µModule Users Manual

Revision: 1.0

Introduction

µModules are small microcontroller modules that perform a specific peripheral function, such as gathering sensor inputs, controlling servo motors and so one. These small modules can be interconnected together and to a central controlling computer through a two-wire bus interface to easily create complex and systems. The central controlling computer talks to the individual modules and orchestrates the operation of the individual modules, by querying their status and sending commands to them. Most modules perform a specific function and in general multiple modules are used in a single system. This way highly customized systems can be implemented with standard µModule components.

A little bit of history

µModules are an off-spring of Modular Circuits' H-Storm project. It uses the PnP side-band bus of H-Storm as it's communication medium. While H-Storm targets much more powerful, single-processor environment and provides a framework for high-speed peripheral interfacing to powerful CPUs, µModules are designed to develop distributed systems, with some local intelligence and (relatively-speaking) low-speed communication requirements. The two approaches though quite different, are not exclusionary. In fact an H-Storm central controlling system is ideal to be used with µModule peripherals to interface to some parts of the external world.

Table of Contents

Legalese

All of the documentation and software included on the µModule project website (including this document) is copyrighted by Andras Tantos and Modular Design. Since the µModule project is closely related to the H-Storm project, the same licenses apply to both.

The documents on this website are covered by various licenses, depending on their content, origin, or intent.

Information on H-Storm in general, like pin-outs, electrical specifications, and so on is free for almost any kind of use, provided that you don't claim that you created that information. This document falls into that category as well. Documents of this kind are covered by a BSD-like license, called H-Storm Open License. According to this license, for example, you can use this documentation to build a gigabit-Ethernet interface card with an H-Storm interface. You can publish the design, if you wish or you can keep it for yourselves if that's what you like. You can sell it or you can build it into your own bigger solutions. Almost the only thing you can't do is to claim that H-Storm was your idea. Content falling into this category is either not marked or marked with this symbol: HSOL

There are also designs (schematics, PCB layouts) presented on the H-Storm project website. These designs are free for non-commercial use i.e. as long as you don't make any money out of them. You are free to re-build the circuits in their original or in a modified form; you can derive your own design from them. You are not required to publish your work based on these designs but you might do so, again, as long as you don't make money off of them. If you however would like to 'productize' some work based on the designs on this website, you have to get my prior written approval for that. I reserve the right to deny such an approval or request some form of compensation for it, so please contact Modular Circuits at sales@modularcircuits.com! This type of information is covered under the H-Storm Non-Commercial License. Content falling into this category is marked with this symbol: HSNCL. Also, any circuit diagram, PCB layout, automatically falls into this category unless marked otherwise.

Source code and binary software presented on this web-page is usually covered under some form of open-license (one of the above two , GPL, LGPL, BSD, etc.) depending on what is the source of the software. The type of license is marked on the web-page by the download link to the particular piece of code and in the source code as well.

And as always: if in doubt, ask us at sales@modularcircuits.com!

This document is covered by the H-Storm Open License (HSOL).

H-Storm Open License (HSOL)

Redistribution and use in source or binary forms, or incorporated into a physical (hardware) product, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in any other form must contain in printed or electronical format the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- All advertising materials mentioning features or use of this technology must display the following acknowledgment:
  This product includes H-Storm technology developed by Andras Tantos and Modular Design.
- Neither the name of Andras Tantos or Modular Design may be used to endorse or promote products derived from or using this technology without specific prior written permission.

ALL THE INFORMATION, TECHNOLOGY, AND SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ANDRAS TANTOS, MODULAR DESIGN OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR TECHNOLOGY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

H-Storm Non-Commercial license (HSNCL)

Redistribution and use in source or binary forms, or incorporated into a physical (hardware) product, with or without modification, are permitted for **non-commercial use only**, provided that the following conditions are met:

- **The redistribution doesn't result in financial gain.**
- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in any other form must contain in printed or electronical format the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this technology must display the following acknowledgment:
  This product includes H-Storm technology developed by Andras Tantos and Modular Design.
- Neither the name of Andras Tantos or Modular Design may be used to endorse or promote products derived from or using this technology without specific prior written permission.

ALL THE INFORMATION, TECHNOLOGY, AND SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ANDRAS TANTOS, MODULAR DESIGN OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING

IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR TECHNOLOGY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Other licenses

No information in this document is covered by either of the following licenses. Some software, hardware or documentation however on the site can be covered by some of these (or another) license. For this reason a reference to these licenses are included here:

The GNU General Public License (GPL):
http://www.opensource.org/licenses/gpl-license.php

GNU Lesser General Public License (LGPL):
http://www.opensource.org/licenses/lgpl-license.php

BSD License (BSD):
http://www.opensource.org/licenses/bsd-license.php

Features

µModules are small, targeted intelligent peripherals to perform a simple interface function. The modules are design to be easily interconnected through a two-wire bus systems to implement complex functions. The primary target for these modules is hobby robotics.

High-level features

- Intelligent modules to perform specific interfacing functions
- Multiple modules can be interconnected in a single system using a simple two-wire bus interconnect
- A central controlling unit (like an H-Storm CPU board an a personal computer) orchestrates to operation of the modules.
- Up to 127 modules can be connected to the same bus, though signal integrity problems usually limit this number to around 10.
- Fully plug-and-play configuration with discoverability over the bus
- Low-cost solutions for fast adoption
- Expandable to accommodate future needs
- Targeted towards hobbyist and low-volume manufacturing
- Open technology
- Free sample designs
- Simple to use and easy to manufacture mechanical design

Technology details

- Two-wire bus interface (I2C compatible)
- Power distribution over the bus interface
- Optional RS-232 serial interface on many modules

- Compatible with the H-Storm side-band configuration and plug-and-play bus
- Low-power operation with 3.3V signaling. Optional 5V signaling is available

µModule bus operation

The µModule-bus uses the standard two-wire interface (similar, but not identical to the I²C™ bus). For the sake of completeness, a short introduction to this bus will be provided. For the full documentation on the two-wire interface can be obtained from Phillips and other vendors.

Two-wire-interface introduction

Device types

The two-wire-interface interconnects multiple devices. Any device on the bus can communicate to any other device. Communication happens in information packets, or transactions. The device, that initiates a communication with another device, is called the **master** device. The master device selects one or more **slave** devices to communicate with. After the selection is made all other devices on the bus remain passive for the rest of the transaction. During the transaction, information can be sent in either direction, i.e. from the master to the slave, or from the slave to the master. A device that is sending information, it is called the **transmitter**, and the device, that receives that information is called the **receiver**. Note that these two notions are orthogonal to each other. A master device can be a transmitter or a receiver so can a slave device be. In fact the direction of the communication can change multiple times during a transaction.

Signal names and signaling levels

This bus contains two signals, a clock (CLK) and a data (DAT). These signals are driven by open-drain drivers and are pulled up to the power rail usually by a passive resistor. Every device on the bus is connected to both of these signals. A master device is required to be able to drive both lines. A slave device is required to drive the DAT line and might optionally choose to drive the CLK signal as well. All devices are required to detect the level of both lines. The signals on the bus are normal digital signals. High levels are close to the digital power supply, low levels are close to ground.

