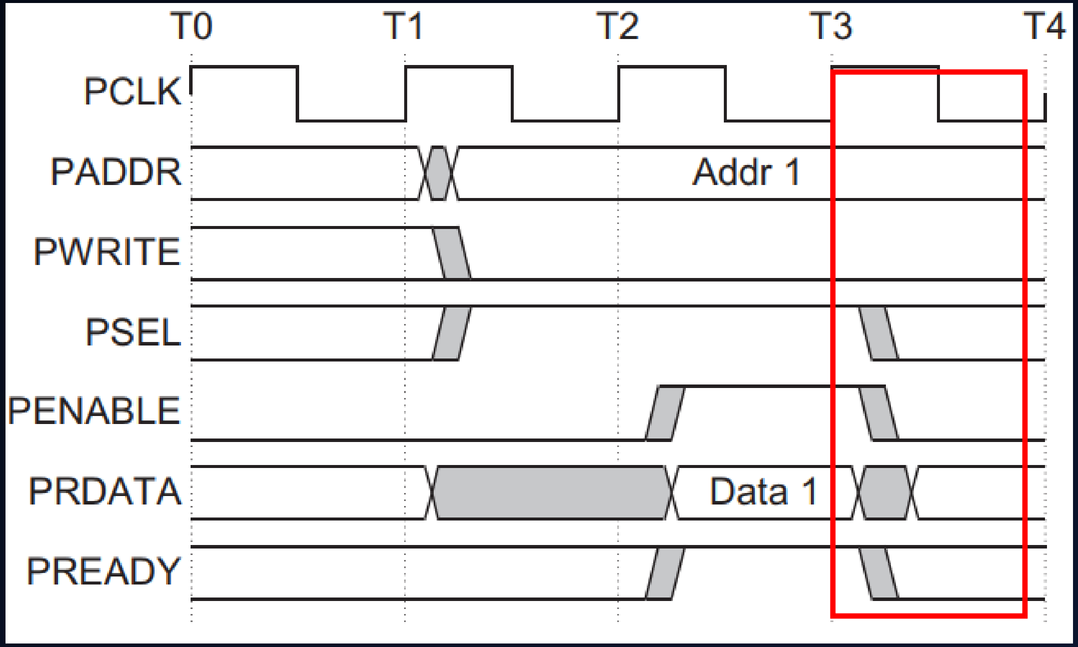
AMBA3 APB – Read Transfer with no wait states

- Access phase:
  - After the following clock edge the enable signal is asserted, PENABLE, and this indicates that the Access phase is taking place.
  - The slave must provide the data before the end of the read transfer.



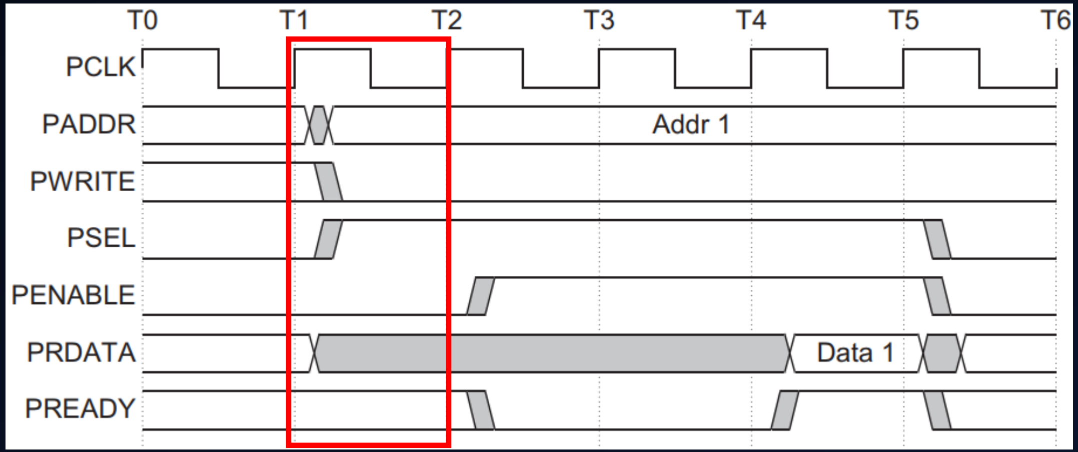
AMBA3 APB – Read Transfer with no wait states

- Final phase:
  - The enable signal, PENABLE, is deasserted at the end of the transfer.
  - The select signal, PSELx, also goes LOW unless the transfer is to be followed immediately by another transfer to the same peripheral.



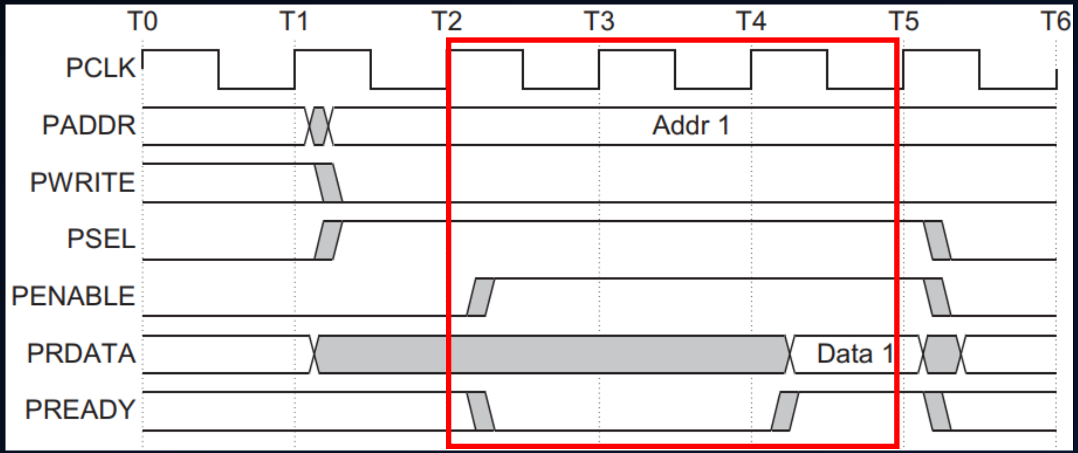
AMBA3 APB – Read Transfer with wait states

- Setup Phase:
  - The write transfer starts with the address, write data, write signal and select signal all changing after the rising edge of the clock.
  - Same as with no wait states.



AMBA3 APB – Read Transfer with wait states

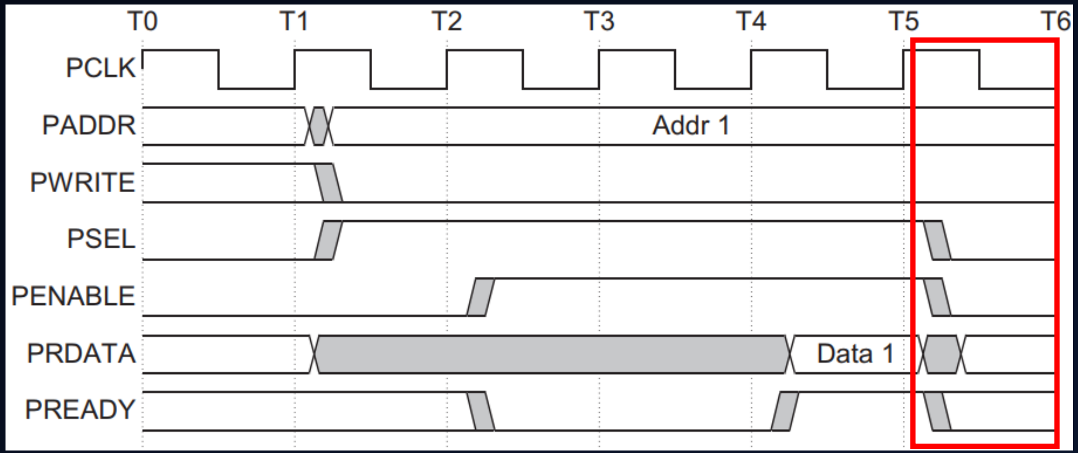
- Access phase:
  - During an Access phase, when PENABLE is HIGH, the transfer can be extended by driving PREADY LOW.
  - Remain unchanged:
    - PADDR
    - PWRITE
    - PSEL
    - PENABLE



2 additional cycles

AMBA3 APB – Read Transfer with wait states

- Final phase:
  - The enable signal, PENABLE, is deasserted at the end of the transfer.
  - The select signal, PSELx, also goes LOW unless the transfer is to be followed immediately by another transfer to the same peripheral.