Bit-level signaling

In the idle state of the bus both the clock and the data lines are at high level. A transaction over the bus is started by pulling the data line low, while keeping the clock line high. This is called the **start condition** or **start-bit**. The device that generated the start-condition on the bus will be the master for the duration of the transaction. Any other device that takes a part in the communication is called a slave device. After the start-bit the transaction continues in 8-bit chunks with one **acknowledge bit** between each 8-bit data. Acknowledge bits are low if the device acknowledges the transfer, and high if its not. The direction of the communication can change at each byte-boundary, though usually there's at most one direction change during the transaction. The clock is always supplied by the master, the data-bits are provided by the transmitter which can be either the master or the slave(s). The acknowledge bit is always sent by the receiver, i.e. if the slave provided the data bits, the acknowledge comes from the master, and if the data bits are provided by the master, the acknowledge is generated by the salve. During the transaction the data line is allowed to change when the clock line is low. The data-line has to be stable while the clock-line is high. The end of the transaction is signaled by pulling the data line high while the clock line is high. This is called an **end-condition** or **end-bit**.

Note that the start and stop conditions are signaled by an edge on the data-line, while the clock signal is high: a falling edge signifies a start condition, while a rising edge encodes the stop-condition. Data-bits never change the state of the data-line while the clock signal is high.

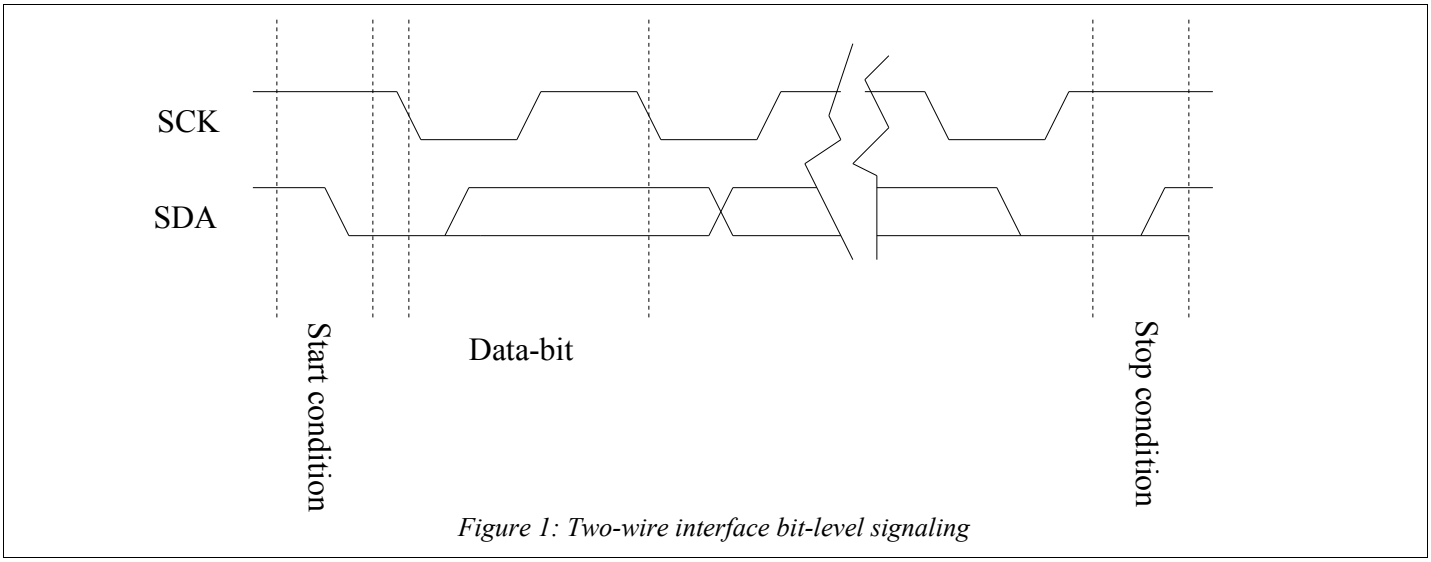

Figure 1: Two-wire interface bit-level signaling

If a stop condition is detected on the line, the current transaction has to be terminated, no matter how many data or acknowledge bits have been transmitted and at what point the communication is. All devices (both masters and slaves) should return to their idle state, however some internal state-changes (like data already written into some memory) are not required to be discarded.

If a start condition is detected without a stop condition preceding it (called are-start condition), a stop-condition is implied. All devices should treat the re-start conditions as if the previous transaction has been completed and a new one has been started.

Note that since all devices drive both the clock and data lines with open-drain or open-collector outputs, the level on the bus will be the logical OR of all drivers outputs. In other words, the clock and the data level will be low if any of the drivers is driving it and will only be high if nobody drives the line. This wired-or behavior of the lines is used for many different functions of the bus and is an essential feature of the interface.

Also note that since the wires are actively drive low, but passively pulled high, the falling edge of the signal can be much faster than the rising edge. The value of the pull-up resistor and the aggregated load capacitance over the wire will determine the maximum operating frequency of the bus. It however cannot be higher than **400kHz**. Some slave devices might not be fast enough to cope with such a high clock rate. These devices can use a technique, called **clock-stretching** to slow down the master during a transaction to a level that the slave device is capable of handling. Clock-stretching is done by the salve by pulling the clock line down during the first half of the bit transmission time. At this period the master is also driving the clock signal low, so the signal level on the wire will not change. However once the master releases the clock line to enter the second half of the bit-period, the clock line will remain low, since the slave device is still driving it. The master device is required to recognize this condition and wait until proceeding to the second phase of the bit-transfer until the slave releases the clock line and it returned to the high level. With this technique a slow slave device can extend the bit-period, and effectively slow down the communication to an acceptable level. Note that, when clock-stretching is used, the duty-cycle of the clock can arbitrary change, even from bit to bit. Also note, that, since the period of the bit-time where the data-line is allowed to change is also when the clock line is low, the slave can use the clock-stretching technique to request more time to prepare the data for transmission in

transmitter mode, not only to delay the reception of the next bit in receiver mode. The last important note here is that there is no upper limit to the time for how long a slave device can extend the bit-time. This means that a slave device can effectively block communication and jam the bus by abusing clock-stretching.

Bytes on the wire are transmitted with MSB first bit-ordering.

Transactions

A transaction is started by a start-bit and ended by a stop-bit or a restart bit and is always initiated by a master device. As discussed before a transaction contains data transferred one byte at a time and each byte is acknowledged by the receiver by an acknowledge bit.

The first byte after a start-bit is always transmitted by the master and contains a slave address and a read/write bit. This byte addresses one or more salves on the two-wire interface bus. The read-write bit, denoted as 'R' on the figure below, is '1' for read transactions and '0' for write ones.

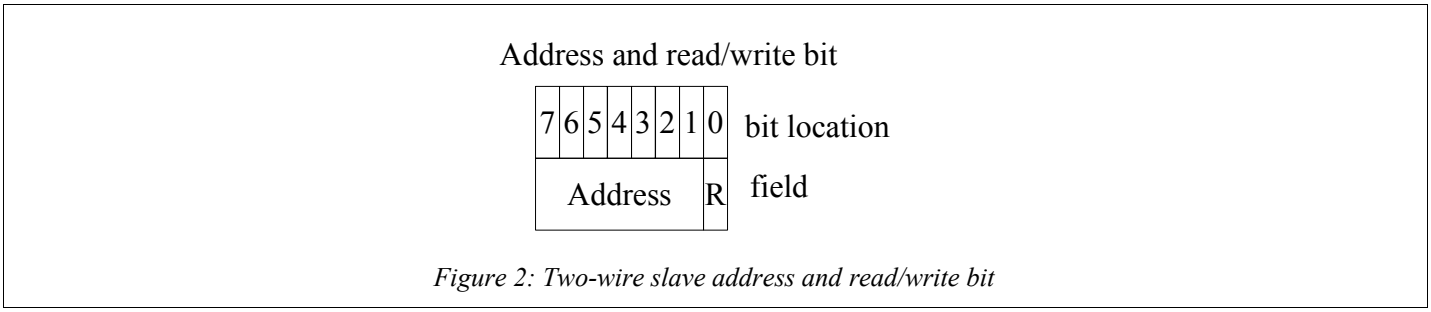

Figure 2: Two-wire slave address and read/write bit

Since all other devices that are not addressed by the transaction become passive and listen for the stop-bit only, the remaining part of the communication depends on the addressed slave. In fact, even the meaning of the read-write bit can be re-defined by the slave device as long as the master understands the communication. Traditionally however, the read-write bit specifies the direction of the communication for the rest of the transaction, and permanently assigns receiver and transmitter roles to the host and the target. If the bit is '1', denoting a read transaction, all following bytes are transmitted by the slave, and received (and acknowledged) by the master. If the read-write bit is '0', it identifies a write transaction and all following bytes are transmitted by the master and received (and acknowledged) by the slave.

Each slave device should have a unique address on the bus so that they can be individually addressed. These addresses are traditionally assigned by Philips to products of licensees of the I2C license. There are other addressing modes (10-bit addressing) as well that are irrelevant to this document. Please see the original Philips I2C specification for details.

One special address, address 0 is reserved. It is called 'general call address'. A device, if encounters a transaction on the general call address that it cannot understand, it should ignore it. If a device does not understand any general-call address transactions, it should completely ignore the general call address. Ignoring a transaction in this sense means, that the device should not change its visible or discoverable state and it should not acknowledge the transaction. This feature makes general call addresses a good extensible interface for broadcast calls. Not many pre-allocated general call address transactions are there and even fewer devices that actually use them.

Acknowledge bit in transactions

Normally the receiver in a transaction should acknowledge each byte of the transaction. A positive acknowledge (logical '0') means that the byte has been successfully received. A negative acknowledge (logical

'1') means that the byte was not successfully received. The master, after the reception of the negative acknowledge should generate a stop-bit (or re-start bit) and terminate the transaction. The master can decide at a later point whether to retry the transaction at a latter point. The master can however decide to terminate a transaction after a positive acknowledge from the slave (in the case when the master is the transmitter) as well. If however the master is a receiver, and it received the last byte of the transaction, it should use a negative acknowledge to signal the slave that the transaction is about to end. The master than is required to terminate the transaction by a stop or re-start bit.

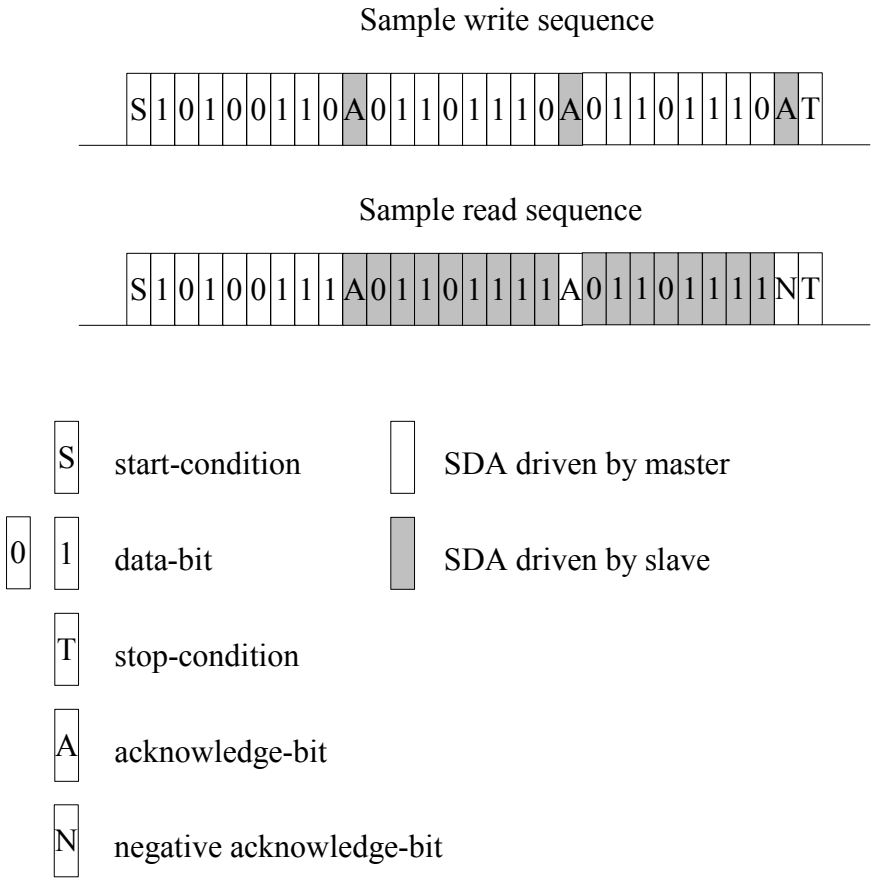

Figure 3: Two-wire interface transactions

Read transactions and re-start bits

If the normal read/write bit semantics is used, it can be seen that no information is transferred from the master to the slave other then the device ID which is used to address the slave in the first place. This means that there is no way to convey what type of information the master tries to retrieve from the slave in the read transaction. In many cases for example a register index should be specified by the master, the index of the register whose value the slave should return. This problem is handled by a trick in the two-wire interface: a first write transaction is used to set the request information on the slave and a second read transaction is used to retrieve the data associated to the request type. As an example, the master could start a write transaction, in which it transmits the register index in the second byte of the transaction to the slave, but terminates the transaction before any value to the register would be written. After that, the master would issue a read transaction to which the slave would return the content of the previously indexed register.

This technique however has a drawn-back: since the slave uses information from the previous transaction to fulfill the request of the current transaction, there is a dependency between multiple transactions in the process. If multiple masters exist on the bus, there is a chance, that in between the read and the write transaction from master 'A', master 'B' gets control of the bus, and overrides that piece of information in the slave that is needed for the second transaction. If that happens, the read transaction from master 'A' would return invalid data, since the information dependency between the two transaction has been violated.