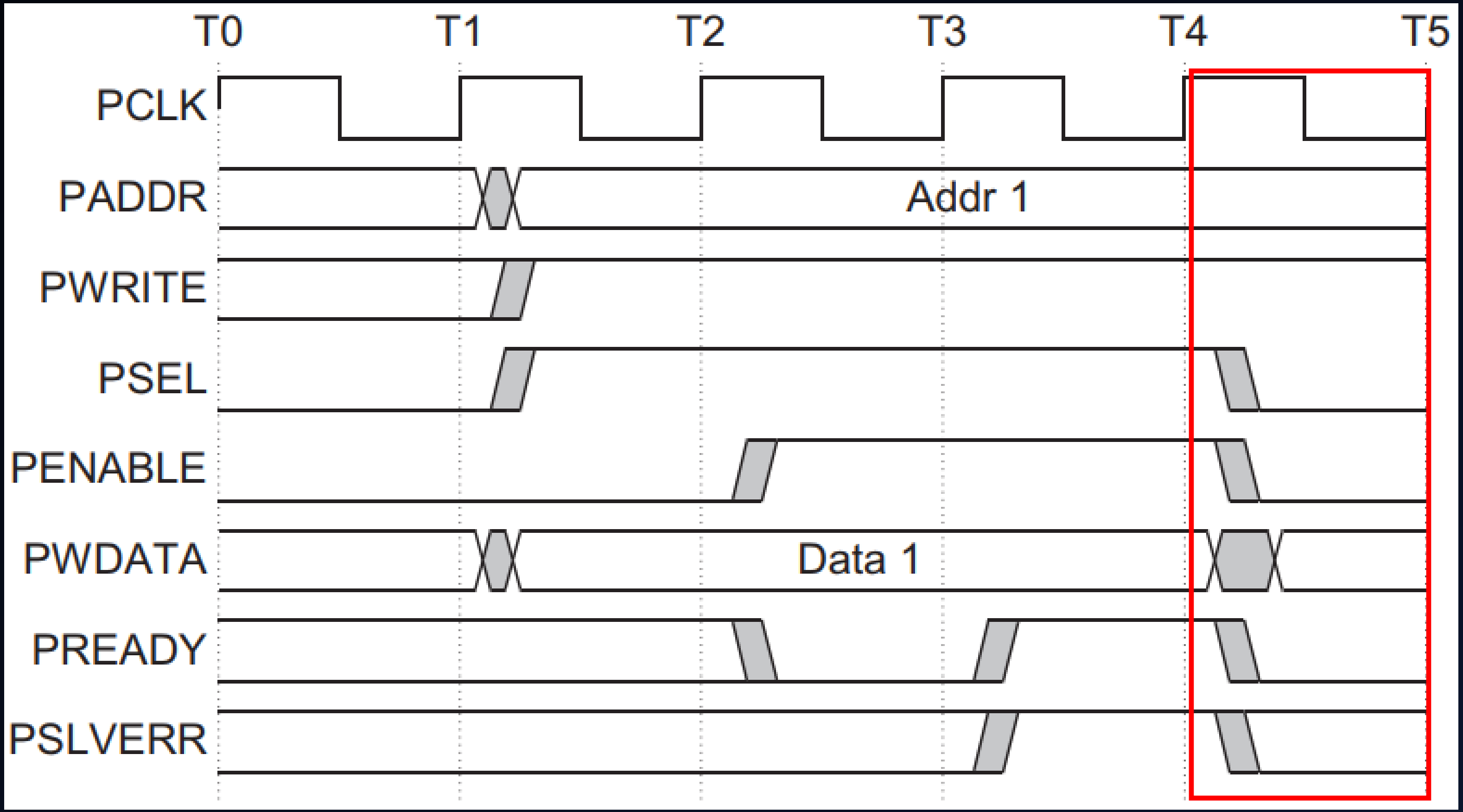
AMBA3 APB – Error Response

- You can use PSLVERR to indicate an error condition on an APB transfer.
- Error conditions can occur on both:
  - Read transactions.
  - Write transactions.
- PSLVERR is only considered valid during the last cycle of an APB transfer, when PSEL, PENABLE, and PREADY are all HIGH.

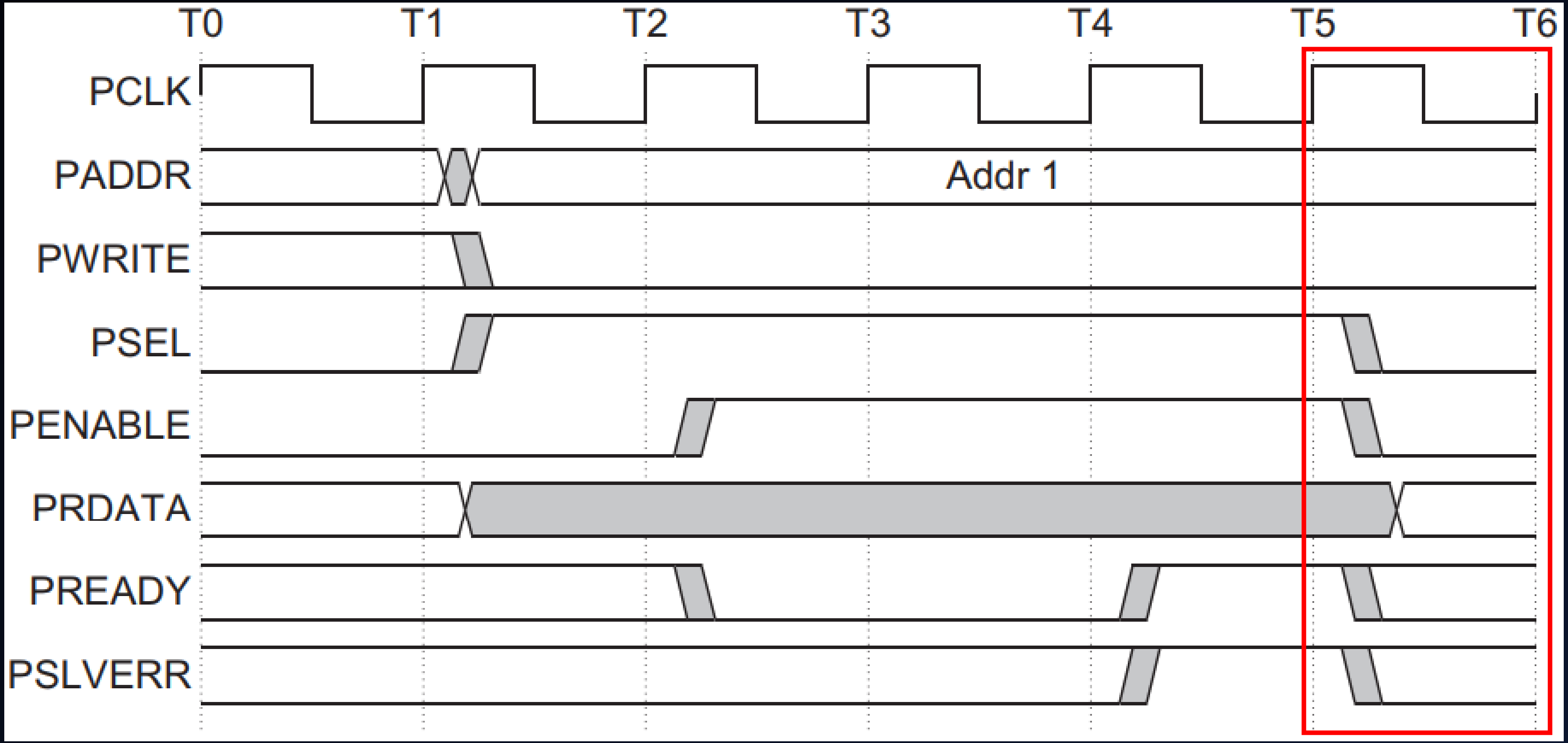
AMBA3 APB – Error Response

- When an error occurs:
  - Peripheral might or might not have changed state.
  - The register within the peripheral might or might not have been written to.
  - Data read might be invalid.
- Support for PSLVERR signal is not required in APB peripherals.
- Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

AMBA3 APB – Error Response on Write Transfer



AMBA3 APB – Error Response on Read Transfer

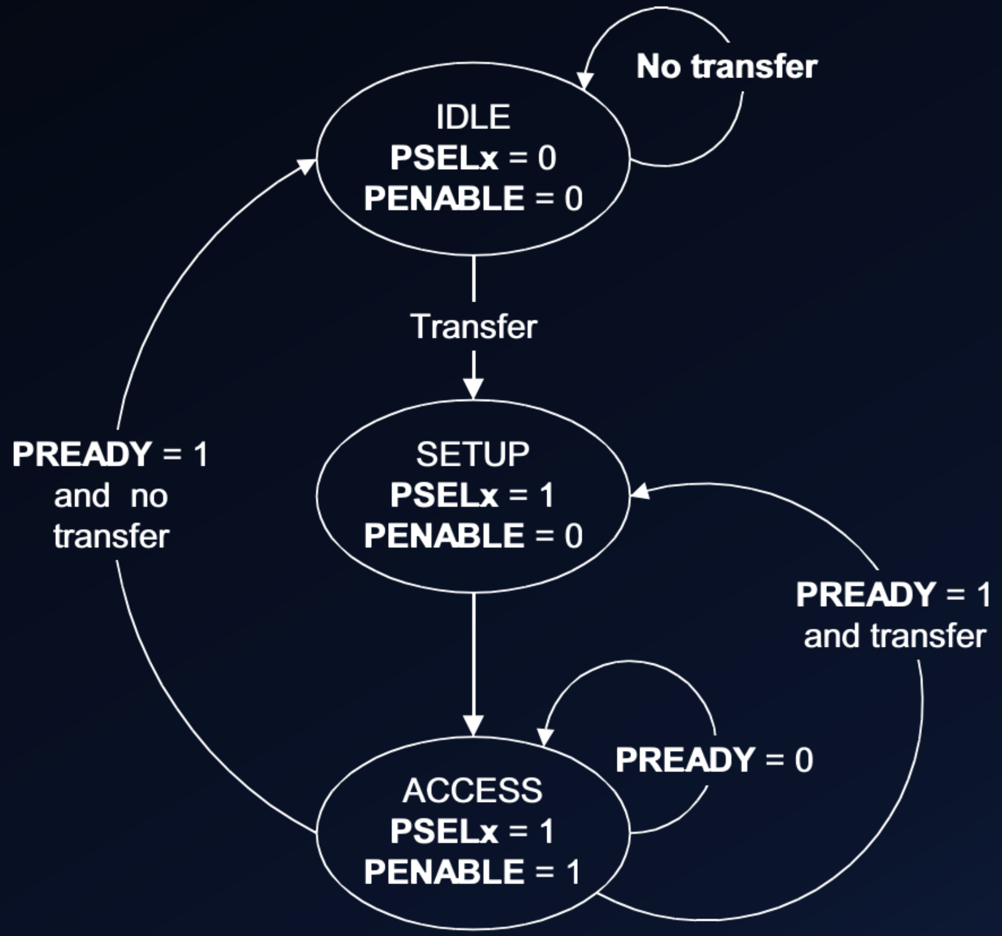


AMBA3 APB – Error Response on AXI/AHB

- From AXI to APB:
  - An APB error is mapped back to RRESP/BRESP = SLVERR. This is achieved by mapping PSLVERR to the AXI signals RRESP[1] for reads and BRESP[1] for writes.
- From AHB to APB:
  - PSLVERR is mapped back to HRESP = ERROR for both reads and writes. This is achieved by mapping PSLVERR to the AHB signal HRESP[0].