The solution to this problem is the use of re-start bits between the write and the read transactions. Since there was no stop condition in the whole process, the bus never returned to the idle state, only after the second transaction. Since a master can only start a transmission if the bus is idle, the dependency between the two transaction can be guaranteed.

Of course there's no reason why more than two transactions could not be linked together this way if it is important to guarantee that no other master can interact with the slave until the last transaction has been finished. Using re-start conditions between transactions is the way in the two-wire interface to lock the bus to a single master device. Of course care must be taken if this technique is used not to lock the bus for extended periods of time since that can prevent other masters to get control of the bus and complete their transactions.

Arbitration

One of the unique features of the two-wire interface, described here is the simple conflict-resolution and arbitration between multiple transmitters on a single bus. Since both masters and slaves can be transmitters this conflict resolution works between multiple masters and multiple slaves as well. It is not possible to have a conflict between a master and a slave transmitter so arbitration for that scenario is not needed.

Arbitration uses the wired-or logic of the data line of the bus. This feature ensures that in a case of multiple drivers of the line, the ones that drive it to low, will win. The wire will always have a valid logic level, even in the case of a collision. Now, a transmitter can detect a collision by monitoring the data line. If the line is at the level to which the device tried to set it to, it considers the transmission not being collided and continues the transmission. If however it detects a different logic level, it concludes that a collision occurred and it lost it's battle for bus ownership. At that point the transmitter detaches from the bus, and aborts its transaction. Now, from the receivers perspective, the data on the line was still valid, since there was at least one transmitter that won the arbitration. That transmitter will finish the transmission of the current byte, and receive the acknowledge from the receiver. Note, that if multiple transmitters are transmitting the same data, the collision will not be detected. That is true, however also note, that since they are transmitting the same data, the receiver of the information still got the right message, so the purpose of the communication (information transfer) was achieved. Note also, that the logic level of the data-line for data-bits is allowed to change while the clock-line is low. Thus the check for whether the intended transmitted data made its way to the line must happen when the clock returned to high.

If a master wants to initiate a transaction, it first must check the bus for idle, and only start the transaction if that condition is satisfied. The only way two masters can collide thus is if they start a transaction at the same time. Their start-bit will be transmitted and as long as they transmit the same address and (optionally data bytes for write transaction) are the same, all is fine. Once, however they try to transmit different data, the one that first transferred a '0' data-bit will win. The other master will detect the collision and will abort its transaction (but will not generate a stop-condition), wait until the bus returns to idle, and retry the transaction. The master that won the arbitration will continue and successfully finish the transaction since it's transmission has not been corrupted. Two (or more) masters that are not arbitrated yet will not have collisions on the clock line, they will use the previously mentioned clock-stretching technique to synchronize to the slowest of them all.

In case a master addresses more than one slave at a time for a read transaction (using the general call address or in a case of a device address collision), again, the same data-line arbitration will make sure only one of the slaves will complete the transaction, provided they try to transmit different information.

Note, that write-transactions from a single master to multiple slaves (through the general-call address or due to address-collisions) will not collide, no arbitration will happen, and all addressed slaves will receive and acknowledge the whole transaction.

There are a couple of conditions, where arbitration is not allowed:

- Between a re-start, or stop condition and a data-bit
- Between a re-start and a stop condition.

If it cannot be guarantied that these conditions will not occur, other means of arbitration between masters must be employed in the system. Since slaves never generate start, re-start or stop conditions, arbitration between slaves is always allowed.

Note, that if a master contains a slave function as well, and it started a transaction, but lost due to arbitration it is possible that the winning master tried to address the slave function. Special care must be taken so to the design of combined master-slave devices: the data transmitted by the master must not be ignored by the slave device, and upon a lost arbitration of the master, the device must immediately switch to slave-receiver mode, to catch the rest of the transaction. Since the arbitration can be lost at any bit-position this transition from master to slave mode can happen at any time not only on byte-boundaries.

Two-wire interface in µModules

The above described, rather generalized description applies to many two-wire interface devices and intended as a general introduction to the nomenclature, techniques and technologies used in this documentation. µModules use the previously described interface to interface to the central controller. Each µModule has two different operating modes. The PnP discovery mode is used to scan the bus, locate µModules on it, and assign device addresses to them. These addresses are latter can be used to talk to those devices and take the place of the device address in the second byte of each transaction. The module-specific mode starts, once the PnP discovery and address-assignment has been finished. The operations and communication in this mode depend on the µModule, though many modules share a rather similar register-bank interface.

Differences to other popular two-wire interface implementations

There are a couple of noteworthy differences between the two-wire interface implementation used by the µModule and other manufacturers' implementations:

- µModules use use byte-boundary slave arbitration for the PnP protocol. Other implementations often do not allow collisions between multiple slaves. Since the collisions-detection is done on the byte-boundary, only a limited number of symbols from the 256 available ones can be transmitted until the arbitration is completed. Slave-arbitration is not allowed for non PnP operations. The reason for not using the bit-level arbitration in slave-modes is that the two-wire interfaces in the microcontroller mostly used for µModules cannot handle arbitration in slave-transmitter mode. Arbitration is thus emulated in software but that emulation is not capable of disabling the transmitter in the middle of a byte. The employed solution is completely backward compatible with bit-level arbitration, though requires the use of a limited symbol-set.
- The modules use LVCMOS level of interfacing, with optional CMOS-level interfacing. The pins are not guarantied, but usually are 5V tolerant when operated in LVCMOS mode.

- The general-call address is used for the PnP protocol in a way that's not specified for other implementations. This however should not cause interoperability issues with those solutions for two reasons. One, general-call address is hardly ever used, second, the specifications requires to ignore unknown general-call address transaction.
- µModules need to do some processing after the stop-bit for the current transaction finished before they are ready to accept the next transactions' start-bit. At this time however they can't use the clock-stretching technique to request extra time from the master, since the clock line is already inactive, and the bus is idle. For that reason, a minimum idle period is specified for µModules. This might cause some interoperability issues with other two-wire interface masters, though those problems could most likely easily be solved by software modifications
- The PnP discovery protocol is not compatible with the SMBus ARP protocol. The reason for that is that the microcontrollers used in most µModules are not capable of handling the ARP protocol that the SMBus specification defines. Most importantly, they are not capable of handling two valid salve addresses at the same time, neither of ones being the general-call address.

The PnP discovery, enumeration and Address Assignment protocol (TWI_AAP)

The PnP discovery, enumeration and Address Assignment protocol is used to locate, identify and configure various µModule devices on a two-wire interface bus. The protocol uses the general call address to address all or some of the modules on the bus, and slave arbitration to sort out the bus contention between multiple slaves trying to answer to calls on the general call address. During the process, each module can be identified, it's device and device class ID can be retrieved and a device address can be assigned to it, that latter on can be used to access the device using standard two-wire interface processes. It is also possible to assign a device address permanently to the µModule, so that the address assignment process need not to be re-executed upon every power-on.

The TWI_AAP protocol is based on commands. All command is started by the general call address and the proper read/write bit. If the read/write bit identifies a write transaction, the second byte in the transmission is the command code, followed by optional additional bytes describing parameters to the command. If the read/write bit identifies a read transaction, information from the slave(s) is returned to the master. The type of returned information depends on the last command sent to the slave(s). In the case when more than one slave received and interpreted the read command, bus contention occurs and the slave bus-arbitration protocol makes sure that at least one slave will complete the transaction without a collision, and that the data that the master received is valid. The construction of the commands and the type of the returned data guarantees that by the end of the transmission, exactly one slave remains active, and it will be winner of the arbitration process.

To work around the limitations in the microcontroller used in µModules, the protocol is designed such that byte-boundary arbitration can be used. This requires some parts of the communication to be specially encoded. The design guarantees that by the time this specially encoded part of the read transaction finished exactly one slave device remains active on the bus.

Every µModule device can have two main states with regards to the TWI_AAP protocol. They may or may not have a device address assigned to them. This device address can be assigned permanently, in which case upon power-on the device starts from the address-assigned state with the permanent device address. If no permanent device address is assigned to the µModule, it start up in the address-unassigned state. The reaction of the device to any TWI_AAP protocol command depends on its state.

The protocol identifies the following commands:

Command symbol	*Command code*	*Explanation*
`TWI_AAPCmd_Start`	`0x20`	Signals the start of an address assignment process
`TWI_AAPCmd_End`	`0x21`	Signals the end of an address assignment process
`TWI_AAPCmd_ResetDevice`	`0x22`	Resets the state-machine to the address unassigned state.
`TWI_AAPCmd_ResetToPermAddress`	`0x23`	Resets the state-machine to the address unassigned state if there is no permanent device address assigned to the device
`TWI_AAPCmd_GetConfig`	`0x24`	Gets the configuration record from the device(s) that don't yet have an assigned address.
`TWI_AAPCmd_AssignAddress`	`0x25`	Assigns an optionally permanent device address to the device

IDs, GUIDs and the configuration record

Each µModule has three IDs associated to it. The **device ID** or **DID** identifies the function of the device. Examples would be a 32-channel servo controller or a tilt sensor or a 20A H-Bridge. The **class ID** or **CID** identifies the device class, this module belongs to. Examples would be servo driver class, sensor class, motor driver class. The third ID is a factory-assigned **unique ID** or **UID**, that is used only to identify and differentiate between µModules on the same bus.

The DIDs and UIDs are 128-bit globally unique identifiers, or GUIDs. The UID is a 48-byte long identifier, that is generated from a 128-bit GUID as well, with a process described below. The most important property of these IDs is that GUID generation techniques exist with which unique IDs can be generated on isolated machines, such that the probability of colliding IDs being generated is practically 0. This way, new class, device and unique IDs can be generated and assigned to devices without a central database and administration of the already assigned IDs.

Many techniques exist to generate such numbers. OS-es from the Windows family have built-in APIs for generating GUIDs. On other OS-es if no such built-in API is provided, a generation algorithm can be implemented in application software.

It has to be noted, that these long GUIDs are used only in this AAP protocol to uniquely identify each module. After such identification is done, a 7-bit address is assigned to each module in the system, and all following communication to the module is done by using that 7-bit device address. This way the additional overhead of using long IDs is minimal and is restricted to system initialization time.

Out of the three IDs, the UID is used to arbitrate between slaves, and in order to facilitate byte-boundary arbitration, it is specially encoded. The generation of UIDs is the follwing:

First, a 128-bit GUID is generated the same way as DIDs and CIDs are. Than, this 128-bit ID is extended to 144 bits, with '0' bits at the LSB bits. After that the resulting number is cut into 48 3-bit chunks. Each 3-bit chunk is then replaced with an 8-bit code, selected from the following table:

3-bit chunk	*assigned code*	*symbol*
000b (0x00)	00000000b (0x00)	`num0`
001b (0x01)	00000001b (0x01)	`num1`
010b (0x02)	00000011b (0x03)	`num2`
011b (0x03)	00000111b (0x07)	`num3`
100b (0x04)	00001111b (0x0f)	`num4`
101b (0x05)	00011111b (0x1f)	`num5`
110b (0x06)	00111111b (0x3f)	`num6`
111b (0x07)	01111111b (0x7f)	`num7`

As it can be seen, that the value of the chunk determines the number of '1' in the code. It also can be seen that code 0xff (symbol `num8`) is not used. That code is reserved for the special value, of 'unassigned ID'. With this transformation, the original 128-bit ID has been replaced with the 48-byte UID sequence.

A special UID (`UniqueIDUnassigned`) is reserved for unassigned UIDs:

```
num0,num1,num2,num3,     \
num4,num5,num6,num7,     \
num8,num7,num6,num5,     \
num4,num3,num2,num1,     \
num0,num1,num2,num3,     \
num4,num5,num6,num7,     \
num8,num7,num6,num5,     \
num4,num3,num2,num1,     \
num0,num1,num2,num3,     \
num4,num5,num6,num7,     \
num8,num7,num6,num5,     \
num4,num3,num2,num1;
```

Notice the use of `num8` symbol in the sequence. No valid UID can have that number in it. Also, due to the arbitration process, if a device with this invalid UID is on the two-wire interface, it will loose against any devices with a valid UID. The result of that is that first all valid device IDs will be enumerated, leaving the invalid ones to the end. At that point, the central controller can detect that there are device(s) with invalid UIDs on the bus, and decide the corrective action. Since those devices have no device addresses assigned to them, they will not affect the operation of the rest of the µModules, and the bus sill remains operational.

The configuration record contains these three IDs in the following order:

```
struct sConfigRecord {
     uint8_t UniqueID[UniqueID_Size];
     GUID ClassID;
     GUID DevID;
```

```
};
```

Assigned IDs

On the top of the `UniqueIDUnassigned` above, the following DIDs and CIDs have been assigned so far:

CIDs:

Symbol	*Value*
`DevClassReserved`	`{00000000-0000-0000-0000-000000000000}`
`DevClassUnassigned`	`{D8745C7D-71A5-4977-A989-2E1EF853CCB8}`
`DevClassServoController`	`{15D37F46-A4F2-416d-8157-FAB7D1A66BE8}`
`DevClassHBridge`	`{D3C8787E-DCE8-445b-B1C7-86C2BF85D144}`

DIDs:

Symbol	*Value*
`DevReserved`	`{00000000-0000-0000-0000-000000000000}`
`DevUnassigned`	`{E59F78CE-6328-4316-B1DE-F0AA5E45F34F}`
`DevUmServo32`	`{E3470D73-9110-4d61-915E-916FF8A89BA6}`
`DevUmHBridge`	`{CF232A58-497F-4bba-9E13-D9AB7938B418}`