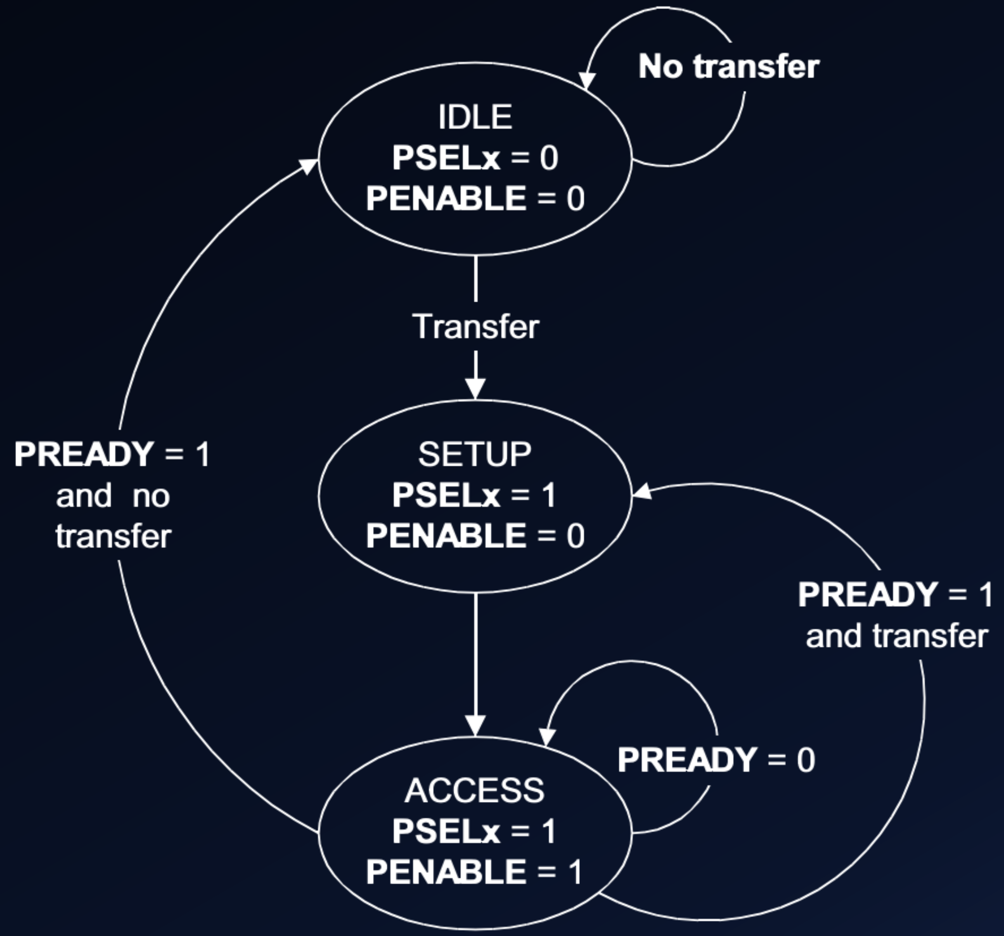
AMBA3 APB – Operating States

- IDLE:
  - Default state of the APB
  - When a transfer is required the bus moves into the SETUP



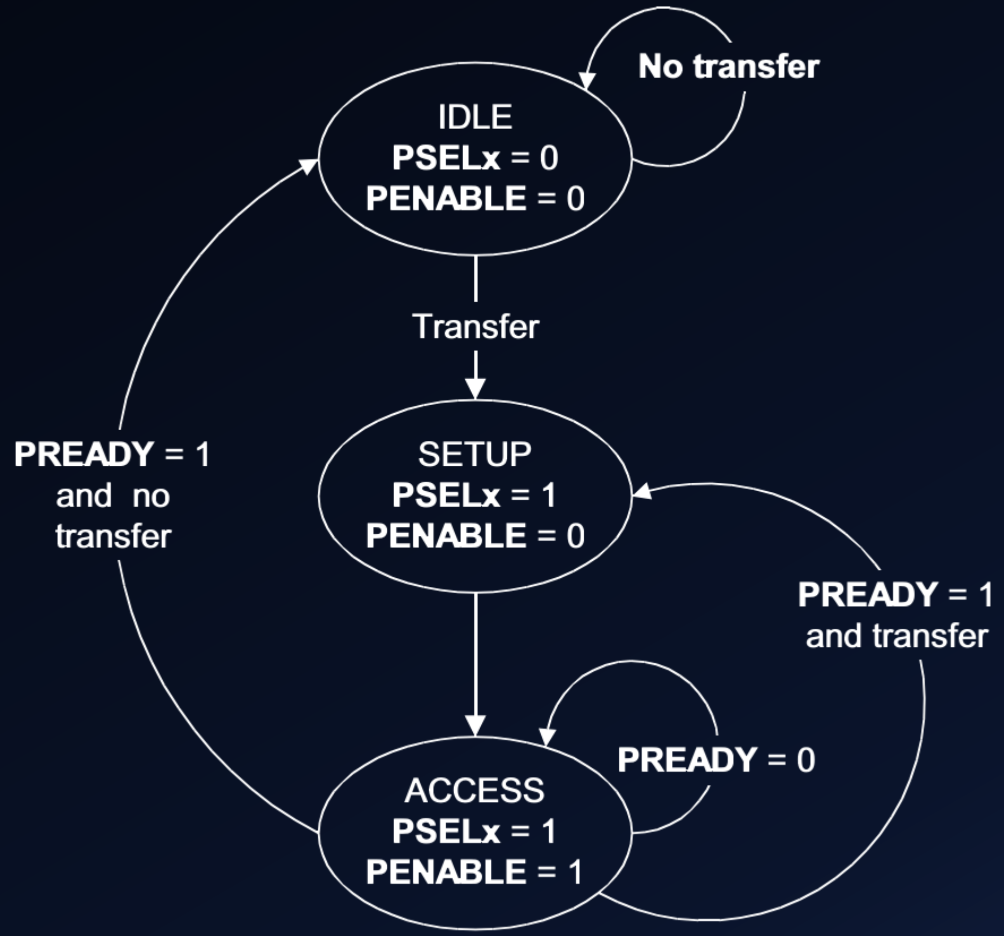
AMBA3 APB – Operating States

- SETUP:
  - The appropriate select signal, PSELx, is asserted.
  - The bus only remains in the SETUP state for one clock cycle and always moves to the ACCESS state on the next rising edge of the clock.
  - The address, write, select, and write data signals must remain stable during the transition from the SETUP to ACCESS state.

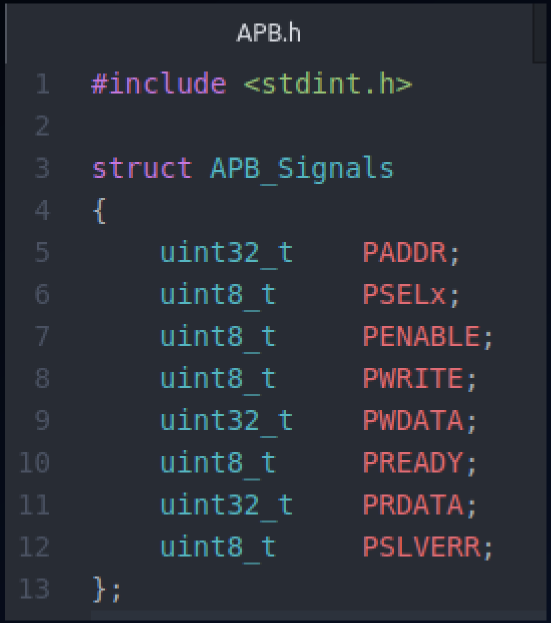


AMBA3 APB – Operating States

- ACCESS:
  - The enable signal, PENABLE, is asserted.
  - Exit from the ACCESS state is controlled by the PREADY signal from the slave:
    - If PREADY is held LOW by the slave then the peripheral bus remains in the ACCESS state.
    - If PREADY is driven HIGH by the slave then the ACCESS state is exited and the bus returns to the IDLE state if no more transfers are required. Alternatively, the bus moves directly to the SETUP state if another transfer follows.