The Address Assignment Process

1. At the beginning of the address assignment process the central controller have to make sure that no other controller has such a process in progress. This means that controllers have to make sure that no other host issued a `TWI_AAPCmd_Start` without its corresponding `TWI_AAPCmd_End` command. Once that condition is fulfilled the controller issues a `TWI_AAPCmd_Start` command to signal the start of the assignment process. This command is ignored by all slaves, but is recognized and interpreted by all masters. The command contains the master UID, a 48 byte unique identifier. This is not directly used in the communication, it's purpose is to sort out any bus-contention if more than one masters decided to start the process at the same time. The controller that successfully finished the transaction is the one that considers itself the driver of the discovery. All other controllers will invalidate their internal address to device mapping and will wait for the end of the discovery process, signaled by the `TWI_AAPCmd_End` command.
2. Next the central controller issues the `TWI_AAPCmd_ResetDevice` or `TWI_AAPCmd_ResetToPermAddress` command. The `TWI_AAPCmd_ResetDevice` command will be interpreted by all µModules in the system, independent of their state. The `TWI_AAPCmd_ResetToPermAddress` command however will only be interpreted by µModule that don't have an assigned permanent device address. All modules that interpreted the command will reset their state which means, they delete their assigned device-address. Modules with a valid temporary device address will not respond to the `TW_AAPCmd_GetConfig` command, unless their device address was specifically specified in that command. After the processing of this command however, all µModules (or in the case of the `TWI_AAPCmd_ResetToPermAddress` command all modules with no permanently assigned device address) will listen to all general `TW_AAPCmd_GetConfig` commands. This is the reason for this to be the first command to be issued by the central controller: to make sure that all µModules will listen to the following TWI_AAP commands.

3. If there are non-PnP devices on the bus, or the controller issued a `TWI_AAPCmd_ResetToPermAddress` command, there might be devices on the bus that still have an assigned a device address. Thus the controller first have to scan the bus for any occupied addresses. The controller can use an empty write transaction (address+write and a stop bit) for that. If there is a device listening on the transmitted device address, it will acknowledge the transaction.
4. The controller issues a `TWI_AAPCmd_GetConfig` command with each learned device address to grab the configuration record. If the device with the detected device address is not a µModule, it will not answer the `TWI_AAPCmd_GetConfig` command.
5. The central controller issues a read transaction on the general call address. If a µModule successfully received the previous transaction, it will answer with its configuration record. If no µModule have the previously transmitted device address assigned, no modules will acknowledge the transaction.
6. The controller repeats step 4-5 for all learned addresses. With this process the central controller gained knowledge of all pre-assigned device-addresses and whether that device-address is assigned to a µModule.
7. The central controller issues the first `TWI_AAPCmd_GetConfig` command with 0 as the target device address. This will be received by all µModule which don't have an assigned device address.
8. The central controller issues a read transaction, to which all µModule that received the `TW_AAPCmd_GetConfig` command will answer. They transmit their configuration record and use the two-wire interface bus-arbitration protocol to decide which one to answer the request. Since the first item in the record is the UID, that is the one that gets transmitted first. The UID is guaranteed to be unique and is encoded in such a way that byte-boundary arbitration can be performed on it. By the time the UID has been transmitted, only a single device is left on the bus, so the rest of the configuration record can be transmitted without any slave-side arbitration.
9. The central controller chooses an unused device address to the µModule whose configuration record it received, and issues a `TWI_AAPCmd_AssignAddress` command to assign this address to the device. The central controller at this point has the option to send the assigned address as a permanent device address, that the µModule will preserve between power-cycles or as a temporary one, that the device will keep only until the next power-down, reset, `TWI_AAPCmd_ResetDevice` or `TWI_AAPCmd_ResetToPermAddress` command. The central controller uses the received UID to identify the device its assigning the address to. It transmits this UID as part of the command. All µModule who don't have yet an assigned device address will listen to this command, compare the transmitted UID with their own, and disconnect and abort the transaction if they find a difference. By the time the whole UID has been transmitted, only a single µModule is on the bus, listening for the rest of the command, the one that successfully transferred its configuration record in the previous step. That module will receive the assigned device-address that is transmitted as the last byte of the command. A special form of the command is used to clear any previously assigned permanent device address.
10. The µModule, that successfully received the device-address, will optionally store it permanently, if it was commanded so, and will use the assigned address for all further transactions on the bus. It will use the newly assigned device-address, even if it has a permanent address and the two does not match. It will not listen to `TW_AAPCmd_GetConfig` commands on the 0 address any more, only on its newly assigned address. It will start listening to transactions on the assigned device-address on which the normal communication with the µModule happens. The form of the communication over that address is module-specific. For that module the PnP discovery sequence have been finished and normal operation started.
11. The central processor iterates through steps 7 to 10 until it gets no response to the `TW_AAPCmd_GetConfig` command. At that point it can assume that all µModule on the bus have been

identified and assigned soft addresses. It can start communicating to the identified µModules using their assigned addresses and consider the address assignment phase done. In the case it receives a configuration record with the `UniqueIDUnassigned` UID in it, it can assume that all valid µModule have been initialized, but there is at least one µModule with an invalid UID. At that point the controller can issue an error message, or can just ignore those modules.

12. The central processor issues a `TWI_AAPCmd_EndPnPEnumeration` command to signal the end of the PnP discovery process. This command is ignored by all slaves but interpreted by all masters. Those masters should start a PnP discovery process to re-establish their internal address map. That process is discussed latter.

The following sections will discuss each command commands in detail.

Start Address Assignment command (`TWI_AAPCmd_Start; 0x20`)

Command format:

byte	0	1	2..50
value	0x00	0x20	0x00-0xff
explanation	General call address + write	`TWI_AAPCmd_Start`	UID of the controller

Central controller behavior:

The central controller should clear any internal module-address assignments. After the execution of the command no µModules will have an assigned address, so any previous assignments will be invalid, if used. All controllers should not only be able transmit this command but listen to and process it. If another controller issues a command, the controller should invalidate it's internal address map, wait for the corresponding `TWI_AAPCmd_End` command and upon reception of it, should start a PnP discovery process to re-establish its address-map.

µModule behavior:

µModules should ignore this command. It is not used to communicate useful information to the modules but to co-ordinate discovery and address-assignment processes between multiple controllers.

Reset PnP engine command (`TWI_AAPCmd_End; 0x21`)

Command format:

byte	0	1
value	0x00	0x21
explanation	General call address + write	`TW_AAPCmd_End`

Central controller behavior:

The central controller should send this command after it finished the address-assignment protocol. All other controllers should listen to this command, and start a PnP discovery process upon the reception of the command to re-establish their internal address-map.

µModule behavior:

µModules should ignore this command. It is not used to communicate useful information to the modules but to co-ordinate discovery and address-assignment processes between multiple controllers.

Reset PnP engine command (`TWI_AAPCmd_ResetDevice; 0x22`)

Command format:

byte	0	1
value	0x00	0x22
explanation	General call address + write	`TWI_AAPCmd_ResetDevice`

Central controller behavior:

The central controller should clear any internal module-address assignments. After the execution of the command no µModules will have an assigned address, so any previous assignments will be invalid, if used.

µModule behavior:

All µModules should interpret this command, independent of their state. They should clear their currently assigned device-address, and no longer answer to transactions on that previous device-address. They should only listen to transactions on the general-call address. If a module have a permanent device-address, it should not clear it. It should, however still be ready to be assigned a new (temporary or permanent) device address and eventually accept transactions over that new address. This means that the permanent device-address and the actual device address are not necessarily the same. Either of them can be valid or invalid at any given time. The main use of the permanent address (if valid) is to initialize the actual device address with it upon power-on of the module.

Reset PnP engine if no permanent address command (`TWI_AAPCmd_ResetToPermAddress; 0x21`)

Command format:

byte	0	1
value	0x00	0x23
explanation	General call address + write	`TWI_AAPCmd_ResetToPermAddress`

Central controller behavior:

The central controller should clear any internal module-address assignments. After the execution of the command some µModules will not have an assigned address, so any previous assignments will be invalid, if used.

µModule behavior:

All µModules should interpret this command, independent of their state. They should clear their currently assigned device-address to the permanent default, if there is one. If no permanent address is assigned to the module, its actions to this command is the same as to the in the `TW_AAPCmd_ResetDevice` command. If

the device had a permanent address, it will listen to transactions on that address as well as on the general call address.

Get ID command (`TWI_AAPCmd_GetConfig; 0x22`)

Command format:

byte	0	1	2
value	0x00	0x24	0x00-0xfe
explanation	General call address + write	`TWI_AAPCmd_GetConfig`	device address

byte	0	1..80
value	0x00	0x00-0xff
explanation	General call address + read	configuration record contents from µModule(s)

Central controller behavior:

The central controller issues this command to retrieve the configuration record of one or more µModules. The controller should issue a read transaction over the general call address after this command to receive the configuration record. The configuration record contains 80 bytes of information. The controller probably needs to store the retrieved information, but other system implementations are possible as well. The central controller can use the data contained in the configuration record to learn the function of the module (from the CID and DID members) and to assign a device-address to the module (using the UID member).

The command has two formats: if a valid device address is used, than at most one µModule will respond to the call. This operating mode can be used to discover µModule on the bus that already have an assigned device address. It can be used to scan the bus without re-configuring it, in the case for example when a secondary master wants to learn about the state of the bus. The other format is when the device address is 0. That command will address all µModule that doesn't have a currently assigned (temporary) device address. They all will start transmitting data in response to the following read transaction and use slave-side byte-boundary arbitration to sort out bus ownership by the time the first 48 bytes of the configuration record has been transmitted. This usage of the command can be used to learn about modules on the bus that need configuration and is the main vehicle for the PnP operation on the two-wire interface.

In systems where multiple masters exist on the bus, the first (write) and the second (read) transactions has to be delimited by a re-start condition to prevent loss of bus-ownership.

If more than one masters are capable of assigning device-addresses on the bus, it is advised to use re-start conditions for all the transactions in the loop for address assignment (step 6 to 9 in the previous flow description).

µModule behavior:

All µModules should interpret this command, independent of their state. They should however respond to the second read transaction only if the transmitted device address (2nd byte) in the write transaction matches the currently assigned device address or if the transmitted device address is 0 and the module doesn't have a temporary assigned device address. If the module doesn't have a temporary assigned device address, but has a permanent device address, it should still answer to the read transaction.

If a module answers to the read transaction, it has to transmit its configuration record. The module has to be aware of the fact that other modules might answer the command as well, so slave-side bus arbitration is used to determine bus-ownership. The content of the configuration record however makes it possible to use byte-boundary arbitration on the first 48 bytes of the transmitted data. It is also guaranteed that the first 48 bytes will result in the full arbitration being completed with only a single transmitter remaining on the bus.

Assign device address command (`TWI_AAPCmd_AssignAddress; 0x23`)

byte	0	1	2..50	51
value	0x00	0x25	0x00-0xff	0x00-0xff
explanation	General call address + write	`TWI_AAPCmd_AssignAddress`	UID	device address + permanent bit

Central controller behavior:

The central controller issues this command to assign a temporary or permanent device address to a µModule. The controller uses the 48-byte device UID to uniquely address a single module. It than transfers the device address. The device address occupies the upper 7 bits of the last byte. If the LSB is 0, it indicates that the assigned address is temporary. If the LSB is 1, it indicates that the assigned address is permanent. In the latter case, the device will store the address in a non-volatile memory and will use that device address on the next power-on. It will also use the assigned address as the currently assigned temporary device address.

The command can also be used to selectively delete the temporary or permanent device address on a single µModule. In that mode, the transmitted address is 0. Again, the LSB specifies whether to clear the temporary assignment or both the temporary and the permanent one.

The bit-assignments of the address byte is the following:

MSB							***LSB***
Device address							Permanent

µModule behavior:

All µModules should interpret this command, independent of their state. They should however respond to the command if they see their UID transmitted in the command payload. If they find a mismatch, they should not acknowledge the transaction after the mismatch is detected and stop interpreting the rest of the command. Since UID are unique on the system, it is therefore guaranteed that by the time the whole UID is transmitted, only a single µModule will be listening to the rest of the transaction.

If the module found a matching UID, it should receive the address in the last byte of the transaction. If the permanent bit is set, it should store the assigned address in non-volatile memory. It should also use the newly assigned address as its temporary device address for any further transactions. If the special device address 0 is assigned, it should clear its temporary (and if specified permanent) device address.

The PnP discovery process

This process is used by central controllers who have detected that another central controller issued an Address assignment process. This detection is done by listening to `TWI_AAPCmd_Start` commands. All central

controllers (except for the one that completed the Address Assignment process) should start a PnP discovery process every time they see a TWI_AAPCmd_End command. Even so, they must listen to both commands since after the start command they cannot trust their internal address map to be valid and should refrain from addressing any µModules until they could re-establish that internal map.

1. If there are non-PnP devices on the bus, there might be devices on the bus that still have an assigned a device address. Thus the controller first have to scan the bus for any occupied addresses. The controller can use an empty write transaction (address+write and a stop bit) for that. If there is a device listening on the transmitted device address, it will acknowledge the transaction.
2. The controller issues a `TWI_AAPCmd_GetConfig` command with each learned device address to grab the configuration record. If the device with the detected device address is not a µModule, it will not answer the `TWI_AAPCmd_GetConfig` command.
3. The central controller issues a read transaction on the general call address. If a µModule successfully received the previous transaction, it will answer with its configuration record. If no µModule have the previously transmitted device address assigned, no modules will acknowledge the transaction.
4. The controller repeats step 1-3 for all learned addresses. With this process the central controller gained knowledge of all assigned device-addresses and whether that device-address is assigned to a µModule. This completes the PnP discovery processing

Note that if there are multiple central controllers (more than two) than a number of them will start the PnP discovery process after an address assignment process by one of them. This will lead to multiple, conflicting transactions on the bus, so master-side arbitration will be used a lot. Also, it is paramount that between step 2 and three no stop condition is issued to the bus, so that the dependency between the write transaction of step 2 and the read transaction of step 3 is maintained.