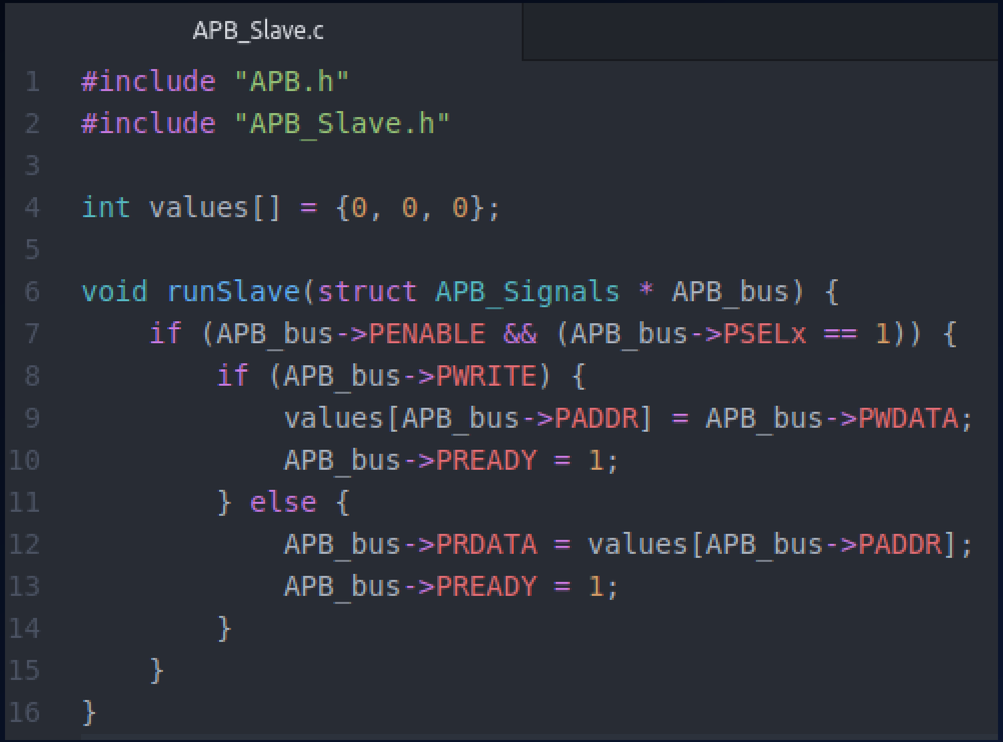


AMBA3 APB – Implementation



APB.h

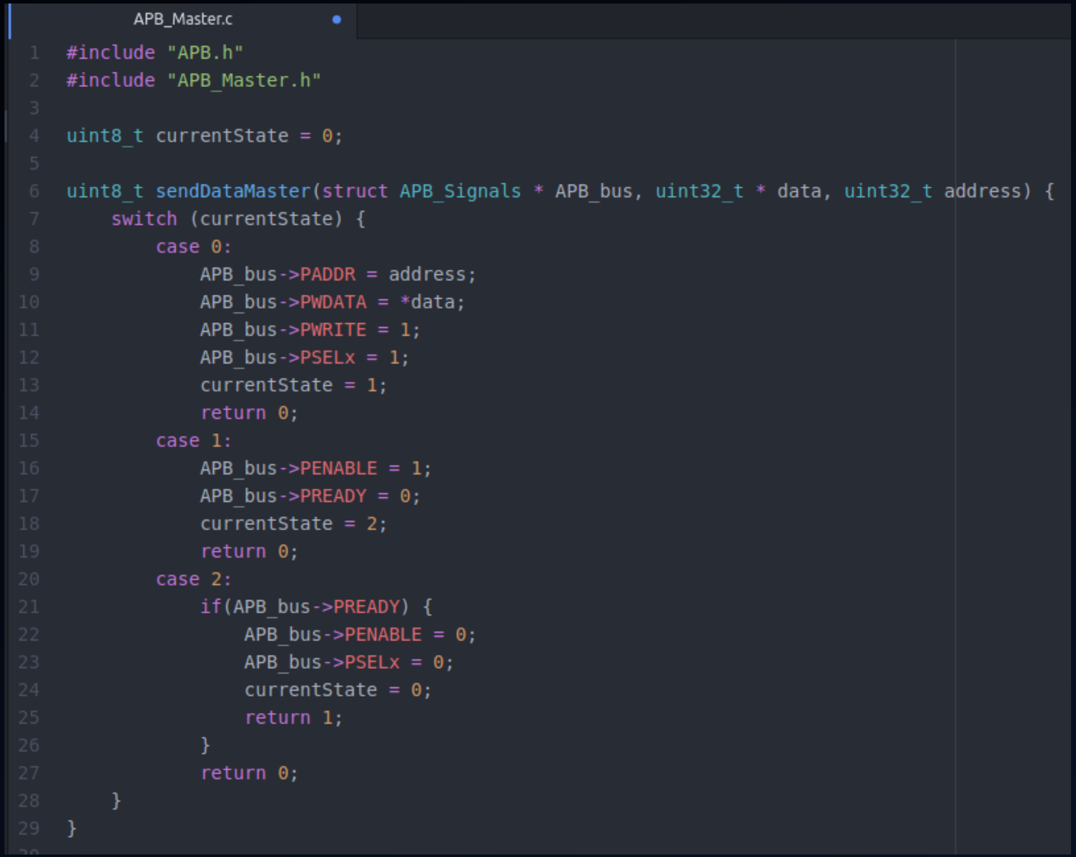
```c
#include <stdint.h>

struct APB_Signals
{
    uint32_t    PADDR;
    uint8_t     PSELx;
    uint8_t     PENABLE;
    uint8_t     PWRITE;
    uint32_t    PWDATA;
    uint8_t     PREADY;
    uint32_t    PRDATA;
    uint8_t     PSLVERR;
};
```



APB_Slave.c

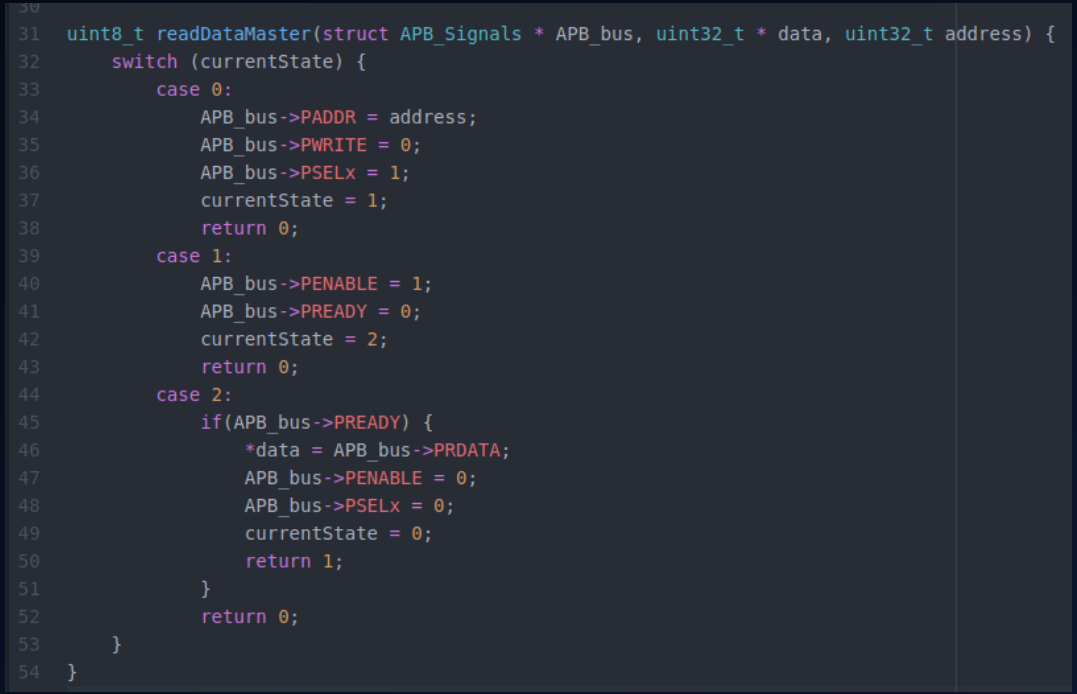
```c
#include "APB.h"
#include "APB_Slave.h"

int values[] = {0, 0, 0};

void runSlave(struct APB_Signals * APB_bus) {
    if (APB_bus->PENABLE && (APB_bus->PSELx == 1)) {
        if (APB_bus->PWRITE) {
            values[APB_bus->PADDR] = APB_bus->PWDATA;
            APB_bus->PREADY = 1;
        } else {
            APB_bus->PRDATA = values[APB_bus->PADDR];
            APB_bus->PREADY = 1;
        }
    }
}
```

AMBA3 APB – Implementation



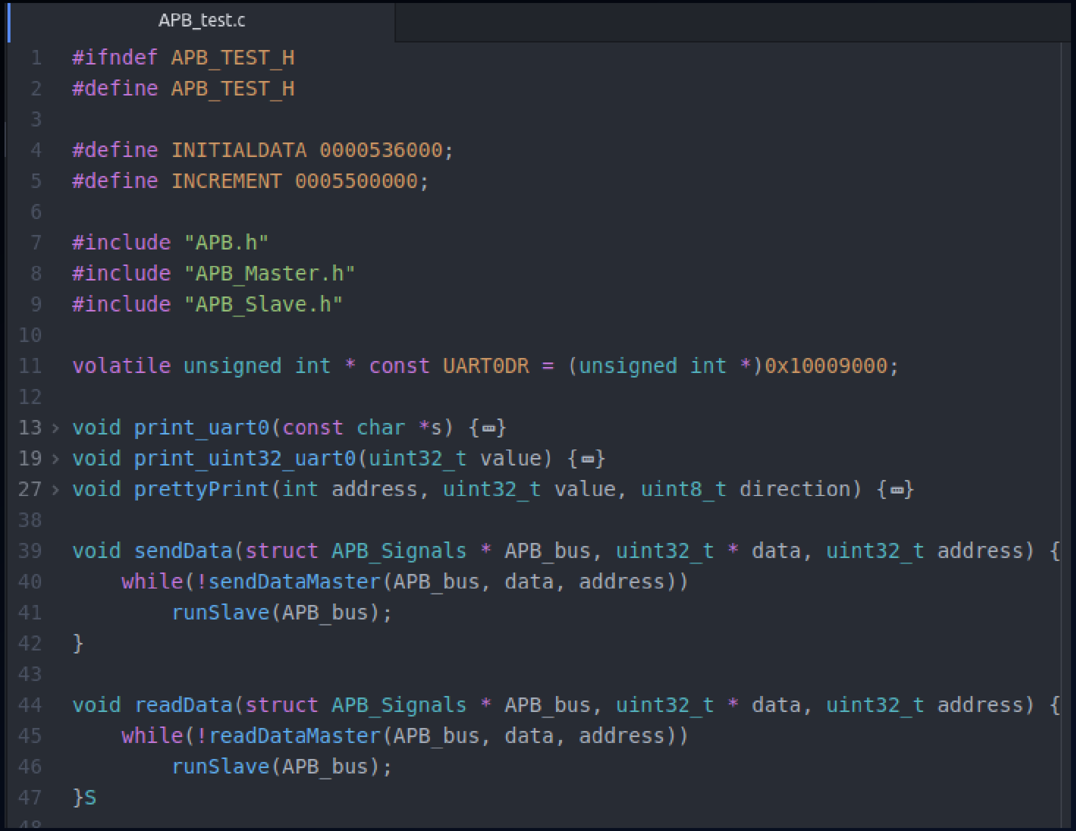
APB_Master.c

```c
#include "APB.h"
#include "APB_Master.h"

uint8_t currentState = 0;

uint8_t sendDataMaster(struct APB_Signals * APB_bus, uint32_t * data, uint32_t address) {
    switch (currentState) {
        case 0:
            APB_bus->PADDR = address;
            APB_bus->PWDATA = *data;
            APB_bus->PWRITE = 1;
            APB_bus->PSELx = 1;
            currentState = 1;
            return 0;
        case 1:
            APB_bus->PENABLE = 1;
            APB_bus->PREADY = 0;
            currentState = 2;
            return 0;
        case 2:
            if(APB_bus->PREADY) {
                APB_bus->PENABLE = 0;
                APB_bus->PSELx = 0;
                currentState = 0;
                return 1;
            }
            return 0;
    }
}
```



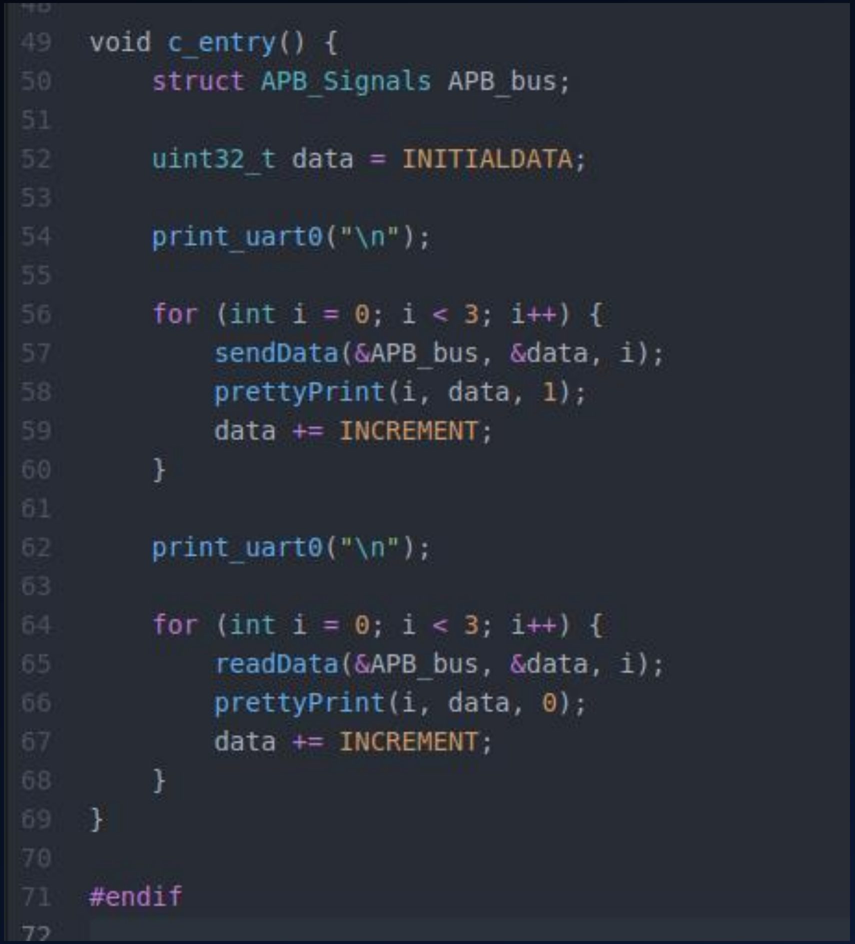
```c
uint8_t readDataMaster(struct APB_Signals * APB_bus, uint32_t * data, uint32_t address) {
    switch (currentState) {
        case 0:
            APB_bus->PADDR = address;
            APB_bus->PWRITE = 0;
            APB_bus->PSELx = 1;
            currentState = 1;
            return 0;
        case 1:
            APB_bus->PENABLE = 1;
            APB_bus->PREADY = 0;
            currentState = 2;
            return 0;
        case 2:
            if(APB_bus->PREADY) {
                *data = APB_bus->PRDATA;
                APB_bus->PENABLE = 0;
                APB_bus->PSELx = 0;
                currentState = 0;
                return 1;
            }
            return 0;
    }
}
```

AMBA3 APB – Implementation



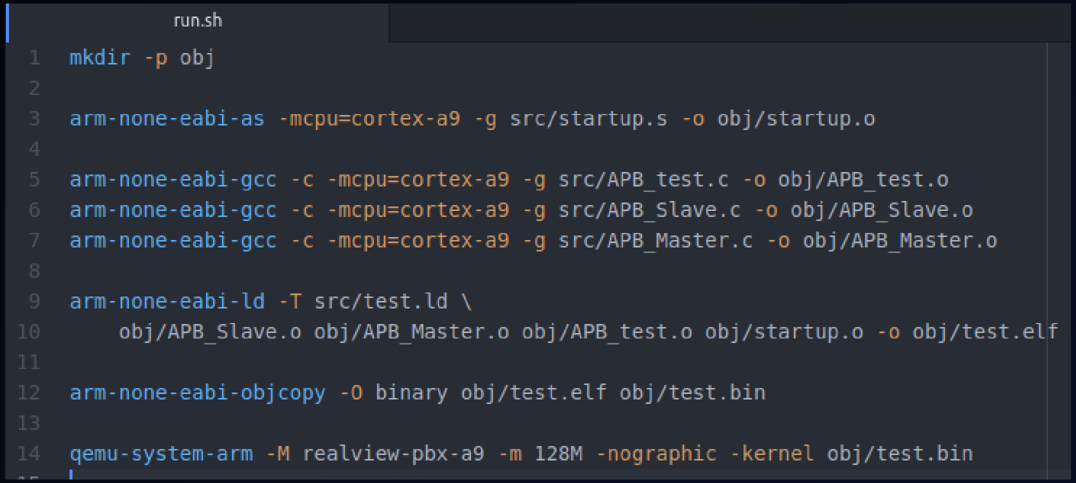
APB_test.c

```c
#ifndef APB_TEST_H
#define APB_TEST_H

#define INITIALDATA 0000536000;
#define INCREMENT 0005500000;

#include "APB.h"
#include "APB_Master.h"
#include "APB_Slave.h"

volatile unsigned int * const UART0DR = (unsigned int *)0x10009000;

void print_uart0(const char *s) {⋯}
void print_uint32_uart0(uint32_t value) {⋯}
void prettyPrint(int address, uint32_t value, uint8_t direction) {⋯}

void sendData(struct APB_Signals * APB_bus, uint32_t * data, uint32_t address) {
    while(!sendDataMaster(APB_bus, data, address))
        runSlave(APB_bus);
}

void readData(struct APB_Signals * APB_bus, uint32_t * data, uint32_t address) {
    while(!readDataMaster(APB_bus, data, address))
        runSlave(APB_bus);
}S
```



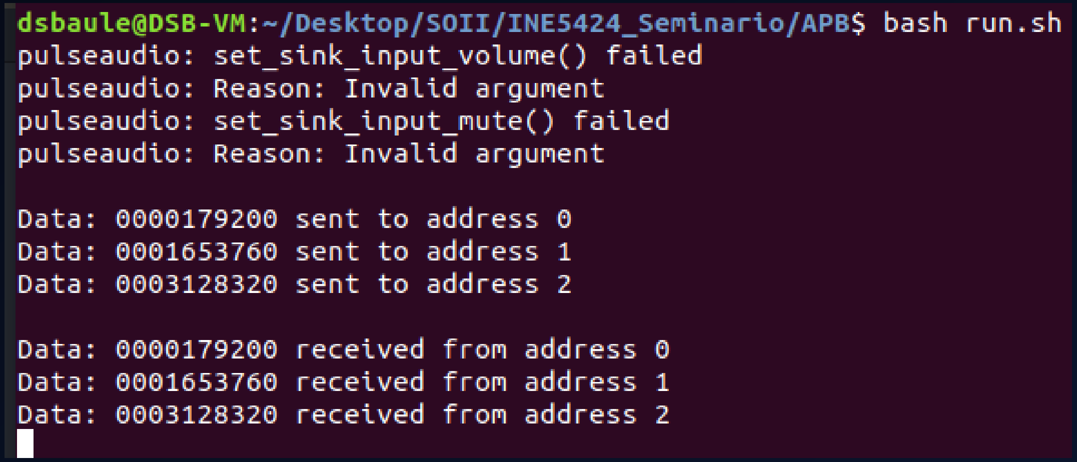
```c
void c_entry() {
    struct APB_Signals APB_bus;

    uint32_t data = INITIALDATA;

    print_uart0("\n");

    for (int i = 0; i < 3; i++) {
        sendData(&APB_bus, &data, i);
        prettyPrint(i, data, 1);
        data += INCREMENT;
    }

    print_uart0("\n");

    for (int i = 0; i < 3; i++) {
        readData(&APB_bus, &data, i);
        prettyPrint(i, data, 0);
        data += INCREMENT;
    }
}

#endif
```

AMBA3 APB – Implementation



run.sh

```
mkdir -p obj

arm-none-eabi-as -mcpu=cortex-a9 -g src/startup.s -o obj/startup.o

arm-none-eabi-gcc -c -mcpu=cortex-a9 -g src/APB_test.c -o obj/APB_test.o
arm-none-eabi-gcc -c -mcpu=cortex-a9 -g src/APB_Slave.c -o obj/APB_Slave.o
arm-none-eabi-gcc -c -mcpu=cortex-a9 -g src/APB_Master.c -o obj/APB_Master.o

arm-none-eabi-ld -T src/test.ld \
    obj/APB_Slave.o obj/APB_Master.o obj/APB_test.o obj/startup.o -o obj/test.elf

arm-none-eabi-objcopy -O binary obj/test.elf obj/test.bin

qemu-system-arm -M realview-pbx-a9 -m 128M -nographic -kernel obj/test.bin
```



```
dsbaule@DSB-VM:~/Desktop/SOII/INE5424_Seminario/APB$ bash run.sh
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument

Data: 0000179200 sent to address 0
Data: 0001653760 sent to address 1
Data: 0003128320 sent to address 2

Data: 0000179200 received from address 0
Data: 0001653760 received from address 1
Data: 0003128320 received from address 2
```

AMBA ATB

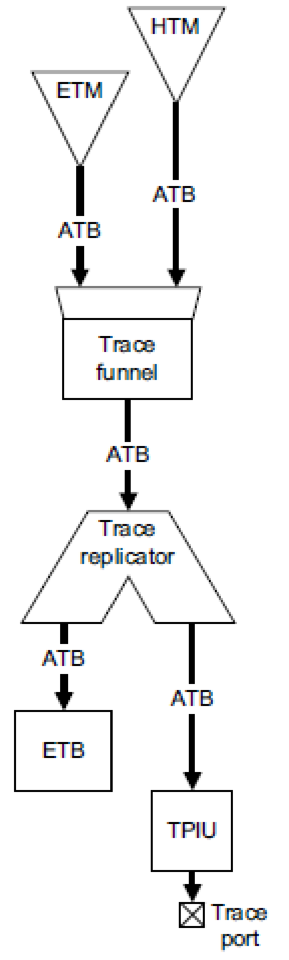


Figure 1-1 ATB relationships

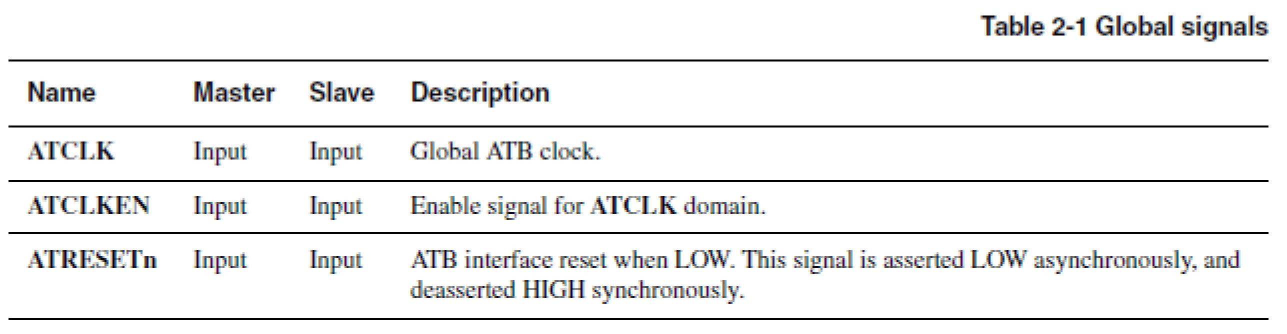


Table 2-1 Global signals

Name	Master	Slave	Description
ATCLK	Input	Input	Global ATB clock.
ATCLKEN	Input	Input	Enable signal for **ATCLK** domain.
ATRESETn	Input	Input	ATB interface reset when LOW. This signal is asserted LOW asynchronously, and deasserted HIGH synchronously.