If central controllers are capable of sniffing the bus, i.e. passively listening to and interpreting communication on the two-wire interface by other masters, they can re-build their internal address mapping just by watching the Address Assignment process of the other controller. In that case no PnP discovery process needs to be executed after the `TW_AAPCmd_End` command is received.

Of course if it can be guaranteed that there's only one central controller on the bus, as it is the case for most of the applications, the above described process can be greatly simplified. Still, if possible, a complete implementation of the protocol is recommended so that additional migration to multi-master scenarios will be easier.

The register-bank protocol

The register-bank protocol is used by many µModules as the main communication protocol with the central controller. This protocol is used after the PnP Address Assignment protocol has been finished, or in case of permanently assigned address, right after power-up. This protocol uses commands issued to the assigned address of the device and is never used over the general call address.

The protocol is closely modeled after the memory access protocol that's used by many EEPROM and FLASH TWI modules.

The protocol provides a set of registers for the central controller to access. Each register has an address associated with it. Registers can be written using a single write transaction, and read by first setting the address with a write transaction, followed by a read transaction. The µModule device maintains an internal address pointer. Every write transaction updates this pointer. Every time a register is read or written, the address pointer is incremented. This provides fast access to multiple registers in a single transaction: once the address

is set, multiple consecutive registers can be written in the same write transaction or be read in the subsequent read transaction. If it can be guaranteed that the address pointer internal to the µModule is in a known state it is not required to issue a write transaction before a read transactions, this technique however is not recommended, especially in multi-master environments.

There are two addressing modes, defined: 8-bit and 16-bit addressing. The main difference between the two is the maximum size of the addressable register-bank. Any given µModule implements only one of these two protocols. Consult the documentation of the µModule for details.

The registers and their layout in the register bank is module-specific however it is common to use the last register in the bank (Address 255 for 8-bit addressing and 65535 for 16-bit addressing) as a command register. The commands themselves are again module specific but a common set of commands is specified. This command register can be used to store for example the current configuration of the µModule as it's startup configuration.

8-bit addressing

In the 8-bit addressing mode, all register-addressed are 8-bit long. This gives the register-bank a maximum length of 256. The actual register file in a given µModule can be lass than this number. Whether the module implements wrap-around for the address register is module specific. It is also module-specific if the last register (address 255) is implemented as a command register and if that address wraps around into the address space of the other registers.

The protocol identifies two transactions.

Write transactions

byte	0	1	2..n (optional)
Value	(0x00-0xfe) & 0xfe	0x00-0xff	0x00-0xff
Explanation	Device address + write	Transaction start address	Transaction data

This transaction starts with the address of the device and the read/write bit cleared to identify a write transaction. It is followed by a register address in the second byte. At this point the addressed µModule updates its internal address register. If subsequent bytes are followed, these bytes a re written to the register in the bank, pointed by the address register, after which the address-pointer is incremented. This process follows until a stop, or restart condition terminates the transaction. It is important to note, that if 0 data bytes follow the register-address, no updates to the register-bank takes place, but the address-register is assigned a new value. This is the preferred way of updating the address register to a known state prior to a read transaction. It is also important to note that if such a write transaction is followed by a read transaction, a restart-condition is advised to use to separate the two from each other. This way no other master can grab control of the bus between the two transactions and potentially corrupt the just set up address register in the µModule.

Read transactions

byte	0	2..n
Value	(0x00-0xfe) | 0x01	0x00-0xff
Explanation	Device address + read	Transaction data

Read transactions start with the device address and the read/write bit set to identify a read transaction. Subsequent bytes are read from the µModule. The module first outputs the content of the register pointed by

the address register. It than increments the address register. The host acknowledges all but the last byte of the transmission. The last byte is required to have a negative acknowledgment. After the negative acknowledgment, a stop or re-start condition must follow, and thus the transaction must be terminated.

16-bit addressing

The 16-bit addressing mode is very similar to the 8-bit addressing mode, only the first two bytes of a write-transaction will update the address-register, instead only the first byte like in the 8-bit addressing mode. This gives access to large register-banks of a maximum length of 65536 bytes. The actual register file in a given µModule can be lass than this number. Whether the module implements wrap-around for the address register is module specific. It is also module-specific if the last register (address 65535) is implemented as a command register and if that address wraps around into the address space of the other registers.

The protocol identifies two transactions.

Write transactions

byte	0	1	2	3..n (optional)
Value	(0x00-0xfe) & 0xfe	0x00-0xff	0x00-0xff	0x00-0xff
Explanation	Device address + write	Transaction start address		Transaction data

This transaction starts with the address of the device and the read/write bit cleared to identify a write transaction. It is followed by a register address in the second and the third byte. Whether the address is little- or big-endien is module-specific, must most modules use big-endien encoding. It is invalid to terminate the transaction after only the first byte of the address being transferred. The content of the address-register in the µModule after such a transaction is undefined. After the second byte of the register-address is received, the addressed µModule updates its internal address register. If subsequent bytes are followed, these bytes a re written to the register in the bank, pointed by the address register, after which the address-pointer is incremented. This process follows until a stop, or restart condition terminates the transaction. It is important to note, that if 0 data bytes follow the register-address, no updates to the register-bank takes place, but the address-register is assigned a new value. This is the preferred way of updating the address register to a known state prior to a read transaction. It is also important to note that if such a write transaction is followed by a read transaction, a restart-condition is advised to use to separate the two from each other. This way no other master can grab control of the bus between the two transactions and potentially corrupt the just set up address register in the µModule.

Read transactions

byte	0	2..n
Value	(0x00-0xfe) | 0x01	0x00-0xff
Explanation	Device address + read	Transaction data

Read transactions start with the device address and the read/write bit set to identify a read transaction. Subsequent bytes are read from the µModule. The module first outputs the content of the register pointed by the address register. It than increments the address register. The host acknowledges all but the last byte of the transmission. The last byte is required to have a negative acknowledgment. After the negative acknowledgment, a stop or re-start condition must follow, and thus the transaction must be terminated.

Multi-byte registers

If a register in the register-bank is 16- or 32-bit long, the content of the whole register must be modified or queried. It is not possible and should not be attempted to read only the first byte of a 32-bit register or write only the upper byte of a 16-bit register. Whether multi-byte registers are little- or big-endien is module-specific, but most modules use big-endien registers. It is the responsibility of the µModule to ensure atomic access to multi-byte registers. For write operations, the new value is affective only after the last byte of the register is updated. For read operations, the sampling of the whole register takes place right before the first byte of the register is read. This way the software in the Central Controller can be sure that no updates happen in between the first and the second byte of a 16-bit register is read for example or no internal operation of the µModule will see a half-updated 32-bit register value.

Command-port codes

The command register is the last addressable register in the register bank (address 255 for 8-bit addressing and 65535 for 16-bit addressing). The following common command codes have been identified:

STORE_CONFIG_CMD(0xff):

This command stores the current configuration of the µModule in permanent storage. Upon next power-up the module will restore this configuration and will start operation from this state. Note, that not all registers in the register bank can be saved to permanent storage. Check the specification of the µModule for details.