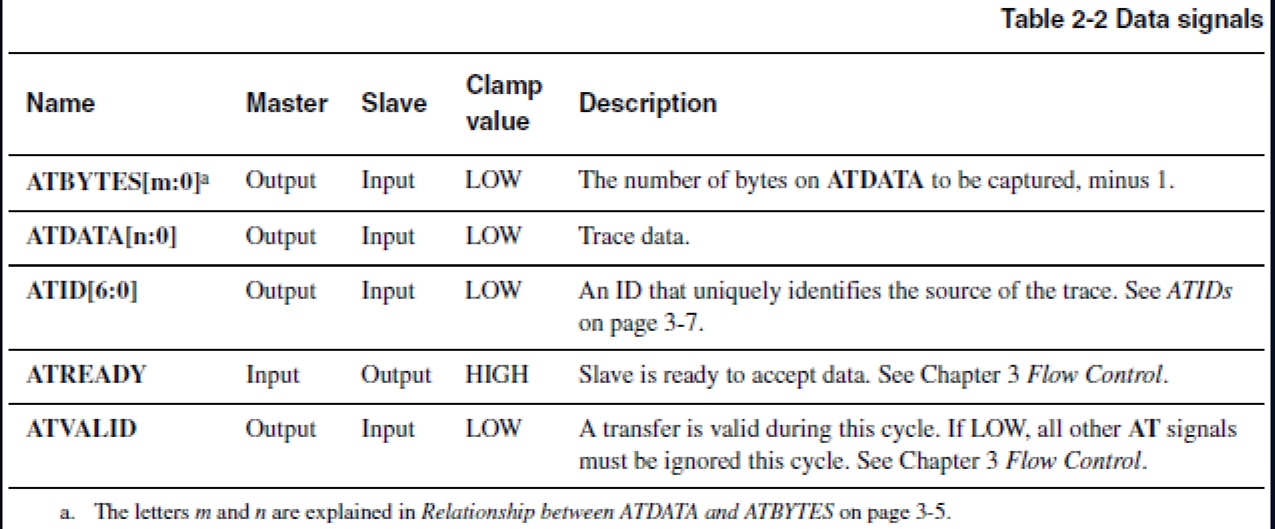


Table 2-2 Data signals

Name	Master	Slave	Clamp value	Description
ATBYTES[m:0][a]	Output	Input	LOW	The number of bytes on **ATDATA** to be captured, minus 1.
ATDATA[n:0]	Output	Input	LOW	Trace data.
ATID[6:0]	Output	Input	LOW	An ID that uniquely identifies the source of the trace. See *ATIDs* on page 3-7.
ATREADY	Input	Output	HIGH	Slave is ready to accept data. See Chapter 3 *Flow Control*.
ATVALID	Output	Input	LOW	A transfer is valid during this cycle. If LOW, all other **AT** signals must be ignored this cycle. See Chapter 3 *Flow Control*.

a. The letters *m* and *n* are explained in *Relationship between ATDATA and ATBYTES* on page 3-5.

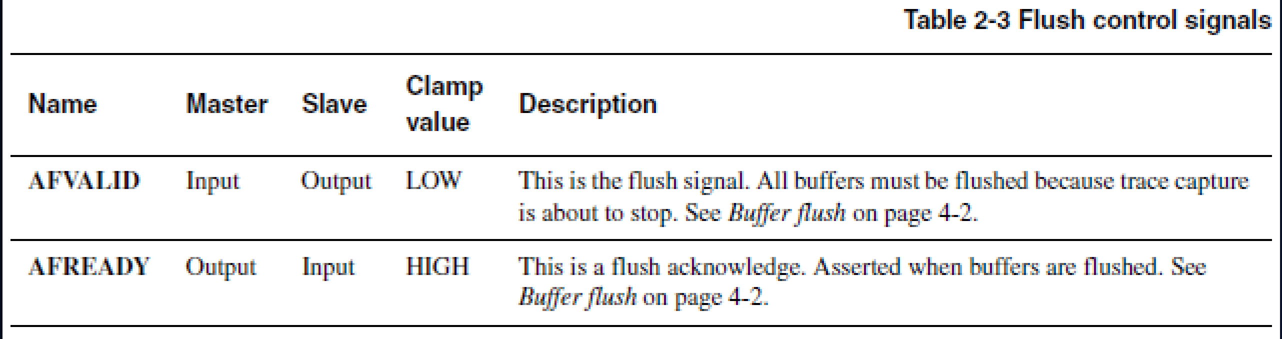


Table 2-3 Flush control signals

Name	Master	Slave	Clamp value	Description
AFVALID	Input	Output	LOW	This is the flush signal. All buffers must be flushed because trace capture is about to stop. See *Buffer flush* on page 4-2.
AFREADY	Output	Input	HIGH	This is a flush acknowledge. Asserted when buffers are flushed. See *Buffer flush* on page 4-2.

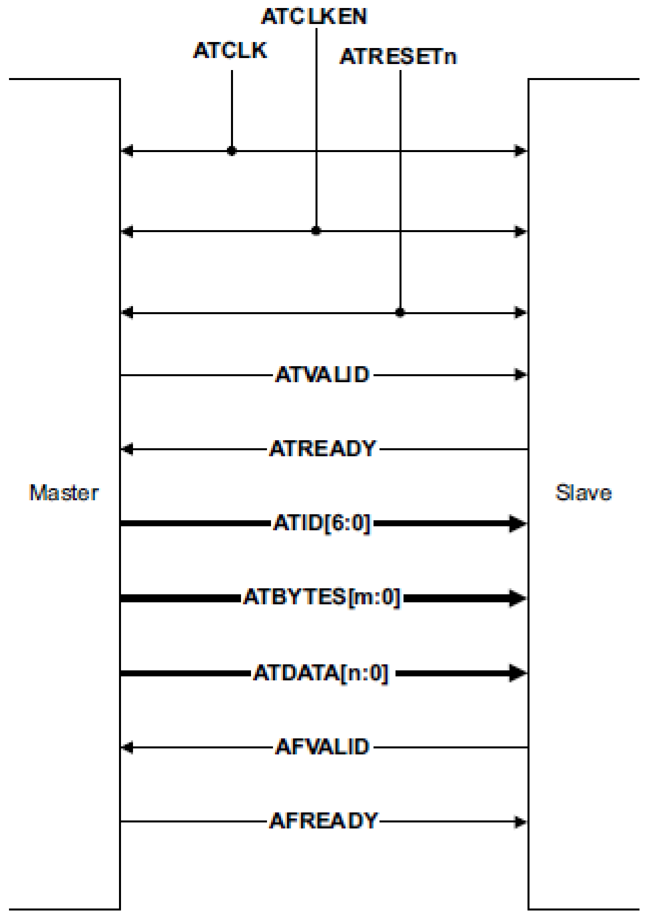


Figure 3-1 Flow of trace data between master and slave

Figure 3-2 shows normal ATVALID and ATREADY flow control.

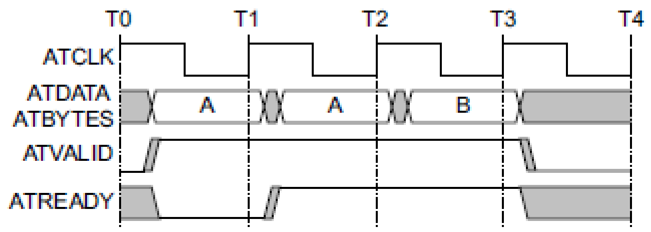


Figure 3-2 Normal ATVALID and ATREADY flow control

Table 3-1 shows the states of the ATB relationship to Figure 3-2.

Table 3-1 ATB states

Clock cycle	State
T1	Stalled, **ATREADY**
T2	A accepted
T3	B accepted
T4	Ignored, not valid

Table 3-2 Width relationship of ATDATA and ATBYTES

ATDATA[n:0]	ATBYTES[m:0]
n = 7	**ATBYTES** not required
n = 15	m = 0
n = 31	m = 1
n = 63	m = 2
n = 127	m = 3

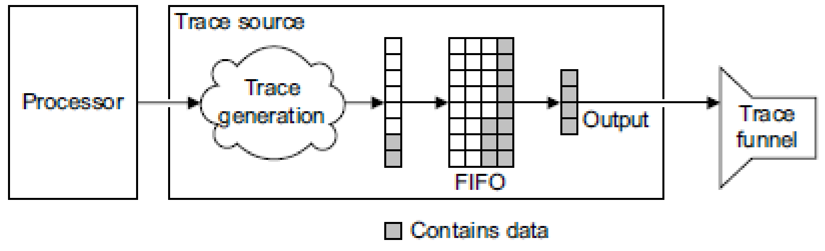
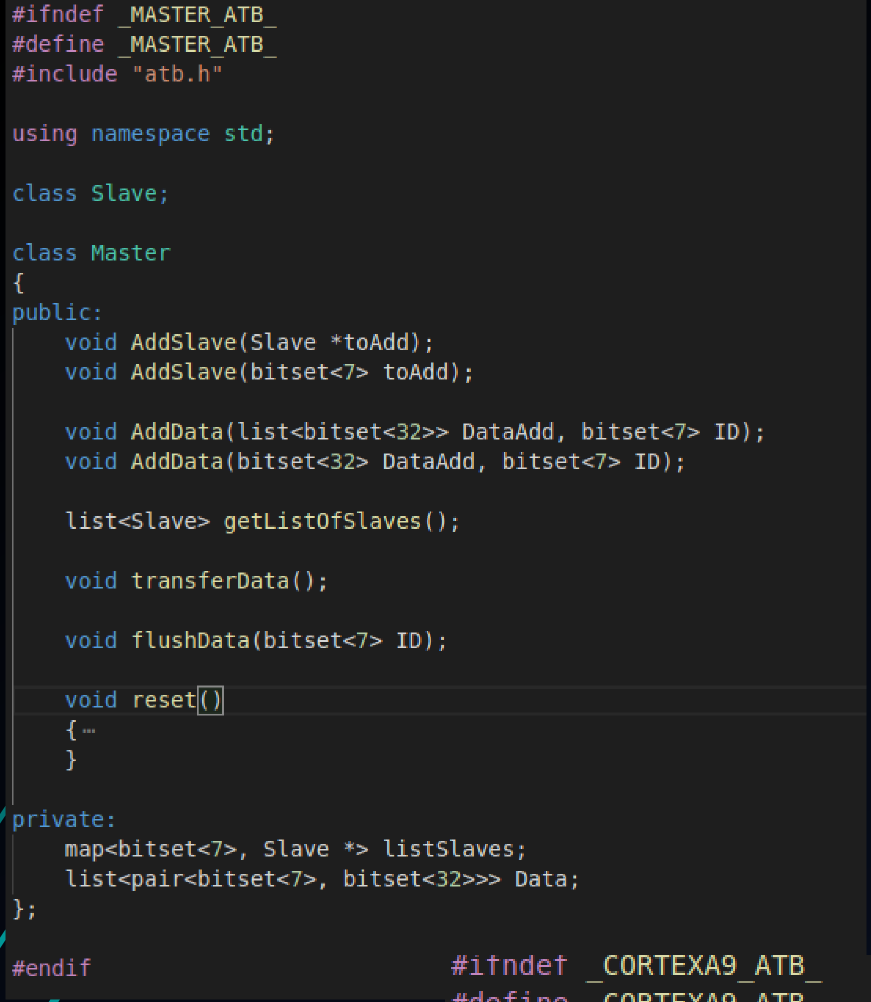
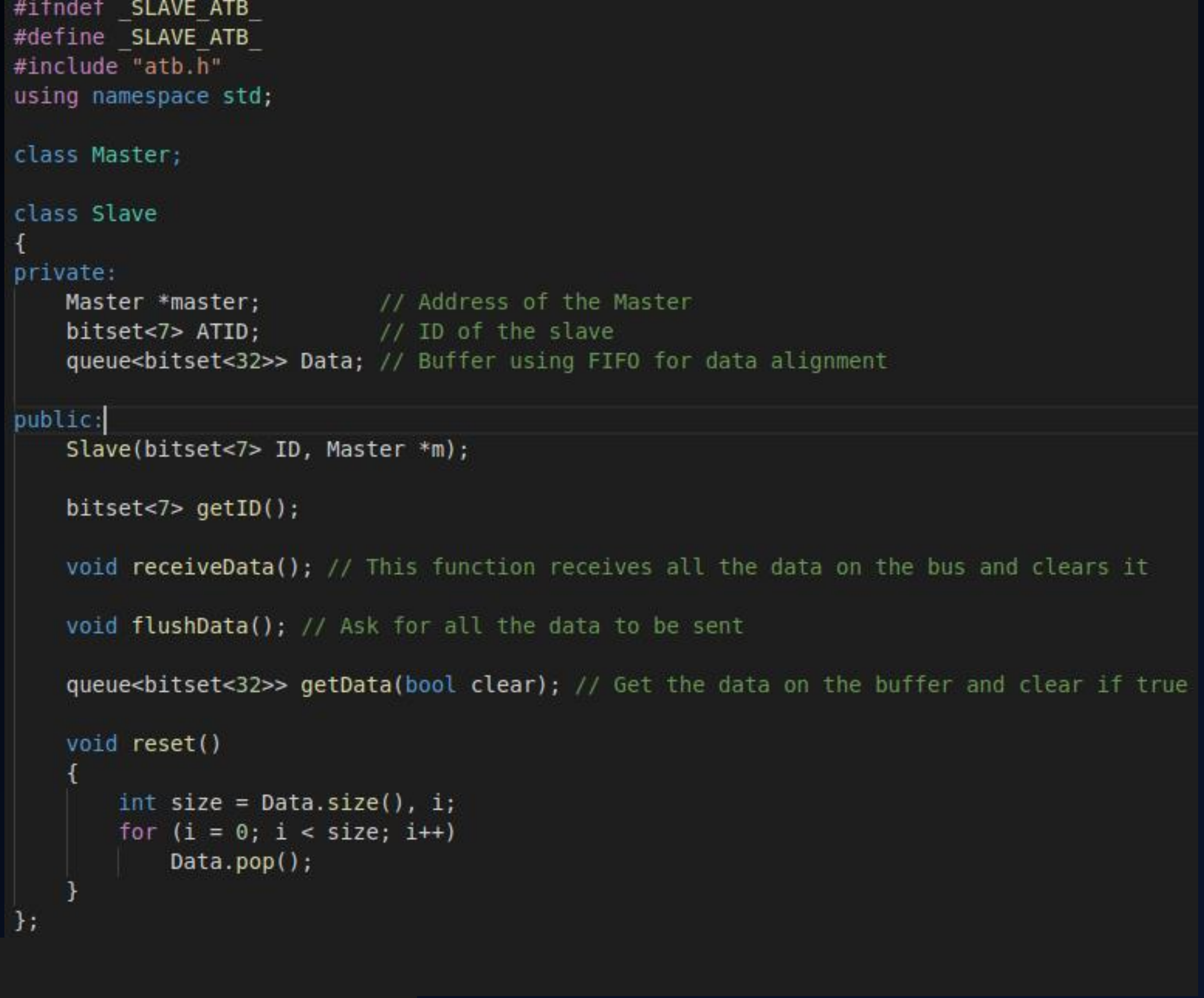


Figure 4-1 Trace generation and output

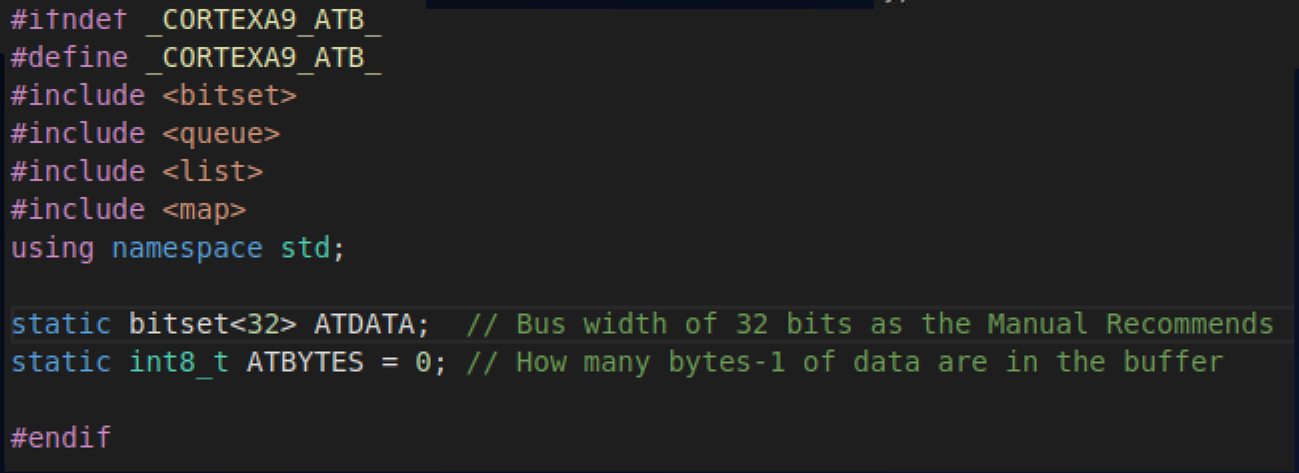
Files.h



```cpp
#ifndef _MASTER_ATB_
#define _MASTER_ATB_
#include "atb.h"

using namespace std;

class Slave;

class Master
{
public:
    void AddSlave(Slave *toAdd);
    void AddSlave(bitset<7> toAdd);

    void AddData(list<bitset<32>> DataAdd, bitset<7> ID);
    void AddData(bitset<32> DataAdd, bitset<7> ID);

    list<Slave> getListOfSlaves();

    void transferData();

    void flushData(bitset<7> ID);

    void reset()
    {…
    }

private:
    map<bitset<7>, Slave *> listSlaves;
    list<pair<bitset<7>, bitset<32>>> Data;
};

#endif
```

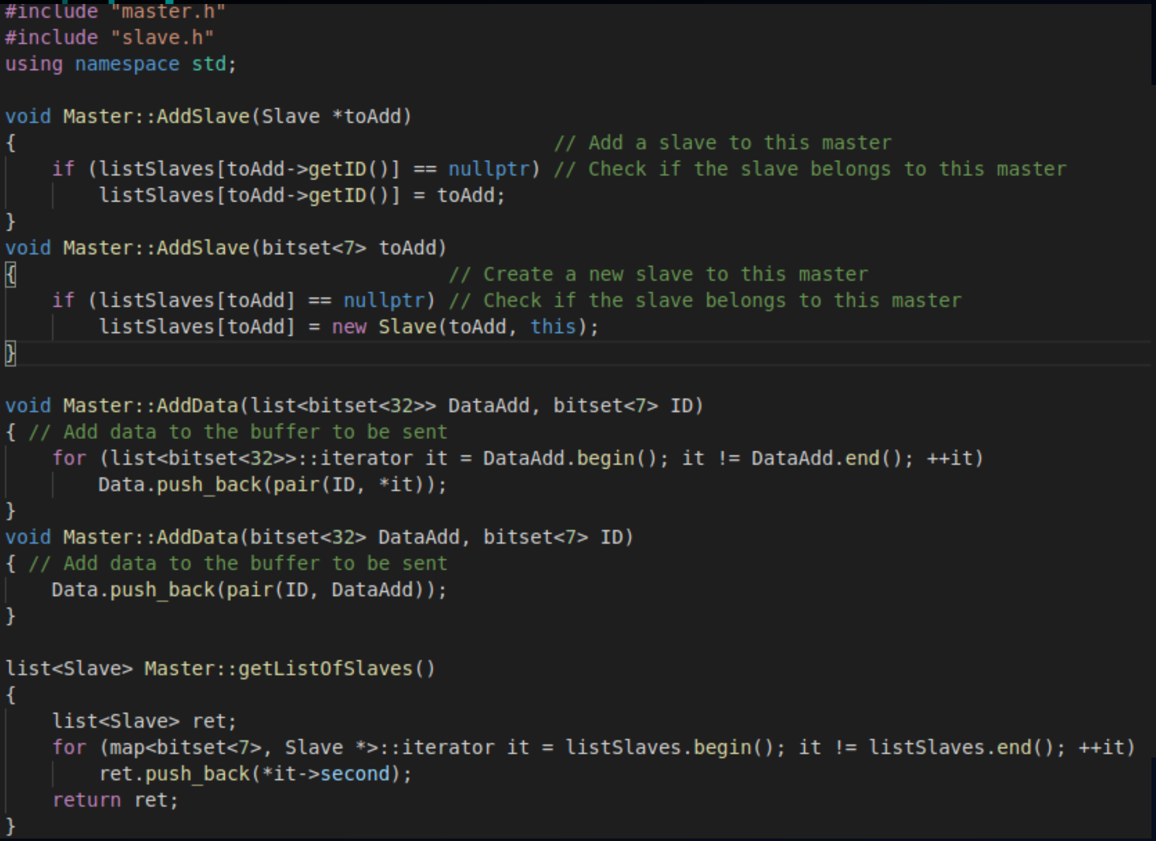


```cpp
#ifndef _SLAVE_ATB_
#define _SLAVE_ATB_
#include "atb.h"
using namespace std;

class Master;

class Slave
{
private:
    Master *master;          // Address of the Master
    bitset<7> ATID;          // ID of the slave
    queue<bitset<32>> Data; // Buffer using FIFO for data alignment

public:
    Slave(bitset<7> ID, Master *m);

    bitset<7> getID();

    void receiveData(); // This function receives all the data on the bus and clears it

    void flushData(); // Ask for all the data to be sent

    queue<bitset<32>> getData(bool clear); // Get the data on the buffer and clear if true

    void reset()
    {
        int size = Data.size(), i;
        for (i = 0; i < size; i++)
            Data.pop();
    }
};
```

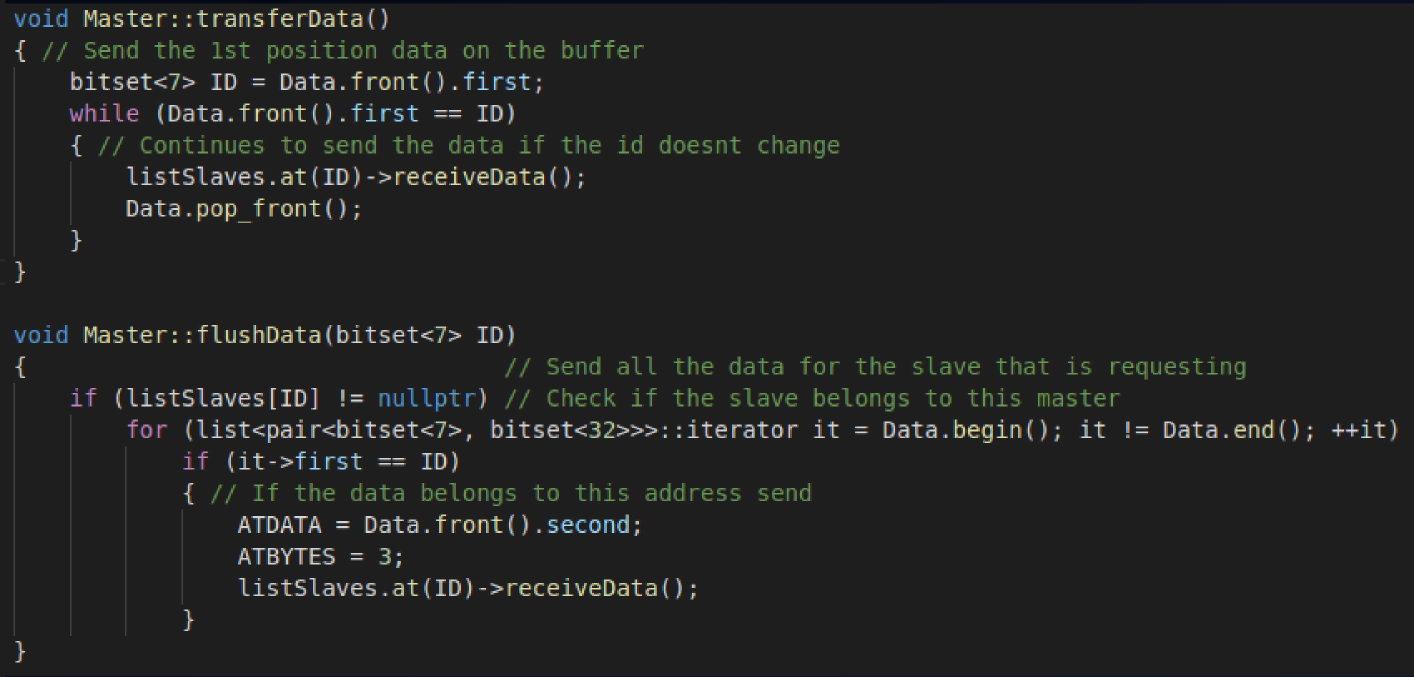


```cpp
#ifndef _CORTEXA9_ATB_
#define _CORTEXA9_ATB_
#include <bitset>
#include <queue>
#include <list>
#include <map>
using namespace std;

static bitset<32> ATDATA;  // Bus width of 32 bits as the Manual Recommends
static int8_t ATBYTES = 0; // How many bytes-1 of data are in the buffer

#endif
```

Master.cpp

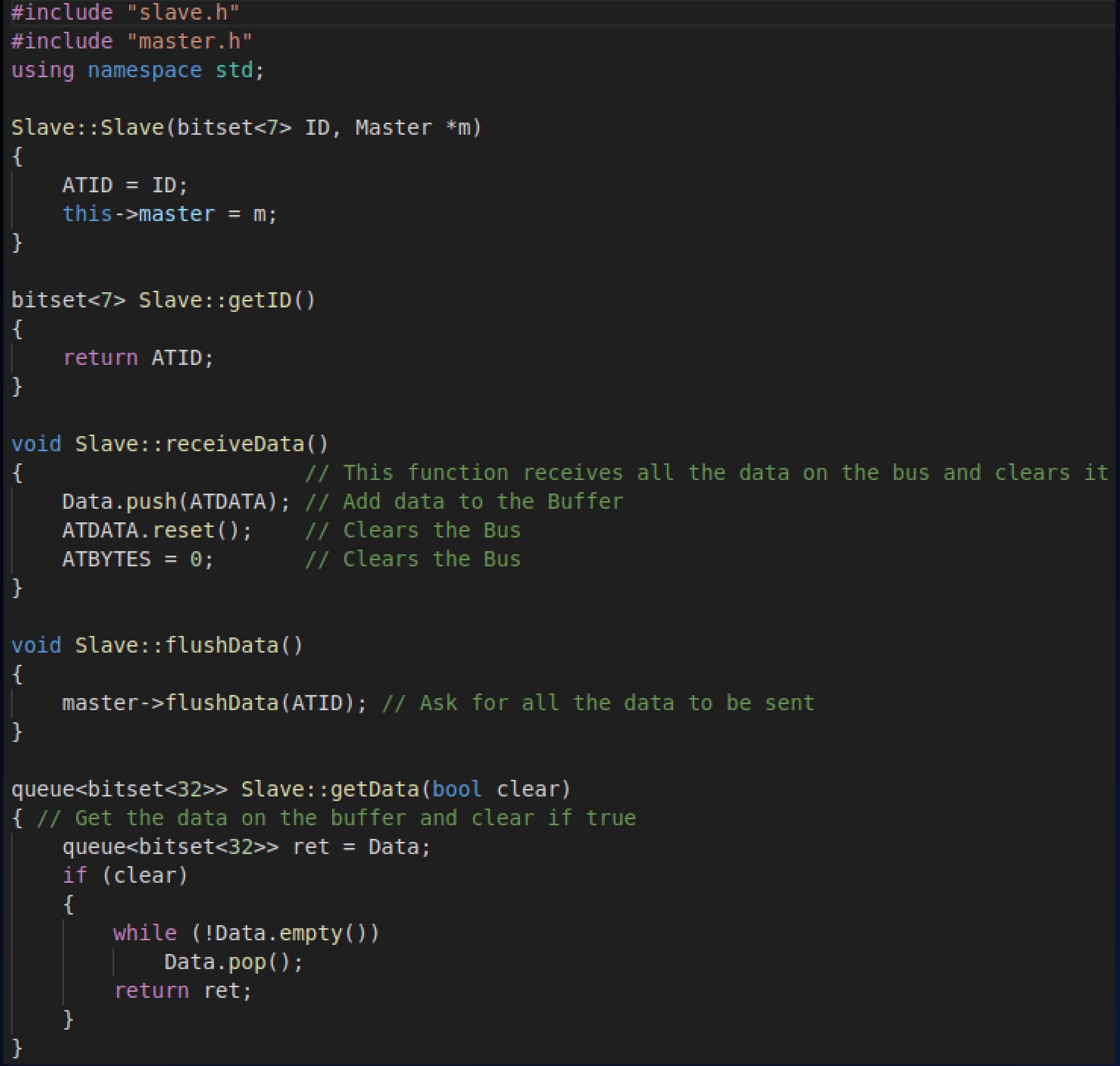


```cpp
#include "master.h"
#include "slave.h"
using namespace std;

void Master::AddSlave(Slave *toAdd)
{                                                // Add a slave to this master
    if (listSlaves[toAdd->getID()] == nullptr) // Check if the slave belongs to this master
        listSlaves[toAdd->getID()] = toAdd;
}
void Master::AddSlave(bitset<7> toAdd)
{                                       // Create a new slave to this master
    if (listSlaves[toAdd] == nullptr) // Check if the slave belongs to this master
        listSlaves[toAdd] = new Slave(toAdd, this);
}

void Master::AddData(list<bitset<32>> DataAdd, bitset<7> ID)
{ // Add data to the buffer to be sent
    for (list<bitset<32>>::iterator it = DataAdd.begin(); it != DataAdd.end(); ++it)
        Data.push_back(pair(ID, *it));
}
void Master::AddData(bitset<32> DataAdd, bitset<7> ID)
{ // Add data to the buffer to be sent
    Data.push_back(pair(ID, DataAdd));
}

list<Slave> Master::getListOfSlaves()
{
    list<Slave> ret;
    for (map<bitset<7>, Slave *>::iterator it = listSlaves.begin(); it != listSlaves.end(); ++it)
        ret.push_back(*it->second);
    return ret;
}
```



```cpp
void Master::transferData()
{ // Send the 1st position data on the buffer
    bitset<7> ID = Data.front().first;
    while (Data.front().first == ID)
    { // Continues to send the data if the id doesnt change
        listSlaves.at(ID)->receiveData();
        Data.pop_front();
    }
}

void Master::flushData(bitset<7> ID)
{                                          // Send all the data for the slave that is requesting
    if (listSlaves[ID] != nullptr) // Check if the slave belongs to this master
        for (list<pair<bitset<7>, bitset<32>>>::iterator it = Data.begin(); it != Data.end(); ++it)
            if (it->first == ID)
            { // If the data belongs to this address send
                ATDATA = Data.front().second;
                ATBYTES = 3;
                listSlaves.at(ID)->receiveData();
            }
}
```

Slave.cpp



```cpp
#include "slave.h"
#include "master.h"
using namespace std;

Slave::Slave(bitset<7> ID, Master *m)
{
    ATID = ID;
    this->master = m;
}

bitset<7> Slave::getID()
{
    return ATID;
}

void Slave::receiveData()
{                       // This function receives all the data on the bus and clears it
    Data.push(ATDATA); // Add data to the Buffer
    ATDATA.reset();    // Clears the Bus
    ATBYTES = 0;       // Clears the Bus
}

void Slave::flushData()
{
    master->flushData(ATID); // Ask for all the data to be sent
}

queue<bitset<32>> Slave::getData(bool clear)
{ // Get the data on the buffer and clear if true
    queue<bitset<32>> ret = Data;
    if (clear)
    {
        while (!Data.empty())
            Data.pop();
        return ret;
    }
}
```

AMBA AHB